Chapter No. 10
"Tuning I/O"
In this package, you will find:

A Biography of the author of the book
A preview chapter from the book, Chapter NO.10 "Tuning I/O"
A synopsis of the book’s content
Information on where to buy this book

About the Author

**Ciro Fiorillo** is an IT professional and consultant with more than a decade of experience in different roles (Developer, Analyst, DBA, Project Manager, Data and Software Architect) among software industries. He is an Oracle DBA Certified Professional and he has worked on different technologies and architectures, such as Oracle, SQL Server, Delphi, C# and .NET Framework, C/C++, Java, Flex, PHP, COBOL, Fortran, and Tibco.

He is based in Italy, near Naples, in the beautiful and historic Ercolano.

Ciro is currently employed as Information Systems Manager in a Financial Organization in Italy, and he is in charge of databases and systems management and development, coordinating the IT staff.

As a freelancer, he writes articles for websites and printed magazines about software and computing, participates in workshops, and teaches C++ parallel programming with Intel Software tools.

Ciro can be reached at ciro@cirofiorillo.com.

For More Information:

People use databases to organize and to manage their data. Oracle Database is the leader in the relational database management systems market, with a broad adoption in many industries. Using the best tool is not enough to be sure that the results of our efforts will be satisfactory—driving the fastest car in a Formula 1 competition, though better than driving the slowest, doesn’t guarantee the first place at the checkered flag.

Every developer—and every manager—knows that applications have to be responsive, because users hate to spend their time waiting for a transaction to end, looking at an hourglass. To meet this goal, it’s important to adopt a correct tuning strategy, which starts at the same time as the application design, then moves forward together, and will continue even when the application and the database are in production.

Even though this is a cookbook on performance tuning, there are no silver bullets. Every recipe in this book will show us how to solve a problem with the correct approach, so when a similar problem arises in one of our databases, we can apply the correct solution even in different situations than the ones presented in the book.

Before we start a database performance tuning process, we have to define what the tuning goals that we aim to reach are. "As fast as possible" is not a tuning goal. The primary tuning goal, generally speaking, is to reduce the response time or to reduce the resources needed to do a certain amount of work in the same time.

At a lower level, to minimize response time we will try to:

- Reduce or eliminate waits
- Cache the largest number of blocks in memory
- Access the least number of data blocks (from disks)

To increase the throughput and availability we will try to:

- Increment hit ratios
- Decrease system memory utilization
- Eliminate paging and swapping
- Reduce recovery time, decreasing the Mean Time To Recovery (MTTR)
- Increase load balancing (distributing data files to different disks) to reduce I/O times
- Increase scalability
Before starting a tuning session, we have to define which are the goals, in terms of SLA, or define precise and measurable objectives. So at the end of the tuning process, we will know if we have reached the expected results. We will work to reduce the workload—so the same task will consume less resources, allowing other tasks to use those resources—and to minimize the response time.

In this book, we will find many recipes that can help us reach these goals. Have a good read!

**What This Book Covers**

*Chapter 1, Starting with Performance Tuning* will show how to set up the example database, how to adopt a performance tuning process that can help in solving performance problems, and how to collect and analyze data from Oracle Database using various methods.

*Chapter 2, Optimizing Application Design* presents the most common application design issues that prevent an application from running without performance issues. You will see how to improve database performance by sharing reusable code and by reducing the number of requests to the database by using various database objects.

*Chapter 3, Optimizing Storage Structures* will show how to optimize the use of different database storage structures, presenting the optimal use for tables, clusters, indexes, and partitioning. You will see how to choose the appropriate structure to improve access time to your data, also analyzing the possible drawbacks in the operations that modify the data.

*Chapter 4, Optimizing SQL Code* is focused on SQL code optimization. Throughout the chapter you will find many methods to diagnose and solve typical performance problems caused by poorly written SQL code. You will find answers on how (and when) to avoid full table scans, how to use indexes, bulk operations and arrays, join and subquery optimization. You will also see how to trace SQL activity to diagnose problems.

*Chapter 5, Optimizing Sort Operations* will show the importance of optimizing sort operations to achieve better performance even when you don’t see any explicit sort operations in your SQL code. In this chapter, we will see the difference between in-memory and on-disk sort, how an index can improve the performance by reducing or avoiding sort operations, how to perform *top-n* queries, and how to use aggregate functions, and the use of set operations.

*Chapter 6, Optimizing PL/SQL Code* will show how to optimize PL/SQL code in stored procedures, triggers, and user-defined functions. You will see the advantages of using bulk-collect and array processing, native compilation and function result cache.

---

For More Information:

Chapter 7, *Improving the Oracle Optimizer* is focused on how to help the Oracle Optimizer in choosing the best execution plan using various tools, tricks, and tips, to obtain better performance. You will see the use of hints, statistics, histograms, stored outlines, adaptive cursor sharing, SQL tuning sets, and SQL baselines.

Chapter 8, *Other Optimizations* will show how to use Client Side Result Cache, parallel SQL, `CREATE TABLE AS SELECT`, and direct path inserting to optimize performance in both queries and DML operations. You will also see how to use SQL*Loader and Data Pump to load data into your Oracle Database.

Chapter 9, *Tuning Memory* will show how to avoid different memory-related issues, starting with Operating System paging. You will learn how to properly configure the library cache, the shared pool, the Program Global Area (PGA), the User Global Area (UGA), and the database buffer cache.

Chapter 10, *Tuning I/O* will focus on how to optimize the I/O, learning how to distribute Oracle files and stripe objects on different disks, what RAID level is better for each type of database files. The use of asynchronous I/O, checkpoint and redo logs tuning are also discussed in this chapter.

Chapter 11, *Tuning Contention* will show how to prevent, detect, and tune contention-related issues. You will see both lock and latch contention, why they occur, and how to prevent and solve any issue related to concurrency and contention in your database.

In Appendix A, *Dynamic Performance Views* you will find a list of the most used dynamic performance views; for each view you will find a brief description and a list of the most useful fields of the view, to be used as a reference in your daily work.

In Appendix B, *A Summary of Oracle Packages Used for Performance Tuning* you will find a brief summary of Oracle supplied packages useful in order to solve performance-related problems.

For More Information:
In this chapter, we will cover:

- Tuning at the disk level and strategies to distribute Oracle files
- Striping objects across multiple disks
- Choosing different RAID levels for different Oracle files
- Using asynchronous I/O
- Tuning checkpoints
- Tuning redo logs

Introduction

In the previous chapter, we have seen some methods to tune the memory used by Oracle processes to obtain best performance from our hardware.

The database is made up of datafiles on disks; typically, I/O time from disk is slower by one order of magnitude than I/O from memory. So, tuning the disk I/O subsystem can gain significant performance improvements for the database.

In this chapter, we will see the different types of files used by the Oracle database and the available options to tune each of them. Due to their specific use, we can see that there is a different solution to be implemented to optimize the I/O.

At the end of this chapter, you will also see how to tune checkpoints and redo logs to optimize according to their related disk activities.

For More Information:
Tuning at the disk level and strategies to distribute Oracle files

There are many Oracle background and foreground processes involved in a database instance; each of them specializes in a certain operation. In this recipe we will see what operations are executed by each process and what type of interaction takes place between files. On this basis, we will establish a strategy to distribute the Oracle files on different disks to help improve performance.

In *Chapter 9, Tuning Memory*, we have seen that the Oracle database uses different O/S processes on *nix machines, and different threads inside the same process on Windows machines, to obtain the same functionalities. In this chapter, when we refer to processes, we are talking about either *nix O/S processes or Windows threads.

**Getting ready**

To monitor and diagnose I/O performance issues, we need to enable timed statistics in the database, by setting the appropriate initialization parameter:

```
ALTER SYSTEM SET TIMED_STATISTICS = TRUE;
```

Without enabling this parameter we will not be able to see, in the statistics, the time required to complete an I/O operation; this value is needed to tune the I/O subsystem.

An appropriate Oracle Tuning Management Pack license is required.

**How to do it...**

The following steps will show how to distribute Oracle files to increase performance:

1. Connect to the database as `SYSDBA`:
   ```
   CONNECT / AS SYSDBA
   ```

For More Information:

2. Read the statistics about I/O on data files querying \texttt{V$FILESTAT} dynamic performance view:

\begin{verbatim}
COL FILE_NAME FOR A40
SELECT DF.FILE_NAME, FS.PHYRDS, 
    FS.PHYWRTS, FS.READTIM, FS.WRITETIM 
FROM V$FILESTAT FS, DBA_DATA_FILES DF 
WHERE FS.FILE# = DF.FILE_ID;
\end{verbatim}

3. Read the statistics about I/O on temporary files querying \texttt{V$TEMPSTAT} dynamic performance view:

\begin{verbatim}
SELECT DF.FILE_NAME, FS.PHYRDS, 
    FS.PHYWRTS, FS.READTIM, FS.WRITETIM 
FROM V$TEMPSTAT FS, DBA_DATA_FILES DF 
WHERE FS.FILE# = DF.FILE_ID;
\end{verbatim}

4. Identify the log files by querying \texttt{V$LOGFILE} dynamic performance view:

\begin{verbatim}
COL MEMBER FOR A40
SELECT * FROM V$LOGFILE;
\end{verbatim}

5. Put redo log files on disk without other activities; to move log files perform the following steps, otherwise go to step 10. Shut down the database:

\begin{verbatim}
SHUTDOWN IMMEDIATE
\end{verbatim}

6. Move the log files using the O/S commands:

\begin{verbatim}
!mv /u01/oradata/TESTDB/redo01.log
    /u01/oradata/TESTDB2/redo01.log
!mv /u01/oradata/TESTDB/redo02.log
    /u01/oradata/TESTDB2/redo02.log
!mv /u01/oradata/TESTDB/redo03.log
    /u01/oradata/TESTDB2/redo03.log
\end{verbatim}

7. Mount the database:

\begin{verbatim}
STARTUP MOUNT
\end{verbatim}

For More Information:

\url{www.packtpub.com/oracle-database-11g-r2-performance-tuning-cookbook/book}
8. Alter the location of the log files:

```sql
ALTER DATABASE RENAME FILE '/u01/oradata/TESTDB/redo01.log'
    TO '/u01/oradata/TESTDB2/redo01.log';
ALTER DATABASE RENAME FILE '/u01/oradata/TESTDB/redo02.log'
    TO '/u01/oradata/TESTDB2/redo02.log';
ALTER DATABASE RENAME FILE '/u01/oradata/TESTDB/redo03.log'
    TO '/u01/oradata/TESTDB2/redo03.log';
```

9. Open the database:

```sql
ALTER DATABASE OPEN;
```

10. Separate redo log files and archived redo logs, placing them on separate disks. To change the destination of archived redo logs, execute the following steps; otherwise, go to step 14. Shut down the database:

```sql
SHUTDOWN IMMEDIATE
```

11. Mount the database:

```sql
STARTUP MOUNT
```

12. Alter the parameter for archived redo log locations:

```sql
ALTER SYSTEM SET LOG_ARCHIVE_DEST_1 =
    'LOCATION=/u01/oradata/disk1/archive';
ALTER SYSTEM SET LOG_ARCHIVE_DEST_2 =
    'LOCATION=/u01/oradata/disk2/archive';
ALTER SYSTEM SET LOG_ARCHIVE_DEST_3 =
    'LOCATION=/u01/oradata/disk3/archive';
```

13. Open the database:

```sql
ALTER DATABASE OPEN;
```

14. Move heavily-accessed files to a separate disk. If you want to move the EXAMPLE tablespace to another disk, perform steps similar to step 5 through step 9. Take the tablespace offline:

```sql
ALTER TABLESPACE EXAMPLE OFFLINE;
```

15. Move the data files using the O/S commands:

```bash
!mv /u01/oradata/TESTDB/example01.dbf
    /u01/oradata/TESTDB2/example01.dbf
```

16. Alter the location of the data files:

```sql
ALTER DATABASE RENAME FILE '/u01/oradata/TESTDB/example01.dbf'
    TO '/u01/oradata/TESTDB2/example01.dbf';
```

For More Information:

17. Take the tablespace online:

```sql
ALTER TABLESPACE EXAMPLE ONLINE;
```

18. Remember to keep on separate disks data that is not related to the database. The last step may seem obvious, but, in many installations, the same disks are shared with other applications, such as web or application servers. The disks used by the database should not be shared among other applications to ensure optimal performance.

**How it works...**

In step 2, we query the I/O statistics on data files, obtaining the result shown in the following screenshot:

![Screenshot](image)

For each file, we can see the number of physical reads and writes performed and the time spent (in milliseconds) for these operations. By observing this, we can identify heavily-accessed data files.

In step 3, we execute a query similar to the one previously mentioned (regarding temporary files), obtaining the results shown in the following screenshot:

![Screenshot](image)

For More Information:

For good performance we need to distribute the online redo log files on different disks, on which little or no other I/O is performed. To do so, in step 4, we retrieve their current position, as shown in the following screenshot:

In step 5 through step 9, we moved the online redo log files to another disk, mounted in /u01/oradata/TESTDB2. We need to stop the database by executing a SHUTDOWN command and then use the mv command to physically move the redo log files identified in step 4.

In step 7, we mounted the database and, before opening it, informed the system of the new redo log file positions, by executing the ALTER statements in step 8.

In the following screenshot, we can see the result of all these operations:

For More Information:
When we execute the database in ARCHIVELOG mode (the norm for a production environment), we need to separate the disks in which online redo logs and archived redo logs are stored. If we want to change the location of our archived redo logs, we can follow step 10 through step 13.

Also, in this situation, we need to shut down the database and open and mount it, as in step 10 and step 11. In step 12, we set three different locations for our archived redo logs, storing them on three different disks. In step 13 we open the database and start using the new destinations for archived redo log.

Please note that when we change the archived redo log file locations, we need to change backup procedures accordingly, to reflect the changes made to the locations. We could obtain a RMAN-06207 error, and to fix it we can execute the CROSSCHECK COPY command in the RMAN prompt.

You can see the output of the preceding operations in the following screenshot:

In step 14 through step 17, we have seen how to move data files to different disks. When we heavily access data files and we know which file is executing the query against the statistics (shown in step 2 and step 3), we can gain in performance by separating the heavily-accessed data files on different disks.

For More Information:
In production environments, we usually check the disk controller-level information to see if the same disk controller is handling both the file systems. We distribute the data files based on disk controller allocation—distributing them on different mount points managed by the same disk controller brings no performance improvement.

The operations to be performed are similar to those executed when moving the online redo log files, but, this time, we don't need to shut down the database. We can take the tablespace offline, and move the desired data files to the new locations using O/S commands—we used `mv` in the example, inform the database about the new data file locations, as in step 17, and then bring the tablespace back online.

By executing the all the steps until now, you can see the following output:

```
In step 19, there is a final tip about distributing Oracle files on machines where the Oracle database shares resources with other applications. We need to keep separate disks for the database to avoid possible issues and slowing down of simultaneous access to the disk by other applications.

**There's more...**

When we discuss different disks, we obviously refer to different physical disks, possibly using different controllers. It's important to know which process uses which type of database file.

Data files are written mostly by `DBWn` processes; the `CKPT` process reads and writes only data file headers. Server processes read data from data files.

Redo log files are sequentially written by the `LGWR` process and read by `ARCh` processes, when the database is in `ARCHIVELOG` mode. Archived redo logs are only written by `ARCh` processes, under the same conditions.

The `CKPT`, `LGWR`, and `ARCh` processes can only read and write control files.
LGWR writes online redo logs sequentially, using a RAID 5 on the
disks, where online redo logs are stored. This can lead to poor
performance due to the slower write times that characterize this
type of disk array—using RAID 0+1 is preferable.

See also

- The Choosing different RAID levels for different Oracle files recipe in this chapter

Striping objects across multiple disks

In the previous recipe, we have seen how to distribute Oracle files on different disks to
obtain better performance. In this recipe, we will see how to stripe objects using different
tablespaces or data files, to improve performance.

How to do it...

The following steps will demonstrate how to stripe objects across multiple disks:

1. Connect to the database as SYSDBA:
   
   ```
   CONNECT / AS SYSDBA
   ```

2. Create a new tablespace, EXAMPLE2, on a different disk:
   
   ```
   CREATE TABLESPACE EXAMPLE2
   DATAFILE '/u01/oradata/TESTDB2/example2.dbf' SIZE 100M;
   ```

3. Move the CUSTOMERS table of the SH schema to the newly-created tablespace:
   
   ```
   ALTER TABLE SH.CUSTOMERS MOVE TABLESPACE EXAMPLE2 NOLOGGING;
   ```

4. Identify the indexes that need to be rebuilt:
   
   ```
   SELECT INDEX_NAME, STATUS FROM ALL_INDEXES
   WHERE TABLE_OWNER = 'SH' AND TABLE_NAME = 'CUSTOMERS';
   ```

5. Rebuild the indexes:
   
   ```
   ALTER INDEX SH.CUSTOMERS_PK REBUILD;
   ALTER INDEX SH.CUSTOMERS_GENDER_BIX REBUILD;
   ALTER INDEX SH.CUSTOMERS_MARITAL_BIX REBUILD;
   ALTER INDEX SH.CUSTOMERS_YOB_BIX REBUILD;
   ```

For More Information:

6. Add a data file to the EXAMPLE tablespace on a different disk:

   ```sql
   ALTER TABLESPACE EXAMPLE ADD
   DATAFILE '/u01/oradata/TESTDB2/example_DISK2.dbf' SIZE 100M;
   ```

7. Allocate an extent for the COUNTRIES table of the SH schema on the newly-created data file:

   ```sql
   ALTER TABLE SH.COUNTRIES ALLOCATE EXTENT
   (DATAFILE '/u01/oradata/TESTDB2/example_DISK2.dbf' SIZE 1M);
   ```

**How it works...**

We want to spread our objects to different disks, to obtain better performance. To do so, we can use multiple tablespaces, allocating them to different disks and distributing objects among different tablespaces, or we can add multiple data files—spread among different disks—to the same tablespace and allocate extents for our objects to these data files. In this recipe, we have followed both the methods.

We can use the `DBA_HIST_SEG_STAT` view to identify the most-accessed segments from instance startup.

In step 2, we created a new tablespace, named `EXAMPLE2`, made by a single data file on a disk mounted under the `/u01/oradata/TESTDB2/` path.

In step 3, we moved the CUSTOMERS table of the SH schema from tablespace EXAMPLE to tablespace EXAMPLE2. We have used the `NOLOGGING` option to avoid logging all the data movements—only the change in a data dictionary is logged.

We can see the results of these operations in the following screenshot:

For More Information:

As always, be careful when executing NOLOGGING operations. To avoid possible data loss, plan this operation with the database backup administrators.

In step 4, we have queried the status of indexes on the moved table. As seen in the following screenshot, all the indexes on the CUSTOMERS table are showing the UNUSABLE status, so they need to be rebuilt:

![SQL query for index status](image)

In step 5, we rebuild the indexes, obtaining the following output:

![SQL ALTER INDEX command output](image)

From step 6 onwards, we follow a different strategy—adding a data file to an existing tablespace and storing it on a different disk. In step 6, we add the new data file to the EXAMPLE tablespace.

In step 7, we allocate an extent for the COUNTRIES table of the SH schema, on the newly created data file. From now onwards, the data in the COUNTRIES table will be stored on a different disk.

For More Information:
Tuning I/O

The results of these operations can be seen in the following screenshot:

![Screenshot of SQL commands](image)

**There's more...**

In the latter steps of the recipe, we have manually striped the objects. But if we have many objects in our database, manual striping can be a nightmare. In such situations, consider moving objects to different tablespaces or using partitions.

We can also distribute tables and related indexes on different disks, to obtain performance gain in both read and write operations. If we have tables and indexes on the same disk, we need to read and write in two different places on the same disk. By dividing the work between two disks, we can perform an index range scan faster than when the index and the table are on the same disk.

**See also**

- The *Using partitioning* recipe in Chapter 3, *Optimizing Storage Structures*

**Choosing different RAID levels for different Oracle files**

In this recipe, we will see the characteristics of different RAID levels available in the market and learn what to choose for each different Oracle file type.

**RAID** is the acronym for *Redundant Arrays of Inexpensive Disks*, a common configuration in a storage subsystem. It is used to obtain low-cost, fault-tolerant configurations for high performance in the non-mainframe market, by using multiple "inexpensive" disks in different configurations.

For More Information:
Getting ready

Despite the "I" in RAID, we need a minimum of two or three drives, depending on the RAID level we want to implement.

Every RAID level has different requirements and offers different performance and data integrity levels. In this recipe, we will illustrate these requirements for each RAID level.

There is a common requirement, that is, the RAID controller; it can be software-based at the operating system, firmware level, or hardware-based. The latter offers guaranteed performance and no overhead on the CPU.

How to do it...

The following steps will demonstrate the various RAID levels; you can chose the right RAID level by considering the following:

1. RAID 0+1 is preferable for your Oracle database installations.
2. RAID 5 has significant write penalty, so don't use it for storing write-intensive data files (if RAID 0+1 is available), redo log files, archived redo log files, and undo segments. You can use it for control files and for data files with a moderate write activity.

How it works...

This recipe has only two steps, which are simple tips to choose the right hardware to obtain better performance. More than a buy list it's a no-buy list:

1. Starting with step 1, in terms of performance the best RAID level for an Oracle database is RAID 0+1, also known as RAID 10. The drawback of this option is the cost of the solution, because it is twice the cost of storage due to mirroring. Additionally, it uses more complex procedures and hardware to manage the striping.
2. In step 2, we warn against RAID 5. There is a significant write penalty when using this RAID level, so storing frequently-updated data is not a good choice. Also, redo log files and archived redo logs don't fit well in this environment, because, on OLTP databases, there is an intense write activity on these files, and also because undo segments experience heavy load.

There's more...

To understand different RAID levels better, we will cover the differences between the most common RAID configurations, highlighting performance considerations related to storing Oracle database files.

For More Information:

Tuning I/O

**RAID level 0**

This is the simplest RAID level, and it's not recommended to use it in any production environment by Oracle itself.

In this RAID level only non-redundant striping is implemented, that is, the data is striped across multiple disks to obtain better read/write performance. However, this is a non-redundant solution, so any disk failure causes the outage of the entire array and hence data loss. It's very cheap because it doesn't need more storage capacity than the actual space required.

**RAID level 1**

RAID level 1 implements a disk mirroring strategy. For each disk drive there is at least one identical disk drive on which an exact copy of the data is maintained. There can be \( n-1 \) outages, where \( n \) is the number of drives on which the data is simultaneously stored, without any data loss. If hot-swappable drives are used there could be no application outage even after \( n-1 \) outages in different disks.

The main defect of this implementation is the cost per megabyte, because for each megabyte of data stored we need \( n \) megabytes of storage to ensure the design redundancy.

Performance, as compared to the performance of a single drive, is slightly better in read times if the controller can choose the disk from which to read the data considering the least I/O cost.

We can use RAID level 1 to store control files, system tablespace, flashback recovery area, undo segments, and data files. It's not a good idea to store redo logs and temporary segments on RAID level 1 configurations, because redo logs should be multiplexed on different disks, and there is no need to protect temporary segments from failures.

**RAID level 5**

RAID level 5 introduces the concepts of Block-interleave and distributed parity.

In a RAID 3 array, there is one disk dedicated to the storage of parity data for the other disks in the array, and data is striped using a strip of 1 bit. RAID 5 introduces two variations of the schema—stripes size is configurable and parity data is not stored on a single drive but is distributed among all disks of the array.

The reason to choose RAID level 5 is to obtain a redundant storage solution, cheaper than RAID level 1. Due to storage of parity data, however, write operation performance suffers. This is because, when writing some data, we need to read the old data and parity value, and write the new data and parity value, resulting in four I/O operations.

Read performance is excellent when the data fits in a single striping segment, allowing a heavy concurrency on data. Except undo segments and intensive-write data files, all other Oracle files can be stored in a RAID level 5 array—when you cannot use RAID level 0+1, especially with read-only data files.

For More Information:

RAID level 0+1

RAID level 0+1 is also known as RAID level 10; it's the result of RAID level 0 and RAID level 1 arrays being used together. So, we have a striping and mirroring solution that allows excellent performance in read operations and very good performance in write operations, due to the additional write operation required by mirroring. RAID level 0+1 ensures optimal protection against failures, thanks to mirroring.

The only defect of RAID level 0+1 subsystems is the cost of this solution, but it can be used to store every type of Oracle database file with optimal performance.

See also

- The Tuning memory to avoid Operating System paging recipe in Chapter 9, Tuning Memory

Using asynchronous I/O

In this recipe we will see how to use asynchronous I/O to obtain better performance from our I/O subsystem.

How to do it...

The following steps will describe the use of asynchronous I/O:

1. Connect to the database as SYSDBA:
   
   ```sql
   CONNECT / AS SYSDBA
   ```

2. Verify whether asynchronous I/O is enabled:
   
   ```sql
   SHOW PARAMETER FILESYSTEMIO_OPTIONS
   ```

3. Enable asynchronous I/O, if it is not enabled:
   
   ```sql
   ALTER SYSTEM SET FILESYSTEMIO_OPTIONS=SETALL SCOPE=SPFILE;
   ```

4. Shut down the database instance:
   
   ```sql
   SHUTDOWN IMMEDIATE
   ```

5. Start and open the database:
   
   ```sql
   STARTUP OPEN
   ```

6. Verify the change in system configuration:
   
   ```sql
   SHOW PARAMETER FILESYSTEMIO_OPTIONS
   ```

For More Information:

How it works...

The Oracle database may use synchronous or asynchronous I/O calls. With synchronous I/O, when an I/O request is submitted to the Operating System, the write process will block until the operation is completed.

Using asynchronous I/O, while the I/O request is still executing, the calling process continues its work without blocking. This is the reason why asynchronous I/O can lead to performance gain in processing writes to Oracle database files.

There is an important downside to keep in mind while using asynchronous writes—the blocks may not get written immediately to the file system, and this behavior may lead to missing data, or corruption, in case of a failure.

In step 2, we verify if the asynchronous I/O option is enabled in our database instance. In step 3, we set the value to SETALL, enabling both asynchronous and direct I/O on system files. Using direct I/O allows the process to bypass the Operating System cache. The Oracle database already uses the database buffer cache, so we can access database files directly, without consuming resources required by the OS cache.

In steps 4 and 5, we restart the database to set the new parameters, and, in step 6, we verify the new value for the FILESYSTEMIO_OPTIONS parameter.

In the following screenshot, we can see the output for the previous operations:

For More Information:
There's more...

Other options for the FILESYSTEMIO_OPTIONS parameter include:

- **NONE**: this disables both asynchronous I/O and direct I/O
- **ASYNCH**: this enables asynchronous operations
- **DIRECTIO**: this enables only the direct I/O option, thus bypassing the O/S cache

On Windows systems the OS cache is never used, so every request will bypass the Operating System cache and go directly to the disk.

On Windows systems, starting from Oracle 10g, asynchronous I/O is supported for all files on Windows Server 2003 x64 bit upward.

On *nix systems we need to pay attention to certain operations that don't use database buffer cache. Using direct I/O option may lead to a performance dip. Those operations include I/O on the temporary tablespace, using NOCACHE LOBs, and parallel query slaves reading data.

Asynchronous I/O is important for **DBWR** and **LGWR** processes, because it allows **DBWR** to use the available hardware bandwidth completely. The **LGWR** writes to multiple log file members and can overlap multiple writes due to rapidly subsequent committed transactions occurring simultaneously.

Using asynchronous I/O on redo log files and temp files eliminates some contention related to the file system read/write locks, resulting in increased performance. Using asynchronous I/O on data files instead doesn't affect performance but scalability, that is, the database can handle more requests at a time. The **DBWR** processes, responsible for writing to the data files, work asynchronously, so the performance of user processes is not affected by the use of asynchronous I/O. However, using asynchronous I/O allows **DBWR** processes to use all the available bandwidth, and use it in a more efficient way, so higher workloads can be managed by the system when asynchronous I/O is enabled.

On platforms that don't support asynchronous I/O, we can enable multiple database writer **slave processes**. A single **DBWR** process will use multiple slave processes to write data on disks, simulating something similar to asynchronous I/O. Please note that multiple **DBWR** processes and multiple **DBWR** slaves cannot be run together, and the last option takes precedence.

To enable multiple database writer slave processes, you need to set the initialization parameter **DBWR_IO_SLAVES** to a non-zero value, setting the number of slave processes to use.

See also

- Refer to the Tuning the buffer cache recipe in Chapter 9, Tuning Memory

For More Information:
Tuning checkpoints

A checkpoint is used to ensure consistency in the database; during this operation, all data files are synchronized with the data blocks in memory.

The process responsible for signaling a checkpoint is the CKPT process, which signals the DBWR processes to write the dirty (modified) buffers from database buffer cache in memory to the data files.

During this operation data, file headers and control files are updated to store the last System Change Number (SCN), to ensure data block consistency.

In this recipe, we will see how to tune checkpoints in an Oracle database, to optimize all these write operations involved in checkpoints, balancing the trade-off between the redo log size and recovery time, in case of instance failure.

How to do it...

The following steps will demonstrate checkpoints in an Oracle database:

1. Connect to the database as SYSDBA:
   ```
   CONNECT / AS SYSDBA
   ```

2. Verify the value for the LOG_CHECKPOINTS_TO_ALERT parameter:
   ```
   SHOW PARAMETER LOG_CHECKPOINTS_TO_ALERT
   ```

3. Alter the LOG_CHECKPOINTS_TO_ALERT parameter to trace checkpoints to the alert log:
   ```
   ALTER SYSTEM SET LOG_CHECKPOINTS_TO_ALERT=TRUE SCOPE=SPFILE;
   ```

4. Switch the log file to force a checkpoint to occur:
   ```
   ALTER SYSTEM SWITCH LOGFILE;
   ```

5. Verify the checkpoint event has been traced in the alert log:
   ```
   !tail /u01/app/diag/rdbms/testdb/TESTDB/trace/alert_TESTDB.log
   ```

6. Query the V$SYSSTAT dynamic performance view to monitor checkpoint process activity:
   ```
   SELECT NAME, VALUE FROM V$SYSSTAT
   WHERE NAME LIKE 'background check%';
   ```

   ```
   SELECT NAME, VALUE FROM V$SYSSTAT
   WHERE NAME LIKE 'DBWR check%';
   ```

For More Information:

7. Verify other parameters involved in the checkpoint:

   SHOW PARAMETER LOG_CHECKPOINT

8. List parameters related to Fast-Start checkpointing:

   SHOW PARAMETER FAST_START

9. Query the V$INSTANCE_RECOVERY dynamic performance view to obtain estimations on recovery time:

   SELECT
       RECOVERY_ESTIMATED_IOS,
       ESTIMATED_MTTR,
       TARGET_MTTR,
       LOG_FILE_SIZE_REDO_BLKS,
       LOG_CHKPT_INTERVAL_REDO_BLKS,
       LOG_CHKPT_TIMEOUT_REDO_BLKS
   FROM V$INSTANCE_RECOVERY;

**How it works...**

In step 2, we verify the value for the LOG_CHECKPOINTS_TO_ALERT parameter, and in step 3 we set it to TRUE, to record checkpoint information to the alert log. In step 4, we force a checkpoint to occur by switching log files.

In the following screenshot, you can see the output of the previous operations:

![Screenshot of SQL operations](image)

In step 5, we verify the content of the alert log to be sure that the checkpoint information is being stored in it.

For More Information:

Tuning I/O

In the following screenshot, you can see that the checkpoint information was written to the alert log:

![Alert Log Screenshot]

In step 6, we query `V$SYSSTAT` to monitor some statistics related to the redo log files and checkpoint. The result of the query is shown in the following screenshot:

![SYSSTAT Query Screenshot]

If the number of started checkpoints is greater than the value of completed checkpoints by more than one, in the first query, you need to enlarge redo log file size. In this situation, checkpoints are not completed between log file switches, because the log file switches occur too often, as log files are very small. Increasing redo log file size will limit the number of log switches required, allowing checkpoints to complete between them.

A redo log switch should occur every 15 to 30 minutes (as a rule of thumb); switching too often leads to performance issues, whilst switching not often enough often may cause a recovery operation to take longer.

For More Information:
The second query shows the number of data blocks written by DBWR and the number of checkpoints. This is useful for monitoring the number of blocks written by DBWR during checkpoints.

In step 7, we view the actual value of parameters, that influence checkpoints, obtaining the following output:

We have seen the use of the LOG_CHECKPOINTS_TO_ALERT parameter in step 2 and step 3.

You can set LOG_CHECKPOINT_INTERVAL to the maximum number of redo log blocks—equivalent to the size of O/S file blocks. It will be left in the redo log before a checkpoint occurs. This value cannot exceed 90 percent of the number of redo blocks that can be stored in the smallest redo log file, to ensure that there won't be log switch between checkpoints.

The LOG_CHECKPOINT_TIMEOUT is used to set the maximum number of seconds for which a dirty block can stay in the memory before it's written to disk. The default value is 1800 seconds. In step 8, we show other initialization parameters related to Fast-Start checkpointing. The output of this operation is shown in the following screenshot:

Fast-Start checkpointing is configured to assure that the instance recovery time is acceptable; this target can be achieved by setting one of the parameters, as shown in the previous image.

We can set the FAST_START_MTTR_TARGET parameter to the expected Mean Time To Recover, that is, the number of seconds required to recover the instance after a crash.

When you set FAST_START_MTTR_TARGET, you cannot use the parameters LOG_CHECKPOINT_INTERVAL and LOG_CHECKPOINT_TIMEOUT, shown earlier.

For More Information:
In step 9, we query the `V$INSTANCE_RECOVERY` view to obtain an estimate of recovery time, collecting data required to choose the correct values for the parameters shown earlier. In the following screenshot, you can observe the output of this query:

![SQL query output](image)

You can find the description for each field in `V$INSTANCE_RECOVERY` view in Appendix A, Dynamic Performance Views.

**There's more...**

In this recipe, we have seen how different parameters can influence the checkpoint behavior.

The **checkpoint queue** is a list of dirty buffers (that is, blocks in the buffer cache modified and not already written to the disk) waiting to be written to the disk by the `DBWR` processes.

There is a trade-off between a short checkpoint queue that ensures faster recovery times in case of instance crash, and a long checkpoint queue which avoids frequent `DBWR` writes that can affect performance.

As always, we need to evaluate the ideal configuration to satisfy our Service Level Agreements. If we assure the maximum recovery time, we will shorten the checkpoint queue, resulting in some more `DBWR` writes.

---

**For More Information:**

The last query of this recipe helps us to obtain RECOVERY_ESTIMATED_IOS. This is an estimate of the number of data blocks to be processed during recovery. ESTIMATED_MTTR indicates the estimated recovery time, based on the current system load. TARGET_MTTR is based on the value of the FAST_START_MTTR_TARGET parameter and on the system performance and limitations. LOG_FILE_SIZE_REDO_BLKS is the number of redo blocks required to make sure that a log switch doesn't occur before the checkpoint completes.

The value for LOG_CHKPT_INTERVAL_REDO_BLKS and LOG_CHKPT_TIMEOUT_REDO_BLKS indicates the number of redo blocks that will be processed during recovery to satisfy the LOG_CHECKPOINT_INTERVAL and LOG_CHECKPOINT_TIMEOUT parameters, respectively.

See also

- The Tuning redo logs recipe that follows

Tuning redo logs

In this recipe, we will see how to monitor redo logs.

How to do it...

The following steps will demonstrate monitoring of redo logs:

1. Connect to the database as SYSDBA:
   ```sql
   CONNECT / AS SYSDBA
   ```
2. Verify possible problems by inspecting the V$SYSTEM_EVENT dynamic performance view:
   ```sql
   SELECT EVENT, TOTAL_WAITS, TIME_WAITED FROM V$SYSTEM_EVENT
   WHERE EVENT LIKE 'log file%';
   ```
3. Query the data dictionary about the redo log files:
   ```sql
   COL MEMBER FOR A40
   SELECT * FROM V$LOGFILE;
   CLEAR COL
   ```
4. Query the data dictionary about redo log details:
   ```sql
   SELECT * FROM V$LOG;
   ```
5. Query the historical log switch data:
   ```sql
   SELECT * FROM V$LOG_HISTORY ORDER BY RECID;
   ```

For More Information:

Tuning I/O

How it works...

In step 2, we query the V$SYSTEM_EVENT dynamic performance view to inspect problems related to redo logs. In the following screenshot, we can see the results obtained on a test database:

The important events to be observed are log file sync and log file parallel write. Often, a high value for the latter statistic is not evidence of a problem. It indicates a wait in LGWR activity, but doesn't specify whether waits affect user processes or not.

The log file sync statistic is more reliable. The symptoms of I/O issues, related to slow disks on which redo log files are written, are highlighted by a high value for log file sync statistic. Often a high value for the log file parallel write is confirmed by a high value for this parameter. In these situations, you need to solve the issue—the high waits—related to redo log file writes.

In step 3, we query the V$LOGFILE to know the redo log files in our database and some information on their status. We can see the results obtained in the following screenshot:

We can see that there are three redo log groups in the database, with only one member each. In a production database, you need at least two members for each group, and, according to the transaction load on the database, more redo log groups could be required.

For More Information:
In step 4, we query the V$LOG dynamic performance view, in which we can find more details regarding redo logs, as in the following screenshot:

![Screenshot of V$LOG view](image)

In the excerpt shown in the previous screenshot, you can see the redo log group (GROUP#), the sequence number (the first SCN, System Change Number) in each group (SEQUENCE#), the size of each group's members (BYTES), the status (STATUS), and whether the log is archived or not (ARC).

In step 5, finally, we query the V$LOG_HISTORY dynamic performance view, obtaining a row for each log switch. This view is useful to verify the log switch frequency, and indicates the first and last change numbers in the log file.

This information is stored in the control file; the MAXLOGHISTORY clause, specified when the control file was created, indicates the length of time for which the information is retained in the control file itself.

In the following screenshot, you can see an excerpt of the results obtained by the query executed in step 5:

![Screenshot of V$LOG_HISTORY view](image)

For More Information:
There's more...

We can also execute a statspack report and search for the **File IO Stats** section or for **log file parallel write** wait events in the **Event** section.

See also

- The *Tuning checkpoints* recipe in this chapter
- The *Analyzing data using Statspack report* recipe in *Chapter 1, Starting with Performance Tuning*

For More Information:

Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.