**ORACLE体系结构**

云和恩墨(北京)信息技术有限公司

技术顾问 燕鑫

http://www.enmotech.com

**文档控制：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **序** | **版本号** | **更改人** | **日期** | **备注** |
| 1 | 1.0版 | 燕鑫 | 2018-07-04 | 初始版本 |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[1. 体系结构概述 - 4 -](#_Toc519202270)

[2. INSTANCE - 4 -](#_Toc519202271)

[2.1 内存结构（Memory Structures） - 5 -](#_Toc519202272)

[2.1.1 SGA（System Global Area） - 5 -](#_Toc519202273)

[2.1.2 PGA（Program Global Area） - 15 -](#_Toc519202274)

[2.1.3 UGA（User Global Area） - 16 -](#_Toc519202275)

[2.2 进程架构（Process Architecture） - 17 -](#_Toc519202276)

[2.2.1 客户端进程（Client Processes） - 17 -](#_Toc519202277)

[2.2.2 服务器进程（Server Processes） - 22 -](#_Toc519202278)

[2.2.3 后台进程（Background Processes） - 23 -](#_Toc519202279)

[3. DATABASE - 28 -](#_Toc519202280)

[3.1 物理存储结构（Physical Storage Structures） - 28 -](#_Toc519202281)

[3.1.1 数据文件（Data Files） - 29 -](#_Toc519202282)

[3.1.2 控制文件（Control Files） - 31 -](#_Toc519202283)

[3.1.3 在线重做日志（Online Redo Log） - 32 -](#_Toc519202284)

[3.2 逻辑存储结构（Logical Storage Structures） - 36 -](#_Toc519202285)

[3.2.1 逻辑结构概要 - 36 -](#_Toc519202286)

[3.2.2 Data Blocks - 37 -](#_Toc519202287)

[3.2.3 Index Blocks - 46 -](#_Toc519202288)

[3.2.4 Extents - 46 -](#_Toc519202289)

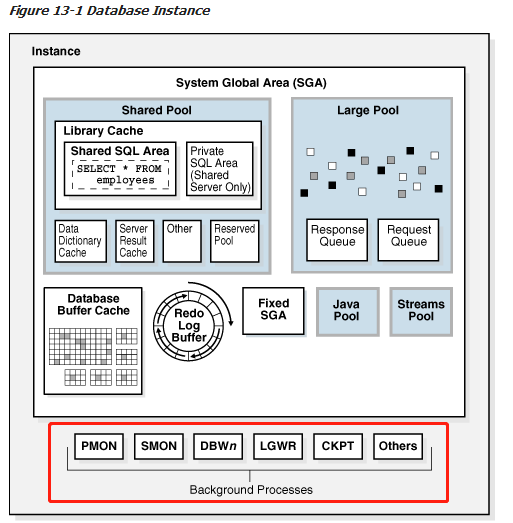
[3.2.5 Segments - 46 -](#_Toc519202290)

# 体系结构概述

ORACLE体系结构分database和instance。

# INSTANCE

SGA + background process = instance，如下图：

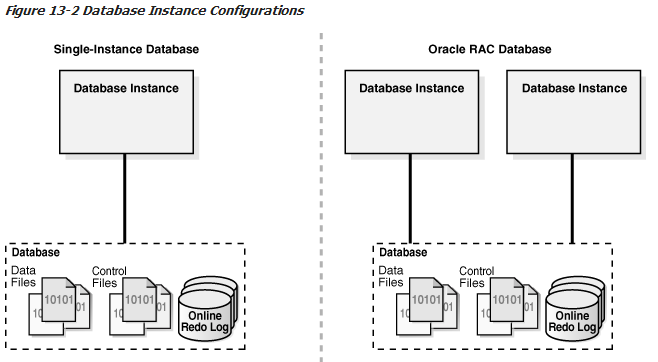


## 实例概述

### Instance Configurations

有如下俩种配置：

1. Single-instance configuration：实例与数据库是1对1。
2. Oracle Real Application Clusters (Oracle RAC) configuration：实例与数据库是多对1。



一个实例同时只能对应一个数据库。

### Duration of a Instance

就是从执行了startup到shutdown immediate。

重点是可以通过如下查询获得本次实例启动的时间：

SQL> SELECT TO\_CHAR(STARTUP\_TIME,'MON-DD-RR HH24:MI:SS') AS "Inst Start Time" FROM V$INSTANCE;

Inst Start Time

---------------------------

MAY-30-18 18:10:15

### Oracle System Identifier (SID)

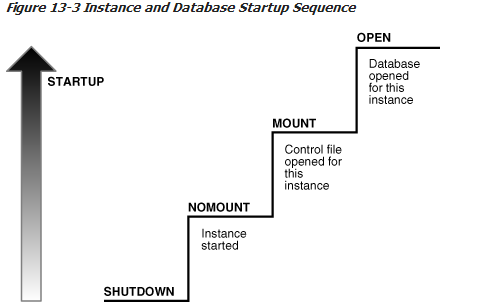
SID是instance在一个特定的host上的unique name。这个最明显的体现在RAC上，不同的host上的SID是不同的。

ORACLE数据库默认是用SID可以用来定位pfile，pfile定位control file。

由ORACLE\_SID和ORACLE\_HOME唯一确定一台host上的一个数据库。

### 实例的startup和shutdown

#### 实例和数据库启动流程



首先，操作系统认证的登陆方式就不说了。如果想通过网络连接数据库进行实例启动，需要满足俩个条件：

1. 数据库必须静态注册在监听中；
2. 必须以sysdba的身份连接数据库

##### No Mount阶段

首先是启动数据库到nomount阶段，经历如下流程：

1. 如果指定pfile=xxx，那么就会读指定的pfile。  
   如果没指定pfile=xxx，那么就先找spfile（server parameter file），找不到就到默认的位置（$ORACLE\_HOME/dbs）下找pfile（initialization parameter file）。
2. 读参数文件中记录的初始化的参数值。
3. 按照pfile里指定的参数值来分配SGA，要是pfile里面没指定，就按系统默认分配，总之这一步要分配SGA。
4. 启动后台进程（background processes）
5. 写alert日志，会把pfile里面写的参数配置都显示出来。

##### Mount阶段

这个阶段就是实例把跟该实例相对应的数据库启动到mount阶段。

实例读参数文件中的CONTROL\_FILES的值，找到控制文件并打开它们，读到其中的数据文件和在线重做日志的位置。

但是一般的数据库操作并不能进行，因为此时数据文件和redo日志还是可读写状态。

##### Open状态

数据库打开了，那么就可以进行正常的操作，比如说就能查用户的表的数据了。那么打开一个数据库，数据库主要做了如下几件事情：

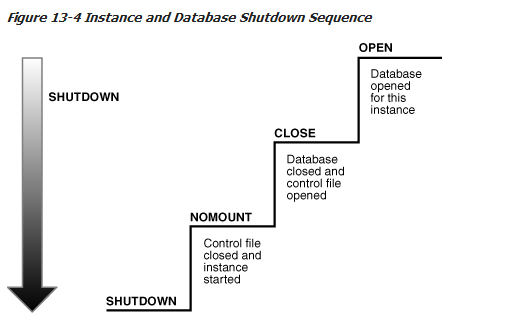
1. 打开online状态的数据文件（不包括undo数据文件），如果是上一次数据库关闭前就是offline的，数据库再open这些文件仍然是offline的；
2. 获得undo表空间：如果有多个undo表空间，那么就使用参数文件中UNDO\_TABLESPACE指定的undo表空间，如果该参数没指定，那就是用第一个可用的undo表空间。
3. 打开online redo log files

##### Read Only模式

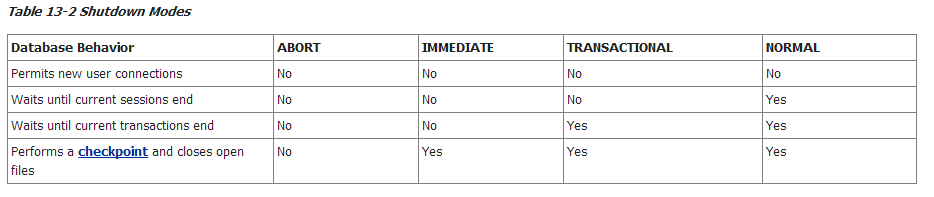
数据库如果处于read only模式，我总结一下几点我认为比较关键的：

1. datafiles和redo log files是不能写了。
2. 但是临时表空间还能写
3. 控制文件、alert日志、trace文件照常工作

#### 实例和数据库关闭流程



##### Shutdown的模式



Abort：强行关闭，不等SGA中的内容写到datafile中就关数据库，再启动时需要实例恢复，如果是在PDB中shutdown abort跟shutdown immediate是一个效果。

Immediate:一致性关闭，这种关库速度第二快，这种关闭方式会中断正在进行的事务和查询，将未提交的已修改的数据回滚，把提交了的修改写入数据文件。所以说，如果执行shutdown immediate，那么把LOCAL=NO的进程杀了再执行时没问题的，一样的，你做比数据库自己做还快。

Transactional：等事务完了再关。

Normal：默认关闭方式，也就是说只输入一个shutdown就是这种关闭方式。这种关闭方式要等所有连接都自己断开了才关闭数据库。

##### Close状态

这时执行了shutdown（除了abort）以后的第一个状态，此时，数据库将SGA中的修改写入datafiles和redo log files并关闭它们。此时control file还保持打开状态。

如果时abort就没有这些流程。

##### Unmount状态

在close之后，ORACLE在这一步把instance和database的关系切断。此时controlfile也被关闭，但是此时SGA还未释放且后台进程还存在。

##### 彻底关闭

最后把SGA释放占有的系统内存，后台进程被终止掉，一个实例就此结束。

### 实例恢复（Instance Recovery）

#### 实例恢复概述

首先，有俩种情况需要实例恢复：

1. 所有实例异常挂掉（包括shutdown abort），注意是所有，此时再open数据库进行的是crash recovery。
2. 集群中个别实例挂掉。

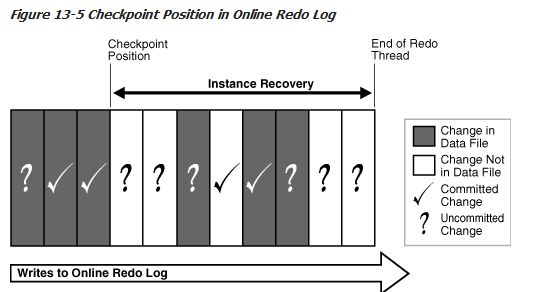
实例恢复是自动的由smon进行的，那么实例恢复发生的依据是什么？

当数据库正常运作的时候，control file中标记redo log是open的；当数据库处于关闭状态时，control file中标记redo log是close的；那么当control file中标记redo log是open的，但此时没有一个活的实例来匹配对应thread的redo log时，就要进行instance recovery。

实例恢复主要做以下俩件事，目的就是要保证数据的consistency：

1. 提交的事务记录在redo中，但是修改的块还没来及写到数据文件上，那么要把这部分数据恢复到数据文件上。
2. 没提交的事务，但是已有部分块写到数据文件上了，那么要把这部分数据回退掉。

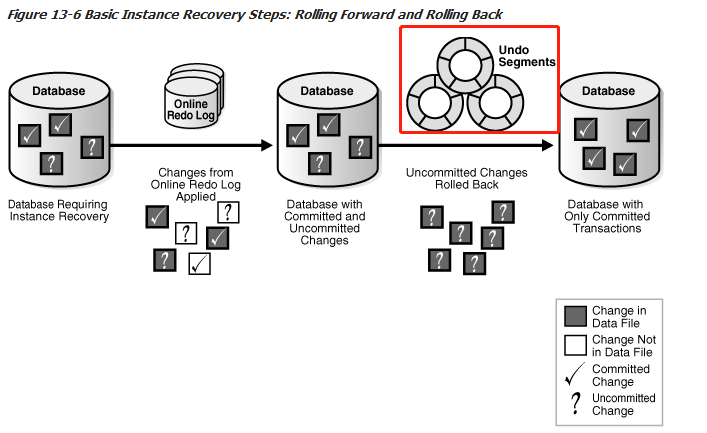
下图是实例恢复的内容：



在实例恢复的时候，可能有些改变已经写在数据文件上了，那么这些不会再重复写，只会把还没写在数据文件上的补上去。

#### 实例恢复的阶段

实例恢复有俩个阶段rolling forward（crash recovery）和rolling back（transaction recovery）。如下图所示：



第一个阶段前滚，是把redo log上最新的checkpoint到redo log最后的sql都执行一遍，这其中包括执行了一半没提交的事务。该步骤会产生相应的undo。该阶段是确保把所有已经提交了的事务造成的数据修改都恢复到数据文件上了。

第二个阶段是回滚，那么还有一部分数据是在事务还没提交就已经写到数据文件上了，同时第一阶段（crash recovery）把没有提交的事务也做了一遍，这部分数据也写到了磁盘上。这俩部分数据都要进行回滚。

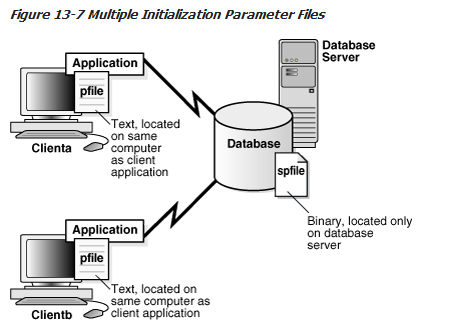
### 参数文件（Parameter Files）

参数文件就俩种格式：spfile(server parameter file)和pfile。

前面在实例启动部分说过，spfile是实例启动到nomount时首先默认查找的参数文件。

Spfile是二进制文件，pfile是文本文件。

一个实例对一个参数文件，各实例自己的参数文件是放在实例的host上的。那么像RAC这种多实例，多参数文件怎么办呢？如下图：

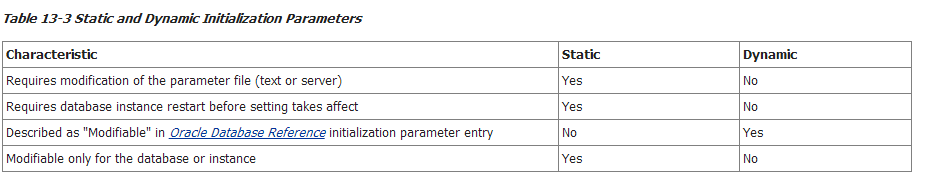


还是一个节点一个pfile，注意，一定是pfile，pfile中就一条记录，就是spfile在共享存储上的位置。

有30个左右的参数可能需要设置调整，其余的默认就可以。这些参数比如db\_block\_size呀等等。如下是12.2的数据库实例参数规范：



系统参数分静态（static）和动态（dynamic），作用范围和区别如下：



修改参数的时候可以指定scope：

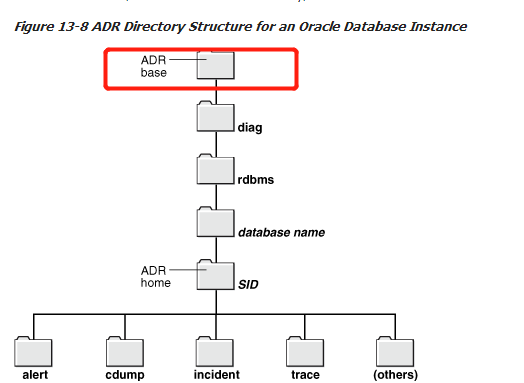
1. scope=memory参数的修改只在本次实例生命周期内生效，重启数据库修改将失效。
2. scope=spfile参数修改是修改在参数文件上的，是永久的，数据库重启了才生效（前提是得有spfile）。
3. scope=both就是立马生效，同时写在spfile上永久生效。如果不指定scope，这就是默认选项。

### 诊断文件（Diagnostic Files）

Automatic Diagnostic Repository (ADR)的主要特点：

1. Unified directory structure
2. Consistent diagnostic data formats
3. Unified tool set

主要还是统一的目录结构，我们看下这个目录结构：



这个ADRbase默认是ORACLE\_BASE，也就是说ASM实例有自己的ADR，数据库实例也有自己的ADR。

然后说下这几个目录都放些啥：

1. alert目录：放的是alert日志的xml版本
2. incident目录：放的是incident事件的trace文件，该目录下面还分了很多子目录（应该是按incident\_id分的），这些子目录下放的才是对应的incident的trc文件和trm文件。
3. trace目录：放的是alert日志的text版本，和各种trace文件。Trace文件的文件格式分俩种。  
   后台进程的：SID\_bg process name\_process number.trc（mytest\_reco\_10355.trc）  
   服务进程的：SID\_ora\_process number.trc（mytest\_ora\_10304.trc）  
   扩展名是trm的文件放的是对应trc文件的metadata。  
   trace文件最多可以分5片，受MAX\_DUMP\_FILE\_SIZE参数影响，总大小超过这个参数后，会自动覆盖其中一片，然后继续写trace，一般不会覆盖第一片。
4. log目录下的ddl目录：放的是ddl log。

可以进行如下查询，查看各路径：

SQL> SELECT NAME, VALUE FROM V$DIAG\_INFO;

NAME VALUE

--------------------- ------------------------------------------------------

Diag Enabled TRUE

ADR Base /u01/app/oracle

ADR Home /u01/app/oracle/diag/rdbms/prodcdb/PRODCDB

Diag Trace /u01/app/oracle/diag/rdbms/prodcdb/PRODCDB/trace

Diag Alert /u01/app/oracle/diag/rdbms/prodcdb/PRODCDB/alert

Diag Incident /u01/app/oracle/diag/rdbms/prodcdb/PRODCDB/incident

Diag Cdump /u01/app/oracle/diag/rdbms/prodcdb/PRODCDB/cdump

Health Monitor /u01/app/oracle/diag/rdbms/prodcdb/PRODCDB/hm

Default Trace File /u01/app/oracle/diag/rdbms/prodcdb/PRODCDB/trace/PRODCDB\_ora\_72981.trc

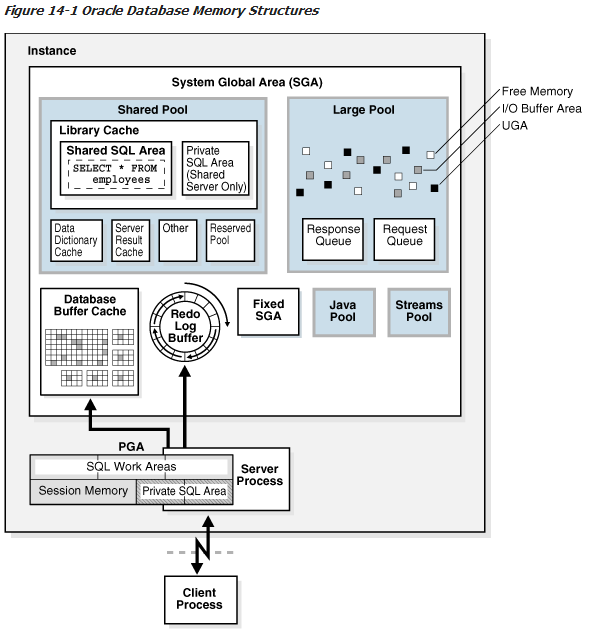
Active Problem Count 1

Active Incident Count 17

下面再列举一下alert日志记录的东西：

1. internal errors (ORA-600), block corruption errors (ORA-1578), deadlock errors (ORA-60)
2. Administrative operations ：STARTUP, SHUTDOWN, ARCHIVE LOG, and RECOVER
3. 共享服务器进程和dispatcher进程相关的一些错误和信息
4. 自动刷新物化视图发生的错误

## 内存结构（Memory Structures）



### SGA（System Global Area）

SGA和后台进程共同构成了instance，也就是说每个实例都有自己得SGA。

SGA是在实例启动的时候ORACLE自动分配的一块内存区域，当实例关闭的时候这块内存会被回收(reclaim)。

SGA的components有：

1. Database Buffer Cache
2. Redo Log Buffer
3. Shared Pool
4. Large Pool
5. Java Pool
6. Streams Pool
7. Fixed SGA

SGA的组件（components）中，除了redo log buffer以外，剩余的组件都是由一段段连续的内存够成的，这每一段内存的大小是统一，名字叫做granule（粒度）。

Granule size（粒度大小）由SGA的大小确定。

#### SGA的组件

##### Database Buffer Cache

Buffer cache里放的是从数据文件里读出来的块的copy。

###### Buffer cache的作用

1. 优化物理IO，Oracle在buffer cache中更新了数据块，先把这个改变写入到redo log buffer，等到事务提交后，Oracle只是把redo log buffer中的内容先写到online redo log中，但并不会立马将buffer cache中的数据块写到数据文件，而是稍后等dbwr在后台进行lazy writes；
2. 将经常访问的块放在buffer cache中，不常访问的写到磁盘中。

###### Buffer的3种状态

1. unused：从来没用过的或者目前没在用的buffer，数据库会优先使用这样的buffer；
2. clean：这样的buffer虽然块里有数据，但是块的数据已经没用了，不需要被checkpoint了，那这样的块也可以被数据库pin住并且再使用。
3. dirty：这样的buffer里的数据还没有写到数据文件中，必须等到数据库checkpoint了之后，才能被重新使用。

Buffer只有俩种访问模式（access mode）：pinned或者free（unpinned），当一个会话访问一个被pin住的buffer时，该buffer是不会被踢出内存的，同时一个buffer不能同时被多个会话修改。

###### Buffer 的俩种模式（mode）

1. current mode：current mode get就是当前事务你修改后就能读到你当前修改的块
2. consistent mode：consistent read get就是在有事务没提交的时候，读到的是undo的块，是当前修改的块的前镜像（consistent read clone）。

###### Buffer的IO

就是所谓的逻辑IO（logical I/O）,就是在buffer cache中对buffer们进行读写，如果没有找到需要的buffer，那就要发生物理IO，即物理读：

1. 如果是从datafile读就通过物理IO（physical I/O）从硬盘上copy需要到块到buffer cache中，然后再进行一次逻辑读。
2. 如果是从tempfile读（当内存不够，oracle不得不把一部分数据写到临时表中，只后再读这部分数据），就只有物理IO，而绕过buffer cache，不发生逻辑读。

###### Buffer的替换算法（replacement algorithm）:

说的就是LRU（least recently used）算法。就是有一个LRU链表，链表上有指针指向脏块和非脏块。同时LRU链表有热端（hot end）和（cold end），靠近热端的是常使用的hot buffer，靠近冷端的就是不常使用的cold buffer。当然是hot buffer更容易长久留在buffer cache中。

理论上只有一条LRU链表，但是当存在对同一块并发访问时，实际上会存在多条LRU。（啥意思？）<https://blog.csdn.net/haiross/article/details/40395877>

###### 写Buffer到磁盘：

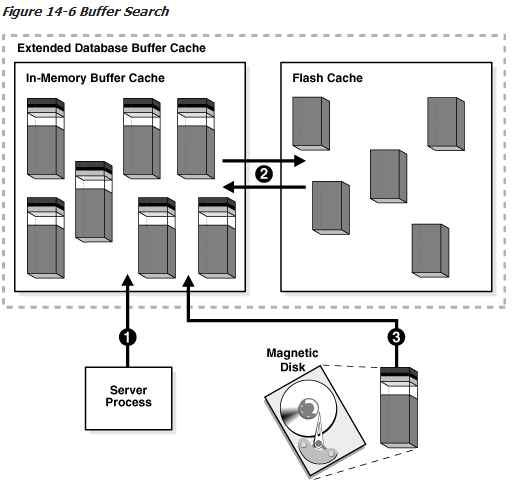
DBW（database writer）负责把cold and dirty buffers写到磁盘中，下列3中情况可触发DBW：

1. 服务进程（server process）为新读入buffer cache的块找不到clean buffer了，oracle就会把cold and dirty的块从LRU移到写队列（write quene），然后DBW把写队列的块们写到磁盘上（尽量会使用多块写）；
2. 发生checkpoint的时候；
3. 表空间被置为只读或者offline时。

###### 从Buffer cache中读buffer

如果要读新块的数据到buffer cache，就要把clean状态的buffer“清理干净”，放新块的copy放进来，但是如果这写被清理掉的块后面又要被使用了，在不开flash cache的情况下，要从磁盘再重新读到buffer cache中。

如果开了flash cache（受系统限制），那就是extended buffer cache，举个图例就不深究了：



###### Buffer Touch Counts

数据库通过计数的方式来衡量一个buffer的访问热度，方法就是，当一个buffer被pin住的时候（被访问的时候），那每隔3秒就增加一次该buffer的touch count，touch count每3秒一变的规则是为了防止一个buffer再短时间内被频繁访问而导致计数暴涨。

当一个buffer已经处于LRU的cold端，但是touch count却很高，那它会被“移动”到hot端。注意，这里所谓的移动，移动的并不是buffer，而是指向buffer的指针。

Touch count的引入是为了避免LRU链表频繁的进行指针的移动。

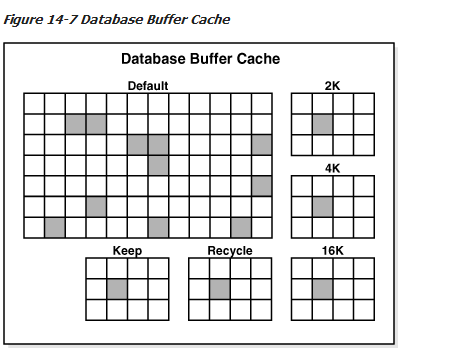
###### Buffer Pools

所谓buffer pool就是一组buffer的集合。数据库把buffer cache分成一个或多个buffer pool来管理blocks。这些pool是在本质上是没区别的，包括对块的cache和踢除的算法都是没区别的。

主要有如下3种buffer pool：

1. Default pool：  
    这个池就是进行正常的块的cache或者age out的buffer pool，如果不手动添加别的buffer pool那这个pool就是buffer cache，或者说buffer cache就这一个pool。
2. Keep pool：  
    这个池放的就是default pool里没地方放但是你又想经常用的块。
3. Recycle pool：  
    这个池放的是不常用的块，就是你用完就扔，不想让这些块去占default pool的地方。
4. Db\_nk\_cache：  
    这是一类池，oracle默认块大小是8k，但如果想建非8k块的表空间，比如16k大小的块，那就得为16k的块也指定一块和default pool一样的内存区域，来专门cache 16k块。这样的池有2k、4k、16k。

如下就是一个啥都有的buffer cache示例图：



###### 全表扫描对buffer cache的使用

正常情况下，当需要将一部分新的表块存入到buffer cache中时，oracle会将这些新块放在LRU中部，这样之前的热块依然时热块，而也不会因为这部分新块还没来得及确定能否变为热块，就被再次新进来的新块挤出buffer cache。

当进行全表扫描的时候，Oracle会进行判断是否对其全表的块（HWM以下的块）进行cache。

主要分为3种情况：

1. 小表：所谓小，就是表的大小于buffer cache大小比值很小。这种情况，数据库会把全表的表块放到LRU链表尾部。这样做是合理的，事实上在OLTP系统中我们并不希望出现全表扫描，即使有也并不希望其高频次的执行。
2. 大表：所谓大，就是表的大小比buffer cache小不了多少，甚至比buffer cache还要大。这样的表，oracle会选择将其直接放入PGA中，绕过SGA，进行直接路径读（direct path read），纯粹的物理读，即这样的物理读之后，并不会有逻辑读。
3. 中表：对于中型表的判断，oracle有自己一套复杂的算法，会根据该表本次扫描于上次扫描的时间间隔、上次被踢出buffer cache的时间、buffer cache剩余的空间来综合判断。判断结果有俩种，按小表处理进行cache，放入LRU尾端；按大表处理，放入PGA，绕过SGA，进行直接路径读。

当开并行的时候，如果内存足够大，而且有充足的资源（大概率不是OLTP系统），oracle会把并行读取的数据块放入buffer cache中，而不是通过PGA进行直接路径读。

这里有俩个属性着重进行强调：

1. alter table … cache：cache属性可以当表被cache到buffer cache中时（意思就是本身这条指令并不会使表cache到buffer cache中），表块被放在最近最多使用（MRU）的尾端，而不是像对其他全表扫描那样把表缓存在最近最少使用（LRU）尾端，这样，表的数据块们就不那么容易被age out了，但是放进来以后，就同其他块一样同等对待。所以，这是个给小表用的！
2. alter table … storage buffer\_pool keep：显而易见，这句的意思就是把该表缓存到keep pool中，关于keep pool的介绍前面有，这就就不赘述了。

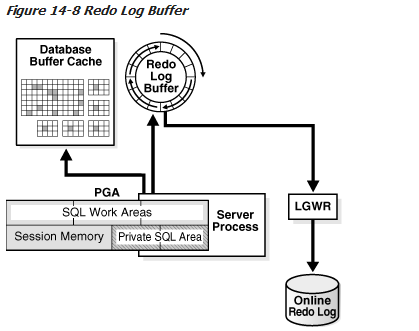
当然，指定了上述俩个属性的表，并不代表一定能放到buffer cache中，要是本来就放不下的话，还是要进行直接路径读。

##### Redo Log Buffer

Redo log buffer是缓存数据库改变信息的，它有如下特点：

1. 是一个循环缓存区（circular buffer），每一组信息，也就是说一个redo条目（redo entry），是一块连续的缓存。
2. 会把其中缓存的信息通过LGWR写到active状态的在线重做日志组中，这个写是一个连续写（sequential writes），不同于DBWR的离散写（scattered writes），所以很快。
3. 大小由LOG\_BUFFER初始参数决定。
4. 同SGA其他组件不同，分配的内存不以granule size为单位。

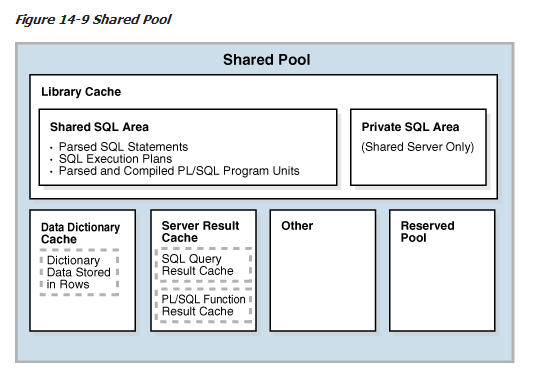
最后我们来上个图：



##### Shared Pool

这个区域是真正处理数据库各种操作的地方，所以非常重要！

这里直接用一副图来展示其大体构成，然后我们在分别来看其中的每一部分：



主要部分有4大块：

1. Library Cache
2. Data Dictionary Cache
3. Server Result Cache
4. Reserved Pool

###### Library Cache（库缓存）

这部分在《游标和绑定变量》中有详细的分析和实验。这里做一些总结性的概述。

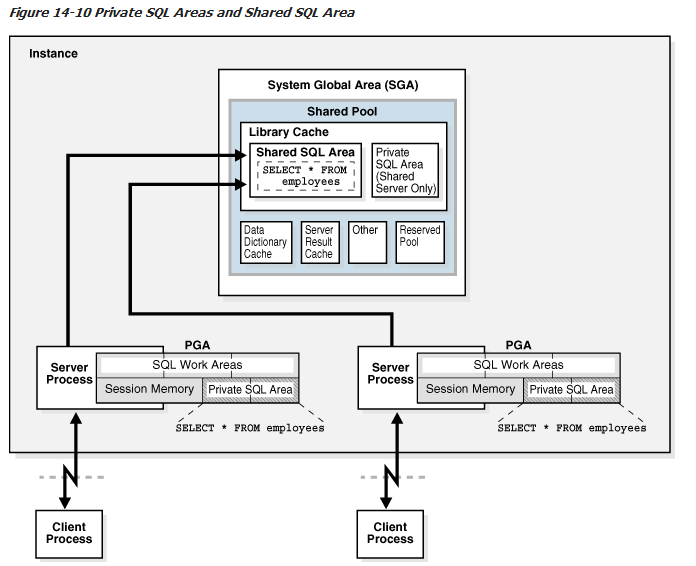
库缓存主要存放的就是可执行的sql（也就是通过语法语义检查后的）和PL/SQL的代码。库缓存大体分为俩部分：

1. Shared Sql Area
2. Private Sql Area（共享连接专属）

我们先说必有的也是主要的构成部分shared sql area：

每一个sql执行都要经过俩个area，如果是专有连接，就先要经过PGA的private sql area，如果是共享连接，那就会经过上面所说的库缓存中的private sql area，不管是哪个private sql area，都可以根据session cursor的句柄找到唯一的Shared Sql Area。

下面展示的是专有连接时的情况：



Oracle将PL/SQL代码+java classes称作program units，对于program units的处理跟说白了也是处理其中的sql，所以跟处理sql是完全一样的。

库缓存的内存也是遵循LRU算法，我们下面来罗列一下cursor被踢出shared sql area的几种主要情况：

1. sql相关的对象收集过统计信息，那这类cursor会在一段时间后被踢出shared sql area，为的是不久之后sql再次被执行时，新的执行计划可以反映新的统计信息。
2. sql相关的对象被ddl修改过，增删列、修改字段、增删索引都会使cursor失效，sql再次执行的时候会重新解析。
3. 偷梁换柱的方式修改profile，并绑定执行计划。
4. 修改dbname，shared pool里的信息会全部被清空。
5. 执行alter system flush shared\_pool也可以把shared pool直接清干净。

###### Data Dictionary Cache

就是进行sql查询的时候，要查数据字典记录的信息（因为数据字典记录对象的信息），这时候oracle就会把基表中的相关对象的记录的这一行读到data dictionary cache中（而不像buffer cache中把包含这一行的块都读到内存中），所以有叫做row cache。

如果 shared pool 设置过小，那么 data dictionary cache 也会相应的过小，没有足够的空间存储ORACLE的系统数据字典信息，从而导致ORACLE需要从硬盘读取数据字典信息，我们查看执行计划的时候看到的recursive calls就是代表的是从磁盘读取数据字典的次数。

查看data dictionary cache的命中率：

SQL> select sum(gets),sum(getmisses),(1-(sum(getmisses)/(sum(gets)+sum(getmisses)))) hitratio from v$rowcache;

SUM(GETS) SUM(GETMISSES) HITRATIO

---------- -------------- ----------

49856561 1596402 .968973565

###### Server Result Cache

就是结果集缓存，用处不大，有俩部分：

1. SQL Query Result Cache
2. PL/SQL Function Result Cache

###### Reserved Pool

这个池是预留给shared pool的，当shared pool没有足够的空间放连续的大块时，会将这些大块放入该池。

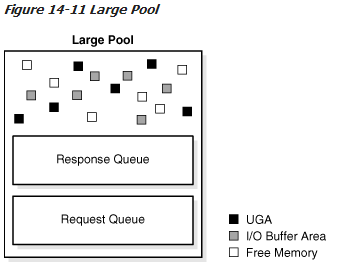
##### Large Pool

大池是一块可选择配置的内存，意思就是可以不配，如果不配，共享连接的UGA就要从shared pool中分配。

大池顾名思义，就是一块大的内存池，主要能为如下3种情况提供内存：

1. 共享连接的UGA和Oracle XA接口
2. 并行执行sql的消息缓存区（Message buffers）
3. RMAN的I/O Slaves

大池并不像shared pool中的reserved pool，reserved pool也是使用的LRU链表，但是大池中没有LRU链表，大池中分配给某一会话的内存只有当会话结束才会释放。



##### Java Pool

##### Streams Pool

##### Fixed SGA

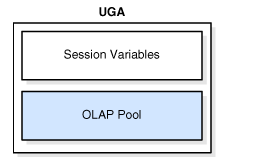
固定的一块内存，数据库决定其大小，随版本不同而不同，手动改不了。

放着后台进程需要访问的一些数据库和实例的状态信息，进程之间通信的信息（比如锁的信息）。

#### SGA的管理

### PGA（Program Global Area）

### UGA（User Global Area）



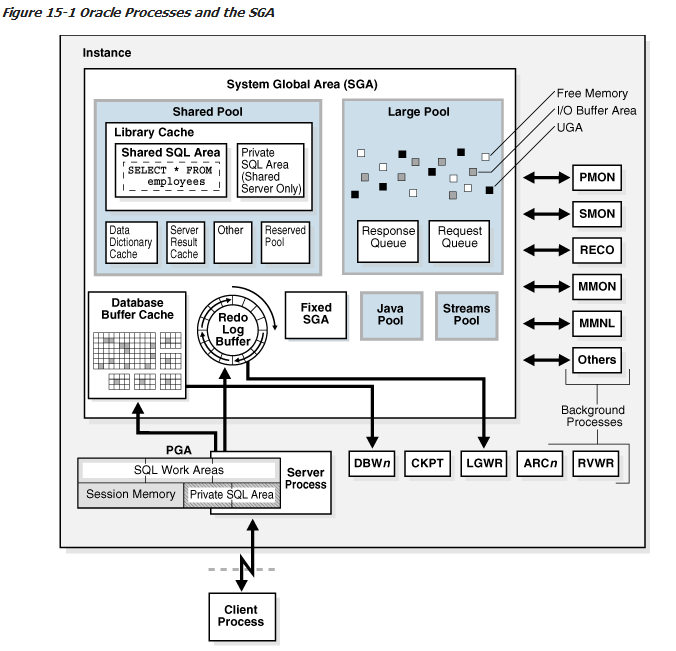
UGA是session memory，分俩部分：

1. Session Variables存着会话的状态，包括登陆信息还有一些会话的额外信息。
2. OLAP Pool

UGA必须同会话是一一对应的，也就是会话在UGA在，所以对于共享服务连接模式，UGA只能存在于SGA中，因为一个会话对应得服务器进程可能会发生变化，所以不能存在于PGA中。

如果是专有连接模式，UGA是存在于PGA中的。

## 进程架构（Process Architecture）



### 客户端进程（Client Processes）

客户端进程是当你使用工具（比如sqlplus）连接数据库的时候产生的一个进程，这个进程是客户端上的，有时候也叫user process，就是用来跑用户应用程序的。

只所以着重强调这一点，是因为它跟后面要总结的服务器进程完全不是一回事儿，做的事情都不一样。

#### 与服务器进程的区别

事实上，不管你连不连数据库，你只要在客户端上开启一个应用，就会有一个进程，这个进程就是为了运行该程序的。如果这个程序要连数据库，那么就由这个client process去跟数据库所在的服务器的server process通信，把代码交给server process，再由server process拿着代码在数据库中运行出结果，再返回给client process。当然，这说的是专有连接，如果是共享连接的话，就是client process跟调度器通信，而不是直接跟server process通信，但道理是完全一样的。

Client process可以在任何一台主机发起，但是server process只能在服务器创建；server process可以读写SGA，但是client process绝对做不到这一点。

现在我们在服务器的主机上，开一个sqlplus：

[oracle@host01 ~]$ sqlplus sys/oracle@pdbprod1 as sysdba

SQL\*Plus: Release 12.2.0.1.0 Production on Sat Jul 7 12:06:03 2018

Copyright (c) 1982, 2016, Oracle. All rights reserved.

Connected to:

Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production

SQL>

然后我们来看进程：

[oracle@host01 ~]$ ps -ef | grep -e PRODCDB -e sqlplus | grep -v grep|grep -v ora\_

oracle 108213 103799 0 12:06 pts/6 00:00:00 sqlplus as sysdba

oracle 108215 1 0 12:06 ? 00:00:00 oraclePRODCDB (LOCAL=NO)

这里注意，虽然我们连的是pdb，但是查要查cdb的进程，pdb是没有自己的进程的，详细的这里先不说。

我们看到108213就是我们的客户端进程，108215就是服务器进程（注意LOCAL=NO），此时host01作为程序发起者（执行sqlplus）是客户端，而作为数据库服务器（即信息接收者），host01同时也是服务器端。

#### 连接与会话的区别（Connections and Sessions）

连接（connection）是一条从client process到数据库实例的物理（pysical）的通信路径。

一般来说，一个连接是发生在client process到server process或者dispatcher或者Oracle Connection Manager (CMAN)之间。

会话（session）是一个逻辑概念，就是数据库实例内存中的一块logical entity，代表的是当前登陆到数据的用户的状态。

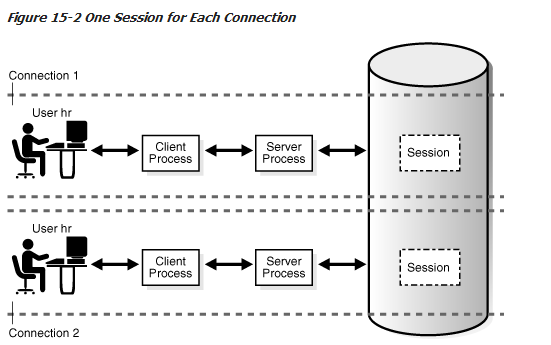
当一个用户通过了数据库的密码验证，那么数据库就会为这个用户创建一个会话。直到这个用户退出了连接数据库的应用程序或者断开了连接（这个不一定），会话才被终止。

如果配了Oracle Net connection pooling，那可能连接断开了，但会话还留着。

一个连接可以有0个、1个、多个会话建立在之上。而会话是独立的，互相不受影响的（各做各的事务）。

##### 一个连接一个会话

我们先看最普遍的一个连接一个会话：



##### 一个连接多个会话

那么一个连接怎么会有多个会话呢？举个例子：如果我们在一个会话中发起autotrace，会是什么效果呢：

首先我们先连接数据库（专有连接），并查询当前会话的sid：

[oracle@host01 ~]$ sqlplus sys/oracle@pdbprod1 as sysdba

SQL\*Plus: Release 12.2.0.1.0 Production on Sat Jul 7 12:06:03 2018

Copyright (c) 1982, 2016, Oracle. All rights reserved.

Connected to:

Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production

SQL> select sid,con\_id from v$mystat where rownum=1;

SID CON\_ID

---------- ----------

629 3

然后查看当前sid对应的服务器进程地址：

SQL> select sid,paddr,status from v$session where sid='629' ;

SID PADDR STATUS

---------- ---------------- --------

629 000000007B53B5B0 ACTIVE

我们来按PADDR进行查询：

SQL> select sid,paddr,status from v$session where paddr='000000007B53B5B0';

SID PADDR STATUS

---------- ---------------- --------

629 000000007B53B5B0 ACTIVE

现在我们打开autotrace再查：

SQL> set autotrace on

SQL> select sid,paddr,status from v$session where paddr='000000007B53B5B0';

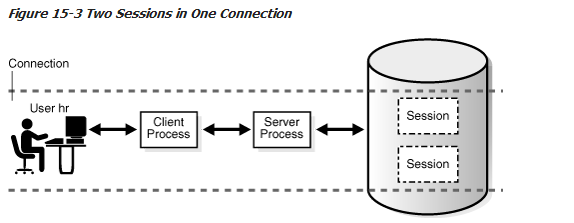
SID PADDR STATUS

---------- ---------------- --------

623 000000007B53B5B0 INACTIVE

629 000000007B53B5B0 ACTIVE

我们看到出现了一个inactive的会话，这个会话就是oracle为autotrace建立的一个监控会话。我们看到服务器进程还是同一个，但同时存在俩个会话。



##### 一个连接0个会话

现在我们再来看一个会话没了，但是服务器进程还在的例子，也就是所谓的0个会话：

退出上个会话（使用exit）：

SQL> select sid,sql\_trace,event,module,paddr,status from v$session where paddr='000000007B53B5B0';

no rows selected

重新使用sqlplus sys/oracle@pdbprod1 as sysdba连接数据库，并查询本会话的sid和服务器进程：

SQL> select sid from v$mystat where rownum=1;

SID

----------

629

SQL> select sid,paddr,program from v$session where sid=629;

SID PADDR PROGRAM

---------- ---------------- ------------------------------------------------

629 000000007B53B5B0 sqlplus@host01 (TNS V1-V3)

SQL> select addr,spid from v$process where addr='000000007B53B5B0';

ADDR SPID

---------------- ------------------------

000000007B53B5B0 116415

然后我们不使用exit，而使用disconnect来断开会话连接：

SQL> disconnect;

Disconnected from Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production

现在我们在另一个窗口新开一个会话来查询刚才这个会话：

SQL> select sid,paddr,program from v$session where sid=629;

no rows selected

会话确实没了，我们继续查服务器进程：

[oracle@host01 ~]$ ps -ef | grep 116415 | grep -v grep

oracle 116415 1 0 14:08 ? 00:00:00 oraclePRODCDB (LOCAL=NO)

但是服务器进程还在。

现在我们使用exit，退出sqlplus，再查：

[oracle@host01 ~]$ ps -ef|grep 116415|grep -v grep

现在服务器进程消失了。

### 服务器进程（Server Processes）

服务器进程就是代表数据库跟客户端进程进行通信的，处理client process“携带”的各种请求。

服务器进程主要可实现如下功能：

1. 处理sql，包括生成sql执行计划并执行该计划；
2. 执行PLSQL代码；
3. 把数据文件中的块读到buffer cache中；
4. 返回处理结果（查询结果）给client proccess。

Server process可分为专有服务器进程和共享服务器进程。

#### Dedicated Server Processes

就是专有连接时的服务器进程。专有服务器进程有如下特点：

1. 此时服务器进程与客户端进程一一对应。
2. Client process直接与server process通信，在整个会话生命周期中，该server process只对该client process服务。
3. UGA在server process的PGA中。

#### Shared Server Processes

就是共享连接时的服务器进程。共享服务器进程有如下特点：

1. 多个client process通过网络与一个dispatcher process通信；
2. dispatcher process将client process们带来的请求放到large pool的request quene中；
3. 一个shared server process从请求队列中拿走一个请求。注意不是一个client process的所有请求，一个client process可能有多个请求，这多个请求不一定是顺序排列在请求队列当中的，可能中间还穿插着别的client process发来的请求。而每次共享服务器进程只拿走队列中的一个请求进行进行处理。
4. Shared server process将处理结果放到response quene中，dispatcher再挨个把这些处理结果提交给client process。
5. Shared server process也有自己的PGA，但是共享连接时，UGA不在PGA中，而在SGA中（也是在大池中）。

### 后台进程（Background Processes）

后台进程是在实例启动时，数据库自动创建的。每个后台进程都有自己独立的任务，但是会相互合作。

在数据库中使用如下sql进行后台进程的查询：

SQL> SELECT PNAME FROM V$PROCESS WHERE PNAME IS NOT NULL ORDER BY PNAME;

后台进程有3个分类：

1. 必须的
2. 可选的
3. slave processes

#### 必须的后台进程（Mandatory Background Processes）

必须的后台进程有：

1. PMON
2. LREG
3. SMON
4. DBW
5. LGWR
6. CKPT
7. MMON and MMNL
8. RECO

下面我们来一一介绍。

##### Process Monitor Process（PMON）

进程监控进程，作用如下：

1. 监控其他后台进程，并当server process或者dispatcher process异常挂掉的时候进行进程恢复（process recovery）
2. 清理buffer cache，并且释放client process使用过的资源。例如：释放不再被请求的锁；从活动的进程列表中移除process id；重置active transaction table的状态。

##### Listener Registration Process (LREG)

注册实例信息和dispatcher processes到监听程序，12c之前这个事情是由PMON做的。

注意，如果监听没启动，那么LREG就会隔一段时间就和监听联系一下，看看监听起没启动，启动了就把实例信息和dispatcher processes注册到监听。

注意，这里LREG联系的监听是默认的1521端口的监听，如果想要LREG联系其他监听，需要设置local\_listener的参数。

##### System Monitor Process (SMON)

系统监控进程主要负责处理各种各样系统级别的清理工作，它的主要工作如下：

1. 实例启动时进行实例恢复（如果有必要的话），在RAC环境中为挂了的节点进行实例恢复。
2. 清理不再使用的临时段，比如说建索引失败了，分配个该操作的临时空间就要由smon进行清理。
3. 数据字典管理的表空间（dictionary-managed tablespaces）的空闲的区（free extent）由smon来进行合并。

SMON自己会时常看有需要的它时情况没，同时别的进程需要它的时候也需要唤醒它来处理。

##### Database Writer Process (DBW)

DBW进程就是把buffer cache中的脏块写入到数据文件中，DBW触发的条件有：

1. server process找不到clean状态的buffer时，就要触发DBW把脏块写到磁盘给buffer cache 腾空间放新块。DBW写数据文件时是异步写。
2. DBW定期写脏块到数据文件，来推进日志文件中的checkpoint（advance checkpoint）。Checkpoint的位置取决于buffer cache中最早旧的脏块。注意，实例恢复是从redo log中的最新的检查点开始恢复的。

除了DBW0，DBW可以额外配置多个，从DBW1-9、DBWa-z、DBW36-99。但是对于单处理器的系统就没意义了。

注意DBW的写在可能的情况下会进行多块写，多快写一次能写多少块因操作系统而异。

##### Log Writer Process (LGWR)

日志写进程是顺序的将redo log buffer中的重做日志条目（redo entry）写入到在线重做日志文件中（磁盘上）。

LGWR触发条件如下：

1. 一个用户提交了一个事务；
2. 日志切换；
3. 距离上一次触发已经过去3秒了；
4. Redo log buffer三分之一满，或者已经包含了1M的缓冲数据；
5. DBW必须要把脏块写到磁盘了（日志先行的原则，导致此时必须要先写触发lgwr）

日志先行的原则就是：DBW要写了脏块到磁盘了，发现还有redo没记录到日志文件，那就必须等日志被LGWR都写到磁盘上了，然后DBW再写脏块。因为日志写（是顺序写）非常快，即使DBW被异常中断，没关系，重做日志已经把操作都记下来了，实例恢复的时候重新执行一遍就ok了。

当一个用户执行了提交，那么事务就会被分配一个scn号，然后LGWR就会很快的把commit SCN和redo entries写到磁盘上。

事实上，LGWR在事务提交之前就会把redo log entries写到日志文件上，但是只有当事务真正提交后，LGWR才会将该记录永久化。

同时，当高并发的时候，LGWR会把几个提交用一次写入到日志中，因为LGWR再写一个事务的redo时，后面提交的事务的redo就要等着这个先写完。为了避免这个问题，LGWR会一次写几个提交到重做日志文件中。

每个Redo日志组，只要不存在日志文件还没归档就全损坏了，那LGWR就会正常工作。不过如果有有某个日志文件损坏了，alert日志中会记录，LGWR的trace file也会记录

##### Checkpoint Process (CKPT)

Chekpoint是保证数据库一致性、实例恢复的重要机制。

从data structure的角度讲，它也叫做checkpoint position，就是使用redo进行实例恢复的时候开始位置的SCN。它存储在控制文件和每一个数据文件头（datafile header）中。

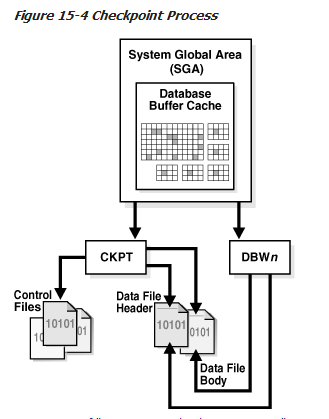
同时，checkpoint也是一种行为，就是触发脏块从buffer cache写入到磁盘中的一种行为。

使用checkpoint的目的如下：

1. 当实例异常停止的后，再次启动实例，可以减少实例恢复的时间；
2. 确保数据库正常的将脏块从buffer cache中写入数据文件中；
3. 确保数据库一致性关闭的时候（shutdown immediate），数据库把所有已提交的数据写到数据文件中。

Checkpoint的触发条件如下：

1. thread checkpoints:数据库一致性关闭、alter system checkpoint、日志切换、alter database begin backup
2. tablespace and datafile checkpoints：正常的只读或者offline一个表空间、shrink一个数据文件、alter database begin backup
3. Incremental checkpoints：ORACLE每3秒触发CKPT把checkpoint position先写到控制文件中，但不写入datafile header中。
4. 其他：当有对象被drop或者truncate。



##### Manageability Monitor Processes (MMON and MMNL)

MMON进程是为AWR做事的。MMNL是Manageability Monitor Lite process，作用是：当SGA中的ASH buffer满了的时候，把ASH buffer里的数据写到磁盘上。

##### Recoverer Process (RECO)

RECO是专门为分布式事务（distributed transactions）服务的进程。

#### 可选的后台进程（Optional Background Processes）

##### Archiver Processes (ARCn)

该进程开了归档就有。日志切换的时候会触发归档。如果有备库，ARCn还需要把归档日志传输给备库。

##### Job Queue Processes (CJQ0 and Jnnn)

当设置系统参数job\_queue\_processes（最大可同时存在的job\_queue\_process数）不为0时，就会产生一个调度进程CJQ0（job coordinator process），ORACLE Scheduler会根据需要起停CJQ0。

[oracle@host01 ~]$ ps -ef|grep cjq0|grep -v grep

oracle 11009 1 0 Jul04 ? 00:21:04 ora\_cjq0\_PRODCDB

大致的工作流程就是CJQ0会从JOB$这张表中按时间顺序选job出来执行，执行一个job的时候需要启动从属进程Jnnn来跑这个job。这就算一个job queue process。一个job queue process只能跑一个job，跑完一个job才可以被分配给下一个job。跑到没job跑的时候，就进入sleep state。

##### Flashback Data Archive Process (FBDA)

FBDA将跟踪的表的归档到Flashback Data Archives。当一个包含了对被track表的DML的事务提交后，FBDA会把该表被改动的行的前镜像（pre-image）记录到Flashback Data Archive。

#### Slave Processes

##### I/O Slave Processes

##### Parallel Execution (PX) Server Processes

# DATABASE

## 物理存储结构（Physical Storage Structures）

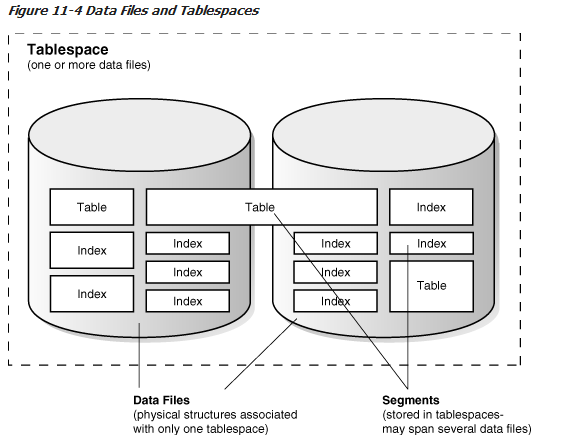
所谓物理存储结构，就是肉眼可见的，实实在在的磁盘上的文件。

### 数据文件（Data Files）

#### 表空间和数据文件

数据库建表建索引都是建在表空间上的，而表空间就是多个数据文件的逻辑体现。这里说一下表空间与数据文件的关系：

1. 一个表空间可以有多个数据文件，但是一个数据文件只能属于一个表空间；
2. 一个segment可以跨越多个数据文件，但是只能属于一个表空间；
3. 一个数据库必须得有system表空间和sysaux表空间，数据库创建的时候自动分配的第一个数据文件（datafile 1）就是给system表空间



#### 永久和临时数据文件（Permanent and Temporary Data Files）

永久数据文件就是一般的数据文件，建表空间时tablespace前是不带temporary的。

那临时表数据文件（temp files）就是临时表空间的数据文件，create temporary tablespace…。

临时表空间里的对象只存在于某一会话期间，也就是说它是会话级别的。比如建了一张临时表，那么往这张临时表里插入数据也就只有该会话自己能看到，其他会话是看不到的。

临时表空间用来进行排序、hash join、merge join、建索引等等，还有内存不足时，会把数据暂时放到临时表空间上。

然后说下俩者的区别：

1. 永久的数据对象，例如表索引这些不会存在临时表空间中；
2. 临时数据文件永远时nologging的，也就是说临时表空间上的操作不记录redo，实例恢复时也不会去读临时数据文件；
3. rman备份也不会对临时数据文件备份，意味着异机恢复时、dg的备库都需要手动创建临时表空间，或者配置OMF，重启实例数据库会跟据控制文件的记录自动创建跟源库或主库同样大小的临时表空间；
4. 不能使用alter database去创建一个tempfile；
5. 临时数据文件在linux上是稀疏文件（sparse files），也就是说你建一个32g的临时数据文件，并不会在系统是真的创建出一个32g的文件，它会随着被使用的情况增长，32g只是上限，所以建临时表空间的时候特别快；
6. 临时数据文件在v$tempfile和dba\_temp\_files中查，永久数据文件在v$datafile和dba\_data\_files中查。

#### 在线和离线数据文件（Online and Offline Data Files）

数据文件可以是online状态或者offline状态，online时才能被读取访问。

offline一个数据文件可能是需要offline backup或者数据文件坏了。

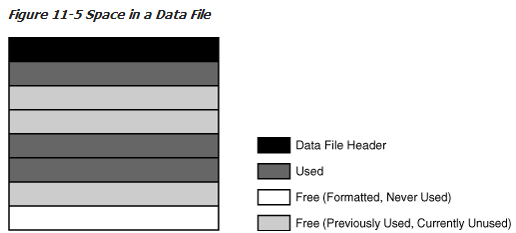
如果只offline一个表空间的某个数据文件，并不会使整个表空间offline，而如果offline一个表空间，就会把所有这个表空间的数据文件全部offline。

12c有个新技术：ALTER DATABASE MOVE DATAFILE…该技术可以把一个在线的数据文件即使是在被访问的情况下从一个物理文件转移到另一个文件。（后面研究了再补充）

该技术目前我觉得牛逼的用处有俩个：

1. 把数据文件从一个存储设备移到另一个存储设备上；
2. Move a database into Oracle ASM

#### 数据文件结构



数据库文件头包含数据文件的大小和checkpoint SCN等数据文件的metadata。文件头还包含文件号（absolute file number）和相对文件号（relative file number）。

随着对对象的更新和删除，会产生很多零碎的free空间，但是又不够大存放新的数据，那这些就是fragmented free space（碎片）。

### 控制文件（Control Files）

#### 控制文件的内容

控制文件是一个二进制文件，一个数据库只有一个控制文件，但可以有多个copies。

控制文件用来定位数据文件的位置和管理数据库的状态。

下面看下控制文件里面放些什么：

1. database name和database unique identifier (DBID)
2. 数据库创建的时间
3. 数据文件、redo log、archive log的信息
4. 表空间的信息
5. RMAN backups

#### 控制文件的用途

控制文件主要有以下用途：

1. 控制文件会跟踪数据库的结构变化，比如说添加、删除一个数据文件或者redo log，控制文件的记录都会随之变化；
2. 控制文件会记录checkpoint SCN，实例恢复的时候就是从这个点开始进行恢复，这个点之前的所有已提交的数据变更都应该记录到数据文件中，每隔3秒CKPT就会把redo里的checkpoint往控制文件里面记录一次。

#### 控制文件多路复用

其实就是控制文件的灾备，建议放在不同的磁盘上。

#### 控制文件的结构

控制文件主要包括俩种类型的记录：

1. Circular reuse records：这类记录是过期了就可以被覆盖的，比如归档日志的信息、RMAN备份的信息
2. Noncircular reuse records：这类记录是一般不变的，控制文件写满了，有新纪录进来就会增大控制文件，而不是覆盖旧的记录。比如说数据文件、redo log files、redo log thread等，除非相应的文件被删除了，否则这部分空间是不会被重用的。

#### 控制文件的读写

ORACLE数据库读写控制文件是把控制文件的块从磁盘读到PGA中，每个process会专门在PGA中分配出一块空间放控制文件的block。

### 在线重做日志（Online Redo Log）

#### redo log的用途

实例恢复最重要的就是redo log，因为redo log记录着data files的数据变化。

server process会把每一个事务同步的写入到redo log buffer，然后LGWR把redo log buffer里的内容记录到redo log中，未提交的事务也会被记录在内（主要还是为了加快写日志的速度）。

undo段的改变也会记录在redo log中，永久对象的undo数据会记录在其中，也可以设置把临时对象的undo数据也记录在redo log中，但是最好是把临时对象的undo数据放在temporary undo segment（可以节省空间又可以提高性能）。

redo log唯一的用途就是恢复用的！当然可以通过logminer进行日志挖掘查看redo log记录的内容，来诊断一些历史误操作呀等等。

#### redo log的切换

数据库实例的redo log叫做redo thread，一个实例一个thread，这个概念主要用于集群，因为单实例肯定就一个thread。那对于集群，每个实例都有自己的redo thread。因为每个实例都有自己的sga，有自己的redo log buffer，所以有自己的redo log才是合理的，实例恢复的时候用自己的redo log就好了。

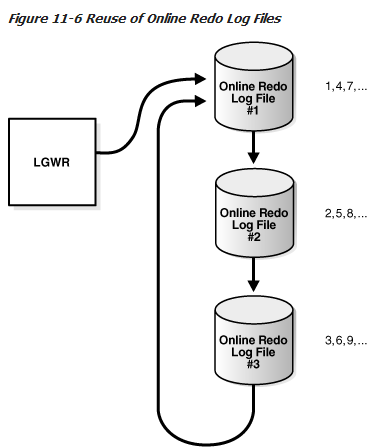
ORACLE要求至少得有俩个redo log，这样一个在清空的或者归档的时候，另一个还可以继续工作。

同一时间实例只会使用一个redo log，正在被使用的redo log状态时current，即current online redo log file。

日志切换发生情况有俩个：

1. current online redo log file写满了；
2. 另一个就是手动切换。

日志文件会循环的使用，一张图就解释的清楚：



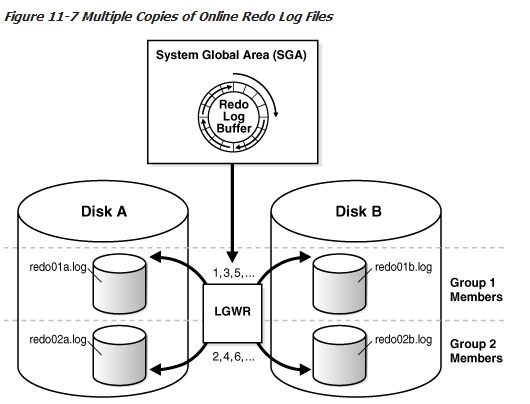
从图中我们看到的“1,4,7,…”这样的数字是数据库分配给每个日志文件的序列号（log sequence number），当发生一次日志切换时，就会产生一个新的序列号给current redo log。

因为日志写的很快，可能因为日志写满了，切换了，但是还有脏块没写入到数据文件。所以current redo log切换后会变为active online redo log file，也就是说active状态的日志里放着实例恢复可能需要的数据，那该redo log暂时不能被重新使用，当再次切换，active redo log变成了inactive redo log的时候，说明日志文件里的内容已经没用了，那这时该文件可以被重用了。

#### redo log的成员

其实跟control file的多路复用一个道理，就是个redo log的灾备。

现在就是数据库在同一时间只会使用一个日志组，对于日志组的每个成员都是同步写入的。所以同样，建议把不同的成员放在不同的磁盘。如下图：



当然，成员数量的增加会带来额外的I/O开销、磁盘开销，所以到底用几个日志组，每个日志组有几个成员，因情况而定。

#### redo log的归档

当数据开启了归档模式，那在发生日志切换后，ARCn就会把刚才那个日志组的某一个可用的成员copy一份保存起来。归档日志是offline的，redo log是online的。

归档日志可以有如下用途：

1. 恢复数据库时，增量追平（跟实例恢复一个道理）
2. 更新备库，其实也就是增量追平
3. 因为时redo log的copy，所以也可以用logminer从里面挖掘历史操作。

#### redo log的结构

redo log包含的时redo records，redo records是一组记录数据块改变的向量（vector）。比如更新employees的salary列，redo records就会包含对表段数据块的变化、undo段数据块的变化、undo段的transaction table的变化的描述。

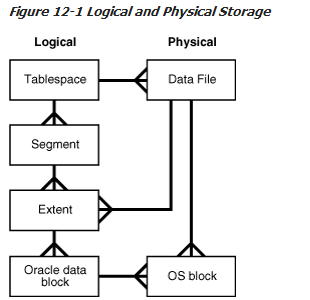
redo records主要包含如下内容：

1. 发生变化时的时间点和SCN号
2. 产生该变化的事务的事务ID（Transaction ID）
3. 事务提交时的时间点和SCN号
4. 导致该变化的操作类型（update、insert…）
5. 被修改的数据段（data segment）的名称和类型（xx表、yy索引…）

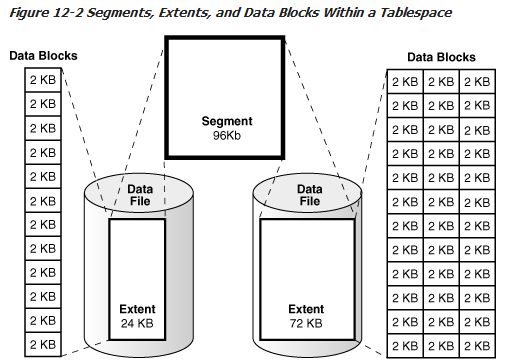
## 逻辑存储结构（Logical Storage Structures）

### 逻辑结构概要

ORACLE的逻辑结构中最小的单位是data block，一个data block可能对应物理结构的几个operating system blocks。下图很明了的展示了逻辑存储结构和物理存储结构的对应关系。



ORACLE的逻辑结构中有块、区、段、表空间，那么他们的等级关系（Logical Storage Hierarchy），还是下面一幅图说的很清楚：



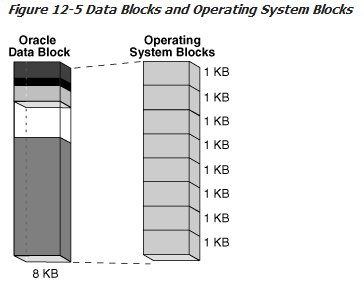
这里还是重点强调一下segment和extent。

每一个占用存储空间的数据库对象都有只属于自己的唯一的一个段。就是一个段对应一个数据库对象（LOB字段也算数据库对象）。一个表空间中有多个段，但是一个段只能对应一个表空间。一个段可以是俩个来自不同数据文件的的区构成的。但是一个extent只能属于一个数据文件。

### Data Blocks

#### Data Block概要

前面已经提过，数据库的data block和操作系统的block是一对多的关系。当数据库需要一个数据块时，操作系统会把这个操作翻译为从物理存储上取数据的一个行为。



这样把数据库的块和操作系统块区分开来的好处是：

1. 应用并不需要去判断数据在磁盘上的物理地址；
2. 数据库的块可以在多张磁盘上条带化或者镜像。

那么data block到底时多大呢？

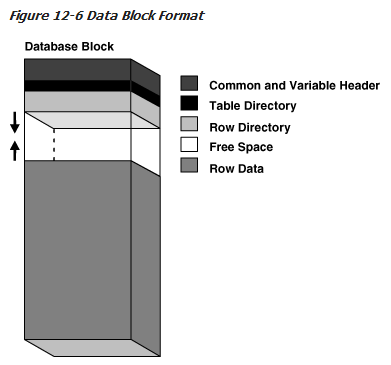
这个取决于参数db\_block\_size，这个在建数据库的时候指定，一旦指定了，想改就只能重建数据库。默认是8k，这个一旦指定，那么system表空间、sysaux表空间的块大小就指定了，数据库再创建其他的表空间，默认的块大小也是这个值。

如果一定要指定一个别的值，那么这个值必须是操作系统块大小的整数倍。

当然在建表空间的时候也可以指定表空间的块大小不是db\_block\_size，那么就如SGA部分所说，需要专门指定一块内存区域来缓存这个表空间的块。

#### Data Block Format

每一个数据块都有一个内部结构来使数据库可以方便的找到块中的数据和可用的空间。数据块的大体结构使一样的，无论其中放的使表的数据还是索引的数据或者使table cluster的数据。下面展示的使为经压缩的数据块（uncompressed data block）的结构。



下面我们一部分一部分的总结。

##### Data Block Overhead

数据库使用块的data block overhead来管理块本身。

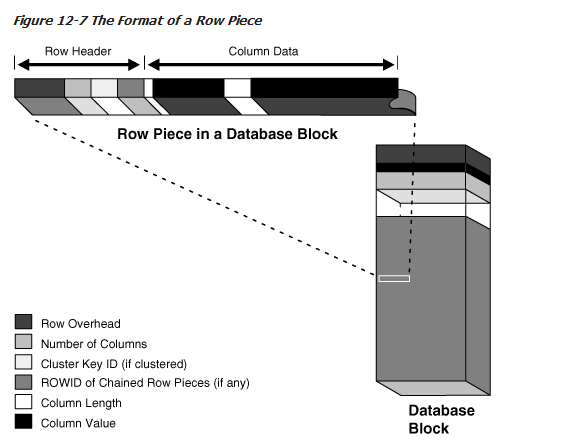
虽然block overhead部分区域于大小是固定的，但是block overhead总的大小是可变的，一般来说data block overhead大小在84-107字节。

Block overhead不放任何用户数据，它有如下几个部分：

1. Block header：存着整个块的概要信息：disk adress和segment type。如果是涉及事务的块，块头还会包含活动的和历史的事务信息。  
   当一个事务更新块时就会产生一个事务条目（transaction entry）记录事务变化，一个事务就有一个事务条目，块头装不下的话就会把事务条目放在free space。事务条目占空间的大小跟操作系统有关，一般是23字节左右。
2. Table directory：对于一个堆组织表（heap-organized table），这里放的就是所属表的metadata。  
   如果是一个table cluster，这里可能放的就是多张表的元数据，因为table cluster的一个块可以存多个表的行数据。
3. Row directory：这里放的是行信息，行地址被放在row directory vector的一个slot中。  
   rowid是对象id+相对文件号+块号+row number，这个row number就是row directory entry用来找对应行位置的一个索引，row directory entry包含指向数据行在块中位置的指针，如果行的位置在块中发生了变化，数据库只会修改row directory中指向该行指针的指向，并不会修改row number，也就是说rowid是不会变得。  
   row directory的空间不会因为删除行而被释放，只有当一个会话向该块insert了一行，才会重用之前的空间。

##### Row Format

Row data部分放的就是表的行数据或者索引的key value。Oracle存放行数据是使用的一个可变长度的record。一个row会由一个或多个row piece组成，每个row piece都有自己的row header和column data。如下图：



Row header包含的信息如下：

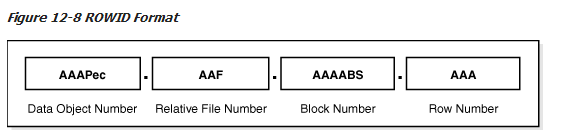
1. row piece中包含了该表的哪些列
2. 同一行的但在其他block中的row piece的信息（这个就很关键了）  
   一般来说如果一行数据能放在一个块中，那么一个row就只有一个row piece。  
   如果插入一行一个数据块放不下，或者更新一行导致原本放的下现在放不下了，就会产生新的row piece放在其他块中
3. Table cluster的cluster keys。

如果一行全都在一个块中，那row header至少有3 bytes。

Column data包含的就是行的真实数据了，从上图可以看到，column data部分有列的value和列的length，这俩部分信息是分开存储的，列的value占多大的空间，取决于列的length。一般来说列的存储顺序就是建表的时候写的列的顺序，但是比如列的字段类型是long，那该列可能会被放在column data的最后。

Rowid（extended rowid）的构成：

data\_object\_id#（table cluster的object\_id是一样的） + rfile# + block#（相对于数据文件的，而不是表空间，一个表空间的不同的数据文件的俩个row可以有完全一样的block#） + row#



Rowid一般不会改变，有俩种情况会变：

1. 启用了row movement，并对表进行了shrink、flashback等。
2. 没启用row movement，使用oracle的工具对行导出再导入。

当然如果一行被删除，然后又重新插入，那rowid也可能会变。

##### Data Block Compression

就是把一些相同的数据用符号（symbol）代替。

##### Data Block的空间管理

这里只说ASSM（automatic segment space management）管理模式下的情况。

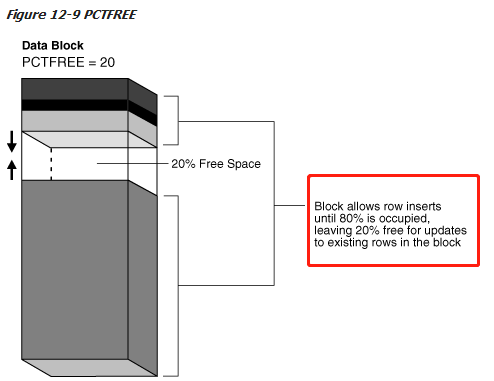
此时创建segment只需要指定pctfree，那么这个pctfree到底有什么用呢？

当执行insert操作的时候，对数据块的数据填充是由下往上的，而随着row data部分不断的上涨，block overhead也在不断的向下增大，那么row data于block overhead中间这部分free space就会越来越小。指定pctfree，就是限制这部分free space在block中的占比。

比如说按如下方式建表，指定pctfree是20：

CREATE TABLE test\_table (n NUMBER) PCTFREE 20;

那么该表在插入（insert）一些数据后，块的情况就如下图：



这个参数的意思就是，当block overhead与row data的总大小占块百分之80的空间时，就不允许再往这个块里插数据了。

那留下的这百分之20的free space到底是干嘛的？

当更新数据的时候，比如更新将行尾的null值更新为非null值，这时候row data的column data就要变长，那么多出来的这部分就是使用的这剩余的百分之20的free space，也就是说这部分free space是留给row data进行update的。

那么如果没有这部分free space，当发生上述更新时，block中没有足够的空间存放column data增加的部分，就会产生一个新的row piece放到一个新块里，这样就减慢了update的速度，如果发生大量这样的情况，那么就会导致性能问题。

所以pctfree的作用就是：

1. 防止row migration
2. 同时避免空间浪费

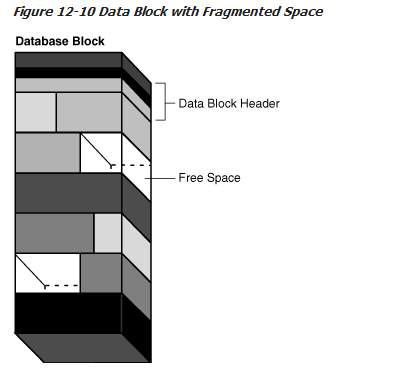
可以使free space变大的操作如下：

1. delete语句
2. update一个长值变成一个短值，或者update导致发生了row migrate

那么free space变大了，什么操作能使用这部分腾出来的空间呢？

1. 同一个事务中的有前面释放快空间的操作，接着执行的insert可以使用这部分空间。
2. 不同事务的话，执行一条insert，必须得等存在释放快空间操作的事务提交后，且没有从没用过的空间可以放这条insert进来的row data了，才会利用被释放出来的空间。

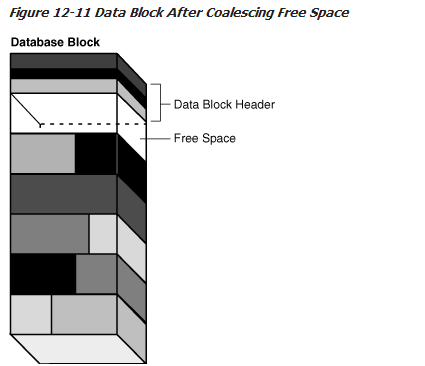
释放出来的空间不一定是连续的，那么如果释放出来这些不连续的空间，再装不下后面要insert进来的row data怎么办？那么这部分空间就只能是碎片空间（Fragmented Space）。如下图：



官文说ORACLE会俩种情况下自动合并这些fragmented space，我总结一下其实就是一种情况，就是想放一个row piece进来，总的free space是够的，但是不连续，那ORACLE就会把这些碎片合并成连续的空间来放新进来的row piece。

也就是说块里的碎片，ORACLE在需要的时候会自动合并利用，千万不要跟高水位线搞混，HWM是针对段说的。

下图是合并后的效果：



##### Chained and Migrated Rows

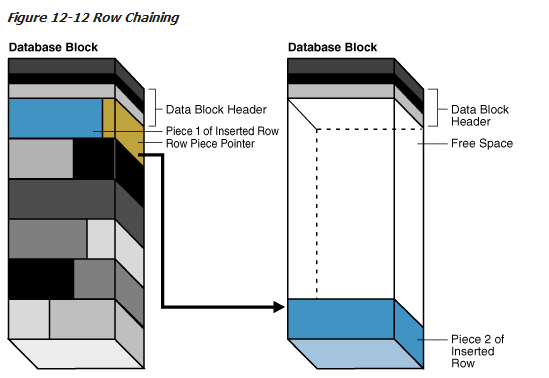
Chain和Migrate是ORACLE对一个块放不下一个row data的俩种处理方式。

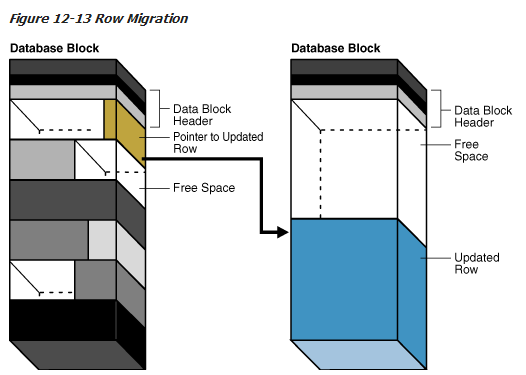
有三种可能的情况：

1. 一行数据第一次试图insert该块，但是太大了，放不下；
2. 一行数据原本是在该块中，但是一次update导致row data变长，致使该块装不下更新后的这个row data；
3. 一行数据超过了255个列

下面总结ORACLE对三种情况进行处理的方式：

1. 对于第一种和第三种情况，ORACLE会使用chain row的方式，就是把row data存放在a chain of data blocks reserved for segment。row chaining经常发生在一行数据存在long或者long raw类型的列，亦或者这一行本身就有很多列（超过255个列属于这种情况的极端情况）。
2. 对于第二种情况，ORACLE会使用migrate row，就是把一整行row移动到新的block中，但是原有的row piece还在（应该是就留了一个row header），而该row piece中会有一个指针指向存放了被迁移的行的新块。  
   这样做就既保证了migrate一个row后空间被释放了出来又保证了rowid不会发生变化。
3. 第三种情况，就是一个知识点，一个row piece最多放255列





当读取chained或者migrated rows时会产生额外的I/O，所以尽量避免发生row chaining或者row migrating。

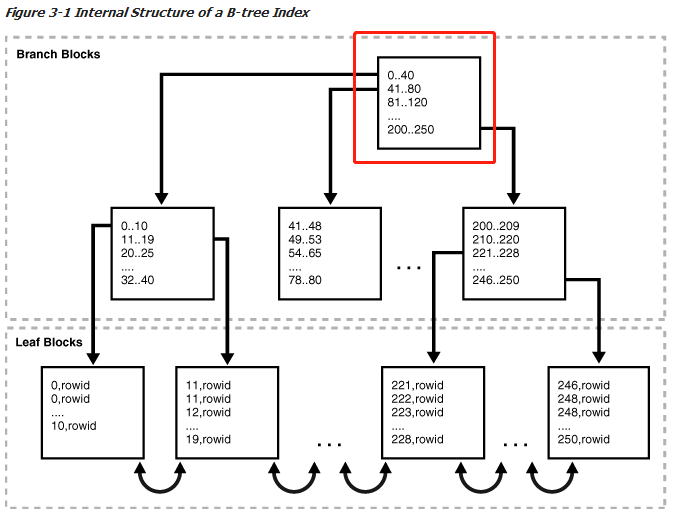
### Index Blocks

index block是一类特殊的data block，对于空间的管理不同于table block。

#### Index Blocks的分类

index block就三种类型：

1. root block
2. branch blocks
3. leaf blocks：叶子块中有序的存着key vlaue和对应的rowid



红框标注的就是root block。

#### Index条目的存储（Storage of Index Entries）

索引虽然是有序的，但是块体（block body）中的index entries（也就是真的数据部分key+rowid）依然同table block一样，是以堆（heap）的方式一行一行的由下向上往块头方向累加，而不是按照key value排序后存放。

那么索引的有序性是怎么来的呢？同table block一样，在block overhead中的row directory的row directroy entry会存有指向index entry的指针。那么同table block不一样的是，这些row directroy entries会按照key value进行排序存放。

那么在row directory中排序的好处是什么呢？如果进行一个index range scan，oracle只需要在叶子块的row directory里面找到first index key，然后顺序在row directory中找到last index key，这样就避免了读取leaf block body中所有的index entry的key value。

同时我认为就从index条目的存储角度来说，当新的index entry进入到块中后，ORACLE只需要对row dierctory entries重新排序，而不需要大量移动index entry进行重新排序。因为这个过程同前面所说的碎片空间的合并类似，而且消耗应该更大。

#### Reuse of Slots in an Index Block

一个索引块往往要比表块存储更多的行，这样可以减少的块分裂的情况，同时也可以减少I/O。

为了更好的实现这一目的，ORACLE允许index block中的空闲的空间是可以被reuse的。比如你删除了一条数据，那么这条数据对应的index entry也会被从索引块中删除，而此时，你又插入了一条数据，注意，这时ORACLE发现新进来的数据的索引条目也应该插入到该所索引块中（比如你刚删了个5，然后又插入了一个6），同时刚才空出来的space可以放的下新的index entry，那么ORACLE就会reuse this slot for new index entry。

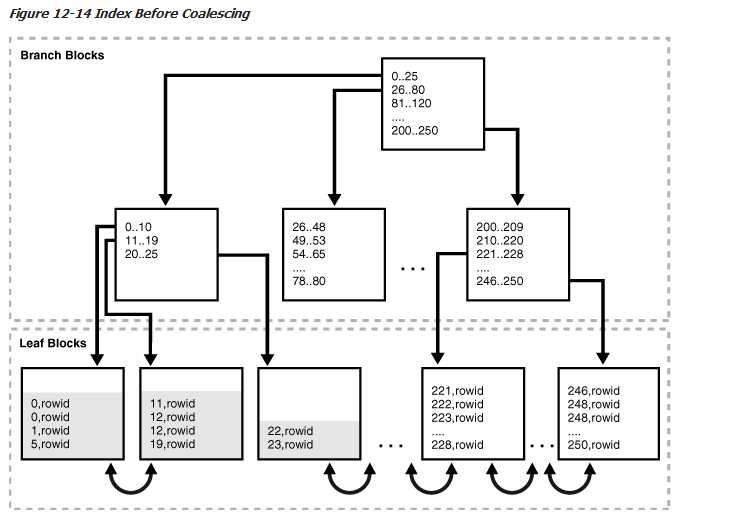
但是index block与table block不同，不会自动coalesce fragmented space。而这些碎片空间，也只有当合适（就是key差不多，且该碎片空间放的下）的index entry进来才会重用。

那么这就产生一个问题，假如一个时间列（或者说是递增列）有索引，我不断的插入的是新的时间，且定期删除历史数据，那么就会产生很多空的索引块，但并不能被重用，同时索引会越来越大。

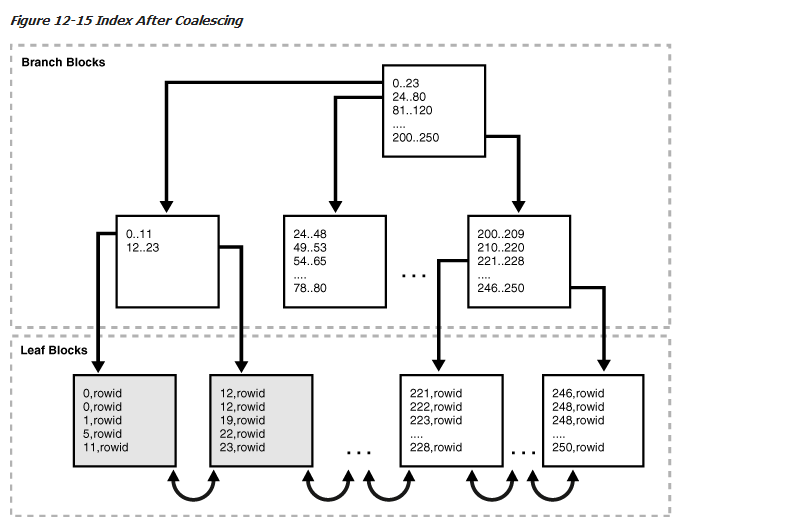
#### Coalescing an Index Block

那么如何来对空闲的空间、空闲的index block进行合并呢？我们必须手动执行alter index … rebuild 或者coalesce。下面我们看一个示例。

Coalesce前：



Coalesce后：



我们注意到，不仅块的空间被再次充分利用，而且还把第三个块释放出来，可以做别的用途，比如被用作table block呀之类的。

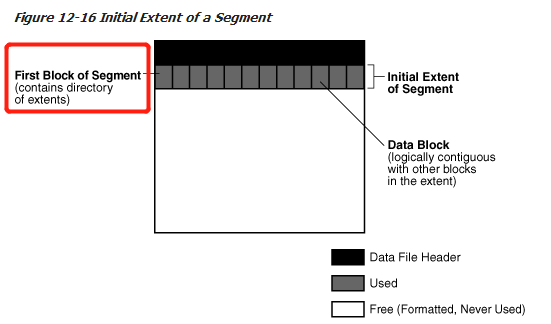
### Extents

区是一块由连续的data block组成的逻辑存储单位，但是物理上并不一定是连续的（比如做了RAID呀等等）。

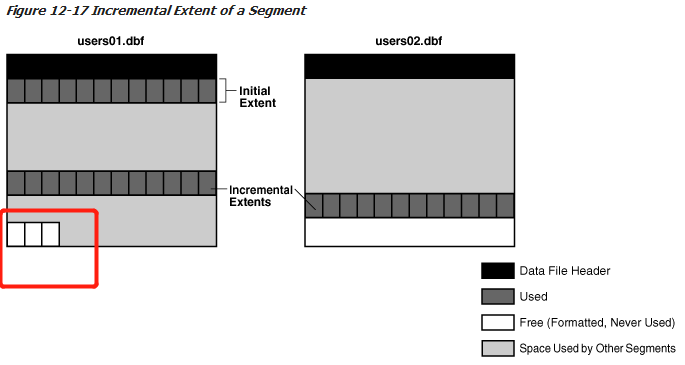
#### 区的分配（allocation）

当一个段创建的时候，数据库会分配给这个段一个区（initial extent）。

这第一个区的第一个个block（当然也就是segment的第一个block）存储了一个记录改segment中的所有的extents的dictionary。



当第一个区满了，数据库就会在本数据文件中找一组新的连续的没人用的块（free block），如果不够构成一个新的extent的话，那么就会到同一个tablespace的另一个datafile中找一组连续的free block构成一个新extent（incremental extent）



注意，一个extent只能属于一个datafile，一个segment可以由不同处于不同datafile的extent构成。

#### 区的重分配（deallocation）

已分配给一个segment的extent，即使数据被删除，仍属于这个segment，不能被别的segment所使用，除非把segment drop掉。比如delete from一张表，这张表的extents仍属于这张表。（automatic undo management mode下的undo段例外）。

我们知道full table scan是扫描的HWM一下的所有块，同时也以为着这些块还要被cache到内存中，所以数据不紧凑，很多空闲extent不仅对空间浪费，还对cache的性能、查询的性能有恶劣的影响。

那么如何手动deallocate一张表的extent呢？可以是使用如下方法free extent：

1. 使用online segment shrink；
2. 把数据移动到另一个新的segment中，之前那个segment就drop了，不要了。
3. 对于index segment，可以使用rebuild或者coallesce；
4. 可以使用truncate table或者table partition来删除表的数据或者分区的数据，这样操作之后，segment会留下MINEXTENTS参数指定的大小的区，其他的incremental extent会被释放。11gR2以后可以使用truncate的drop all storage选项来把整个segment移除。

对于locally managed tablespace，当extents被free后，数据文件会修改自己的bitmap来反映新的extent分布。

#### 区的存储参数

按优先级罗列如下：

1. Segment storage clause
2. Tablespace storage clause
3. Tablespace storage clause

对于locally managed tablespace，可以使用uniform extent size（统一区大小），默认统一区大小是1M，当然可以在建表空间的时候指定，统一区大小的意思这个表空间里分配的每一个区大小都是一样的。

这是对应于让ORACLE自动决定区大小而言的，就是ORACLE会在一个segment的头几个区使用一种区大小，后面就会使用更大的extent size。

### Segments

先说明，我们所有的总结都是基于ASSM管理的前提。Segment就是一组extents构成的一个逻辑结构。一张表是个table segment，一张表的一个索引是个index segment，LOB字段是个LOB segment。

#### User Segment

##### User Segment的简述

User segment的类型有如下几种，当然还有别的：

1. Table, table partition, or table cluster
2. LOB or LOB partition
3. Index or index partition

每一个非分区对象和每一个分区都有自己的段，比如一个索引有5个分区，那么这个索引就有5个segment构成。

##### User Segment的创建

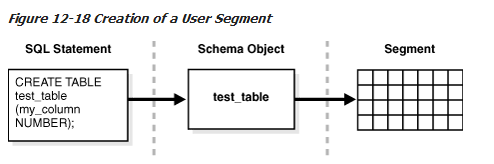
ORACLE创建segment使用的是deferred segment creation的方法，就是使用create创建对象，但是并不真的为其分配segment，只是把metadata写到数据库中。只有当对对象插入第一行数据的时候，才会真正的为其分配segment。

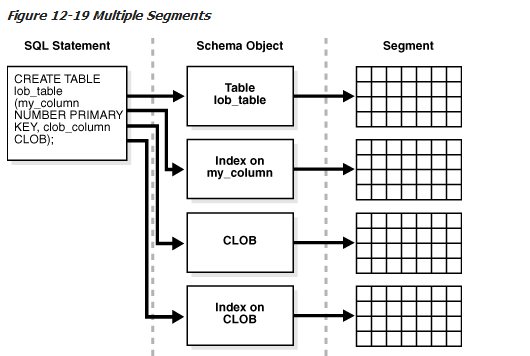
这样做的好处是显而易见的，因为一个应用可能会同时创建数百千张表，但可能很多表最终都不会使用，只是空表，那如果建表就为其分配段，就会浪费很多磁盘空间。

可以是用DBMS\_SPACE\_ADMIN包来管理空对象的段：

1. 手动实体化（materialize）那些没有被创建segment的empty table或者empty partition，也就是说给他们把段建上；
2. 把那些还有segment但已经是空表或者空的分区的segment移除掉。

下面展示一下表的创建和段的创建的关系，为了清楚的表示它们的关系，我们假设deffered segment creation是关着的。





我们重点说下第二种情况，建一个表，同时创建了主键，那么要为表创建一个segment，为索引也要创建一个segment。

但是列中还有一个LOB字段，那么就要为这个LOB字段创建俩个segment，一个segment type是LOB，另一个是LOB INDEX。

#### Temporary Segments

当执行一条sql的时候，临时表空间往往被数据库用作中介环节（intermediate stages）。常见的有sort、hash、merge。

建索引的时候也会使用临时段，先在临时段里把索引段建好，再把索引段转成永久的段。

下面我们分开总结临时段的分配

##### For Queries

ORACLE会为一个查询分配一个临时表段（如果需要的话），等查询完了就把这个临时表段删了（但是这些块就留在临时表空间了）。

ORACLE在用户默认的临时表空间中创建临时段。

locally manage的system 表空间不能作为默认的temporary storage，必须创建专门的临时表空间。正常来说我们也是这样做的。

##### For Temporary Tables and Indexes

临时表是会话级别的，也就是说为临时表分配的extents只能有该会话用户可以访问到，其他会话用户是访问不到的。

在临时表被第一次插入数据的时候，才会为临时表创建临时段。

同为普通表创建段一样，当想临时表插入数据的时候，ORACLE也会为临时表的索引，lob字段分别创建段。只是这些段用完了就删。

#### Undo Segments

有事务就会有undo data，undo data的作用如下：

1. 回滚一个active transaction；
2. 恢复一个被终止的事务
3. 提供读一致性
4. Perform some logical flashback operations

##### Temporary Undo Segments

就记住，TEMP\_UNDO\_ENABLED为true的时候，temporary undo segments是在临时表空间创建的。