Oracle IO 问题解析

作者：fuyuncat
来源：WWW.HelloDBA.COM

作者简介

黄玮，男，99 年开始从事 DBA 工作，有多年的水利、军工、电信及航运行业大型数据库 Oracle 开发、设计和维护经验。

曾供职于南方某著名电信设备制造商——H 公司。期间，作为 DB 组长，负责设计、开发和维护彩铃业务的数据库系统。目前，H 公司的彩铃系统是世界上终端用户最多的彩铃系统。最终用户数过亿。

目前供职于某世界著名物流公司，负责公司的电子物流系统的数据库开发、维护工作。

msn: fuyuncat@hotmail.com
Email：fuyuncat@gmail.com

数据库的作用就是实现对数据的管理和查询。任何一个数据库系统，必然存在对数据的大量读或者写或者两中操作都大量存在。IO 问题也往往是导致数据库性能问题的重要原因。在这篇文章中，主要帮助大家在理解 Oracle 的读写操作机制的基础上，灵活解决遇到的各种常见的 IO 问题。

1 Oracle 中 IO 的产生

IO 当然包括了读、写两部分，先介绍 Oracle 中写操作的产生。

1.1 写

介绍写操作之前，先简单的看下 Oracle 的物理结构：oracle 的物理文件包括以下三种文件：控制文件（Control Files）、重做日志文件（Redo Log Files）、数据文件（datafiles）。而数据文件中，根据功能的不同，还可以分为系统数据文件、临时空间文件、回滚段文件和用户数据文件。另外，如果数据库的 Archive Log 模式被激活，还存在归档日志文件。Oracle 的 IO 产生，就是对这些文件的数据读、写操作。下面再详细看下几种主要写操作的产生及其过程。

1.1.1 控制文件

控制文件中记录了整个数据库的物理结构信息，如数据库名字、数据文件及日志文件名字和位置、事件戳信息等等。任何数据库的结构变化（如果创建新的数据文件）都会引起 Oracle 修改控制文件。同时控制文件还记录系统和各个数据文件的 SCN（System

Change Number，关于 SCN 可以参见文章《Oracle SCN 机制详解》）信息，以用于数据恢复，因此数据文件上的 SCN 变化后，Oracle 也会相应修改控制文件上的 SCN 信息。

1.1.2 用户数据修改

由于内存的读写效率比磁盘的读写效率高万倍，因此，为了降低 IO wait，oracle 会将数据 cache 在内存（Buffer Cache，对 Buffer Cache 的详细介绍可以参见《Oracle 内存全面分析》）中，对数据的读写尽量在内存中完成。当 Buffer Cache 中的数据缓存块被修改过了，它就被标记为“脏”数据。根据 LRU（Least Recently Used）算法，如果一个数据块最近很少被使用，它就称为“冷”数据块。进程 DBWn（系统中可以存在多个 DBW 进程，n 为序号）负责将“冷”的“脏”数据写入数据文件中去。DBWn 进程会在以下两种情况下将“脏”数据写入磁盘中去：

- 当服务进程扫描一定数量（阀值）的 Buffer Cache 后还没有找到干净、可重用的缓存块后，它会通知 DBWn 进程将“脏”数据写入文件中去，以释放出空闲缓存；
- 当发生检查点（Checkpoint）时。

1.1.3 Redo Log

在非直接写（Direct Write）的情况下，事务中的写操作都会产生 Redo Log，作为数据块异常关闭时的恢复记录。同样，和写用户数据类似，Redo Log 也不会被直接写入 Redo Log 文件，而是先写入 Log Buffer 中。

Log Buffer 是一个可以循环重用的缓存区。LGWR 进程负责将 Log Buffer 中的记录写入 Redo Log File 中去。一旦 Log Buffer 中的条目被写入了 Redo Log 文件中，就可以被重用了。

为了保证事务尽快获得 Log Buffer，LGWR 进程一般会尽快将 Log Buffer 中的数据写入 Redo Log 文件中去。在以下几种情况下，LGWR 回将一个连续的 Log Buffer 写入 Redo Log 文件中去：

- 当一个事务提交（COMMIT）时；
- 每 3 秒钟写一次 Log Buffer；
- 当 Log Buffer 到达 1/3 满时；
- 当 DBWn 进程将“脏”数据写入磁盘时；

1.1.4 Archive Log

当据库的 Archive Log 模式被激活后，所有 Redo Log 数据都会被写入 Archive Log 文件中以便日后进行恢复。当发生日志组切换时，ARCn（Archive 进程，可以存在多个）进程就会 Redo Log 文件拷贝到指定存储目录中去，成为 Archive Log 文件。

1.1.5 临时表空间

当 Oracle 在执行一些 SQL 时，会需要一些临时空间来存储执行语句时产生的中间数据。这些临时空间由 Oracle 从指定的临时表空间中分配给进程。主要有三种情况会占用临时空间：临时表/索引操作、排序和临时 LOB 操作。

- 临时表/索引

在会话中，当第一次对临时表进行 INSERT（包括 CTAS）时，Oracle 会从临时表空间中为临时表及其索引分配临时空间一存储数据。

- 排序

任何会使用到排序的操作，包括 JOIN、创建（重建）INDEX、ORDER BY、聚合计算（GROUP BY）以及统计数据收集，都可能使用到临时表空间。

排序操作首先会选择在内存中的 Sort Area 进行（Sort In Memory），一旦 Sort Area 不足，则会使用临时空间进行排序操作（Sort In Disk）。看以下例子：

```
SQL> alter session set sort_area_size = 10000000;

Session altered.

SQL> select owner, object_name from t_test1
  2  order by object_id;

47582 rows selected.

Execution Plan
----------------------------------------------------------
Plan hash value: 1312425564

--------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         | 47582 |  1486K|   155   (4)| 00:00:02 |
|   1 |  SORT ORDER BY     |         | 47582 |  1486K|   155   (4)| 00:00:02 |
|   2 |   TABLE ACCESS FULL| T_TEST1 | 47582 |  1486K|   150   (1)| 00:00:02 |
--------------------------------------------------------------------------------


Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
        658  consistent gets
          0  physical reads
          0  redo size
    1566184  bytes sent via SQL*Net to client
      35277  bytes received via SQL*Net from client
       3174  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
      47582  rows processed

SQL> alter session set sort_area_size = 10000;

Session altered.

SQL> select owner, object_name from t_test1
  2  order by object_id;

47582 rows selected.

Execution Plan
----------------------------------------------------------
Plan hash value: 1312425564

--------------------------------------------------------------------------------
```

```
| Id  | Operation          | Name    | Rows  | Bytes |TempSpc| Cost (%CPU)| Time|
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         | 47582 |  1486K|       |  1251   (1)| 00:0
0:16 |
|   1 |  SORT ORDER BY     |         | 47582 |  1486K|  4136K|  1251   (1)| 00:0
0:16 |
|   2 |   TABLE ACCESS FULL| T_TEST1 | 47582 |  1486K|       |   150   (1)| 00:0
0:02 |

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


Statistics
----------------------------------------------------------
          6  recursive calls
         20  db block gets
        658  consistent gets
        629  physical reads
          0  redo size
    1566184  bytes sent via SQL*Net to client
      35277  bytes received via SQL*Net from client
       3174  SQL*Net roundtrips to/from client
          0  sorts (memory)
          1  sorts (disk)
      47582  rows processed
```

- 临时 LOB 对象

LOB 对象包括 BLOB、CLOB、NCLOB、和 BFILE。在 PLSQL 程序块中，如果定义了 LOB 变量，则这些 LOB 变量就是临时 LOB 对象。临时 LOB 对象被创建在临时表空间上，直到 LOB 数据被释放，或者会话结束。

1.1.6 回滚段

我们知道，一个事务在未被提交前，其做的任何修改都是可以被回滚（Rollback）的。这些回滚数据就被放到回滚段（Rollback Segment）上。此外，一致性读（Read Consistency）、数据库恢复（Recover）都会用到回滚段。

任何数据块的修改都会被记录在回滚段中，甚至 Redo Log 也会产生回滚记录。当任何一个非只读（只有查询）的事务开始时，oracle 会自动为其指定下一个可用的回滚段。事务中任何数据变化都被写入回滚段中。如果事务回滚，oracle 根据回滚段中的回滚记录将 buffer cache 中的“脏”数据恢复，释放回滚段空间。当事务被提交，由于要保证一致性读，oracle 并不会立即释放回滚段中的数据，而是会保留一段时间。

1.1.7 Direct-Path Insert

这里，我们还要介绍一种特殊的写操作——Direct-Path Insert（直接路径插入）。Direct-Path Insert 通过直接在表中已存在的数据后面添加数据，直接将数据写入数据文件中，而忽略掉了 Buffer Cache。

我们前面提到，为了能在意外时恢复数据，每一个数据修改都会被记录到 Redo Log 中。然而，由于 Redo Log 需要写入到物理文件中去，是一个比较消耗性能的操作。为了提高性能，我们在批量写入数据时就可以通过 Direct-Path Insert 的指定 NOLOGING 的方式来避免写 Redo Log。

有多种方法可以指定 Direct-Path Insert：CTAS（CREATE TABLE AS SELECT）；SQL*Loader 指定 Direct 参数；在语句中指定 APPEND 提示。

1.2 读

1.2.1 物理读

产生物理读主要有以下几种情况：

- 第一次读取

当数据块第一次被读取到，Oracle 会先将其从磁盘上读入 Buffer Cache 中，并将他们放在 LRU（Last Recently Used）链表的 MRU（Most Recently Used）端。再次访问数据块时就可以直接从 Buffer Cache 中读取、修改了。看以下例子：

```
SQL> select owner, index_name from t_test3;

2856 rows selected.

Execution Plan
----------------------------------------------------------
Plan hash value: 2878488296

--------------------------------------------------------------------------------
| Id  | Operation         | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |         |  2856 | 68544 |    22   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| T_TEST3 |  2856 | 68544 |    22   (0)| 00:00:01 |
--------------------------------------------------------------------------------


Statistics
----------------------------------------------------------
        407  recursive calls
         32  db block gets
        344  consistent gets
         89  physical reads
          0  redo size
     103888  bytes sent via SQL*Net to client
       2475  bytes received via SQL*Net from client
        192  SQL*Net roundtrips to/from client
          9  sorts (memory)
          0  sorts (disk)
       2856  rows processed

SQL> select owner, index_name from t_test3;

2856 rows selected.

Elapsed: 00:00:00.03

Execution Plan
----------------------------------------------------------
Plan hash value: 2878488296

--------------------------------------------------------------------------------
| Id  | Operation         | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |         |  2856 | 68544 |    22   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| T_TEST3 |  2856 | 68544 |    22   (0)| 00:00:01 |
--------------------------------------------------------------------------------


Statistics
----------------------------------------------------------
```

```
          0  recursive calls
          0  db block gets
        276  consistent gets
          0  physical reads
          0  redo size
     103888  bytes sent via SQL*Net to client
       2475  bytes received via SQL*Net from client
        192  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
       2856  rows processed
```

- 数据块被重新读入 Buffer Cache

如果有新的数据需要被读入 Buffer Cache 中，而 Buffer Cache 又没有足够的空闲空间，Oracle 就根据 LRU 算法将 LRU 链表中 LRU 端的数据置换出去。当这些数据被再次访问到时，需要重新从磁盘读入。

```
SQL> select owner, table_name from t_test2
  2  where owner = 'SYS';

718 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 1900296288
-------------------------------------------------------------------------------------
| Id  | Operation                   | Name         | Rows  | Bytes | Cost (%CPU)
| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |              |    99 |  2178 |    10   (0)
| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T_TEST2      |    99 |  2178 |    10   (0)
| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | T_TEST2_IDX1 |    99 |       |     1   (0)
| 00:00:01 |
-------------------------------------------------------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("OWNER"='SYS')


Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        145  consistent gets
          0  physical reads
          0  redo size
      21690  bytes sent via SQL*Net to client
        902  bytes received via SQL*Net from client
         49  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
        718  rows processed

SQL> select * from t_test1; --占用 Buffer Cache
```

```
47582 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 1883417357

--------------------------------------------------------------------------------
| Id  | Operation         | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |         | 47582 |  3996K|   151   (2)| 00:00:02 |
|   1 |  TABLE ACCESS FULL| T_TEST1 | 47582 |  3996K|   151   (2)| 00:00:02 |
--------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
        195  recursive calls
          0  db block gets
       3835  consistent gets
          5  physical reads
          0  redo size
    5102247  bytes sent via SQL*Net to client
      35277  bytes received via SQL*Net from client
       3174  SQL*Net roundtrips to/from client
          5  sorts (memory)
          0  sorts (disk)
      47582  rows processed

SQL> select owner, table_name from t_test2
  2  where owner = 'SYS';

718 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 1900296288

-------------------------------------------------------------------------------------
| Id  | Operation                   | Name         | Rows  | Bytes | Cost (%CPU)
| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |              |    99 |  2178 |    10   (0)
| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T_TEST2      |    99 |  2178 |    10   (0)
| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | T_TEST2_IDX1 |    99 |       |     1   (0)
| 00:00:01 |
-------------------------------------------------------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("OWNER"='SYS')


Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        145  consistent gets
```

```
         54  physical reads
          0  redo size
      21690  bytes sent via SQL*Net to client
        902  bytes received via SQL*Net from client
         49  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
        718  rows processed
```

- 全表扫描

当发生全表扫描（Full Table Scan）时，用户进程读取表的数据块，并将他们放在 LRU 链表的 LRU 端（和上面不同，不是放在 MRU 端）。这样做的目的是为了使全表扫描的数据尽快被移出。因为全表扫描一般发生的频率较低，并且全表扫描的数据块大部分在以后都不会被经常使用到。

而如果你希望全表扫描的数据能被 cache 住，使之在扫描时放在 MRU 端，可以通过在创建或修改表（或簇）时，指定 CACHE 参数。

1.2.2 逻辑读

逻辑读指的就是从（或者视图从）Buffer Cache 中读取数据块。按照访问数据块的模式不同，可以分为即时读（Current Read）和一致性读（Consistent Read）。注意：逻辑 IO 只有逻辑读，没有逻辑写。

- 即时读

即时读即读取数据块当前的最新数据。任何时候在 Buffer Cache 中都只有一份当前数据块。即时读通常发生在对数据进行修改、删除操作时。这时，进程会给数据加上行级锁，并且标识数据为“脏”数据。

```
SQL> select * from t_test1 where owner='SYS' for update;

22858 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 3323170753

-------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         | 22858 |  1919K|   151   (2)| 00:00:02 |
|   1 |  FOR UPDATE        |         |       |       |            |          |
|*  2 |   TABLE ACCESS FULL| T_TEST1 | 22858 |  1919K|   151   (2)| 00:00:02 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("OWNER"='SYS')


Statistics
----------------------------------------------------------
         44  recursive calls
      23386  db block gets
       2833  consistent gets
```

```
          0  physical reads
    5044956  redo size
    2029221  bytes sent via SQL*Net to client
      17138  bytes received via SQL*Net from client
       1525  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      22858  rows processed
```

- 一致性读

Oracle 是一个多用户系统。当一个会话开始读取数据还未结束读取之前，可能会有其他会话修改它将要读取的数据。如果会话读取到修改后的数据，就会造成数据的不一致。一致性读就是为了保证数据的一致性。在 Buffer Cache 中的数据块上都会有最后一次修改数据块时的 SCN。如果一个事务需要修改数据块中数据，会先在回滚段中保存一份修改前数据和 SCN 的数据块，然后再更新 Buffer Cache 中的数据块的数据及其 SCN，并标识其为“脏”数据。当其他进程读取数据块时，会先比较数据块上的 SCN 和自己的 SCN。如果数据块上的 SCN 小于等于进程本身的 SCN，则直接读取数据块上的数据；如果数据块上的 SCN 大于进程本身的 SCN，则会从回滚段中找出修改前的数据块读取数据。通常，普通查询都是一致性读。

下面这个例子帮助大家理解一下一致性读：

会话 1 中：

```
SQL> select object_name from t_test1 where object_id = 66;

OBJECT_NAME
------------------------------
I_SUPEROBJ1

SQL> update t_test1 set object_name = 'TEST' where object_id = 66;

1 row updated.
```

会话 2 中：

```
SQL> select object_name from t_test1 where object_id = 66;

OBJECT_NAME
------------------------------
I_SUPEROBJ1


Execution Plan
----------------------------------------------------------
Plan hash value: 1883417357

--------------------------------------------------------------------------------
| Id  | Operation         | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |         |     1 |    27 |   151   (2)| 00:00:02 |
|*  1 |  TABLE ACCESS FULL| T_TEST1 |     1 |    27 |   151   (2)| 00:00:02 |
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
```

```
   1 - filter("OBJECT_ID"=66)


Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        661  consistent gets
          0  physical reads
        108  redo size
        423  bytes sent via SQL*Net to client
        385  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

1.2.3 查找数据

在一个查询操作中，大量的读操作都产生于数据的查找过程中。减少查找过程是我们优化 IO 性能问题的重要目标。

下面介绍几种主要的数据查找方式。

- Full Table Scan

当查询条件无法命中任何索引、或者扫描索引的代价大于全表扫描代价的某一比例时（由参数 optimizer_index_cost_adj 设定），Oracle 会采用全表扫描的方式查找数据。当发生全表扫描时，Oracle 会自下向上一次读取一定数量（由参数 db_file_multiblock_read_count 设定）的数据块，一直读取到高水位标志（HWM，High Water Mark）下。Full Table Scan 会引起 db file scattered read 事件。

- INDEX UNIQUE SCAN

全表扫描查找数据的效率是非常低的。而索引能大幅提高查找效率。普通索引的数据结构是 B-Tree，树的叶子节点中包含数据的 ROWID，指向数据记录，同时还有指针指向前一个/后一个叶子节点。索引扫描每次读取一个数据块，索引扫描是“连续的”（Sequential）。当索引为 UNIQUE 索引时，每个叶子节点只会指向一条数据。如果 Oracle 能预知扫描结果只有 0 或 1 条记录时，会采用 INDEX UNIQUE SCAN。当对 Unique Index 中的所有字段进行完全匹配时，会发生 INDEX UNIQUE SCAN。

```
SQL> select object_name from t_test1
  2  where object_id = 66;


Execution Plan
----------------------------------------------------------
Plan hash value: 2634232531

-------------------------------------------------------------------------------------------
| Id  | Operation                   | Name       | Rows  | Bytes | Cost (%CPU)|
Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |            |     1 |    27 |     1   (0)|
00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T_TEST1    |     1 |    27 |     1   (0)|
00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | T_TEST1_PK |     1 |       |     1   (0)|
```

```
00:00:01 |
-------------------------------------------------------------------------------
```

INDEX UNIQUE SCAN 的查找过程如下：

1. 从数的根节点数据块开始查找；
2. 查找根节点块中所有 key 值中大于或等于要查找的值的最小 key 值；
3. 如果 key 值大于查找值，则继续查找这个 key 值之前一个 key 值所指向的子节点数据块；
4. 如果 key 值等于查找值，则继续查找这个 key 值所指向的子节点数据块；
5. 如果没有 key 值大于或等于查找值，则继续查找最大 key 值所指向的子节点数据块；
6. 如果继续查找的节点数据块是数一个分支节点，则重复 2~4 步；
7. 如果查找的节点是叶子节点数据块，则在数据块中查找等于查找值的 key 值；
8. 如果找到相等的 key 值，则返回数据和 ROWID；
9. 如果没找到相等的 key 值，则说明没有符合条件的数据，返回 NULL。

- INDEX RANGE SCAN

如果通过索引查找数据时，Oracle 认为会返回数据可能会大于 1，会进行 INDEX RANGE SCAN，例如 Unique Index 中字段不完全匹配查找时、非 Unique Index 查找时。

```
SQL> select object_name from t_test1
  2  where object_id < 66;

64 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 1635545337

-------------------------------------------------------------------------------------
| Id  | Operation                    | Name       | Rows  | Bytes | Cost (%CPU)|
Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |            |    57 |  1539 |     2   (0)|
00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T_TEST1    |    57 |  1539 |     2   (0)|
00:00:01 |
|*  2 |   INDEX RANGE SCAN           | T_TEST1_PK |    57 |       |     1   (0)|
00:00:01 |
-------------------------------------------------------------------------------------
```

INDEX RANGE SCAN 分为闭包（有前后查找边界）和非闭包（只有一边或者没有边界）。返回数据会依据索引增序排序，多个相同值则会按照 ROWID 的增序排序。以下的查找条件都是闭包的：

```
WHERE column = 'Value'
WHERE column like 'value%'
WHERE column between 'value1' and 'value2'
WHERE column in ('value1', 'value2')
```

以下查找条件非闭包：

```
WHERE column < 'value1'
WHERE column > 'value2'
```

闭包条件下的 INDEX RANGE SCAN 的查找过程如下：

1. 从数的根节点数据块开始查找；
2. 查找根节点块中所有 key 值中大于或等于要查找的起始值的最小 key 值；
3. 如果 key 值大于起始值，则继续查找这个 key 值之前一个 key 值所指向的子节点数据块；
4. 如果 key 值等于起始值，则继续查找这个 key 值所指向的子节点数据块；
5. 如果没有 key 值大于或等于起始值，则继续查找最大 key 值所指向的子节点数据块；
6. 如果继续查找的节点数据块是数一个分支节点，则重复 2~4 步；
7. 如果查找的节点是叶子节点数据块，则在数据块中大于或等于要查找的起始值的最小 key 值；
8. 如果 Key 值小于或等于结束值，则：如果所有 Key 字段都符合 WHERE 字句中的查找条件，则返回数据和 ROWID；否则继续查找当前叶子节点所指向的右边的叶子节点。

INDEX UNIQUE SCAN 和 INDEX RANGE SCAN 都会引起 db file sequential read 事件。

- TABLE ACCESS BY INDEX ROWID

当发生索引扫描时，如果需要返回的字段都在索引上，则直接返回索引上的数据，而如果还需要返回非索引上的字段的值，Oracle 则需要根据从索引上查找的 ROWID 到对应的数据块上取回数据，这时就是 TABLE ACCESS BY INDEX ROWID。

- INDEX FAST FULL SCAN & INDEX FULL SCAN

索引快速全扫描和全表扫描类似，一次读取 db_file_multiblock_read_count 个数据块来描所有索引的叶子节点。INDEX FAST FULL SCAN 和其他索引扫描不同，它不会从树的根节点开始读取，而是直接扫描所有叶子节点；也不会一次读取一个数据块，而是一次读取 db_file_multiblock_read_count 个数据块。INDEX FAST FULL SCAN 会引起 db file scattered read 事件。

```
SQL> select count(1) from t_test1 where object_id < 21314;

Execution Plan
----------------------------------------------------------
Plan hash value: 1586700957

-------------------------------------------------------------------------------
| Id  | Operation             | Name       | Rows | Bytes| Cost (%CPU)| Time    |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |            |     1|     4|    24   (5)| 00:00:01|
|   1 |  SORT AGGREGATE       |            |     1|     4|            |         |
|*  2 |   INDEX FAST FULL SCAN| T_TEST1_PK | 18264| 73056|    24   (5)| 00:00:01|
-------------------------------------------------------------------------------
```

在某些情况下，如 db_file_multiblock_read_count 值过小、强制使用索引扫描时，会发生 INDEX FULL SCAN。INDEX FULL SCAN 和 INDEX FAST FULL SCAN 不同，它是一种索引扫描，按照 B-Tree 的查找法从树的根节点开始扫描，遍历整棵树，并且一次读取一个数据块。它会引起 db file sequential read 事件。

```
SQL> select /*+index(a t_test1_pk)*/count(1) from t_test1 a;


Execution Plan
----------------------------------------------------------
Plan hash value: 138350774

-------------------------------------------------------------------------------
| Id  | Operation         | Name       | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |            |     1 |    61   (2)| 00:00:01 |
|   1 |  SORT AGGREGATE   |            |     1 |            |          |
|   2 |   INDEX FULL SCAN| T_TEST1_PK | 47582 |    61   (2)| 00:00:01 |
-------------------------------------------------------------------------------
```

2 IO 系统的设计和配置

要控制好数据库的整体 IO 性能，在规划数据库架构时就需要做好 IO 系统的设计和配置。例如，将对 IO 要求不同的文件放置在不同的存储设备上；规划数据文件的分布、均衡 IO 负担等。

2.1 OS 和存储相关

IO 性能是直接和操作系统已经硬件性能相关的。如果能利用操作系统的一些高级 IO 特性，或者采用更高速的磁盘设备，能大大提高 IO 性能。下面介绍一些 OS 的 IO 配置、不同的磁盘硬件设备以及存储技术。

2.1.1 文件系统（File System）和裸设备（Raw Device）

我们知道，内存的读写效率比磁盘高近万倍，因此 Oracle 在内存中开辟了一片区域，称为 Buffer Cache，使数据的读写尽量在 Buffer Cache 中完成。同样，在文件系统中，操作系统为了提高读写效率，也会为文件系统开辟一块 Buffer Cache 用于读写数据的缓存。这样，Oracle 的数据会被缓存 2 次。为了避免 OS 的这次缓存，我们可以采用裸设备做为数据文件的存储设备。裸设备，也称为裸分区（Raw Partition），它是一个没有被加载（Mount）到操作系统的文件系统上、也没有加载到 Oracle 集群文件系统（OCFS Oracle Cluster File System）的磁盘分区，它通过字符设备驱动来访问。裸设备的文件读写不由操作系统控制，而是由应用程序（如 Oracle RDBMS）直接控制。

2.1.2 IO 方式

OS 和文件系统对 IO 的控制存在多种方式，不同的 IO 方式下对于数据库的 IO 性能影响也不同。

2.1.2.1 Direct IO & Concurrent IO

除了裸设备，某些文件系统可以支持 Direct IO，以避开读写缓冲。如果要使用 Direct IO，需要指定 Oracle 参数“filesystemio_options”来设置支持 Direct IO。但是要注意，不同 OS 中的不同文件系统对 Direct IO 的支持也不同：

- Windows　　在 windows 中不需要做特别设置可以直接使用 Direct IO;
- AIX　　在 AIX 中，JFS 文件系统需要通过设置“filesystemio_options”为“SETALL”或者“DIRECTIO”来支持 Direct IO;
- LINUX　　Linux 在内核版本为 2.4.9 以上才支持 Direct IO。NFS 或者 OCFS 文件系统支持 Direct IO。需要设置“filesystemio_options”为“SETALL”或者“DIRECTIO”;
- Solaris　　Solaris 需要在操作系统中设置“forcedirectio”选项，并设置“filesystemio_options”为“SETALL”或者“DIRECTIO”。

参数“filesystemio_options”支持 4 种值：

- ASYNCH: 使 Oracle 支持文件的异步（Asynchronous）IO;
- DIRECTIO：使 Oracle 支持文件的 Direct IO;
- SETALL：使 Oracle 同时支持文件的 Asynchronous IO 和 Direct IO;
- NONE：使 Oracle 关闭对 Asynchronous IO 和 Direct IO 的支持。

在 AIX 的 JFS2 文件系统上，如果“filesystemio_options”为“SETALL”，则会支持 Concurrent IO。CIO 比 DIO 的性能更高，因为 JFS2 的 CIO 支持多个进程同时对一个文件进行读写。

2.1.2.2 Asynchronous IO & Synchronous IO

通常，用的比较多的 IO 模型是同步 IO（Synchronous IO）。在这种模式下，当请求发出之后，应用程序就会阻塞，直到请求满足为止。这种模式最大好处就是调用应用程序在等待 I/O 请求完成时不需要使用 CPU 资源。但是，对于一些强调高响应速度的程序（如 DB）来说，希望这种等待时间越短越好，我们这时就可以考虑采用异步 IO（Asynchronous IO）模式。异步 IO 模式下，进程发出 IO 请求后无需等待 IO 完成，可以去处理其它事情；IO 请求被放入一个队列中，一旦 IO 完成，系统会发出信号通知进程。

异步 IO 可以使需要大量写的 Oracle 进程（如 DBWn 进程）将 IO 请求队列化，以充分利用硬件的 IO 带宽，从而使它们能最大程度实现并行处理。异步 IO 还可以使那些需要进行大量计算的操作（如排序）在它们发出 IO 请求前预先从磁盘取出数据，以使 IO 和计算并行处理。

确认操作系统已经设置支持 AIO 后，还需要设置 Oracle 初始化参数"DISK_ASYNCH_IO"为“true”以支持异步 IO。

2.1.3　负载均衡及条带化（Striping）

当多个进程同时访问一个磁盘时，会出现磁盘冲突。大多数磁盘系统都对访问次数（每秒的 IO 操作）和数据传输率（每秒传输的数据量）有限制。当达到这些限制时，后面要访问磁盘的进程就需要等待，这时就是所谓的磁盘冲突。

避免磁盘冲突是优化 IO 性能的一个目标，这就需要将一个热点磁盘上的 IO 访问负载分担到其他可用磁盘上，也就是 IO 负载均衡。在一些成熟的磁盘负载均衡技术出现之

前，DBA 需要了解/预测各系统的 IO 负载量，通过手工配置每个数据到不同存放位置以分担 IO 负载来达到负载均衡的目的。

条带化技术就是将数据分成很多小部分并把他们分别存储到不同磁盘上的不同文件中去。这就能使多个进程同时访问数据的多个不同部分而不会造成磁盘冲突。很多操作系统、磁盘设备供应商、各种第三方软件都能做到条带化。通过条带化，DBA 可以很轻松的做到 IO 负载均衡而无需去手工配置。

2.1.4 RAID

RAID 的全称是独立磁盘冗余阵列（Redundant Array of Independent Disks）。它通过将多个相对比较便宜的磁盘组合起来，并相互连接，同时都连到一个或多个计算机上，以组成一个磁盘组，使其性能和容量达到或超过一个价格更昂贵的大型磁盘。RAID 分为 6 级。

- RAID-0

RAID-0 只提供纯粹的条带化（Stripping）。条带可以使一个大文件被多个磁盘控制器同时访问，因此支持对数据的并发访问。RAID-0 不提供数据冗余和奇偶保护，它只关注性能。如果 RAID-0 中任何一个磁盘出错，整个数据库都会崩溃。

- RAID-1

RAID-1 提供磁盘镜像（Disk Mirror）。在 RAID-1 中，所有数据都会被写入两个独立的磁盘中，以实现对数据的冗余保护。两块磁盘的数据是同时写入的，以保证其速度不会低于写入单独磁盘的速度。RAID-1 实现了数据的完全冗余，它提供了所有 RAID 级别中最安全可靠的数据保护。在这种模式下，写的性能下降了，但读的性能被提升了。此外，RAID-1 也是最占用磁盘空间的模式

- RAID 0+1

RAID-0 能提供更好的性能，RAID-1 提供最佳的数据保护。如果把两者结合在一起就能同时提供高性能和数据保护，但是也会同时提高磁盘阵列造价。

- RAID-3

在 RAID-3 中，会有一块专门的磁盘驱动被用作存储错误修正或者奇偶校验数据。而其他的磁盘驱动则被条带化。RAID-3 的并行处理能力比较低，它适合于主要是读操作的系统（如决策分析系统 DSS，但是 DSS 会存在大量复杂查询，需要做 JOIN，同样也会存在一些临时的写操作），不适合存在大量写操作的系统（OLTP）。

- RAID-5

RAID-5 不做全磁盘镜像，但它会对每一个写操作做奇偶校验计算并写入奇偶校验数据。奇偶校验磁盘避免了像 RAID-1 那样完全重复写数据。当一个磁盘失效，校验数据被用来重建数据，从而保证系统不会崩溃。为避免磁盘瓶颈，奇偶校验和数据都会被分布到阵列中的各个磁盘。尽管读的效率提高了，但是 RAID-5 需要为每个写操作做奇偶校验，因此它的写的效率很差。

- RAID-S

RAID-S 是 EMC 公司的 RAID-5 的实施方案，它和纯粹的 RAID-5 存在以下区别：

(1) 它条带化奇偶校验，但不条带化数据；

(2) 它与一个带有写缓存的异步硬件环境合并。

这个缓存主要是一种延迟写的机制，因此它能让系统在相对不忙的时候计算和写奇偶校验信息。

- RAID-7

RAID-7 也同样引入了缓存机制，这个缓存是被一个内嵌式操作系统控制。但是，RAID-7 中数据是被条带化的，而奇偶校验不被条带化。奇偶校验信息被存放着一个或者多个专门的磁盘上。

2.1.5 SAN

SAN（Storage Area Network，存储区域网）是一个高速的子网，这个子网中的设备可以从你的主网卸载流量。通常 SAN 由 RAID 阵列连接光纤通道（Fibre Channel）组成，SAN 和服务器和客户机的数据通信通过 SCSI 命令而非 TCP/IP，数据处理是“块级”（block level）。

SAN 通过特定的互连方式连接的若干台存储服务器组成一个单独的数据网络，提供企业级的数据存储服务。 SAN 是一种特殊的高速网络，连接网络服务器和诸如大磁盘阵列或备份磁带库的存储设备，SAN 置于 LAN 之下，而不涉及 LAN。利用 SAN，不仅可以提供大容量的存储数据，而且地域上可以分散，并缓解了大量数据传输对于局域网的影响。SAN 的结构允许任何服务器连接到任何存储阵列，不管数据置放在哪里，服务器都可直接存取所需的数据。

2.1.6 NAS

NAS 是 Network Attached Storage（网络附加存储）的简称。在 NAS 存储结构中，存储系统不再通过 I/O 总线附属于某个服务器或客户机，而直接通过网络接口与网络直接相连，由用户通过网络访问。它是连接到一个计算机网络的文件层的数据存储，它可以为不同网络客户端提供数据存储服务。NAS 的硬件与传统的专用文件服务器相似。它们的不同点在于软件端。NAS 中的操作系统和其他软件只提供数据存储、数据访问功能，以及对这些功能的管理。与传统以服务器为中心的存储系统相比，数据不再通过服务器内存转发，直接在客户机和存储设备间传送，服务器仅起控制管理的作用。

2.2 IO 配置

在借助各种成熟的存储技术的基础上，合理配置系统的 IO 分布及系统 IO 配置能大量减少系统在生产运行中出现 IO 性能及相关问题的几率。当然，这些配置是我们在布置数据库系统时初始建议，对于复杂的系统来说，很多配置（如一些存储相关的参数）是需要根据系统的运行状况进行调优的。

在数据库系统中，如果某个文件或者某块磁盘上存在远远高于其他文件或磁盘的大量 IO 访问，我们就称这个文件或磁盘为热点文件/磁盘。我们在做 IO 规划时的一个重要目标就是要消除系统中热点文件/磁盘的存在，使整个系统的 IO 负载相对平衡。

2.2.1 条带化的设置

由于现在的存储技术成熟、成本降低，大多数系统都采用条带化来实现系统的 IO 负载分担。如果操作系统有 LVM（Logical Volume Manager 逻辑卷管理器）软件或者硬件条带设备，我们就可以利用这些攻击来分布 IO 负载。当使用 LVM 或者硬件条带时，决定因素是条带深度（stripe depth）和条带宽度（stripe width）：

- 条带深度指的是条带的大小，也叫条带单元；
- 条带宽度指的是条带深度的产量或者一个条带集中的驱动数；

需要根据系统的 IO 要求来合理的选择这些数据。对于 Oracle 数据库系统来数，比较合理的条带深度是从 256K 到 1M。下面分析影响条带深度和条带宽度的影响因素。

2.2.1.1 条带深度

为了提高 IO 效率，我们要尽量使一次逻辑 IO 请求由一块磁盘的一次物理 IO 请求。因而影响条带的一个重要因素就是一次逻辑 IO 请求的大小。

此外，系统中 IO 的并发度不同我们对条带的配置要求也不同。例如，在高并发度且 IO 请求的大小都比较小的情况下，我们希望一块磁盘能同时响应多个 IO 操作；而在那些存在大 IO 请求的低并发度系统中，我们可能就需要多块磁盘同时响应一个 IO 请求。无论是一个磁盘还是多个磁盘响应 IO 请求，我们的一个原则是让一次逻辑 IO 能被一次处理完成。

下面先看下影响 IO 大小的操作系统和 Oracle 的相关参数：

- db_block_size：Oracle 中的数据块大小，也决定了 Oracle 一次单个 IO 请求中的数据块的大小；
- db_file_multiblock_read_count：在多数据块读时，一次读取数据块的数量，它和参数 db_block_size 一起决定了一次多数据块读的大小，它们的乘积不能大于操作系统的最大 IO 大小；
- 操作系统的数据块大小：这个参数决定拉 Redo Log 和 Archive Log 操作时的数据块大小，对于大多数 Unix 系统来说，该值为 512K；
- 最大操作系统 IO 大小：决定了一次单个的 IO 操作的 IO 大小的上限，对于大多数 Unix 系统来说，由参数 max_io_size 设置；
- sort_area_size：内存中 sort area 的大小，也决定了并发排序操作时的 IO 大小；
- hash_area_size：内存中 hash area 的大小，也决定了哈希操作的 IO 大小。

其中，前面两个是最关键的两个参数。

在 OLTP 系统中，会存在大量小的并发的 IO 请求。这时就需要考虑选择比较大的条带深度。使条带深度大于 IO 大小就称为粗粒度条带（Coarse Grain Striping）。在高并行度系统中，条带深度为（n * db_block_size），其中 n 为大于 1 的整数。

通过粗粒度条带能实现最大的 IO 吞吐量（一次物理 IO 可以同时响应多个并发的逻辑 IO）。大的条带深度能够使像全表扫描那样的多数据块读操作由一个磁盘驱动来响应，并提高多数据块读操作的性能。

在低并发度的 DSS 系统中，由于 IO 请求比较序列化，为了避免出现热点磁盘，我们需要避免逻辑 IO 之由一块磁盘处理。这是，粗粒度条带就不适合了。我们选择小的条带深度，使一个逻辑 IO 分布到多个磁盘上，从而实现 IO 的负载均衡。这就叫细粒度条带。条带深度的大小为（n * db_block_size），其中 n 为小于多数据块读参数（db_file_multiblock_read_count）大小的整数。

另外，IO 过程中，你无法保证 Oracle 数据块的边界能和条带单元的大小对齐。如果条带深度大小和 Oracle 数据块大小完全相同，而它们的边界没有对齐的话，那么就会存在大量一个单独的 IO 请求被两块磁盘来完成。

在 OLTP 系统中，为了避免一个逻辑 IO 请求被多个物理 IO 操作完成，条带宽度就需要设置为两倍或者两倍以上于 Oracle 数据块大小。例如，如果条带深度是 IO 大小的 N 倍，对于大量并发 IO 请求，我们可以保证最少有（N-1）/ N 的请求是由一块磁盘来完成。

2.2.1.2 条带宽度

正如我们前面所述，无论是一个还是多个磁盘响应一个逻辑 IO，我们都要求 IO 能被一次处理。因而在确定了条带深度的基础上，我们需要保证条带宽度 >= IO 请求的大小 / 条带深度。

此外，考虑到以后系统容量的扩充，我们也需要规划好条带宽度。

如今大多数 LVM 都支持在线动态增加磁盘。也就是在磁盘容量不足时，我们可以随时将新磁盘加入到一个已经使用的逻辑卷中。这样的话，我们在设置逻辑卷时就可以简单地将所有磁盘都归入到一个卷中去。

但是，有些 LVM 可能还不支持动态增加磁盘。这时我们就需要考虑以后的容量扩充对 IO 均衡的影响了。因为你新增加的磁盘无法加入原有卷，而需要组成一个新的卷。但一般扩充的容量和原有容量比较相对比较小，如果原有卷的条带宽度比较大的话，新增加的卷的条带宽度无法达到其大小，这样就会使新、旧卷之间出现 IO 失衡。

例如，一个系统的初始配置是一个包含 64 块磁盘、每块磁盘大小为 16G 的单一逻辑卷。磁盘总的大小是 1T。随着数据库的数据增长，需要增加 80G 的空间。我们把新增加的 5 个 16G 磁盘再组成一个逻辑卷。这样就会导致两个卷上的 IO 失衡。为了避免这种情况。我们可以将原有磁盘配置成每个条带宽度为 8 个磁盘的 8 个逻辑卷，这样在新增加磁盘时可以也增加为 8 个磁盘的新卷。但必须要保证 8 个磁盘的条带宽度能够支持系统的每秒 IO 吞吐量。

如果你的条带宽度设置得比较小，就需要估算出你的各个数据库文件的 IO 负载，并根据负载量不同将他们分别部署到不同卷上一分担 IO 负载。

2.2.2 人工条带

如果系统不支持 LVM 或者硬件条带，IO 负载就必须由 DBA 根据数据库文件的 IO 负载不同手工将他们分散到各个磁盘上去以保证整个系统的 IO 负载均衡。

有许多 DBA 会将哪些使用频率非常高的表和它的索引分开存储。但实际上这种做法并不正确。在一个事务中，索引会先被读取到然后再读取表，它们的 IO 操作是有前后顺序的，因此索引和表存储在同一个磁盘上是没有冲突的。仅仅因为一个数据文件即包含了索引又包含了数据表而将它分割是不可取的。我们需要根据文件上的 IO 负载是否已经影响到了数据库的性能来决定是否将数据文件分割。

为了正确分布文件，我们首先必须先了解各个数据库文件的 IO 负载需求以及 IO 系统的处理能力。鉴定出每个文件的 IO 吞吐量。找出哪些文件的 IO 吞吐率最高而哪些 IO 量很少，将它们分散分布到所有磁盘上去以平衡 IO 吞吐率。

如果你不了解或者无法预计文件的 IO 负载，就只能先估计他们的 IO 负载来规划文件分布，在系统运行过程中再做调整。

2.2.3 文件分离

无论是采用操作系统条带化还是手工 IO 分布方式，如果 IO 系统或者 IO 规划布置无法满足 IO 吞吐率的要求，我们就需要考虑将高 IO 吞吐率的文件和其他文件分离。我们可以在存储规划阶段或者系统运行阶段找出那样的文件。

除了 IO 吞吐率，在决定是否分割文件时，我们还需要考虑可恢复性以及数据容量扩张问题。

但是在分割文件之前，一定要确认存在 IO 瓶颈，然后再根据产生 IO 瓶颈的数据定位到存在高 IO 吞吐率的文件（热点文件）。

2.2.3.1 表、索引和临时表空间

如果具有高 IO 吞吐率的数据文件属于包含表和索引的表空间，我们就需要找出这些文件的 IO 是否可以通过 SQL 语句调优或者优化应用程序来降低。

如果具有高 IO 吞吐率的数据文件属于临时表空间，那我们就需要检查是否可以通过避免或调优 SQL 语句的排序操作来降低 IO。

经过应用调优后，如果 IO 分布仍然无法满足 IO 吞吐的要求，我们就需要考虑分离高 IO 吞吐率的数据文件了。

2.2.3.2 Redo Log 文件

如果具有高 IO 吞吐率的文件是 Redo Log 文件，则需要考虑将 Redo Log 文件与其他文件分离，可以通过以下配置来实现：

- 将所有 Redo Log 文件放到没有任何其他文件的磁盘上去。考虑到可恢复性，需要将一个 Redo Log 组中的成员文件分别放到不同的物理磁盘上去；
- 将每个 Redo Log 组放到一个没有任何其他文件的单独磁盘上；
- 通过操作系统条带化工具，将 Redo Log 文件条带化分布到多个磁盘上去；
- 不要将 Redo Log 文件放到 RAID 5 上去

Redo Log 文件是由 LGWR 进程序列化的写入的。如果在同一个磁盘上不存在并发的其他 IO 操作，写入效率就更高。我们需要确认已经没有其他优化调整空间再考虑分割 Redo Log 文件。如果系统支持 AIO 但还没有激活该特性，可以考虑激活 AIO 看是否能解决 Redo Log 的 IO 性能瓶颈。

2.2.3.3 归档 Redo Log

如果归档变慢，我们也许可以通过使 LGWR 的写操作与 Archive 进程的读操作分离来避免 LGWR 进程鱼 Archive 进程直接的 IO 冲突。我们可以同交替成组存放 Redo Log 文件来实现。

例如，我们有四组 Redo Log，每组包含两个 Log 文件：（A1，A2）、（B1，B2）、（C1，C2）、（D1，D2）。我们就可以以下面这种存放方式将它们分布存储到四个磁盘上去来实现磁盘分离访问：（A1，C1）、（A2、C2）、（B1，D1）、（B2，D2）。

当 LGWR 进程做日志切换时，如从 A 组切换到 B 组，LGWR 开始向 B 组写 Redo Log（第三、四块磁盘），而 Archive 进程则从 B 组读取数据（第一、二块磁盘）写入归档文件中去，他们分别访问的是不同磁盘，因而避免了 IO 冲突。

2.3 三种简单的配置方法

这里给出三种简单的操作系统 IO 配置的例子，包括如何简单地计算来决定磁盘的拓扑结构、条带深度等等。

2.3.1 将所有文件条带化到所有磁盘上去

IO 配置最简单的方法就是建立一个大的逻辑卷，将所有磁盘都条带化到这个卷中去。考虑到可恢复性，这个卷需要被镜像（RAID 1）。每个磁盘的条带深度必须大于频繁执行的 IO 操作的最大 IO 大小。这种配置对大多数情况都能提供足够的性能支持。

2.3.2 将归档日志放到另外的磁盘上去

在归档模式下，如果归档文件也和其他文件放在同一个条带化的卷中，那么当归档进程对 Redo Log 进行归档时，会大大增加磁盘的 IO 负载。将归档日志转移到其他磁盘上有如下好处：

- 归档进程效率提高；
- 当归档时，其他进程受到归档进程的影响

归档日志的磁盘数由归档日志产生的频率以及归档存储容量决定。

2.3.3 将 Redo Log 文件放到另外的磁盘上去

在更新非常频繁的 OLTP 系统中，Redo Log 的写操作非常频繁。将 Redo Log 文件转移到其他磁盘上可以有如下好处：

- 写 Redo Log 的读写效率最高，因而事务的执行也能获得最佳性能；
- 写 Redo Log 操作不会影响任何其他 IO 操作

Redo Log 的磁盘数量有 Redo Log 的大小决定。由于现在的磁盘容量都非常大，通常配置两个磁盘（如果做镜像则需要四块）就足够了。并且，根据我们前面的分析，将 Redo Log 文件交互的存放到两块磁盘上去能避免 LGWR 进程的写操作与 ARCH 进程的读操作之间的 IO 冲突。

3 Oracle 中的 IO 问题及其解决思路

对于负载偏重点不同，我们可以简单的将数据库系统分为 CPU 负载系统（CPU Bound System）和 IO 负载系统（IO Bound System）。顾名思义，CPU 负载系统的资源瓶颈在于 CPU，而 IO 负载系统的瓶颈在于磁盘 IO。

我们可以通过操作系统的一些命令来确认一个系统是否是存在 IO 负载。在 UNIX 下，可以使用"iostat"或者"sar -d"来看系统的 IO 情况；在 windows 下，可以通过系统的性

能监视器查看，但由于性能监控器中看到的 IO 是静态的 IO 总量信息，并不直观，因此也可以用本站的 TopShow 工具来查看实时的 IO 信息。

在 UNIX 系统下，发现 CPU IDEL 很低并不一定代表这是一个 CPU 负载系统。一个 IO 负载系统在表面上看 CPU 的 IDEL 值也可能很低：

```
oracle@db01:/export/home/oracle> sar -u 1 10

HP-UX hkhpdv45 B.11.23 U ia64    10/24/07

09:43:05    %usr    %sys    %wio   %idle
09:43:06      43      25      30       1
09:43:07      44      36      19       1
09:43:08      23      27      44       6
09:43:09      12      37      50       1
09:43:10      10      36      51       3
09:43:11      15      34      42       9
09:43:12      18      36      44       3
09:43:13      17      35      46       2
09:43:14      12      32      52       4
09:43:15      12      31      56       1

Average       21      33      43       3
```

我们可以注意到，实际上 WIO 是引起 CPU IDEL 过低的主要原因。WIO 是当一个进程需要运行或已经运行后，因为需要等待 IO 事件而被阻塞了。事实上 CPU 是处于 IDEL 状态（在某些系统中，已经将 WIO 取消并归为 IDEL），真正的原因是系统中存在 IO 瓶颈。

通过 iostat 或者 sar -d 我们可以找出存在 IO 瓶颈的磁盘设备，如果该磁盘设备是用于 Oracle 数据库存储文件的，我们可以判断出是数据库存在 IO 问题。在 windows 下，可以通过 TopShow 来找出哪个进程正在进行大量 IO 传输，如果是 Oracle 进程，也可以判断为是数据库存在 IO 问题。

确认系统存在 IO 问题后，我们就需要定位到底是什么引起的 IO 问题，该采取什么措施来解决问题。根据我们前面的介绍，Oracle 中存在各种 IO，要定位 IO，最好的工具是 statspack（在 10g 以后，可以用 AWR）。通过 statspack report 的 Top 5 Events，我们可以看到对系统系能影响最大的 5 个等待 event，而不同的 IO 问题会对应不同 Event，所以，我们可以根据这些 event 采取不同的措施来解决 IO 问题。下面是一个典型的 IO 负载系统的 Top 5 Event：

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                                 % Total
Event                                              Waits    Time (s) Ela Time
-------------------------------------------- ------------ ----------- --------
db file sequential read                       70,575,969     344,200    53.34
db file scattered read                        11,240,748     163,242    25.30
log file sync                                    657,241      36,363     5.64
CPU time                                                      35,290     5.47
log file parallel write                          833,799      20,767     3.22
```

可以看到，前两个时间“db file sequential read”和“db file scattered read”分别占了总等待时间的 53.34%和 25.30%，而我们前面提到这两个事件分别是由索引扫面和全表扫面（或快速索引扫面）引起的，因此，能解决索引扫面问题和全表扫面问题就能解决这个系统的 IO 瓶颈。

IO 问题到底对 CPU 有多大影响呢？我们用以上例子中的数据分析一下。从等待时间统计数据中，我们看到的是时间在总等待时间中所占的比例。而系统的“总响应时间 ”= “等待时间 ”+ “CPU 工作时间”（注意，上面 Top 5 事件中的“CPU Time”不是指 CPU 的工作时间，而是指 CPU 的等待时间）。“CPU 工作时间”的数据我们可以在“Instance Activities Stats for DB”这一分类统计数据中找到：

Statistic	Total	per Second	per Trans
CPU used by this session	17,136,868	396.7	15.5

先计算出“总等待时间” = 344,200 * 100% / 53.34% = 645,294s

“总响应时间” = “总等待时间” + “CPU 工作时间” = 645,294 + 17,136,868 = 17,782,162s

我们可以算出“CPU 工作时间”、“db file sequential read”和“db file scattered read”分别在“总响应时间中所占的比例为：

CPU 工作时间 = 17,136,868 / 17,782,162 = 96.4%

“db file sequential read” = 344,200 / 17,782,162 = 1.9%

“db file scattered read” = 163,242 / 17,782,162 = 0.9%

可见，IO 事件所引起的等待时间在总响应时间所占比例并不大。因此，我们在做系统优化之前先分析系统是 CPU 负载系统还是 IO 负载系统对于我们的优化方向和最终的优化效果起很大的作用。

以下事件是可能由 IO 问题引起的等待事件，在 IO 负载系统中，我们要特别关注这些事件：

- 与数据文件相关的 IO 事件

'db file sequential read'
'db file scattered read'
'db file parallel read'
'direct path read'
'direct path write'
'direct path read (lob)'
'direct path write (lob)'

- 与控制文件相关的 IO 事件

'control file parallel write'
'control file sequential read'
'control file single write'

- 与 Redo 日志相关的 IO 事件

'log file parallel write'
'log file sync'
'log file sequential read'
'log file single write'
'switch logfile command'
'log file switch completion'

'log file switch (clearing log file)'
'log file switch (checkpoint incomplete)'
'log switch/archive'
'log file switch (archiving needed)'

- 与 Buffer Cache 相关的 IO 事件

'db file parallel write'
'db file single write'
'write complete waits'
'free buffer waits'

下面我们就分别介绍如何解决 IO 问题。

3.1 IO 调优的思路及常用手段

通过对 statspack 或者 awr 报告的分析，我们可以得知是那些 IO 相关事件引起的 IO 问题。针对不同的事件，可以采取不同的分析、处理方法。而有一些通用的方法并不是针对特定的事件的。我们这里先介绍一下这些方法。

3.1.1 通过 SQL 调优来减少 IO 请求

一个没有任何用户 SQL 的数据库几乎不产生任何 IO。基本上数据库所有的 IO 都是直接或间接由用户提交的 SQL 所导致的。这意味着我们可以通过控制单个 SQL 产生的 IO 来降低数据库总的 IO 请求。而通过 SQL 调优来降低 SQL 查询计划中的 IO 操作次数则是降低 SQL 产生 IO 的最好方法。数据库的性能问题通常是由少数几个 SQL 语句所导致的，它们产生了大量 IO 导致了整个数据库的性能下降。优化几条问题语句往往就能解决整个数据库的 IO 性能问题。

从 Oracle 10g 开始，ADDM 能够自动检测出问题语句，同时，再通过查询优化建议器能够自动优化语句并降低它们对 IO 的消耗。关于 ADDM 和查询优化建议器可以参考文章《Oracle 10G 新特性——ADDM 和查询优化建议器》。

3.1.2 通过调整实例参数来减少 IO 请求

在这种方法中，主要有两种途径来实现对 IO 的优化。

- 使用内存缓存来减少 IO

通过一些内存缓存，如 Buffer Cache、Log Buffer、Sort Area，可以降低数据库对 IO 的请求。

当 Buffer Cache 被增大到一定大小时，绝大多数结果可以直接从缓存中获取到，而无需从磁盘上读取了。而在进行排序操作时，如果 Sort Area 足够大，排序过程中产生的临时数据可以直接放在内存中，而无需占用临时表空间了。

- 调整 multiblock IO（多数据块 IO）的大小

控制 Multiblock IO 的参数叫 DB_FILE_MULTIBLOCK_READ_COUNT，它控制在多数据块读时一次读入数据块的次数。适当增加这个参数大小，能够提高多数据块操作（如全表扫

描）的 IO 效率。例如，读取 100M 数据，如果每次读取 1M 一共读取 100 次的效率就比每次读取 100K 一共读取 1000 次更快。但是这个数字达到一定大小后，再增加就作用不大了：每次 10M 一共读 100 次来读取 1G 的数据的效率和单独一次读取 1G 数据的效率是没有多大区别的。这是因为 IO 效率受到 2 个因素的影响：IO 建立时间和 IO 传输时间。

IO 建立时间对于不同 IO 大小来说都是相同的，它决定了对小 IO 的总的 IO 时间，增大 Multiblock IO 大小可以减少 IO 建立时间；

IO 传输时间与 IO 大小是成正比的，在小 IO 时，IO 传输时间一般比 IO 建立时间少，但对于大 IO 操作来说，IO 传输时间决定了总的 IO 时间。因此 Multiblock IO 大小增大到一定大小时，它对总的 IO 时间影响就不大了。

3.1.3 在操作系统层面优化 IO

如我们前面所介绍的，利用一些操作系统提供的提升 IO 性能的特性，如文件系统的异步 IO、Direct IO 等来优化数据库系统的 IO 性能。另外一种方法就是增加每次传输的最大 IO 大小的限制（大多数 Unix 系统中，由参数 max_io_size 控制）。

3.1.4 通过 Oracle ASM 实现对 IO 的负载均衡

ASM（Automatic Storage Manager 自动存储管理）是从 Oracle 10g 开始引入的。它是一个建立在数据库内核中的文件系统和卷管理器。它能自动将 IO 负载均衡到所有可用的磁盘启动器上去，一避免“热区”。ASM 能防止碎片，因此无需重建数据来回收空间。数据被均衡分布到所有硬盘上。

3.1.5 通过条带化、RAID、SAN 或者 NAS 实现对 IO 的负载均衡

这个方法通过一些成熟的存储技术，如条带化、RAID、SAN 和 NAS，来将数据库 IO 分布到多个可用的物理磁盘实现负载均衡，以避免在还存在空闲可用磁盘时出现的磁盘争用和 IO 瓶颈问题。

关于这几种存储技术，我们文章的前面部分都有做介绍。

3.1.6 通过手工布置数据库文件到不同的文件系统、控制器和物理设备上来重新分布数据库 IO

当数据库系统中缺乏以上各种存储技术手段时，我们可以考虑使用这种方式。这样做的目的是使数据库的 IO 得到均匀分布，从而避免在还有空闲磁盘时出现磁盘争用和 IO 瓶颈问题。当然这种手工分布 IO 方法是无法达到以上的自动分布 IO 的效果的。

3.1.7 其他手段

系统中总会存在一些 IO 是无法消除或降低的。如果采用以上手段还不能满足 IO 性能要求的话，可以考虑这两种方法：

- 将老数据移除你的生产数据库（Housekeep）
- 采用更多、更快的硬件

3.2 数据文件相关的 IO 事件

数据库系统中的大多数的 IO 请求都是针对数据文件的。因此大多数情况下，与数据文件相关的 IO 事件是引起系统 IO 性能的主要原因。这些事件也是我们文章需要重点介绍的事件。下面分别针对不同事件介绍问题的解决思路。

3.2.1 db file sequential read

这个事件是是最常见的 IO 等待事件。它一般发生在读取单独数据块时，如读取索引数据块或者通过索引访问一个表数据块，另外在读取数据文件头数据块时也会发生 db file sequential read 等待事件。

当发现这个等待事件成为系统等待事件中的主要事件，我们可以通过一下方法来处理：

3.2.1.1 优化 Top SQL

从 statspack 或者 awr 报告中的“SQL ordered by Reads”部分或者通过 V$SQL 视图找出系统中的 Top SQL，对 SQL 进行调优以减少 IO 请求。

- 当 SQL 中存在 Index Range Scan 时，如果访问的索引的选择性不好就会导致需要访问过多的数据块，这时可以通过建立一个、或强制 SQL 使用一个已经存在的选择性更好的索引。这样使我们访问更少的数据块来获取到需要的数据。

```
SQL> select object_id, object_name
  2  from t_test1
  3  where owner = 'SYS'
  4  and created > sysdate - 30;

no rows selected

Execution Plan
----------------------------------------------------------
Plan hash value: 4014220762

-------------------------------------------------------------------------------------
| Id  | Operation                   | Name         | Rows  | Bytes | Cost (%CPU)
| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |              |     1 |    39 |    11   (0)
| 00:00:01 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| T_TEST1      |     1 |    39 |    11   (0)
| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | T_TEST1_IDX1 |   576 |       |     1   (0)
| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("OWNER"='SYS' AND "CREATED">SYSDATE@!-30)


Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        658  consistent gets
         45  physical reads
          0  redo size
        339  bytes sent via SQL*Net to client
        374  bytes received via SQL*Net from client
          1  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
```

```
          0  rows processed

SQL> create index t_test1_idx2 on t_test1(owner, created);

Index created.

SQL> select object_id, object_name
  2  from t_test1
  3  where owner = 'SYS'
  4  and created > sysdate - 30;

no rows selected


Execution Plan
----------------------------------------------------------
Plan hash value: 3417015015
-------------------------------------------------------------------------------------
| Id  | Operation                   | Name         | Rows  | Bytes | Cost (%CPU)
| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |              |    49 |  1911 |     2   (0)
| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T_TEST1      |    49 |  1911 |     2   (0)
| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | T_TEST1_IDX2 |    49 |       |     1   (0)
| 00:00:01 |
-------------------------------------------------------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("OWNER"='SYS' AND "CREATED">SYSDATE@!-30)


Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
          2  consistent gets
          1  physical reads
          0  redo size
        339  bytes sent via SQL*Net to client
        374  bytes received via SQL*Net from client
          1  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          0  rows processed
```

- 如果索引存在碎片，那每个索引数据块上的索引数据就更少，会导致我们需要访问更多的索引数据块。这时，我们需要考虑重建索引来释放碎片；

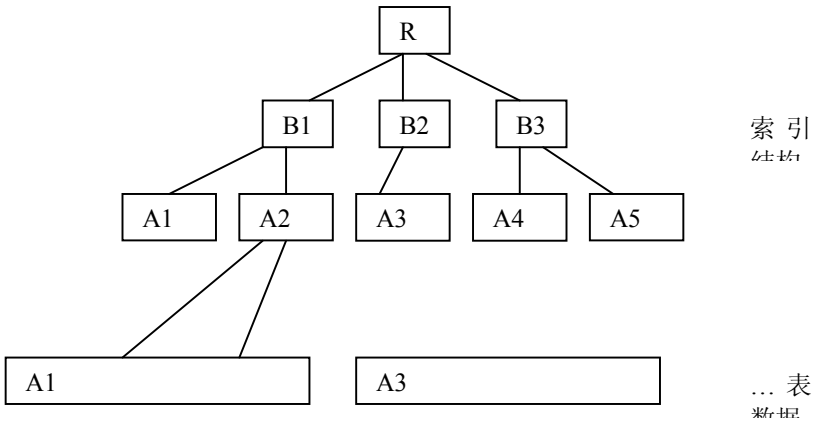
  判断一个所以是否需要重建，我们介绍一个简单的方法：对一个索引进行结构分析后，如果该索引占用超过了一个数据块，且满足以下条件之一：B-tree 树的高度大于 3；使用百分比低于 75%；数据删除率大于 15%，就需要考虑对索引重建：

```
SQL> analyze index t_test1_idx1 compute statistics;
```

```
Index analyzed.

SQL> analyze index t_test1_idx1 validate structure;

Index analyzed.

SQL> select btree_space, -- if > 8192(块的大小)
  2         height, -- if > 3
  3         pct_used, -- if < 75
  4         del_lf_rows/(decode(lf_rows,0,1,lf_rows)) *100 as deleted_pct -- if >
20%
  5  from index_stats;

BTREE_SPACE     HEIGHT   PCT_USED DELETED_PCT
----------- ---------- ---------- -----------
     880032          2         89           0
```

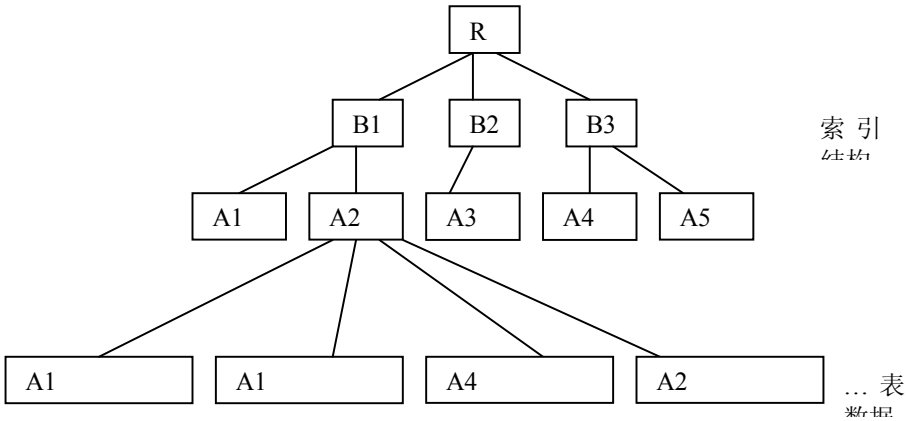
- 如果使用的索引的聚簇因子（Clustering Factor）很大，说明一条索引记录指向多个数据块，在返回结果时需要读取更多的数据块。通过重建表可以降低聚簇因子，因而可以在查找索引时减少表数据块的访问块数。

聚簇因子说明了表数据的物理存储位置相对于一个索引的排序性的符合程度。例如，一个非唯一索引是建立在 A 字段上的，如果表数据的存储是以 A 字段的顺序存储的，则索引与数据的关系如下图：



此时，索引的聚簇因子很低，从图上看到，假如我们需要获取 A=A2 的数据，只需要读取一个数据块就可以了；

相反，如果表数据物理存储顺序和索引顺序相差很大，就会出现下面的情况：



这时该索引的聚簇因子就很大，可以看到，如果需要获取 A=A2 的数据，我们需要读取 4 块或更多的数据块。

对索引进行分析后，我们可以从视图 DBA_INDEXES 中获取到索引的聚簇因子，字段名为 Clustoring_Factor。如果一个索引是一张表主要被使用的索引（或者是该表的唯一索引），且它的聚簇因子过高导致 IO 请求过高的话，我们可以考虑采取以下措施来降低 IO：

1) 以索引字段的顺序重建表以降低聚簇因子，可以用以下语句重建表（当然，你还需要重建触发器、索引等对象，还可能需要重建、重新编译有关联对象）：

```
CREATE new_table AS SELECT * FROM old_table ORDER BY A;
```

2) 建立基于索引字段 IOT（索引表）。

如果该索引不是表的主要索引，只是被少量语句引用到，按照以上方式处理的话反而可能会使其他使用更加频繁的索引的聚簇因子增大，导致系统性能更差。这时我们可以建立包含返回字段的索引，以避免“TABLE ACCESS BY INDEX ROWID”。如以下例子：

```
SQL> set autot trace
SQL> select status from t_test1
  2  where owner = 'DEMO';

576 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 4014220762

--------------------------------------------------------------------------------
| Id  | Operation                   | Name         | Rows  | Bytes | Cost (%CPU)
| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |              |   576 |  6336 |    11   (0)
```

```
| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T_TEST1      |   576 |  6336 |    11   (0)
| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | T_TEST1_IDX1 |   576 |       |     1   (0)
| 00:00:01 |
---------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("OWNER"='DEMO')


Statistics
----------------------------------------------------------
        465  recursive calls
          0  db block gets
        222  consistent gets
         43  physical reads
          0  redo size
       8368  bytes sent via SQL*Net to client
        803  bytes received via SQL*Net from client
         40  SQL*Net roundtrips to/from client
          8  sorts (memory)
          0  sorts (disk)
        576  rows processed

SQL> create index t_test1_idx3 on t_test1(owner, status) compute statistics;

Index created.

SQL> select status from t_test1
  2  where owner = 'DEMO';

576 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 2736516725

-------------------------------------------------------------------------------------
| Id  | Operation        | Name         | Rows  | Bytes | Cost (%CPU)| Time|
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT |              |   576 |  6336 |     2   (0)| 00:00:01|
|*  1 |  INDEX RANGE SCAN| T_TEST1_IDX3 |   576 |  6336 |     2   (0)| 00:00:01|
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("OWNER"='DEMO')


Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
         43  consistent gets
          3  physical reads
          0  redo size
       8152  bytes sent via SQL*Net to client
        803  bytes received via SQL*Net from client
```

```
         40  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
        576  rows processed
```

- 通过分区裁剪（partition pruning）技术来减少的 SQL 对数据块的访问。

采用分区裁剪技术，Oracle 优化器会先分析 FROM 和 WHERE 字句，在建立访问分区列表时将那些不会被访问到的分区排除。例如，我们的表 T_TEST1 的 owner 字段的值有“SYS、SYSTEM、XDB、DEMO、TEST”，如果我们按照 owner 字段建立的是分区表：

```
CREATE TABLE t_test1
(object_id    NUMBER(5),
 object_name  VARCHAR2(30),
 owner        VARCHAR2(20),
 created      DATE)
PARTITION BY LIST(owner)
(
PARTITION owner_sys VALUES('SYS', 'SYSTEM'),
PARTITION owner_xdb VALUES ('XDB'),
PARTITION owner_demo VALUES('DEMO'),
PARTITION owner_test VALUES('TEST'),
PARTITION owner_others VALUES(DEFAULT)
);
```

则对于以下语句：

```
select object_name
from t_test1
where owner in ('DEMO', 'TEST')
and created > sysdate - 30;
```

优化器会先将分区 owner_sys、owner_xdb、owner_others 从分区访问列表中裁剪出去，只访问分区 owner_demo 和 owner_test 上的数据或者通过这两个分区上的索引来访问数据。

3.2.1.2 处理非 SQL 导致的 IO 问题

如果从 statspack 或者 AWR 报告中找不到明显产生 db file sequential read 事件的 SQL，则该等待事件可能是由于以下原因导致的：

- 热点数据文件或磁盘

数据文件所在的磁盘 IO 负荷过重导致对 IO 请求反映慢，这时，我们可以通过 statspack 或 AWR 报告中的“File I/O Statistics”部分（或者通过 V$FILESTAT 视图）来找到热点磁盘：

Statspack report：

```
Tablespace               Filename
------------------------ ----------------------------------------------------
                  Av      Av     Av                     Av        Buffer Av Buf
         Reads Reads/s Rd(ms) Blks/Rd       Writes Writes/s      Waits Wt(ms)
-------------- ------- ------ ------- ------------ -------- ---------- ------
```

```
AFW_DATA                  /export/home/icssprd/data/data17/icssprd_afw_data_01
           726       0     4.3     1.0          381        0          0

AFW_INDX                  /export/home/icssprd/data/data18/icssprd_afw_indx_01
         1,741       0     6.3     1.0        2,104        0          0

CSS_AN_DATA               /export/home/icssprd/data/data03/icssprd_css_an_data
       200,649       5     1.8     3.2       24,192        1          0
                          /export/home/icssprd/data/data04/icssprd_css_an_data
       242,462       6     1.6     3.1       26,985        1          3    6.7

CSS_AN_INDX               /export/home/icssprd/data/data13/icssprd_css_an_indx
        70,789       2     5.0     1.6        5,330        0          0

CSS_AUDIT_RESOURCES_DATA /export/home/icssprd/data/data10/icssprd_css_audit_r
         2,394       0     0.6     1.0        1,781        0          0

CSS_AUDIT_RESOURCES_INDX /export/home/icssprd/data/data11/icssprd_css_audit_r
           248       0     4.3     1.0           52        0          0

... ...
```

视图：

```
SQL> select b.name, phyrds, phywrts
  2  from V$FILESTAT a, V$DATAFILE b
  3  where a.file# = b.file#;

NAME
--------------------------------------------------------------------------------
    PHYRDS    PHYWRTS
---------- ----------
C:\ORACLE\PRODUCT\10.2.0\ORADATA\EDGAR\DATAFILE\O1_MF_SYSTEM_20TFOB4Q_.DBF
    132767      11565

C:\ORACLE\PRODUCT\10.2.0\ORADATA\EDGAR\DATAFILE\O1_MF_UNDOTBS1_20TFQP78_.DBF
      1943      19924

C:\ORACLE\PRODUCT\10.2.0\ORADATA\EDGAR\DATAFILE\O1_MF_SYSAUX_20TFSGC6_.DBF
    659458     100811

... ...
```

找到热点数据文件（磁盘）后，我们可以考虑将数据文件转移到性能更高的存储设备上去，或者利用我们上述说的条带化、RAID 等存储技术来均衡 IO 负荷。

- 热点数据段

从 Oracle9.2 开始，出现了数据段的概念。每个表和索引都存储在自己的数据段中。我们可以通过视图 V$SEGMENT_STATISTICS 查找物理读最多的段来找到热点数据段。通过对热点段的分析，考虑采用重建索引、分区表等方式来降低该数据段上的 IO 负荷。

```
SQL> select owner, object_name, tablespace_name, object_type, value
  2  from V$SEGMENT_STATISTICS
  3  where statistic_name = 'physical reads'
  4  order by value desc;

OWNER                          OBJECT_NAME
------------------------------ ------------------------------
```

```
TABLESPACE_NAME                OBJECT_TYPE             VALUE
------------------------------ ------------------ ----------
SYS                            CONTEXT$
SYSTEM                         TABLE                      71

SYS                            I_CONTEXT
SYSTEM                         INDEX                      70

... ...
```

另外，我们还可以根据视图 v$session_wait 中的 P1（热点段所在的数据文件号）、P2（发生 db file sequential read 事件的起始数据块）、P3（数据块的数量，db file sequential read 读取数据块数量为 1）来定位出热点段：

先找出文件号、起始数据块、数据块数量：

```
SQL> select p1 "fileid", p2 "block_id", p3 "block_num"
  2  from v$session_wait
  3  where event = 'db file sequential read';

    fileid   block_id  block_num
---------- ---------- ----------
       396      44869          1
```

然后根据找出的文件号、起始数据块、数据块数量来定位出数据段：

```
SQL> select
  2     segment_name     "Segment Name",
  3     segment_type     "Segment Type",
  4     block_id         "First Block of Segment",
  5     block_id+blocks  "Last Block of Segment"
  6  from dba_extents
  7  where &fileid = file_id
  8  and &block_id >= block_id
  9  and &block_id <= block_id+blocks;
Enter value for fileid: 396
old   7: where &fileid = file_id
new   7: where 396 = file_id
Enter value for block_id: 44869
old   8: and &block_id >= block_id
new   8: and 44869 >= block_id
Enter value for block_id: 44869
old   9: and &block_id <= block_id+blocks
new   9: and 44869 <= block_id+blocks

Segment Name
--------------------------------------------------------------------------------
Segment Type       First Block of Segment Last Block of Segment
------------------ ---------------------- ---------------------
CSS_TP_SHMT_QUEUE_ACTIVITY
TABLE                               44841                 44873
```

3.2.1.3 调整 Buffer Cache

如果系统中即不存在性能有问题的 SQL 语句，而且所有磁盘的 IO 负载也比较均衡（不存在热地磁盘），则我们需要考虑增加 Buffer Cache 来降低磁盘 IO 请求。

在 8i，主要是根据缓存命中率（Buffer Cache Hit Ratio）来调整 buffer cache。当 Buffer Cache 调整到一定大小，对命中率没什么影响了时，就没有必要在增大 Buffer Cache 了。可以通过以下语句来查看 Buffer Cache 命中率：

```
SQL> select 1-(physical_reads)/(consistent_gets+db_block_gets)
  2  from v$buffer_pool_statistics;

1-(PHYSICAL_READS)/(CONSISTENT_GETS+DB_BLOCK_GETS)
--------------------------------------------------
                                        .95628981
```

在 9i 中，可以利用 statspack report 中的 Buffer Cache 建议部分来调整 Buffer Cache 的大小。

```
Buffer Pool Advisory for DB: ICSSPRD  Instance: icssprd  End Snap: 259
-> Only rows with estimated physical reads >0 are displayed
-> ordered by Block Size, Buffers For Estimate

        Size for  Size      Buffers for  Est Physical          Estimated
P   Estimate (M) Factr         Estimate   Read Factor     Physical Reads
--- ------------ ----- ---------------- ------------- ------------------
D            304    .1           37,715          9.18      5,928,235,496
D            608    .2           75,430          6.88      4,443,709,043
D            912    .3          113,145          5.73      3,699,496,220
D          1,216    .4          150,860          3.87      2,502,670,372
D          1,520    .5          188,575          2.32      1,499,049,228
D          1,824    .6          226,290          1.70      1,099,326,418
D          2,128    .7          264,005          1.41        912,042,579
D          2,432    .8          301,720          1.22        790,925,174
D          2,736    .9          339,435          1.09        703,357,378
D          2,992   1.0          371,195          1.00        645,905,997
D          3,040   1.0          377,150          0.99        636,992,420
D          3,344   1.1          414,865          0.90        583,996,250
D          3,648   1.2          452,580          0.84        542,063,246
D          3,952   1.3          490,295          0.79        508,261,496
D          4,256   1.4          528,010          0.74        480,472,150
D          4,560   1.5          565,725          0.71        455,533,563
D          4,864   1.6          603,440          0.67        434,743,759
D          5,168   1.7          641,155          0.64        416,285,837
D          5,472   1.8          678,870          0.62        400,208,242
D          5,776   1.9          716,585          0.60        385,785,401
D          6,080   2.0          754,300          0.57        365,597,932
          -------------------------------------------------------------
```

这里，Est Physical Read Factor 是估算的从磁盘物理读取次数与从 buffer cache 中读取的次数的比值。从意见估算的图表中，当 Buffer Cache 的增长对该因子影响不大时，则说明无需在增大 Buffer Cache，我们就可以去相应临界点的大小作为 Buffer Cache 的大小。上述例子中，我们可以考虑设置 Buffer Cache 大小为 2992M。

在 Oracle10g 中，引入了新的内存管理特性——自动共享内存管理（Automatic Shared Memory Management ASMM）。基于这一特性，oracle 能够自动根据当前的负荷计算出最优的 Buffer Cache 大小。关于 ASMM，可以参见文章《Oracle 内存全面分析》的 SGA_TARGET 部分。

我们可以采用多尺寸缓冲池技术将热点数据段（表或索引）KEEP 在缓冲池中：

```
SQL> alter table t_test1 storage(buffer_pool keep);

Table altered.
```

关于多尺寸缓冲的更多内容，可以参考文章《Oracle 内存全面分析》的“多缓冲池部分”部分。

3.2.1.4 Housekeep 历史数据

对于一些被频繁访问到的大表，我们需要定期对其做 housekeep，将一些不用的、老的数据从表中移除，以减少访问的数据块。定期对含有时间轴的 Transaction 表做 housekeep 是降低 IO 负载的重要措施。

3.2.2 db file scattered read

这是另外一个常见的引起数据库 IO 性能问题的等待事件。它通常发生在 Oracle 将“多数据块”读取到 Buffer Cache 中的非连续（分散的 Scattered）区域。多数据块读就是我们上述所说的一次读取“DB_FILE_MULTIBLOCK_READ_COUNT”块数据块，前面提到，它通常发生在全表扫描（Full Table Scan）和快速全索引扫描（Fast Full Index Scan）时。当发现 db file scattered read 等待事件是系统引起 IO 性能的主要原因时，我们可以采取以下措施对系统进行优化。

3.2.2.1 优化存在 Full Table Scan 和 Fast Full Index Scan 的 SQL 语句

我们可以首先从 statspack 或者 awr 报告中的“SQL ordered by Reads”部分中找出存在 Full Table Scan 和 Fast Full Index Scan 的 Top SQL。因为这些 Top SQL 往往是整个系统的瓶颈。

从 9i 开始，我们还可以通过视图 V$SQL_PLAN 来查找系统中存在 Full Table Scan 和 Fast Full Index Scan 的 SQL 语句。查找 Full Table Scan 的语句：

```
select sql_text from v$sqlarea t, v$sql_plan p
 where t.hash_value=p.hash_value and p.operation='TABLE ACCESS'
   and p.options='FULL';
```

查找 Fast Full Index Scan 的语句

```
select sql_text from v$sqlarea t, v$sql_plan p
 where t.hash_value=p.hash_value and p.operation='INDEX'
   and p.options='FULL SCAN';
```

Full Table Scan 通常是由于以下几个原因引起的：

- 条件字段上没有索引；

在这种情况下，如果表的数据量比较大，我们就需要在相应字段上建立起索引。

- CBO 中，对象的统计数据不正确

CBO 中，如果对象的统计数据或者其柱状图（Histogram）信息不正确，会导致优化器计算出错误的查询计划，从而选择全表扫描。这种情况下，我们要做的就重新分析（Analyze）表、索引及字段。

- CBO 中，SQL 语句中引用到了无法估算统计数据的对象

在 PLSQL 中，可以建立一些高级的数据类型，如“TABLE OF”、ARRAY 等，通过 TABLE、CAST 函数可以在 SQL 语句中将这些对象当成表来处理。而这些对象的数据只存在于调用 PLSQL 的会话中，因此他们没有相应的统计数据，Oracle 会为他们生产一些假的统计数据以完成查询计划代价估算。但是基于这些假的数据计算出的查询计划一般是错误的。我们可以考虑通过提示来强制 SQL 使用索引或者强制 SQL 采用 RBO 优化器。

此外，如果 SQL 中引用到了临时表（Temporary Table）也会产生同样的问题。其原因和解决方法和上面相同。

- 优化器认为索引扫描代价过高；

在 Oracle 中存在一个参数 optimizer_index_cost_adj，该参数的值代表一个百分数，如果对索引扫描的代价达到或超过全表扫描的代价的这个百分比值时，优化器就采用全表扫描。

optimizer_index_cost_adj 是一个全局性的参数，它的合理值是通过长期调整出来的。一般来说是一个介于 1 到 100 之间的数字。我们可以按照以下方法来选取 optimizer_index_cost_adj 的合理值。

先由以下语句得出 optimizer_index_cost_adj 的一个初始值：

```
SQL> select
  2     a.average_wait                                    "Average Waits FTS"
  3     ,b.average_wait                                   "Average Waits Index Read"
  4     ,a.total_waits /(a.total_waits + b.total_waits)   "Percent of FTS"
  5     ,b.total_waits /(a.total_waits + b.total_waits)   "Percent of Index Scans"
  6     ,(b.average_wait / a.average_wait)*100            "optimizer_index_cost_adj"
  7  from
  8     v$system_event  a,
  9     v$system_event  b
 10  where a.EVENT = 'db file sequential read'
 11    and b.EVENT = 'db file scattered read';

Average Waits FTS Average Waits Index Read Percent of FTS Percent of Index Scans
----------------- ----------------------- -------------- ----------------------
optimizer_index_cost_adj
------------------------
             1.25                    1.06     .041867874             .958132126
                    84.8
```

这里，84.8 是我们系统的初始值。在系统经过一段时间运行后，再次运行上面的语句，重新调整 optimizer_index_cost_adj 的值。经过多次如此反复的调整之后，最终上面语句得出值趋于稳定，这时这个值就是符合我们系统性能需求的最合理的值。

当然这个数值也可以通过 statspack 的历史数据来调整，在 9i 中：

```
select to_char(c.end_interval_time, 'MM/DD/YYYY') "Date",
```

```
       sum(a.time_waited_micro)/sum(a.total_waits)/10000 "Average Waits FTS",

       sum(b.time_waited_micro)/sum(b.total_waits)/10000 "Average Waits Index
Read",

       (sum(a.total_waits) / sum(a.total_waits + b.total_waits)) * 100 "Percent
of FTS",

       (sum(b.total_waits) / sum(a.total_waits + b.total_waits)) * 100 "Percent
of Index Scans",

       (sum(b.time_waited_micro)/sum(b.total_waits)) /

       (sum(a.time_waited_micro)/sum(a.total_waits)) * 100
"optimizer_index_cost_adj"

from dba_hist_system_event a, dba_hist_system_event b, dba_hist_snapshot c

where a.event_name = 'db file scattered read'

and   b.event_name = 'db file sequential read'

and   a.snap_id = c.snap_id

and   b.snap_id = c.snap_id

group by c.end_interval_time

order by 1;
```

10g 中：

```
select to_char(c.snap_time, 'MM/DD/YYYY') "Date",

       sum(a.time_waited_micro)/sum(a.total_waits)/10000 "Average Waits FTS",
```

```
       sum(b.time_waited_micro)/sum(b.total_waits)/10000 "Average Waits Index
Read",

       (sum(a.total_waits) / sum(a.total_waits + b.total_waits)) * 100 "Percent
of FTS",

       (sum(b.total_waits) / sum(a.total_waits + b.total_waits)) * 100 "Percent
of Index Scans",

       (sum(b.time_waited_micro)/sum(b.total_waits)) /

       (sum(a.time_waited_micro)/sum(a.total_waits)) * 100
"optimizer_index_cost_adj"

from stats$system_event a, stats$system_event b, stats$snapshot c

where a.event = 'db file scattered read'

and   b.event = 'db file sequential read'

and   a.snap_id = c.snap_id

and   b.snap_id = c.snap_id

group by c.snap_time

order by 1;
```

当 optimizer_index_cost_adj 的值对于整个系统来说已经是比较合理的值，而某些语句由于该值选择了全表扫描扫描导致了 IO 性能问题时，我们可以考虑通过提示来强制语句命中索引。

- 建立在条件字段上的索引的选择性不高，结合上一条导致全表扫描；

当索引的选择性不高，且其代价过高，系统则会选择全表扫描来读取数据。这时我们可以考虑通过选择/建立选择性比较高的索引，使查询命中索引从而避免全表扫描。

```
SQL> create index t_test1_idx1 on t_test1(owner) compute statistics;

Index created.

SQL> set autot trace
SQL> select object_name
  2  from t_test1
  3  where owner = 'SYS'
  4  and created > sysdate - 30;

no rows selected


Execution Plan
----------------------------------------------------------
Plan hash value: 1883417357

--------------------------------------------------------------------------------
| Id  | Operation         | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |         |    49 |  1715 |   152   (2)| 00:00:02 |
|*  1 |  TABLE ACCESS FULL| T_TEST1 |    49 |  1715 |   152   (2)| 00:00:02 |
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("OWNER"='SYS' AND "CREATED">SYSDATE@!-30)

... ...

SQL> create index t_test1_idx2 on t_test1(owner, created) compute statistics;

Index created.

SQL> select object_name
  2  from t_test1
  3  where owner = 'SYS'
  4  and created > sysdate - 30;

no rows selected


Execution Plan
----------------------------------------------------------
Plan hash value: 3417015015

-------------------------------------------------------------------------------------
| Id  | Operation                   | Name         | Rows  | Bytes | Cost (%CPU)
| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |              |    49 |  1715 |     2   (0)
| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T_TEST1      |    49 |  1715 |     2   (0)
| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | T_TEST1_IDX2 |    49 |       |     1   (0)
| 00:00:01 |
-------------------------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("OWNER"='SYS' AND "CREATED">SYSDATE@!-30)

... ...
```

3.2.2.2 调整 DB_FILE_MULTIBLOCK_READ_COUNT

当 SQL 已经没有优化余地后，问题仍没有解决，我们可以考虑调整 DB_FILE_MULTIBLOCK_READ_COUNT 大小。其作用我们在 3.1.2 中有做叙述，这里不再赘述。不过要注意一点就是，DB_FILE_MULTIBLOCK_READ_COUNT * DB_BLOCK_SIZE 是一次 IO 读取的传输量，它不能大于系统的 max_io_size 大小。

从 Oracle 10gR2 开始，如果没有设置 DB_FILE_MULTIBLOCK_READ_COUNT 的大小，Oracle 会自动为其调整一个默认值，这个默认值的大小与平台最大 IO 大小（max_io_size）相关（对大多数平台来说 max_io_size 是 1M），其大小被设置为（max_io_size / DB_BLOCK_SIZE）。

3.2.2.3 将频繁访问的全扫描的表 CACHE 住

由于通过 Full Table Scan 和 Fast Full Index Scan 读取的数据块会被放置到 Buffer Cache 的 LRU 链表的 LRU 端，从而使数据块尽快从 Buffer Cache 中移出。因此，对于那些会被频繁访问到全扫描的表，且其数据量不大的情况下，我们可以考虑将它们 CACHE 住。

```
SQL> alter table t_test1 cache;

Table altered.
```

对于 Fast Full Index Scan 的索引对象，则可以考虑把它放置在 KEEP 池中。

```
SQL> alter index t_test1_idx1 storage(buffer_pool keep);

Index altered.
```

利用 V$SESSION_EVENT 视图，我们同样可以找到当前系统中发生全扫描的对象。

```
SQL> select p1 "fileid", p2 "block_id", p3 "block_num"
  2  from v$session_wait
  3  where event = 'db file scattered read';

    fileid   block_id  block_num
---------- ---------- ----------
       359     152972         16

SQL> select
  2     segment_name     "Segment Name",
  3     segment_type     "Segment Type",
  4     block_id         "First Block of Segment",
  5     block_id+blocks  "Last Block of Segment"
  6  from dba_extents
  7  where &fileid = file_id
```

```
  8  and &block_id >= block_id
  9  and &block_id <= block_id+blocks;
Enter value for fileid: 359
old   7: where &fileid = file_id
new   7: where 359 = file_id
Enter value for block_id: 152972
old   8: and &block_id >= block_id
new   8: and 152972 >= block_id
Enter value for block_id: 152972
old   9: and &block_id <= block_id+blocks
new   9: and 152972 <= block_id+blocks

Segment Name
--------------------------------------------------------------------------------
Segment Type       First Block of Segment Last Block of Segment
------------------ ---------------------- --------------------
CSS_TP_SHMT_QUEUE
TABLE                             152969               153001
```

3.2.2.4 利用分区表减少全扫描操作读取的数据块数量

前面我们有介绍分区裁剪（Partition Pruning）技术。将表分区，利用分区裁剪技术，在进行全扫描时只会扫描在 WHERE 条件中出现的分区，从而可以减少全扫描所读取到的数据块数量。

3.2.2.5 Housekeep 历史数据

同样，housekeep 不需要的、历史的数据，减少数据段中的数据块数量，也能减少全扫描的 IO 请求次数。

3.2.3 db file parallel read

首先，不要被该事件名称所误导——它和并行 DML 或者并行查询都无关。当从多个数据文件并行读取数据到非联系的内存（PGA、Buffer Cache）缓冲中时，会发生该等待事件。它通常发生在 Recovery 操作或者利用缓冲预提取（Buffer Prefetching）从数据文件并行读取数据时。

我们可以通过以下语句找出发生 db file parallel read 等待事件的数据文件和数据块：

```
select p1 "fileid", p2 "block_id", p3 "requests"
  from v$session_wait
 where event = 'db file parallel read';
```

优化该等待事件的手段可以参考优化 db file sequential read 等待事件中非 SQL 优化方法部分。

3.2.4 direct path read & direct path read (lob)

当直接读取（Direct Read）数据到 PGA（而不是到 Buffer Cache）中去时，会发生 Direct Path Read 等待事件。对 Lob 数据的直接读有一个单独的等待事件——direct path read (lob)。

当 Oracle 设置支持异步 IO 时，进程可以在提交 IO 请求后继续做其他操作，并且在稍后再提取 IO 请求返回的结果，在提取结果时就产生了 direct path read 等待事件。

在没有启用异步 IO 时，IO 请求在完成之前会被阻塞，但在执行 IO 操作时并不会产生等待事件。进程稍后回来提取那些已经读取到的 IO 数据，这时尽管能够很快返回，但仍然会显示 direct path read 等待事件。

和其他 IO 等待事件不同的是，对 Direct Path Read 等待事件要注意以下两点：

- 等待次数并不等于 IO 请求次数；
- 统计（如 statspack 报告中）得出的 Direct Path Read 的等待时间并不一定代表该事件引起的真正等待时间。

事件中的 P1、P2、P3 参数分别代表：
P1：发生等待事件的数据块所在文件号；
P2：发生等待事件的数据块号；
P3：等待事件涉及的连续数据块数量。

直接读（Direct Read）请求一般发生在以下几种情况：

- 磁盘排序 IO（Sort Area 不足时，排序用到的临时数据会被写到临时表空间上去，当读取这些数据时就使用直接读）；
- 并行查询；
- 预读取（当一个进程认为某个数据块将很快被用到而发出 IO 请求时）
- Hash Join（Hash Area 不足）
- IO 负载系统中，服务进程处理缓存的速度比系统 IO 返回数据到缓存的速度更快时

通过视图 V$SESSION_EVENT 我们可以找出当前产生等待的会话，再根据会话中正在进行的操作确定导致等待的原因。针对不同的原因，我们可以采取不同的措施减少 Direct Path Read 等待事件。

3.2.4.1 磁盘排序

首先我们可以考虑优化语句以减少排序操作。排序一般是由以下操作引起的：

- Order By;
- JOIN;
- UNION;
- Group By;
- 聚合操作；
- Select unique;
- Select distinct;

可以尝试在语句中减少没必要的上述操作来避免排序操作。另外，创建索引也会引起排序操作。在专业模式（Dedicated）下，排序所占用的内存是从 PGA 中分配出来的一块区域，叫 Sort Area，由参数 sort_area_size 控制其大小；在 MTS 中，排序区是从 Large Pool 中分配的。当 sort area 大小无法满足排序操作要求时，就会占用临时表空间来存放排序数据，因而产生 Direct Path Read 等待事件。我们可以通过适当增加该参数来减少磁盘排序操作。

这个参数可以在系统范围或会话范围进行修改。对于一些需要做大量排序操作而且又比较独立的会话（如 Create Index），我们可以在会话级别为其设置比较大的 Sort Area 以满足排序需要：

```
SQL> alter session set sort_area_size = 10000000;

Session altered.
```

该参数大小一般推荐设置为 1~3M。在 9i 之后，不推荐设置该参数，我们可以通过设置 PGA_AGGREGATE_TARGET 进行 PGA 内存自动管理（设置 WORKAREA_SIZE_POLICY 为 TRUE）。对于 PGA_AGGREGATE_TARGET 的大小设置，可以参考文章《Oracle 内存全面分析》中的 PGA_AGGREGATE_TARGET 部分。

此外，我们还可以通过以下语句来查找系统中存在磁盘排序的会话及其语句：

```
SELECT a.sid,a.value, b.name, d.sql_text from

V$SESSTAT a, V$STATNAME b, V$SESSION c, V$SQLAREA d

WHERE a.statistic#=b.statistic#

AND b.name = 'sorts (disk)'

and a.sid = c.sid

and c.SQL_ADDRESS = d.ADDRESS(+)

and c.SQL_HASH_VALUE = d.HASH_VALUE(+)

and value > 0

ORDER BY 2 desc,1;
```

3.2.4.2 并行查询

当设置表的并行度非常高时，优化器可能就对表进行并行全表扫描，这时会引起 Direct Path Read 等待。

在使用并行查询前需要慎重考虑，因为并行查询尽管能教师程序的响应时间，但是会消耗比较多的资源。对于低配置的数据库服务器不建议使用并行特性。此外，需要确认并行度的设置要与 IO 系统的配置相符（建议并行度为 2~4 * CPU 数）。在 10g 中，可以考虑使用 ASM。

对于表的并行度，我们不建议直接用 ALERT 修改表的物理并行度：

```
ALTER TABLE t_test1 PARALLEL DEGREE 16;
```

而是推荐针对特定语句使用提示来设置表的并行度：

```
SQL> SELECT /*+ FULL(T) PARALLEL(T, 4)*/ object_name FROM t_test1 t;

47582 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 2467664162

-------------------------------------------------------------------------------------
| Id  | Operation            | Name     | Rows  | Bytes | Cost (%CPU)| Time
|    TQ  |IN-OUT| PQ Distrib |
-------------------------------------------------------------------------------------_____
|   0 | SELECT STATEMENT     |          | 47582 |  1068K|    42   (3)| 00:00:01
|        |      |            |
|   1 |  PX COORDINATOR      |          |       |       |            |
|        |      |            |
|   2 |   PX SEND QC (RANDOM)| :TQ10000 | 47582 |  1068K|    42   (3)| 00:00:01
|  Q1,00 | P->S | QC (RAND)  |
|   3 |    PX BLOCK ITERATOR |          | 47582 |  1068K|    42   (3)| 00:00:01
|  Q1,00 | PCWC |            |
|   4 |     TABLE ACCESS FULL| T_TEST1  | 47582 |  1068K|    42   (3)| 00:00:01
|  Q1,00 | PCWP |            |
-------------------------------------------------------------------------------------
```

3.2.4.3 Hash Join

Hash Area 是用于 hash join 的内存区域。Hash Area 过小会引起 Direct Path Read 等待。当 WORKAREA_SIZE_POLICY 为 FALSE 时，可以考虑增加 hash_area_size 的大小（建议为 sort_area_size 大小的 1.5 倍）；当 WORKAREA_SIZE_POLICY 为 TRUE 时，可以考虑增加 PGA_AGGREGATE_TARGET 大小。

3.2.4.4 Direct path read (lob)

为了减少 LOB 的读写时间，通常我们会设置 LOB 的存储参数 NOCACHE，这时读取 LOB 时会引起 Direct Path Read (lob)等待事件。但当我们发现 Direct path read (lob) 引起了 IO 性能问题，就需要考虑将那些被经常读取的 LOB 字段设置为 CACHE。另外，如果操作系统的文件系统有足够的 Buffer Cache 时可以考虑将 LOB 数据段存储在文件系统上。

3.2.4.5 其他优化措施

当内存资源不足、IO 读取数据到内存效率远远低于内存中数据被处理的效率时，会引起 Direct Path Read 等待事件。作为对上述处理措施的补充，增加内存（PGA）、在确保操作系统支持 AIO 情况下设置 DISK_ASYNCH_IO 为 TRUE 以支持异步 IO、采用效率更高的存储设备都能帮助我们减少 Direct Path Read 等待。

3.2.5 direct path write & direct path write (lob)

直接写（Direct Path Write）允许一个会话先将 IO 写请求放入一个队列中，让操作系统去处理 IO，而自身可以继续处理其他操作。当会话需要知道写操作是否完成（如会话需要一块空闲的缓存块或者会话需要确认内存中所有写操作都被 flush 到磁盘了），会话就会等待写操作完成从而产生 Direct Path Write 等待事件。Direct Path Write (lob) 是在对 LOB 数据段（NOCACHE）直接写时产生的等待事件。

在没有启用异步 IO 时，IO 写请求在完成之前会被阻塞，但在执行 IO 写操作时并不会产生等待事件。进程稍后回来提取那些已经完成的 IO 操作数据，这时尽管能够很快返回，但仍然会显示 direct path write 等待事件。

和 Direct Path Read 等待事件相似，对 Direct Path Write 等待事件也要注意以下两点：

- 等待次数并不等于 IO 请求次数；
- 统计（如 statspack 报告中）得出的 Direct Path Write 的等待时间并不一定代表该事件引起的真正等待时间。

事件中的 P1、P2、P3 参数分别代表：
P1：发生等待事件的数据块所在文件号；
P2：发生等待事件的数据块号；
P3：等待事件涉及的连续数据块数量。

直接写请求一般发生在以下几种情况：

- 直接数据载入操作（如 CTAS、SQL*Loader 设置 Direct 选项等）；
- 并行 DML 操作；
- 磁盘排序（排序内存空间不足，数据写入磁盘）；
- 载入 NOCACHE 数据段；

对 Direct Path Write 的优化处理措施基本上和 Direct Path Write 类似。

3.3 控制文件相关的 IO 事件

这一类等待事件发生在对控制的 IO 操作时。对控制文件的 IO 访问一般都是由 Redo Log 文件切换、Checkpoint 等（如更新 SCN）引起的。因此，对这类事件的优化处理也就主要是对这些操作的调优处理。

3.3.1 control file parallel write

这一等待事件通常发生在一个服务进程在更新所有控制文件时，通常是以下情况：

- 会话启动了一个控制文件事务（在提交事务之前更新所有控制文件为最新）；
- 会话提交了一个事务到控制文件；
- 一个控制文件的条目被修改了，该修改要更新到所有控制文件上去

如果这一事件明显影响到了系统的 IO 性能时，可以考虑用以下手段来进行优化：

- 在保证控制文件的备份数量足够安全（不会出现控制文件全部丢失）的情况下使控制文件数量最少；
- 如果操作系统支持 AIO，设置数据库支持 AIO；
- 将控制文件转移到 IO 负载比较低的磁盘上去。

3.3.2 control file sequential read

这一等待事件通常发生在对一个单独的控制的 IO 读操作时。通常可能是以下情况：

- 备份一个控制文件；
- RAC 中在实例之间共享一个控制文件信息时；
- 读取控制文件的头数据块或者其他数据块时。

用以下语句可以找到是访问哪个控制导致的该等待事件：

```
select P1 as FileName from V$SESSION_WAIT


where EVENT = 'control file sequetial read' and STATE='WAITING';
```

我们可以采取以下手段来降低这一等待：

- 如果操作系统支持 AIO，设置数据库支持 AIO；
- 将控制文件转移到 IO 负载比较低的磁盘上去。

3.3.3 control file single write

这一等待事件通常发生在对一个单独的控制的 IO 写操作时。用以下语句可以找到是访问哪个控制导致的该等待事件：

```
select P1 as FileName from V$SESSION_WAIT


where EVENT = 'control file single write' and STATE='WAITING';
```

我们可以采取以下手段来降低这一等待：

- 如果操作系统支持 AIO，设置数据库支持 AIO；
- 将控制文件转移到 IO 负载比较低的磁盘上去。

3.4 Redo Log 相关的 IO 事件

在写 Redo Log 时，会发生很多等待事件，大部分和 IO 相关。其中最重要的要属“log file parallel write”和“log file sync”。Oracle 的后台 LGWR 进程会等待“log file parallel write”事件而前台进程会等待“log file sync”事件。

3.4.1 log file parallel write

这一等待事件发生在 LGWR 进程等待完成将 Redo 记录写入 Redo Log 文件时。在 LGWR 进程将 Log Buffer 中的数据写入 Log File 时会发生该事件。

当使用了异步 IO 时，这种写操作是并行的，否则只会一个接着一个 Redo 文件的写入。LGWR 进程必须等待所有的 Redo Log 文件都被写入。因而 Redo Log 文件所在磁盘的 IO 效率就直接影响了该等待事件的总的等待时间。

事件中的 P1、P2、P3 参数分别代表：
P1：有多少个 Redo Log 文件在被写入；
P2：有多少个数据块被写入；
P3：IO 请求的次数。

降低 log file parallel write 等待的方法有：

- 不要使表空间长期处于热备状态。当表空间处于热备状态时，表空间不再被更新，Redo Log 会急剧增加；
- 将 Redo Log 文件放在高速存储设备上，千万别放在 RAID5 上，可以考虑放在裸设备上；
- Redo log 文件所在的磁盘应尽量避免有其他 IO 操作的存在；
- 对某些操作，如大批量数据导入，可以设置 NOLOGGIN、UNRECOVERABLE 选项，或者在 SQL 语句中使用提示/*+APPEND*/，以减少 Redo Log 的产生。
- 在确保 Redo Log 数据足够安全（不会发生 Log 文件丢失）的情况下，尽量减少 Redo Log 组的成员数；
- 在配置需要使用到 Redo Log 的功能时，如 Streams 复制、LogMiner、逻辑模式的 DG，尽量设置为最低级别的补充日志（Supplemental Logging）；
- 适当增加 Log_buffer 的大小

我们可以按照以下方法来调整 Log_buffer 的大小，比较 Redo Log buffer 分配的重试（在请求 log buffer 时，无足够 buffer，需要重新提交请求）率，如果该比例大于 0.1%，我们就需要考虑增加 Log_buffer 的大小。

```
select retries.value/entries.value "Redo Log Buffer Retry Ratio"

from V$sysstat retries, V$sysstat entries

where retries.name =  'redo buffer allocation retries'

and entries.name =  'redo entries';
```

另外，如果系统统计数据中 redo log space requests 大于 0，说明有进程在等待分配 Redo Log 文件空间，而不是等待 Buffer 空间。这时我们也需要考虑增加 Log_buffer 的大小。

```
SELECT name, value

FROM v$sysstat

WHERE name = 'redo log space requests';
```

但是，Log_buffer 的大小不要超过 128K*CPU 或 512K（取两个数字中最大的一个）数。

3.4.2 log file sync

当 Oracle 前台进程提交或者回滚事务需要等待提交或回滚完成时会产生该等待事件。部分等待的原因可能是等待 LGWR 进程将会话事务的 Redo 记录从 Log Buffer 中拷贝到磁盘上去。这时，就会出现前台进程等待 Log File Sync，而后台 LGWR 进程在等待 Log File Parallel Write 的情况。

事件中的 P1、P2、P3 参数分别代表：
P1：发生等待时正在等待哪个 Log Buffer 数据被写入 Log 文件；
P2：无意义；
P3：无意义。

实际上，一个 Log File Sync 等待事件包含了多个步骤：

1、 如果 LGWR 空闲则唤醒 LGWR 进程；
2、 LGWR 收集需要写的 Redo 记录并提交 IO 请求；
3、 等待写 Log 的 IO 完成；
4、 LGWR IO 提交处理；
5、 LGWR 提交已经完成了写日志的前台/用户进程；
6、 前台/用户进程被唤醒。

如果配置了 Data Guard，上述步骤中的第三步还需要将 Redo 记录通过网络写入到 standby 数据库的 Redo Log 文件中去。
针对不同步骤的等待时间的不同，我们需要采取不同的优化措施：

- 第二、三步的相关等待数据可以从 statspack 或 awr 的“redo write time”统计项获得；
- 第三步的等待时间和 Log File Parallel Write 的等待时间相同；
- 当系统负载非常高时，第五、六两步的时间就会很长，因为此时尽管 LGWR 进程已经通知了前台/用户进程写日志已经完成，但是系统负载太高，前台/用户进程需要等待操作系统安排其运行计划。

要了解是什么阻滞了 Log File Sync 的关键是比较 Log File Sync 和 Log File Parallel Write 的平均等待时间：

- 如果他们的等待时间差不多，则说明是 Log 文件的 IO 问题（即第三步）导致的

Log File Sync 等待，我们就需要优化 Log 文件的 IO（如上一节所述的方法）；

- 如果 Log File Sync 的等待时间远远大于 Log File Parallel Write 的等待时间，则说明 Log File Sync 是由于在提交或回滚时的其他 Redo Log 机制（非 IO 原因）引起的，如 Latch Free、LGWR wait for copy 等 log buffer 相关的 latch 冲突。

可以用下面的语句来获取 Log File Sync 和 Log File Parallel Write 的平均等待时间的比值：

```
select (sum(decode(name, 'redo synch time', value)) / sum(decode(name, 'redo
synch writes', value)))


       / (sum(decode(name, 'redo write time', value)) / sum(decode(name, 'redo
writes', value)))


       as sync_cost_ratio


from v$sysstat


where name in ('redo synch writes', 'redo synch time', 'redo writes', 'redo write
time');
```

我们还可以采取以下调优手段来降低 Log File Sync 等待：

- 按照上一节中的方法减少 Redo Log 的产生、提供 Redo Log 的 IO 效率、减少 Redo Log 与其他 IO 的冲突；
- 将一些小事务合并成批量事务，以减少提交和回滚次数。

3.4.3 log file single write & log file sequential read

Log file single write 只会发生在打开或关闭一个 Redo Log 文件后，向文件头写入相关信息时。因为文件头的信息中包含了文件号，因此文件头信息不会并行写入多个文件，而是单独一个个写入，因而其等待时间不会被统计到 log file parallel write 之中。

事件中的 P1、P2、P3 参数分别代表：
P1：写入的 Redo Log 文件号；
P2：写入的数据块号；
P3：写入的数据块数。

当进程从 Redo Log 文件中读取 redo 记录时会产生 log file sequential read 等待事件，如 Arch 进程读取 Redo Log 数据。

事件中的 P1、P2、P3 参数分别代表：

P1：读取的 Redo Log 文件号；
P2：开始读取的数据块号；
P3：读取的数据块数。

当 Redo Log 文件存在 IO 问题时，以上两个等待事件通常都会和 log file parallel write 等待事件同时出现。因而可以通过在 3.4.1 中提到的提高 Redo Log 文件 IO 效率、减少 Log 文件 IO 冲突的方法来减少这两个等待。

3.4.4 log file switch completion & switch logfile command & log file switch (clearing log file)

当在产生 Redo Log 时需要等待 LGWR 切换 Log 文件时，会产生 log file switch completion 等待事件；而 switch logfile command 则是等待 DBA 手工执行的切换日志的命令：

```
SQL> alter system switch logfile;

System altered.
```

日志切换由以下步骤组成：

- 从控制文件获取下一日志文件的文件号；
- 获取 Redo Copy 和 Redo Allocation 的 Latch；
- 清空 Redo，将 buffer 中的 Redo 记录写入 Log 文件中去；
- 关闭当前 Redo Log 文件；
- 更新控制文件，包括：
    - 设置下一日志文件为当前日志文件；
    - 设置之前的日志文件为 INACTIVE；
    - 如果在 Archive 模式下，将之前文件加到归档文件列中；
    - 打开新的日志文件组中的所有文件；
    - 将 SCN 写入文件头；
    - 打开允许产生 Redo Log 的开关。

在等待上述的第三步时，则会产生 log file switch (clearing log file) 等待事件。

我们可以用以下语句查看当前的日志文件：

```
SQL> select GROUP#, ARCHIVED, STATUS from v$log;

    GROUP# ARC STATUS
---------- --- ----------------
         1 NO  INACTIVE
         3 NO  INACTIVE
         2 NO  CURRENT
```

我们可以采取前述方法提高 Redo Log 文件 IO 效率来降低这三个等待事件。此外，我们还可以增大 Log 文件大小，降低日志切换频率（一般来说，在系统运行高峰期以 20~30 分钟切换一次为佳）。通过以下语句可以查询日志的切换记录及其切换间隔时间：

```
SQL> SELECT to_char(b.first_time, 'YYYY-MM-DD HH24:MI:SS') as swtich_time,
  2         (b.first_time - a.first_time) * 24 as "switch_interval(hr)"
  3  FROM v$log_history a, v$log_history b
  4  WHERE a.SEQUENCE# + 1 = b.SEQUENCE#
  5  AND ROWNUM <= 10
  6  ORDER BY 1;

SWTICH_TIME         switch_interval(hr)
------------------- -------------------
2007-08-25 00:28:59              2.3975
2007-08-25 06:04:53          5.59833333
2007-08-25 12:15:52          6.18305556
2007-08-25 21:58:13          9.70583333
2007-08-25 23:50:39          1.87388889
2007-08-26 00:28:42          .634166667
2007-08-26 08:32:04          8.05611111
2007-08-26 17:58:05          9.43361111
2007-08-26 23:26:57          5.48111111
2007-08-27 07:21:35          7.91055556
... ...
```

另外，从 Alert 日志中，我们也可以找到日志切换记录。

```
... ...
Beginning log switch checkpoint up to RBA [0x18106.2.10], SCN: 0x0003.93b3fb7d
Thread 1 advanced to log sequence 98566
  Current log# 7 seq# 98566 mem# 0:
/export/home/icssprd/data/data02/icssprd_redo_07a.rdo
  Current log# 7 seq# 98566 mem# 1:
/export/home/icssprd/data/data18/icssprd_redo_07b.rdo
Mon May 28 12:35:14 2007
ARC0: Evaluating archive   log 2 thread 1 sequence 98565
ARC0: Beginning to archive log 2 thread 1 sequence 98565
Creating archive destination LOG_ARCHIVE_DEST_1:
'/export/home/icssprd/admin/arch/1_98565.dbf'
Mon May 28 12:36:47 2007
Completed checkpoint up to RBA [0x18106.2.10], SCN: 0x0003.93b3fb7d
Mon May 28 12:38:02 2007
ARC0: Completed archiving  log 2 thread 1 sequence 98565
Mon May 28 12:41:26 2007
Beginning log switch checkpoint up to RBA [0x18107.2.10], SCN: 0x0003.93b4a3a6
Thread 1 advanced to log sequence 98567
  Current log# 8 seq# 98567 mem# 0:
/export/home/icssprd/data/data10/icssprd_redo_08a.rdo
... ...
```

3.4.5 log file switch (checkpoint incomplete)

当在做日志切换时，同时会做 checkpoint，如果此时有其他 checkpoint 正在进行时，需要等待正在进行的 checkpoint 完成，此时就会产生 log file switch (checkpoint incomplete)等待事件。通常发生这一等待事件时，日志切换的时间都比平时更长。

要降低该等待事件，就需要降低日志切换时引起的 checkpoint 遇上系统中其他 checkpoint 的几率。我们可以通过以下方法来进行调优：

- 增加 Redo Log 文件的大小，使日志切换频率降低；
- 增大参数 Log_checkpoint_interval 大小，该参数设置系统两次 checkpoint 之间 Redo Log 数据块（该数据块的大小由操作系统的数据块大小决定）的数量。但

是 Oracle 会限制这些数据块总的大小要小于最小 log 文件的 90%。如最小 log 文件大小为 100M，操作系统的数据块大小为 512K，则 Log_checkpoint_interval 要小于(100 * 90% / 0.5) = 180

3.4.6 log switch/archive & log file switch (archiving needed)

log switch/archive 等待事件在会话等待所有 Archive 线程对当前 log 文件 Archive 操作完成。当 LGWR 进程切换日志时，如果要切入的日志还没有被 Archive，需要等待其被完成 Archive，这时会产生 log file switch (archiving needed)等待事件。这时，在 alert log 中我们还能发现以下信息：

```
Thread 1 cannot allocate new log, sequence 9556
All online logs needed archiving
```

这两个事件只有数据库在 Archive 模式下才会出现。说明 Archive 操作太慢。对这两个事件的调优主要是针对 Archive 设置的调优。

要提高 Archive 的效率，可以采取以下方法：

1) 调整 Redo Log 文件的个数和大小

大多数情况下，Redo Log 文件的个数越多、大小越大，就能让归档进程有更多时间做 Archive。如果 Redo 记录急剧增加，可以考虑加多 log 文件数量，这样能使归档进程有更多时间均衡归档过程。但是，如果 ARCH 进程无法跟上 LGWR 进程的处理速度时，增加 Log 文件数量就于事无补了。

2）调整 checkpoint 的间隔和效率

增大 checkpoint 的间隔也可以使归档进程有更多时间来处理归档。增大参数 log_checkpoint_interval 可以增大 checkpoint 的间隔。另外，增加 DBWR 进程数量、配置 AIO 都可以提高 checkpoint 的处理效率。

3）配置多 ARCH 进程

通过参数 log_archive_max_processes 可以配置最大 ARCH 进程数量。我们可以做一个脚本来执行'alter system archive log all;'命令，然后设置一个作业以固定间隔时间来执行该脚本。这个命令可以强制归档所有未归档的日志文件。这可以帮助均衡 ARCH 进程的归档处理负担。

4）调整 Archive 进程

增加 Archive 进程的 buffer 大小可以提高 Archive 效率，其大小是由参数 log_archive_buffer_size 控制的。该参数的初始设置为 4K，最大可以增加到 128K。但是，要注意，增加该参数虽然可以提供 Archive 效率，但是可能会使系统的整体性能下降；

另外，我们还可以通过增加 Archive 进程的 buffer 数量来提高 Archive 效率。Buffer 数量由参数 log_archive_buffer 控制，最大为 8.

5）减少系统的 IO 冲突、提供系统 IO 效率

系统整体的 IO 性能及冲突问题也会影响到 Archive 的效率。我们可以用前面介绍的方法来减少系统中存在的 IO 冲突、提高整体 IO 效率来提高 Archive 的效率。

3.5 Buffer Cache 相关的IO 事件

Buffer Cache 是影响 Oracle IO 的重要因素。这里要解决的几个等待事件都是涉及到 DBWR 进程和 IO 从属进程（Slave）的 Buffer Cache 操作引起的等待事件。

3.5.1 db file parallel write

该事件和并行 DML 无关。这个等待事件出现在当 DBWR 进程提交了多 IO 请求来并行将 Buffer Cache 中的脏数据写入磁盘中后，等待所有提交的 IO 请求完成。通常是由于操作系统的 IO 系统导致的该事件的阻滞。

事件中的 P1、P2、P3 参数分别代表：
P1：（9.2.0.5 之前）写入数据的文件号/（9.2.0.5 之后）请求次数；
P2：（9.2.0.5 之前）写入的数据块号/（9.2.0.5 之后）请求中断的次数；
P3：（9.2.0.5 之前）请求次数/（9.2.0.5 之后）请求发生了 Timeout 的时间

这一等待事件一般不会显著影响用户会话。但是当用户会话中有很高的“write complete waits”或“free buffer waits”事件的等待时间时，说明该事件已经影响到了用户会话。有时候这一事件对操作系统 IO 的影响也会影响到进程从同一磁盘读取数据的等待时间。

解决该事件的关键在于减少相关磁盘的 IO 冲突。如果这事件已经影响到用户会话，我们需要结合其他等待事件信息，考虑采取均衡热地磁盘负载、提高存储设备 IO 效率、增加 checkpoint 间隔、增大 Redo log 文件等方法来减低该事件。

3.5.2 db file single write

当 DBWR 进程请求修改数据文件头，在等待 IO 请求完成时，会出现 db file single write 等待事件。

事件中的 P1、P2、P3 参数分别代表：
P1：写入数据的文件号；
P2：写入的数据块号；
P3：写入的数据块数（一般为 1）。

解决这一等待事件的关键还是要处理好磁盘的 IO 冲突问题，特别是发生该事件所在的磁盘。通过相关 SQL 的调优等手段来降低事件发生的磁盘的 IO、采用更高效率的存储设备、均衡磁盘 IO 负载等方法是降低这一等待事件的主要方法。

3.5.3 write complete waits

当会话对一个正在被写写入磁盘的 Buffer 数据块发出请求时，需要等待其被写入磁盘完成，这时就会产生 write complete waits 等待事件。

事件中的 P1、P2、P3 参数分别代表：
P1：要写入数据的文件号；
P2：要写入的数据块号；
P3：无意义

提高 Buffer Cache 脏数据写入磁盘的效率、提高整体 IO 效率是降低该等待事件的主要方法：

- 配置数据库支持 AIO；
- 增加 db_writer_processes（支持 AIO 时）或者 db_io_slaves（不支持 AIO 时）大小以增加 DBWR 进程；
- 其他提高 IO 效率（如采用裸设备等）、减少 IO 冲突的方法

3.5.4 free buffer waits

当会话在 Buffer Cache 中找不到空闲 buffer 块，或者在没有空闲 buffer 块来建立一致性读时，就会产生 free buffer waits 等待事件。

事件中的 P1、P2、P3 参数分别代表：
P1：要读取数据的文件号；
P2：要读取的数据块号；
P3：10gR1 之前无意义，10gR1 后表示在 Buffer Cache 中 LRU 和 LRUW 列表的 SET_ID#

这一等待事件通常表示 Buffer Cache 不足或者从 Buffer Cache 中将脏数据写入磁盘的效率太低。要降低该等待事件，我们就需要分别从这两方面入手：调整 Buffer Cache 的大小（如根据 statspack 的建议器来设置）；按照我们前述的方法来提高存储设备的 IO 效率。

4 结束语

最后要说的是，一旦数据库服务器出现了 IO 问题后，首先要检查操作系统本身的 IO 系统是否有问题，然后再确认是否是 Oracle 出现了 IO 问题。

其次要注意的一点是，上述等待事件在系统出现一定的等待次数对于系统来说是正常的，我们要解决的是对系统 IO 影响最大的一个或几个等待事件，而不是全部事件。

总的来说，要调整 Oracle 中出现的 IO 性能问题，我们有两种手段：一种是针对特定等待事件的相应方法，如相关 SQL 语句的调优、相关参数的修改；另外一种是通过提升整体 IO 效率、减少 IO 冲突来降低 IO 等待，如均衡 IO 负载、使用效率更高的存储设备、激活 AIO 和重新分布对 IO 有不同要求的文件。

事实上，数据库的性能问题大多数是由应用引起的，而其中大部分问题都是 Top SQL 造成。因此，这里要说的一句题外话就是：SQL 调优是每一个 DBA 必须具备的最基本的技能。因为很多时候无论采用什么手段、什么工具来定位问题，通过各种内部机制来分析问题，但最终解决问题的手段就是 SQL 调优。

5 参考文章

1、www.HelloDBA.com
2、Metalink.Oracle.com
3、Oracle OTN
4、Oracle Concept
5、Oracle Database Performance Tuning Guide

--- Fuyuncat ---