深入 SecureFile

——新一代 LOB 揭秘

中国 Oracle 用户组

作者：叶熙昌(yexichang)

http://www.acoug.org

版本	发布时间
1.0	2011/1/30

目录

摘要: 从 11g 开始，Oracle 提供了一种新的 LOB 存储方式——SecureFile。本文详细阐述了 SecureFile LOB 的特点，使用方法，物理存储结构，并对 SecureFile 及 BasicFile 的架构和性能进行了比较。希望能帮助你加深对 SecureFile 的理解。

最近张乐奕（Kamus）在 gtalk 上问了俺一个问题，为啥 SecureFile 类型的 LOB 要比以前的 LOB 性能要好很多，是不是存储结构上发生了什么变化。这两天抽空看了些资料，现在整理出来算是回答他的问题了。

1 SecureFile LOB 的特点

关于 11g 以前的 LOB 类型的实现方式可以参考前一段写的一篇博文《CLOB 的物理存储结构及语言编码详解》。从 11g 开始，Oracle 提供了一种新的 LOB 存储方式叫 SecureFile，以前旧有的 LOB 存储方式就叫 BasicFile 了。Oracle 宣称，“SecureFile 不仅是新一代 LOB，它们还为 LOB 带来了更多的价值，尤其是以前只能在文件系统领域中获得的特性。SecureFile 可以进行加密以确保安全性，可以进行重复消除和压缩以提高存储效率，可以进行缓存（或不进行缓存）以加快访问（或节省缓冲池空间），可以按多个级别记录以减少崩溃后的平均恢复时间。引入 SecureFile 后，您可以在数据库中存储更多的非结构化文档，而不会导致过多的开销，也不会失去 OS 文件系统提供的任何重要功能。”

简单来说就三条：

一、提供了压缩、重复消除、加密等新功能

二、比以前的 LOB 的性能提高很多

三、易用性（无需设置 CHUNK、PCTVERSION、FREELISTS、FREELIST GROUPS、FREEPOOLS 参数）

注意：

> **注意： 压缩需要 Oracle Advanced Compression Option，加密需要 Oracle Advanced Security Option，这两个 option 都是单独购买的，没有包括在 Enterprise Edition 里面。**

2 使用 SecureFile

想要使用 SecureFile LOB 很简单，只需指定 STORE AS SECUREFILE 子句就行了（测试环境为 11gR2）：

```
CREATE TABLE tst.t11 (id number,c1 CLOB) LOB (c1) STORE AS SECUREFILE;
```

Securefile 列标明了是否为 SecureFile 类型的 LOB：

```
SELECT TABLE_NAME,SEGMENT_NAME,INDEX_NAME,SECUREFILE FROM DBA_LOBS WHERE TABLE_NAME='t11';

TABLE_NAME SEGMENT_NAME                   INDEX_NAME                    SECUREFIL
---------- ------------------------------ ------------------------------ ---------

T11        SYS_LOB0000069030C00001$$      SYS_IL0000069030C00001$$      YES
```

使用 Securefile LOB 的表也是自动生成 LOB segment 和 LOB index 的。但是此时 LOB index 只有在使用重复消除功能时才会使用，在其他情况下均不会使用。要注意，Securefile LOB 只能在 ASSM 的表空间（自动管理的表空间）里创建，不过既然从 9i 起 ASSM 表空间就是默认设置了，一般这里不会有多大问题。还要多说一句，只是要求 SecureLOB 所在的 LOB 列数据需要存放在 ASSM 表空间中，而包含 LOB 列的那个表，你还

是可以放在手动管理的表空间中。

想使用 SecureFile LOB，对数据库的参数 DB_SECUREFILE 设置也有一定的要求：

1) PERMITTED：数据库的默认参数。指定 SecureFile 时创建 SecureFile 类型的 LOB；未指定时，或显式指定 BasicFile 时，创建 BasicFile 类型的 LOB。
2) FORCE：无论是否指定 SecureFile，强制创建 SecureFile 类型的 LOB。在手动管理的表空间上创建 LOB 时，无论 STORAGE 子句是否指定 SecureFile，均报 ORA-43853 错。
3) ALWAYS：无论是否指定 SecureFile，强制创建 SecureFile 类型的 LOB。在手动管理的表空间上创建 LOB 时，若 STORAGE 子句未显式指定 LOB 类型，创建为 BasicFile 类型的 LOB；若 STORAGE 子句显式指定 SecureFile 类型，则也报 ORA-43853 错。
4) NEVER：无论是否指定 SecureFile，强制创建 BasicFile 类型的 LOB。指定 SecureFile 类型特有的功能如压缩，加密，重复消除时，报 ORA-43854 错。
5) IGNORE：无论是否指定 SecureFile，强制创建 BasicFile 类型的 LOB。忽略 SecureFile 类型特有的功能，创建 BasicFile 类型的 LOB。

3 BasicFile 和 SecureFile 的架构比较

3.1 可变 Chunk

首先介绍一下 SecureFile 中的可变 Chunk。大家都知道在 BasicFile 的 LOB 中，Chunk 的大小是一定的，最小跟 DB Block 的大小一样，最大为 32KB，这存在一些问题。比如 chunk 比 LOB 的数据小很多的情况下，访问 LOB 就会产生很多 IO，而 chunk 比 LOB 的数据大很多的情况下，又会产生对存储空间的浪费。而在 SecureFile 中，chunk 的 size 是可变的，由 Oracle 自动动态分配，最小跟 DB Block 的大小一样，最大为 64MB。这样在存储较小的 LOB 时，使用比较小的 chunk；在存储比较大的 LOB 时，会使用比较大的 chunk。注意不是说一个 LOB 就放在一个 chunk 里，而是 oracle 根据 LOB data 的数据大小会自动决定 chunk 数和 chunk 的 size，具体可以看下面“SecureFile 的物理存储结构”一节的实验结果。

3.2 LOB index

在 LOB 数据的存储方式上，两种 LOB 也有很大的区别。关于 BasicFile 的存储方式，在《CLOB 的物理存储结构及语言编码详解》一文中有详细的介绍，大概就是表中的 LOB 字段只存储 LOB locator，指向 LOB index，LOB index 再指向 LOB segment 里实际的 LOB 数据。不难看出，这里增加了一个 LOB index 的结构，那么不可避免的，LOB index 就有可能产生竞争，成为瓶颈。在 SecureFile 中，LOB index 只有在使用重复消除功能时才会使用（关于这个结论的验证方法，在 CLOB 那篇文章中有记载，这里不再赘述了）。简而言之，SecureFile 中只要不使用重复消除功能就没 LOB index 什么事，自然性能就上去了。

表 1-1

BasicFile	Col1	LOB col	---->	LOB index	---->	LOB data
SecureFile	Col1	LOB col	---------->			LOB data

3.3 空闲空间搜索

在 BasicFile 里，关于有空间的使用情况的信息是保存在 LOB index 和 LOB segment 里的。在 INSERT 或 UPDATE 操作 LOB segment 时，以下面的顺序来搜索空闲空间：

1) 在 LOG segment 的管理区搜索空闲空间，如果没有，转下一步
2) 访问 LOB index，把可以释放的空间（如已经 commit 的 transaction 使用的 UNDO）释放掉，并更新索引 entry。如果不存在这种可以释放的空间，转下一步
3) 将 HWM 升高，扩大 LOB segment，使用新分配的空间

由此可见，BasicFile 的 LOB 在搜索空闲空间时，可能会去扫描 LOB index。因此 LOB index 的竞争，或者在 LOB 数据很多的情况下，搜索 LOB index 的空闲空间这个操作本身都会造成时间上的花费。

对于空闲空间的管理，SecureFile 将其放入了 shared pool，这比 BasicFile 空闲空间管理的效率有了质的提高。 Shared Pool 里的这个内存结构叫 In-memory dispenser，它把空闲空间的信息 cache 在内存里，因此速度要比访问 LOB index 快了 N 个数量级。In-memory dispenser 负责接受前台进程对于 LOB 空间的请求，并进行 chunk 的分配。

在 In-memory dispenser 中管理的空闲空间不是全部，而只是一部分而已，它的信息由后台进程 SMCO/Wnnn 来定期的更新。SMCO/Wnnn 监视 SecureFile LOB segment 的使用情况，根据需要保证空闲空间的供应。注意 SMCO/Wnnn 也负责普通的 ASSM 表空间的空间动态分配。

1) SMCO 进程（Space Management Coordinator）。负责前瞻式（Proactive）的空间分配，它动态产生 slave 进程 Wnnn 来完成实际的工作。
2) Wnnn（SMCO Worker）每 10 分钟扫描一遍 LOB segment 的状态，根据需要把空 chunk 移动到 In-memory dispenser 中。如果这样空 chunk 还是不够，则扩大 LOB segment。

此时在 INSERT 或 UPDATE 操作 LOB segment 时，以下面的顺序来搜索空闲空间：

1) 前台进程向 In-memory dispenser 发出需要 chunk 的请求
2) In-memory dispenser 里的 chunk 信息里如果空 chunk 数量不足，或者空 chunk 的 size 不够时，在 LOG segment 的管理区搜索空闲空间，将空 chunk 的信息 cache 在 In-memory dispenser 里。如果搜索不到空闲空间，转下一步
3) 将 HWM 升高，扩大 LOB segment，使用新分配的空间

3.4 STORAGE 参数

跟 BasicFile 一样，SecureFile 同样也有 enable storage in row 和 disable storage in row 的区别，在 SecureFile 的 LOB 里默认设置同样也是 enable storage in row。LOB 控制结构 size 加上 LOB 数据 size 一共未满 4000 字节时，enable storage in row 的情况下就存储在源表的 LOB 列内，超出时就存放在 LOB segment 里；而 disable storage in row 的情况下则无论是否超过 4000 字节，LOB 数据均存放在 LOB segment 里。

《CLOB 的物理存储结构及语言编码详解》一文中提到过，enable storage in row 的情况下源表的 LOB 列最多能存放 3964 字节；而在 DB11gR1 的 SecureFile LOB 中，变成了 3740 字节；DB11gR2 时又变成了 3952 字节。均为引用的数据，具体区别尚未弄清。个人认为 Size 的变化都是因为 LOB 的控制信息发生了细微的变化。注意这里的 size 都是在未使用重复消除、加密、压缩选项的情况下得出的。

4 SecureFile 的物理存储结构

下面就是本文的重头戏了，SecureFile 的 LOB 到底在物理上是怎么存储的。在 Table 的 Segment 里的 LOB 列中，存放着两个 layer：Locator Layer（20 字节）和 Inode Layer。Locator 的内容和 BasicFile 里是一样的，也包括了控制信息和 10 字节的 LOB ID。而 Inode Layer 包含了 RCI Header 和 Inode 部分，其中 Inode 部分包含了指向 LOB segment 的指针，或者在 In-line 存储的情况下包含实际的 LOB 数据。RCI(LOB Row-Column Intersection)即表 Segment 里存储的 LOB 列的所有数据，RCI Header 里存储的是 SecureFile 的 LOB 控制信息。具体的定义如下：

2 字节的 Inode Layer 整个的 size，即表 Segment 里 LOB 列中，Locator Layer 之外的 size。如果是 in-line 存储的话也包括 LOB 数据的 size 在内。

1 字节的 flag，具体含义如下

0x01 此 LOB 是有效 LOB
0x02 此 inode 在 Index 中
0x04 此 inode 是 in-line 的
0x08 in-line 的数据是实际的 LOB 数据
0x10 此 inode 是临时 lob 的控制结构
0x40 此 LOB 是 SecureFile 的 LOB

比如我们 dump 出 block 这里是 0x48，就说明这个 LOB 是 SecureFile 的 LOB，而且是 in-line 存储的。1 字节的 SecureFile 的 LOB 的选项 flag：0x1 是启用重复消除，0x2 是启用压缩，0x4 是启用加密。下面举例说明：

上文里创建过 T11 表，T11 表没指定 storage in row 选项，因此就是默认的 enable storage in row。给 T11 表插入两条测试数据，一条是 in-line 方式存储的，一条是 out-of-line 方式存储的：

```
SELECT ID,DBMS_LOB.GETLENGTH(C1) FROM tst.t11;

        ID DBMS_LOB.GETLENGTH(C1)
---------- ----------------------

         1                    56

         2                 25583
ALTER DATABAS FLUSH BUFFER_CACHE;

SELECT DBMS_ROWID.ROWID_TO_ABSOLUTE_FNO(rowid,'TST','T11') fno,

DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID) BNO,ID FROM tst.t11;

       FNO        BNO         ID
---------- ---------- ----------

         5        132          1

         5        132          2
ALTER SYSTEM DUMP DATAFILE 5 BLOCK 132;

SELECT value FROM v$diag_info WHERE name='Default Trace File';

VALUE
```

```
--------------------------------------------------------------------------------
/u01/app/oracle/diag/rdbms/dbeuc/dbeuc/trace/dbeuc_ora_25611.trc
```

先看第一条数据：

tab 0, row 0, @0x1f03

tl: 149 fb: --H-FL-- lb: 0x0 cc: 2

col 0: [2] c1 02

col 1: [142]

```
 00 54 00 01 02 0c 80 80 00 02 00 00 00 01 00 00 00 61 3c c7 00 7a 48 90 00
 <- - - - - - A - - - - - - -> <- - - - - - B - - - - - - -> <- - C - -> <-
 74 00 00 70 01 30 d3 30 b8 30 cd 30 b9 90 4b 55 b6 30 4c … … …
 - - - D - - -> < - - - - - - - - - - - E - - - - - - - - - … … …
```

LOB

Locator:

  Length: 84(142)

  Version: 1

  Byte Length: 2

  LobID: 00.00.00.01.00.00.00.61.3c.c7

  Flags[0x02 0x0c 0x80 0x80]:

    Type: CLOB

    Storage: SecureFile

    Characterset Format: IMPLICIT

    Partitioned Table: No

    Options: VaringWidthReadWrite

  SecureFile Header:

    Length: 122

    Old Flag: 0x48 [DataInRow SecureFile]

    Flag 0: 0x90 [INODE Valid]

    Layers:

      Lengths Array: INODE:116

      INODE:

        00 00 70 01 30 d3 30 b8 30 cd 30 b9 90 4b 55 b6 30 4c 30 88

        … … …

上面 dump 出的十六进制信息含义如下：

A：00 54 00 01 02 0c 80 80 00 02 Lob Locator Header

我认为这里的定义应该跟 BasicFile 没有发生变化：2 字节的 LOB locator 长度 （除这两个长度字节外）+ 2 字节的 LOB locator structure 版本 + 4 字节的 FLAG + 2 字节的字符集里字符的长度

B：00 00 00 01 00 00 00 61 3c c7 LOB ID

C：00 7a 48 90 RCI Header

0x007a（=122 字节）：这个 filed 后的 Inode 的 size（即 D 部分 + E 部分的 size）

0x48：这是 in-line 的 SECUREFILE LOB

0x90：未启用重复消除，压缩，加密

D：00 74 00 00 70 01 Inode 管理信息

0x0074（=116 字节）：这个 field 后面（即后面的 00 00 70 01 四个字节 + E 部分）Inode 数据的 size。116

减 4 字节为 112 字节，这跟上面得到的 LOB 的 length 为 56 是能匹配上的。

0x0000：in-line 存储 LOB data。第二位的 0 表示后面 LOB Data 的 size 是用 1 字节表示

0x70（=112 字节）：LOB Data 的 size

0x01：LOB Data 的 version

E：这里开始是真正的 LOB Data

第二条：

```
tab 0, row 1, @0x1893
tl: 51 fb: --H-FL-- lb: 0x0  cc: 2
col  0: [ 2]  c1 03
col  1: [44]
 00 54 00 01 02 0c 80 80 00 02 00 00 00 01 00 00 00 61 3d 2e 00 18 40 90 00
 <- - - - - - A - - - - - - - -> <- - - - - - B - - - - - - -> <- - C - -> <-
 12 21 00 c7 de 0a 01 01 01 40 33 a2 01 01 01 40 33 14 06
 - - - - D - - - - -> <- - - - - - - - - E - - - - - - - ->
LOB
Locator:
  Length:          84(44)
  Version:         1
  Byte Length:     2
  LobID: 00.00.00.01.00.00.00.61.3d.2e
  Flags[ 0x02 0x0c 0x80 0x80 ]:
    Type: CLOB
    Storage: SecureFile
    Characterset Format: IMPLICIT
    Partitioned Table: No
    Options: VaringWidthReadWrite
  SecureFile Header:
    Length:    24
    Old Flag: 0x40 [ SecureFile ]
    Flag 0:    0x90 [ INODE Valid ]
    Layers:
      Lengths Array: INODE:18
      INODE:
        21 00 c7 de 0a 01 01 01 40 33 a2 01 01 01 40 33 14 06
```

上面 dump 出的十六进制信息含义如下：

A：Lob Locator Header

B：LOB ID

C：00 18 40 90 RCI Header

0x0018（=24 字节）：这个 filed 后的 Inode 的 size（即 D 部分 + E 部分的 size）

0x40：这是 out-line 的 SECUREFILE LOB

0x90：未启用重复消除，压缩，加密

D：00 12 21 00 c7 de 0a 01 Inode 管理信息

0x0012（=18 字节）：这个 field 后面（即后面的四个字节 + E 部分）Inode 数据的 size。

0x2100：第一位的 2 表示后面的 E 部分是 chunk 的 RDBA + size，第二位的 1 表示后面 LOB Data 的 size 是用 2 字节表示

0xc7de（=51166 字节）：LOB Data 的 size

0x0a： LOB Data 的 version

0x01：代表后面有 2 个 chunk，如果是 2 就是 3 个 chunk，以此类推。

E：01 01 40 33 a2 01 01 01 40 33 14 06

01 01 40 33 a2 01：RDBA 以 0x014033a2 开头的 1 个 block

01 01 40 33 14 06：RDBA 以 0x01403314 开头的 6 个 block

我们看一下第一个 chunk：

```
SELECT DBMS_UTILITY.DATA_BLOCK_ADDRESS_BLOCK(TO_NUMBER('14033a2','xxxxxxxx'))
BNO FROM DUAL;
       BNO
----------
     13218
```

得到：

bdba [0x014033a2]

kdlich [0x2b4878d45a4c 56]

flg0 0x28 [ver=0 typ=data lock=y]

flg1 0x00

scn 0x0000.0045ff19

lid 0000000100000613d2e - ->这是 LOB ID

rid 0x00000000.0000

kdlidh [0x2b4878d45a64 24]

flg2 0x00 [ver=0 lid=short-rowid hash=n cmap=n pfill=n]

flg3 0x00

pskip 0

sskip 0

hash 00000000000000000000000000000000000000000

hwm **8060** - ->这个 **block** 里存放了 **8060** 字节的数据

spr 0

data [0x2b4878d45a80 52 8060]

30 d3 30 b8 30 cd 30 b9 90 4b 55 b6 30 4c 30 88 30 8a 89 07 96 d1 30 55 30 92

我们再看一下最后一个 block：

```
SELECT DBMS_UTILITY.DATA_BLOCK_ADDRESS_BLOCK(TO_NUMBER('1403314','xxxxxxxx'))
BNO FROM DUAL;
       BNO
----------
     13076
```

现在需要 dump 从 13076 开始的第 6 个即 13081 号 block，得到：

… … …

hwm **2806**

… … …

一共 8060×6+2806=51166 字节的 LOB data，这与前面的结果是能相互印证的。

下面我们再来看 disable storage in row 的情况下 SecureFile 是如何存储的。

在这种情况下指向 LOB data 的指针可能有两种存储方式：

1) LOB data 的 size 不算很大的情况下，在 Table Segment 里的 LOB 列中以 chunk 的初始 RDBA + size 的方式存储，一个 chunk 信息接着一个 chunk 信息；
2) LOB data 的 size 很大的情况下，在 Table Segment 里的 LOB 列中存储 LHB（Lob Header Block）的信息，在 LHB 中存放所有 chunk 及 size 的列表。

第一种跟上面的第二条数据存储方式差不多，就不再介绍了，下面我们看第二种情况：

```
CREATE TABLE TST.T12 (ID NUMBER,C2 CLOB) LOB (C2) STORE AS SECUREFILE(DISABLE STORAGE IN ROW);
```

插入一条很大的 LOB 数据：

```
SELECT ID,DBMS_LOB.GETLENGTH(C2) FROM TST.T12;

        ID DBMS_LOB.GETLENGTH(C2)
---------- ----------------------

         1              49498672

ALTER DATABAS FLUSH BUFFER_CACHE;

SELECT DBMS_ROWID.ROWID_TO_ABSOLUTE_FNO(ROWID,'TST','T12') FNO,

DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID) BNO,ID FROM TST.T12;

       FNO        BNO         ID
---------- ---------- ----------

         5        173          1

ALTER SYSTEM DUMP DATAFILE 5 BLOCK 173;
```

现在来看一下 Table Segment 里存放 LOB 列的地方是什么信息：

```
tab 0, row 0, @0x1f1b
tl: 44 fb: --H-FL-- lb: 0x1  cc: 2
col  0: [ 2]  c1 02
col  1: [37]
 00 54 00 01 02 0c 80 80 00 02 00 00 00 01 00 00 00 61 3e 58 00 11 40 90 00
 < - - - - - - A - - - - - - > <- - - - - - B - - - - - - -> <- - C - -> <-
 0b 43 00 05 e6 94 60 41 01 40 3b 81
 - - - - D - - - - - - > <- - E - ->
LOB
Locator:
  Length:          84(37)
  Version:         1
  Byte Length:     2
```

LobID: 00.00.00.01.00.00.00.61.3e.58
Flags[0x02 0x0c 0x80 0x80]:
Type: CLOB
Storage: SecureFile
Characterset Format: IMPLICIT
Partitioned Table: No
Options: VaringWidthReadWrite
SecureFile Header:
Length: 17
Old Flag: 0x40 [SecureFile]
Flag 0: 0x90 [INODE Valid]
Layers:
Lengths Array: INODE:11
INODE:
43 00 05 e6 94 60 41 01 40 3b 81

上面 dump 出的十六进制信息含义如下：

A：Lob Locator Header。

B：LOB ID

C：00 11 40 90 RCI Header

0x0018（=17 字节）：这个 filed 后的 Inode 的 size（即 D 部分 + E 部分的 size）

0x40：这是 out-line 的 SECUREFILE LOB

0x90：未启用重复消除，压缩，加密

D：00 0b 43 00 05 e6 94 60 41 Inode 管理信息

0x000b（=11 字节）：这个 field 后面（即后面的四个字节 + E 部分）Inode 数据的 size。

0x4300：第一位的 4 表示后面的 E 部分是 LHB 的 RDBA，第二位的 3 表示 LOB data 的 size 是 4 字节。

0x05e69460（=98997344 字节）：LOB Data 的 size

0x41： LOB Data 的 version

E：01 40 3b 81 这是 LHB 的 RDBA

我们把 LHB 给 dump 出来看看：

```
SELECT DBMS_UTILITY.data_block_address_block(TO_NUMBER(LTRIM('01403b81'),'xxxxxxxx'))
BNO FROM dual;
       BNO
----------
     15233
ALTER SYSTEM DUMP DATAFILE 5 BLOCK 15233;
```

可以看到下面有 chunk 列表，以 block 数 + RDBA 的形式存储：

bdba [0x01403b81]
kdlich [0x2b3f1e77844c 56]
flg0 0x18 [ver=0 typ=lhb lock=y]
flg1 0x00
scn 0x0000.00462283

lid 0x0000001000000613e58
rid 0x00000000.0000
kdlihh [0x2b3f1e778464 24]
flg2 0x00 [ver=0 lid=short-rowid hash=n it=n bt=n xfm=n ovr=n aux=n]
flg3 0x80 [vll=y]
flg4 0x00
flg5 0x00
hash 0x00000000000000000000000000000000000000000000
llen 0.98997344
ver 0.65
#ext 100
asiz 100
hwm 100
ovr 0x00000000.0
dba0 0x00000000
dba1 0x00000000
dba2 0x00000000
dba3 0x00000000
auxp 0x00000000
ldba 0x01406ab5
nblk 12283 **- ->这个 LOB 共占用了多少个 block**
[0] 0x00 0x00 73 0x014039b7 **- ->从 RDBA 0x014039b7 开始的 73 个 block**
[1] 0x00 0x00 7 0x014000c9 **- ->从 RDBA 0x014000c9 开始的 7 个 block**
[2] 0x00 0x00 5 0x014000bb
… … … …
[97] 0x00 0x00 795 0x01406665
[98] 0x00 0x00 224 0x01406585
[99] 0x00 0x00 85 0x01406a61

接下来对于 LOB data 所在的 block 的 dump 这里就不做了，方法跟上述的类似。

5 两种 LOB 性能测试比较

上面说了这么多新的 SecureFile 的 LOB 怎么怎么好，怎么怎么牛，大家一定有疑问了，是不是我在这忽悠大家呢。下面就给大家看些干货，真实的测试数据。

首先介绍下 LOB 参数：

LOGGING：在 CREATE/UPDATE/INSERT LOB 数据时会写入 REDO LOG 文件。但 NOLOGGING 会 Internally 转换成 FILESYSTEM_LIKE_LOGGING，而 FILESYSTEM_LIKE_LOGGING 会确保数据库 CRASH 完整恢复

NOLOGGING：在 CREATE/UPDATE/INSERT LOB 数据时不写入 REDO LOG 文件。

FILESYSTEM_LIKE_LOGGING：数据库只记录 LOB 的 METADATA 到 REDO LOG

NOCACHE：LOB 数据不 CACHE 在 SGA

CACHE：LOB 数据 CACHE 在 SGA

测试环境：

虚拟机 OEL5.5 64bit + DB11.2.0.1，测试数据是一个 110MB 的文本文档 lob.txt：

```
[oracle@cdcjp11vm1 ~]$ du -m lob.txt

110     lob.txt

[oracle@cdcjp11vm1 ~]$ cat lob.txt|wc -l

1916928

[oracle@cdcjp11vm1 ~]$ head -n1 lob.txt

ビジネス運営がより複雑さを増すなかで、IT に対する変化の要求は高まりを見せ、関連するリスクの軽減もあわせて求めら
```

创建了一个存储过程 insert_clob（代码在附录中），作用是插入若干条 CLOB 数据即 lob.txt 的内容，本次测试每次是插 20 条数据，共 2.2GB，记录所花的时间。

表 1-2

存储方式	DML 类型	SecureFile	MB/s	BasicFile	MB/s	性能比%
CACHE + LOGGING	INSERT	54.152 s	40.626	243.726 s	9.027	450.05%
CACHE + NOLOGGING	INSERT	59.398 s	37.038	NOT support	-	-
NOCACHE + LOGGING	INSERT	43.799 s	50.229	289.213 s	7.607	660.23%
NOCACHE + NOLOGGING	INSERT	48.512 s	45.349	293.454 s	7.497	604.90%

结论已经很明显，新 SecureFile 格式的 LOB 性能相比较以前 BasicFile 有了巨大的提升，而且在最典型 LOB 的选项组合 NOCACHE + LOGGING 的情况下，性能提升的比例最大。

两种 LOB 的性能数据也可以参考这篇博文：http://blog.sina.com.cn/s/blog_6058d7c10100nx26.html
他的测试结果如下，有的测试结果跟我的结果相比有一定的出入，可能是环境的问题，也可能是数据的问题（我是 110MB 的文本文件，他是 5MB 的文本文件），也可能是程序的问题（他用的 java，我用的是 PL/SQL）。另外他这篇文章里有关于 SELECT（即 READ）的性能数据，在 NOCACHE + LOGGING 的情况下，性能提升约三倍。

表 1-3

存储方式	DML 类型	SecureFile MB/s	BasicFile MB/s	性能比%
CACHE + LOGGING	INSERT	9.17	8.64	106.13%
CACHE + LOGGING	SELECT	39.52	4.42	894.12%
CACHE + NOLOGGING	INSERT	31.56	-	-
CACHE + NOLOGGING	SELECT	35.31	-	-
NOCACHE + LOGGING	INSERT	36.63	2.32	1578.88%
NOCACHE + LOGGING	SELECT	50.28	16.28	308.85%
NOCACHE + NOLOGGING	INSERT	9.38	2.51	373.71%
NOCACHE + NOLOGGING	SELECT	5.54	11.36	48.76%

6 总结

做个总结吧，我认为 SecureFile 的 LOB 之所以比 BasicFile 的 LOB 性能有提升，就是因为可变 chunk、LOB

index 不再使用、空闲空间搜索放到了 shared pool 里这三大原因共同决定的，尤其是后两者，比起以前的 BasicFile LOB，架构设计上有了飞跃。我们也能看出虽然 Oracle 数据库的发展不像以前那么革命性了，但是在很多方面，新版本的 Oracle 数据库还是取得了巨大的进步。

7 附录

测试表（只写了一种，其他的选项组合类似）：

```
create table tst.LOBTAB(ARTICLE_ID NUMBER PRIMARY KEY,ARTICLE_NAME VARCHAR2(50),
ARTICLE_DATA CLOB) tablespace data lob (ARTICLE_DATA)
store as SECUREFILE (tablespace DATA cache) LOGGING;
```

插入 CLOB 数据的存储过程 insert_clob：

```
create or replace procedure tst.insert_clob (fromid in number,endid in number)
AS
i NUMBER;
V_LOB CLOB;
V_FILE BFILE := BFILENAME('HOME_DIR', 'lob.txt');
V_SOURCE NUMBER := 1;
V_DEST NUMBER := 1;
V_LANG NUMBER := 0;
V_WARN NUMBER;
BEGIN

for i in fromid..endid loop
V_SOURCE := 1;
V_DEST := 1;
INSERT INTO tst.LOBTAB VALUES (i, 'ABC'||to_char(i), 'TEST');
UPDATE tst.LOBTAB SET ARTICLE_DATA = EMPTY_CLOB where ARTICLE_ID=i RETURN ARTICLE_DATA INTO V_LOB;
DBMS_LOB.FILEOPEN(V_FILE);
DBMS_LOB.OPEN(V_LOB, DBMS_LOB.LOB_READWRITE);

DBMS_LOB.LOADCLOBFROMFILE(
V_LOB,
V_FILE,
DBMS_LOB.GETLENGTH(V_FILE),
```

```
V_DEST,
V_SOURCE,
0,
V_LANG,
V_WARN);

DBMS_LOB.CLOSE(V_LOB);
DBMS_LOB.FILECLOSEALL;
COMMIT;
end loop;
END;
/
```

计算 CLOB 的 INSERT 操作的时间差是使用以下的 PL/SQL：

```
declare
a VARCHAR2(50);
b VARCHAR2(50);
begin
select to_char(systimestamp,'HH24:MI:SS.FF3') into a from dual;
TST.insert_clob(1,20);
select to_char(systimestamp,'HH24:MI:SS.FF3') into b from dual;
dbms_output.put_line(a);
dbms_output.put_line(b);
end;
/
```

8 作者介绍

叶熙昌，网名：yexichang

Oracle 10g OCM
TechTarget 特邀技术专家

现供职于 Oracle 软件研发中心，从事 RAC/New Feature/Goldengate/Grid Control/Migration/Upgrade 等 Database 相关的质量保证和技术支持等工作。

之前曾在日本东京担任过一个小银行的 RAC DBA。对于 RAC 数据库的管理、高可用性特性 DG/Streams/Goldengate 有一定的心得，关注 Oracle Database 的各种新特性。业余爱好台球、羽毛球、三国杀，欢迎同好切磋。

个人站点：http://www.orafan.net
Gmail&gtalk：xichangye@gmail.com
MSN：yexichang@hotmail.com