Database Performance

11

What the reader will learn:

- that the different physical and logical aspects of database performance make tuning a complex task
- that optimising for read and write performance is not the same thing and the this can cause conflicts when tuning
- that database tuning is an ongoing requirement in an active OLTP system
- that there are several types of index, each aimed at returning specific kinds of data more rapidly
- that disk operations are amongst the slowest element of any database operation

The examples we deal with in this chapter are primarily based on Client/Server RDBMS technology. Of course, as we saw in Chap. 5, other types of database technology do exist, and do claim to bring high performance in certain scenarios.

The worked examples and diagrams rely heavily on Oracle. We have used Oracle 11g. However, much of what is covered holds true for versions back to 8i. Many architectural diagrams will hold true in principle for SqlServer, MySQL and most other true Client/Server databases.

11.1 What Do We Mean by Performance?

For users—very important stakeholders in any system—database performance is often simply measured in terms of how quickly data returns to their application. And they will often intuitively know when that is "too slow" despite not having timed the process.

Returning data can be a very important element of a database system's requirements. In a Decision Support System (DSS), for example, the system may well spend more than 90 % of its processing time serving out results sets and only ever have data loads occasionally. OLTP systems, on the other hand are often writing new rows, or updating existing ones, whilst relatively less frequently answering queries.

Heavy use of indexing to speed query output will be very likely to slow inserts and updates (since the system has more information to store). The need, therefore, whether your system is read- or write-intensive is a very good starting point.

P. Lake, P. Crowther, *Concise Guide to Databases*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-5601-7_11,

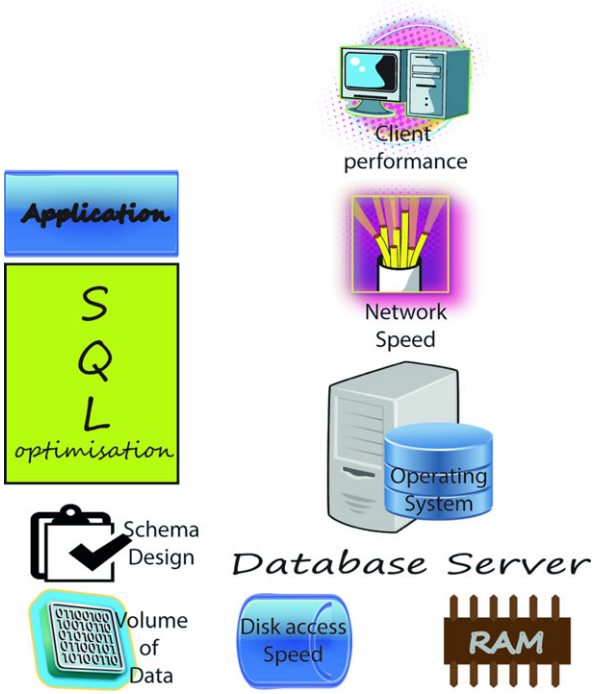


Fig. 11.1 Elements which influence database performance

So performance is not only about the speed with which queries are answered. As we look further at the building blocks of RDBMS we will need to come to a more rounded view. *Performance is about ensuring you use all the system resources optimally in the process of meeting the application's requirements, at the least cost.*

Modern database systems are typically built on several layers of technologies. When running a SQL query, the physical parts of the host server can be just as important in determining how quickly we get the result report runs as any of the processes within the RDBMS itself (see Fig. 11.1).

We can divide the elements that might impact database performance into the following broad, interrelated categories:

Physical layer, such as disks and RAM
Operating System (OS)
Database Server processes
Schema level: Data types, location and volumes
SQL optimisation
Network
Application

Typical production databases are dynamic, growing entities, which means that yesterday's perfectly tuned database can become today's performance dog just because it is being used. We will review the influence the data itself can make on performance later in the chapter.

A good database professional needs to understand all the factors that impact upon the performance of their databases. Proactive performance tuning can save a DBA time in the long run as reactive problem chasing can be both time consuming and detrimental to the business.

Before we start, we should clarify some of the important aspects of database performance:

Workload is the combination of all the online transactions, ad hoc queries, data warehousing and mining, batch jobs, and system-generated commands being actioned by the database at any one time. Workload can vary dramatically. Sometimes workload can be predicted, such as peaks with heavy month-end processing, or lulls after the workforce has gone home. Unfortunately, however, DBAs do not always have the luxury of such predictability. This is especially so in a 24/7 environment, such as an e-commerce system. Naturally, workload has a major influence upon performance.

Throughput is a measure of the database's capability to carry out the processing involved with the workload. There are many factors that can impact upon throughput, including: Disk I/O speed, CPU speed, size of available RAM, the effectiveness of any workload distribution such as parallel processing, and the efficiency of the database server operating system.

Contention is the condition in which two or more components of the workload are attempting to use a single resource. Multiple users wanting to update a single row, for example. As contention increases, throughput decreases. Locks are placed on resources to ensure that tasks are completed before the resource is made available again, resulting in waits. The user notices this and calls it poor performance.

11.2 A Simplified RDBMS Architecture

Before we look at the individual performance factors, it would be good to have some idea of what happens when a user interacts with a database, so that we know where to look for improvements.

Review Fig. 11.1 and, with this diagram in mind, let us walk through one interaction, which is a simple fetch of data:

1. The user starts their client-side application process. This could be a simple command-line process, like SQLPLUS, or a full blown application.
2. The client process connects to the rdbms server process.
3. An area of rdbms managed RAM is set aside to hold inputs and outputs, and to allow server-side processes on behalf of the client, such as sorting
4. The request for the rdbms to process some SQL is passed over the connection from client to server.
5. The SQL string is then compared with the SQL which has been run by the server before, which is stored in the library cache. If this query has been run before the execution plan can be recalled from cache and reused, saving hefty processing on parsing and execution plan generation.

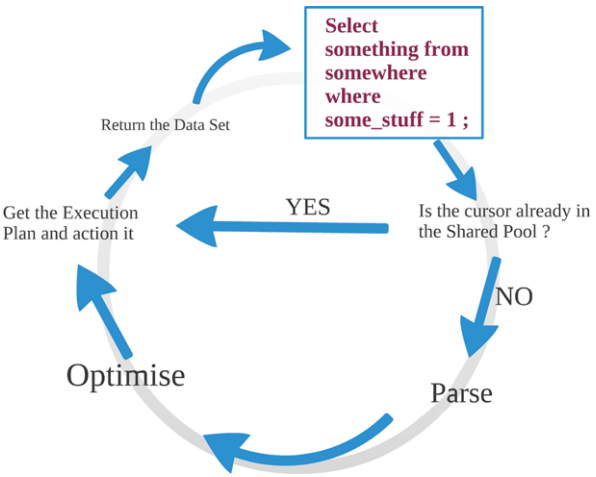


Fig. 11.2 Phases in processing an SQL query

6. If this is a new piece of SQL then the string needs to be parsed. This checks that the objects (tables, indexes etc.) referred to exist and that the user has access rights. It checks that the statement is syntactically correct. It will also gather run-time values if there are bind variables involved in the query.
7. So now we have a valid, new query that we have sufficient privileges to run, we then pass over to the optimiser (see section on optimisation below) to determine the best way to gather the information.
8. The optimiser will generate an execution plan
9. The SQL processor will then carry out the instruction in the plan
10. As this is a query which requires data to be fetched, then the RDBMS will have to maintain read consistency to ensure that data it returns is from a certain point in time and not affected by any changes made by other users during the fetch.
11. If the data has been fetched by another user recently enough for it not to have dropped out of the buffer cache, and it and is clean (it has not been changed), the RDBMS will gather the data from the data buffer rather than use a time consuming disk read.
12. If this data isn't in the cache, it will be read from disk and placed into the buffer cache.
13. If there are any post fetch tasks, such as sorting, these can be carried out at the server or at the client.
14. Once all gathered the requested dataset will be returned to the client for use in the user application.

With this set of core processes in mind, we will now examine each of the layers we outlined above.

11.3 Physical Storage

Unlike RAM, the medium used to store permanent data in a database has to be non-volatile—the data stored must remain the same regardless of whether the device has power to it or not. Most database servers are dependant upon a disk, or array of disks, to enable the permanency of the data. This is a weak point in terms of performance since disk operations are amongst the slowest elements of any database process. The DBA's task, therefore, is more about accepting that there will be some delays as a result of I/O latency, and finding ways to minimise it.

Much of the database's inbuilt processing will already be automatically attempting to minimise the disk I/o. Where possible data that has been fetched from disk will be kept in the server's RAM for as long as possible in case other users may want the same data. Naturally, in a very busy production system this can mean either a huge (and therefore expensive) bank of RAM, or a volatile data pool which has to swap out data from memory frequently, thus causing more disk reads.

11.3.1 Block Size

Of course we should never loose sight of the fact that we are storing ons and offs on a magnetic material when we store our data. The operating system is given the job of managing the disk handling required to seek a piece of information, but what it will return will be a "block" of ons and offs which the RDBMS will have to manipulate. It may have to extract a small part of what is returned. Or it might have to send the disk head reader to several places, extract data from each read, and then glue the portions together.

In an Oracle database objects are stored in a Tablespace. This is a container for tables, indexes Large Objects, and is a continuous area of physical disk that is under Oracle's control and can't be used by other applications. The Tablespace is the Logical-Physical boundary since one of the parameters passed when issuing the Create Tablespace command is the physical o/s file(s) being used to store the contents.

What the operating system will deal in is a block. Block size is set at the OS level. Db block, which is what the RDBMS deal in, needs to be a multiple of that figure in order that we do not waste time retrieving or writing O/S blocks which are only part full of the data we want to collect.

The Oracle Tuning Guide offers this advice:

> A block size of 8 KB is optimal for most systems. However, OLTP systems occasionally use smaller block sizes and DSS systems occasionally use larger block sizes.

Now we have another circumstance when the DBA needs to understand the type of database he is managing. In two extremes, for example, we could have a Data Warehouse which rarely has data added and is always causing the SQL engine to read large datasets from disk. On the other hand we might have an OLTP system recording only small amounts of data from a sales line, and nearly always serving small datasets back to queries.

The task we want to help the HDD avoid is spinning the disk and moving the read-head to a new position, since this is a really slow part of the overall process. If, in the data warehouse example, all the data is physically all together in adjacent disk blocks it would be good if we could "suck up" all those with only one head-positioning movement. This is why Oracle controls an area of disk in a Tablespace. It attempts to manage that portion of disk such that data is most likely to be adjacent to similar data.flash

11.3.2 Disk Arrays and RAID

The main database vendors all support reading and writing to multiple disks. There can be different reasons for doing this (such as robustness and performance), but in recent years the driving force has probably been the low cost nature of HDD, and the ease with which they can be slotted into a rack.

RAID technology is aimed to either improve the I/O performance of the disk subsystem or to make the data more available. There is usually a trade-off to be had, but some RAID configurations attempt to do both. The other variable to consider is cost, since, generally speaking, the more complex the RAID management system needs to be, the more it will cost.

These storage options can impact on a system's performance. Striping your data such that subsequent data writes are made to different disks in a round-robin, can have a large effect on read speed if parallel processing is enabled since data can be read from all disks at once and then glued together in memory. More information about RAID and arrays can be found in Chap. 3.

11.3.3 Alternatives to the HDD

All of this section so far has been written with the standard HDD as the permanent storage device in mind. There are alternatives. In more recent times mobile computing has come to the fore, and the shock-resistant, lightweight and power efficient nature of flash memory makes it a sensible addition to embedded systems or portable computing devices. RAM is still faster than flash, but in read operations flash can outperform a HDD by factors of several hundreds. This is primarily because there is no spinning platter to add the mechanical resistance the HDD suffers from. One drawback however is that erasing, and therefore effectively managing the disk content, is considerably slower for flash. At the time of writing, flash is also considerably more expensive/Mb than a HDD.

Whilst the need to be mobile can excuse the weaknesses of flash in some applications, it is unlikely that many large scale commercial applications will rely on flash just yet, and research papers are being published which identify hybrid solutions which attempt to take advantage of flash to speed certain operations, whilst maintaining the traditional virtues of the HDD.

Perhaps the most extreme form of performance enhancement is to have a totally in-memory database.

Documentation errata are in the README. Extra installation information for each platform is in the Release Notes.		
Readme	HTML	PDF
Release Notes for AIX 5L Based Systems (64-Bit)	HTML	PDF
Release Notes for Apple Mac OS X (Intel)	HTML	PDF
Release Notes for HP OpenVMS	HTML	PDF
Release Notes for HP Tru64 UNIX	HTML	PDF
Release Notes for HP-UX Itanium	HTML	PDF
Release Notes for HP-UX PA-RISC	HTML	PDF
Release Notes for IBM z/OS on System z	HTML	PDF
Release Notes for IBM zSeries Based Linux	HTML	PDF
Release Notes for Linux Itanium	HTML	PDF
Release Notes for Linux on POWER	HTML	PDF
Release Notes for Linux x86	HTML	PDF
Release Notes for Linux x86-64	HTML	PDF
Release Notes for Microsoft Windows (32-Bit)	HTML	PDF
Release Notes for Microsoft Windows (x64)	HTML	PDF
Release Notes for Microsoft Windows Itanium (64-Bit)	HTML	PDF
Release Notes for Solaris Operating System	HTML	PDF
Release Notes for Solaris Operating System (x86)	HTML	PDF
Release Notes for Solaris Operating System (x86-64)	HTML	PDF
Retail Data Model Release Notes	HTML	PDF

Fig. 11.3 Operating Systems supported by Oracle

One example is SAP HANA, used for hefty analytics, business processes, and predictive analysis this combines columnar data storage (see Chap. 5) with parallel processing and in-memory computing. Another example is Oracle's Times Ten in-memory database which can be used for any time-critical database processes as well as analytics.

The similarity for both is that performance comes as a result of processing entirely in memory. In order to do this they are likely to need large amounts of RAM. The downside to this approach is that RAM is nowhere near as cheap as HDD storage, so these are expensive systems for specialist use.

They have to provide durability and this is achieved by some form of logging transactions to disk, so they are not truly disk-free solutions. However, persistence is probably the most important attribute of a database system, so this can't be avoided.

11.3.4 Operating System (OS)

Modern commercial databases run on a variety of OS platforms. The OS platforms supported by Oracle, for example, can be seen from this screen shot of their documentation page (Fig. 11.3).

The major player which bucks the trend of having multiple versions is Microsoft's SQL Server which runs only on a Windows platform. However, although overall they are behind Oracle on sales, on the sale of databases to run on Windows platforms, Microsoft performs much better.

The perception amongst professionals can to be that the Unix OS was always built for true multitasking and is therefore a more robust starting point for any critical production system. Microsoft, naturally, would disagree with this. In actual fact, all OSs have there own strengths and weaknesses. A database professional may indeed have to work in an environment where there is a mix of server OS systems. To a

degree modern RDBMS systems, once installed and running, make the OS invisible. This is especially so since GUI-based management consoles have come along to replace the script-based maintenance. However, even with GUI management consoles, in order to be more employable, a mixture of OS experience improves any CV.

11.3.5 Database Server Processes

Whatever the OS, the RDBMS has to perform a number of processes in order to enable all the rich functionality available in most modern client/server systems. Some of these may not seem likely to have an impact upon performance, but they can cause problems if ignored or misunderstood by the DBA.

The connection process and how that works for example, may not seem immediately likely to be a cause for performance problems. However, taking Oracle as an example, there are a number of parameters which can be altered, depending upon the connection workload and the types of tasks being undertaken at the client end.

The default is for a client session to be given their own server-side process once the connection is made. This, in effect, reduces the server's available RAM, since a portion is now given over to the client. When you have thousands of concurrent users attaching to your system, this can mean that RAM gets used up and swapping to (slow) disk will occur more regularly.

This one-to-one relationship between client and server is the standard connection type in Oracle. If you create such a connection and then go and make a cup of tea the area of RAM is needlessly occupied when it could be used elsewhere. You can, however, opt for a shared server setting in your start-up parameters. This will mean that a large area of RAM is set aside for shared use by all users, with the expectation that pauses and delays that are a natural part in any SQL process will even out the load across the users.

Another RDBMS background process is a process manager. This constantly checks to make sure that all open connections, with their associated server sessions, are active. Any timed-out sessions are killed by this monitor, since every connection takes some resource, even if it isn't active, thus affecting overall performance.

11.3.6 Archive Manager

As we have seen elsewhere there are often conflicting needs that a DBA needs to balance. Availability vs Performance is one area of regular tension. Writing data to the Redo Log is an essential part of making sure the database is not compromised during any outage and ensuring that the service, when it comes back up, needs minimum manual correction.

In the default mode, a number of redo logs, which are preallocated files, are filled in a round-robin fashion such that when the last disk is filled, the Log Writer writes the next value to the first disk, overwriting what is there.

All of this is fine in a development environment, but in a production database you would normally want to be able to recover a database to any point in time, and to do that you need to store all the redo information. In Oracle you do this by issuing this SQL command:

```
Alter database Archivelog;
```

You need to be aware, however, that this will also slow your system down. Usually, large redo log files provide better performance, but obviously they take up more disk space.

11.3.7 Schema Level: Data Types, Location and Volumes

The whole relational thing of relating connected tables, of normalising to reduce data duplication, was largely a response to the expense of disk storage. At the time when Codd's model was beginning to be used in earnest a "big" HDD was around 1 Mb to 5 Mb and would cost about 1/10th of the average annual wage. Today the average weekly wage would allow you to buy a large HDD and have lots of change! The Per Gb cost of HDD storage has dropped in the same period from thousands of dollars to fractions of a dollar. In short, it made sense to reduce the amount of data being stored.

As HDD have become much less expensive, and able to hold so much more data, this driver has become less important. Whereas most database courses stress the importance of normalisation, practitioners began to realise that the join which enabled related data to be reconstructed in user queries was one of the most time consuming tasks undertaken by the database. De-normalisation—purposefully allowing data to be repeated to avoid joins—became a useful performance tool.

To an extent the new wave of NoSQL databases (see Chap. 5) recognises this. MongoDb, for example, is completely schemaless and allows any old mix of data and datatypes and does not have a mechanism equivalent to normalisation.

With the advent of XML as the de facto medium for data exchange between heterogeneous databases DBA's have had to find ways to store and manipulate XML. Oracle created a new datatype to cope with this. XMLTYPE is an extension to the existing Character Large Object (CLOB) type and it provides functionality to the user, such as XQuery if the data is stored in that datatype.

However, yet again, our DBA needs to know what actually is going to happen to the data when it is stored. If there will be many Xqueries to search and manipulate the data then the performance overhead of using the XMLTYPE might well be justified. However, data loads can be many times slower for the same data being stored as XMLType as compared to CLOB. And Xquery has been tested as slower than SQL in a number of circumstances. Another alternative is to map the incoming XML into relational tables, allowing users to access the data using SQL.

As we can see the decisions taken at design time can have a considerable effect upon performance and these decisions need care.

11.3.8 SQL Optimisation

Imagine you need to get this information from your rdbms:

> Select $a.X, a.Y, b.Z$ from employee a, department b where $a.deptid = b.departmentno$ and $b.departmentno = 22$;

Could you have written this SQL any better? Will this bring back the data you want in the quickest way possible?

The good news is that in most cases you don't need to worry about the performance aspects of your query. The parser will pass the requirements over to something called an Optimiser. This inbuilt process is responsible for, in effect, re-writing any query into the most efficient possible. And it often surprises newcomers to optimisation that there are likely to be hundreds of ways to return the data you are looking for.

In actual fact the optimiser will not re-write your query. It will simply try and work out which access paths are the most efficient in terms of returning the dataset. Naturally it will depend upon the design of your database and the data distribution within it. Some of the most straightforward options before it might include:

- Using an index on table a
- Using an index on table b
- Doing a full table scan on b followed by using an index on a
- Using a full table scan on both

This would get more complicated depending upon:

- Whether there is more than one index to choose from
- Whilst $b.departmentno = 22$ may be likely to return less than 15 % of the rows and therefore be a good candidate for the use of an index, the optimiser needs to estimate the proportion of all rows that will be returned from table a, before it can decide on whether or not to use an index

Once these decisions are taken the optimiser will create a Query Plan. You can usually examine the choices made (in Oracle there is **Explain Plan**—see *Tools* section below). And you can force the database to do something different if you know better (see **Hints** in the *Tools* section).

In order to make sensible decisions, the optimiser needs to do better than just guess. In earlier versions of Oracle the process was one of following rules (Rule-based Optimisation—RBO). At their most basic the rules say things like:

- Only do a full table scan if there is no option
- Use a single row fetch using a Rowid rather than a Primary Key if possible

These rules are simple to implement and therefore the processing involved is relatively trivial. However Oracle moved away from RBO and it has been obsoleted since 10g. The default optimiser is now a cost based one (CBO). The CBO approach is a response to the fact that the RBO does not pay any regard to the type and distribution of data being queried, but rather blindly applies a single set of rules.

CBO works by estimating the resources (primarily disk I/o and cpu usage) that will have to be used for each of the possible access paths and then selects the least "expensive" access method. In order to do this estimation the optimiser must have

an understanding of the data stored. It does this by gathering statistics for every table and storing these in the data dictionary.

The use of statistics is, in its own right, a balancing act for the DBA. The statistics stored are about the size, cardinality and distribution of the data stored. But the statistics are only useful if they are reasonably current, allowing accurate estimates to follow. However, the process of gathering the statistics is, itself, an intensive one and slows the database down. A DBA needs to know how regularly to update the statistics. Of course, if the database isn't used at certain periods, such as weekends, this is a job that can be scheduled to run then. But in a 24/7 environment this is more tricky.

11.3.9 Indexes

Perhaps the most well known, and often over-used performance tool used by database creators is the index. In its most usual form the B-Tree index is fundamentally just the same as the key/value pair that we saw in Chap. 5. It's just that the value being stored against the key is the hex address of the row containing the column or columns being referenced (see Fig. 11.4).

11.3.9.1 B-Tree Indexes

There are many different possible ways to locate a series of rows. If an index is available, and we are searching for a small percentage of the total number of rows in the table, then the RDBMS may choose to look up the key from the index, and then pass the required hex address to the disk reader so that it can retrieve the required row.

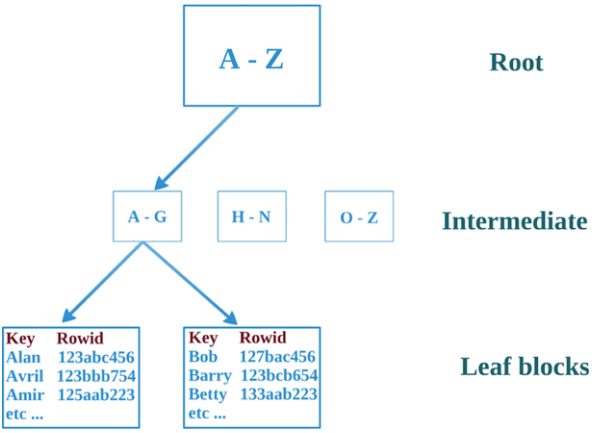
But this needs at least two extra reads (to get to the right leaf node of the index) and more processing than would be required to just read the entire table and disregard unwanted rows (known as a full table scan). In Oracle, if there you are retrieving over 15 % of the rows Oracle will ignore an index, preferring instead the relative simplicity of a full table scan.

Which columns are best suited for an index? The most obvious is the primary key of a table. Most RDBMS will automatically create an index to support your primary key constraint. Oracle does not, however, index a foreign key. Generally foreign keys should be indexed provided the child table is big enough to warrant this. This is because there is an inferred likelihood of some joining going on in queries between tables related in this way.

Small tables should not be indexed. A full table scan is much the speedier way to gather a row from a table containing just a few rows. Indexing would be overkill—even if you were only ever returning one row.

When looking for suitable candidates for secondary indexes, a DBA should be looking for columns that are often involved in selection or join criteria. Yet again, the DBA needs to understand the application the database is supporting. If users are frequently accessing data with a WHERE clause of a non-primary key column, and that query returns fewer than 15 % of the total rows stored, then a secondary index may well help with speeding up reads. However, if that query we have just

Fig. 11.4 B-Tree index



described runs just once a year, whilst the table itself gets written to many times a minute, it may be that the slowing effect during the writes (caused by having to create and update rows in the index and the consequential need to maintain the index's structure) far outweighs the advantage of having an index. Of course, it could be that the once a year report is run by the Managing Director, and therefore the political dynamic changes the decision making process yet again! The DBA's balancing act can be a difficult one!

There are workarounds to the tensions between write- and read-performance effects available to DBAs if they know their system well enough. For example, updating secondary indexes during write operations can slow the entire system due to the extra locking it generates. To avoid this the DBA could store the inserts and updates into a temporary table during the working day, and then apply them all overnight in a single batch job. This is would be unreasonable in a 24/7 operation, but may well work in some circumstances.

Loading large amounts of data can slow the database down. Oracle's bulk loader, SQLLDR, for example, if used in default mode, will generate an insert statement for each row to be loaded. This will generate Redo and Undo and might significantly increase the amount of locking. The workaround may be to delete any indexes before loading, then load the data, then recreate any indexes—preferably at a time when the database is less active anyway. This is another balancing act for the DBA: will the decrease in query response times against the tables when they are without indexes cause sufficiently little overall disruption to justify this approach?

11.3.9.2 Non B-Tree Indexes—Bitmap Indexes

B-tree indexes have been around for decades. Many databases only have this type of index available. Sometimes, however, this form of physical address lookup is not the best. This is particularly true of column data which has low cardinality (very few possible values). Columns containing only values such as Yes/No, for example, or Male/Female, or even colours of the rainbow, are not well suited to B-tree indexes. This is because many valid keys will be stored across potentially many leaf blocks,

which will mean many reads per row returned, and the index leaf blocks may well be well distributed across the tablespace, slowing the read process.

To get around this problem the Bitmap Index type was created. The SQL syntax is the same as for a B-tree standard index, but with the addition of the keyword; Bitmap

```
Create Bitmap Index Gender_idx on Person(Gender) ;
```

In effect the index for each possible key value for this index looks like this:

Male, Start ROWID, End ROWID, 10001101000101000100 10
Female, Start ROWID, End ROWID, 01110010111101011101101

Bitmap indexes use arrays of bits to record whether or not a particular value is present in a row, and answers SQL queries by performing bitwise logic against these arrays. Because these indexes use fast, low level bitwise logic they can significantly reduce query response times and may also reduce storage requirements.

Bitmap indexes can substantially improve the performance of queries in which the individual predicates on low cardinality columns return a large number of rows. It can be especially useful when queries need to use two or more bitmaps to select rows since bitwise logic can rapidly select matches. The downside however, is that the mapping mechanism can be slow during updates and inserts. For this reason bitmap indexes are often used in decision support environments. These environments usually have large amounts of data but few writes.

11.3.9.3 Non B-Tree Indexes—Reverse Key Index

A physical problem that Indexes can cause is that of "hot disk". This is exactly what it sounds like. If the disk head is constantly accessing a particular sector of the disk, perhaps because lots of similar key values are being written into the same index leaf block, then the HDD's capacity to execute I/O will be exceeded which can cause performance problems as users wait for their turn to get to that area of disk.

As an example, think of a relatively straightforward concept of creating a composite index on three columns: date, branch_id and saleperson_id. The scenario is that this is a chain of stores recording its sales information and to speed the monthly reports this index is created.

On Thursday 15th Sept 2013, Store 44, during a busy period, sends this series of rows to the server.

Transaction_ID	143434
Date	15092013
Store	44
Salesperson	11
Item_id	732
Qty	4

Transaction_ID	143435
Date	15092013
Store	44

Salesperson	11
Item_id	861
Qty	1

and so on.

Perhaps during a very busy period all their sales assistants are sending very similar information.

In the space of a few seconds the server may have to write the following leaf key value pairs:

150920134411	hex address of the row with this composite value
150920134412	hex address of the row with this composite value
150920134421	hex address of the row with this composite value
150920134411	hex address of the row with this composite value
150920134405	hex address of the row with this composite value
150920134411	hex address of the row with this composite value
150920134411	hex address of the row with this composite value
150920134407	hex address of the row with this composite value
150920134412	hex address of the row with this composite value

The point here is that all these keys are so similar that they will end up being written to the same leaf block. This may mean some waiting occurs as each key/value takes its turn to take a lock on the leaf so they can write to it, and, even worse, because the head is virtually constantly over the same physical spot on the disk, we may end up with hot disk.

Oracle came up with a solution to this problem called the Reverse key index. A simple algorithm works to transpose what the application wants to write to the system (using the first row in the above example: 150920134411) and reverses it at the point of storage, in this case to: 114431029051. The benefit becomes clear when we look at what happens to the next key value pair: 214431029051. This is significantly different and will very probably need to be written to a different branch/leaf, reducing contention for blocks and minimising the likelihood of hot disk.

This is clearly a specialist solution. The reverse algorithm has to apply when applications try and retrieve data, and there will have to be an overhead in terms of performance for applying this process to every read and write. However, it is probably much better than destroying your HDD!

11.3.9.4 Non B-Tree Indexes—Function-Based Indexes

Sometimes we have occasions when we are not sure what a user is going to input—all we know is that they are always right! Think of an employee table containing a column called last_name which USER A inserts to:

Insert into Employee (*id, last_name*) *values* (*12, 'Jones'*);

Meanwhile User B inserts:

Insert into Employee (*id, last_name*) *values* (*12, 'JONES'*);

What would USER C expect to see if he issues this:

Select ∗ from Employee where last_name = 'Jones';

The answer is that they would not see the row entered by User B.

One solution would be to put a CHECK constraint on the column to ensure all names are inserted in uppercase. However, this would force McEwan to be written as MCEWAN, and this may upset the name's owner.

However, if we do not have a CHECK constraint any mix of case will be allowed, and as the Jones example shows, that leaves us with a problem when attempting to find all people called Jones.

Oracle provide a solution to this, and other problems, by allowing us to create an index which is made from altered source data, whilst not actually changing the source itself. The syntax is not dissimilar to creating an ordinary index. In this example we turn the value in last_name to uppercase before it is stored in the index.

CREATE INDEX upperlastname_idx ON employees (*UPPER*(*last_name*));

This will sort the 'Jones' problem for us without us having to change the column data.

11.3.9.5 Indexes—Final Thoughts

As with other elements of tuning, index use needs ongoing review. Reports are available which show how well used indexes are. If you discover an index is rarely used that index is a candidate for dropping since the overhead cost of keeping it active is generating no reward.

The distribution of the actual data which is being indexed may also change significantly over time. A small table, initial deemed unworthy of an index, may, for example, grow into a large table, which would benefit from an index. Or a column which the designers thought would contain many different values may, it turns out, contain very few, making it worth considering for a bitmap index.

11.3.10 Network

There is often little the DBA can do about network speed. Database clients are often distant, or sometimes very distant from the server and this means there are many places where bottlenecks on the network layer can happen. A DBA needs to work closely with his networking colleagues.

There are, however, a few things a DBA can affect in terms of networking performance. Oracle, for example, encapsulates data into buffers sized according to the DEFAULT_SDU_SIZE parameter. (SDU is *session data unit*.) The default is for this size to be 8192 bytes or less. However, if your application is typically bringing back bigger chunks of data you can consider making this larger so that fewer packages need to be sent on the network for each query.

11.3.11 Application

11.3.11.1 Design with Performance in Mind

Without pointing an accusatory finger at developers, it is true that many database systems are written by developers who know SQL well but don't understand the physical aspects of database design. Many IDEs have SQL generators allowing complex joins to be written by merely drag-and-dropping. Worse, the developer can fall into the trap of only worrying about system outputs. This can result in excessive use of indexes, with the consequent hit on write performance.

One sensible solution, if the organisation can afford it, is to include DBAs in a development team. They can ensure that the design process is implementation focused, and not merely "well designed". There are a number of less well known structures available for specialist use which can speed query responses. But because they are non-standard they often get overlooked.

Indexed Organised Table (IOT) If a table is very frequently, or solely searched via the primary key it is a candidate for being stored as an IOT. In effect the table information gets stored in an B-tree-like index structure, which means that searches for a particular primary key will be faster than a standard table+index structure since there is no need to look up the rowid from the index to gather the row data as it is stored in the leaf block itself. Unlike an ordinary table where data is stored as an unordered collection, data for an index-organized table is stored in a primary key order so range access by the primary key involves minimum block accesses. Because of this storage mechanism secondary indexes become difficult to maintain and are probably best avoided.

Wide tables are not ideal candidates, although there is an ability to overflow data that is seldom required into a separate, non-b-tree area. The syntax for creating an IOT is the same as for an ordinary table with the exception of the ORGANISATION INDEX clause. There has to be a Primary Key. Here is an example:

```
CREATE TABLE WardReqIOT
(
    WardID NUMBER,
    RequestDate DATE,
    Grade NUMBER,
    QtyReq NUMBER,
    AuthorisedBy VARCHAR2(30),
    CONSTRAINT PK_WardReqIOT PRIMARY KEY
(WardID, RequestDate, Grade)
)
ORGANIZATION INDEX INCLUDING QtyReq OVERFLOW Tablespace Users
```

Here all the columns up to and including the one named in the INCLUDING option of the OVERFLOW clause (QtyReq) are stored in primary key order in a B-tree structure. The other data is stored in a table structure in a tablespace called Users.

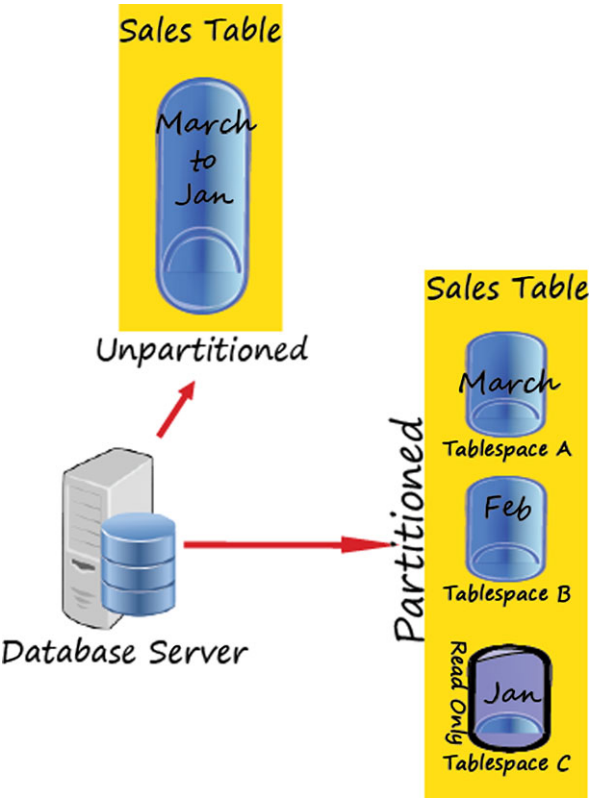


Fig. 11.5 Partitioning data across disks

Partitioning Oracle can manage very large tables comprising many millions of rows. However, performing maintenance operations on such large tables is difficult, particularly in a 24/7 environment. There can be performance issues associated with inserting into, or reading from tables this large. Indexes can help, of course, in reducing the number of full table scans required, but another solution might be to partition the data (see Fig. 11.5).

A Partition will be assigned according to defined criteria for a key column(s). Two examples are:

Range Slices the data according to which range a partitioning key is in. For example you could partition rows where the Transaction_Date column contains data from different months, with partitions for January, February and so on.

List Instead of a range you can partition by creating a list of values. For example all rows where the column Location is any of: Edinburgh, Glasgow, Montrose, or Inverness could be in a partition called Scotland.

Depending upon the type of data being kept rows may be divided into a series of key ranges, such as ones based on date. Although each partition will have the same logical structure as the others, they may have different physical storage properties.

Data may be stored in different tablespaces, or even disks, enabling a sort of striping potentially speeding up data access by allowing parallel reading to occur.

Furthermore, if the historic data is unlikely to be updated, the majority of the partitions in a history table can be stored in a read-only tablespace (good for performance). This approach can also reduce back-up times.

11.4 Tuning

11.4.1 What Is Database Tuning?

There are several steps in the process of ensuring a database is performing adequately. This is often called ***tuning***. Different professionals have different thoughts about what the process should be. Is it an Art or a Science?

At its most abstract, however, the steps will usually be:

1. Monitor
2. Analyse
3. Select the appropriate type of action, which could be any or all of:
   (a) Make changes to the hardware, usually by adding RAM or disk
   (b) Alter the RDBMS parameters to allocate resources more efficiently
   (c) Modify the SQL to run more efficiently.
   (d) Change the application and the way it interacts with the RDBMS
4. Go back to step 1 to assess the success of the actions taken

Ideally this should be an ongoing process. As we have already highlighted, the data that is loaded today may well have made the database less well tuned. Sometimes production systems start off performing well. Any slowing that happens may happen slowly over a number of months. This can make the analysis phase much lengthier. However, in a busy DBA section, when your users aren't complaining, it is very easy to loose sight of the need to constantly monitor.

Modern RDBMS can help (see **Tools** section below). They have inbuilt performance monitoring and can flash alert on the DBA dashboards when thing begin to go wrong. There may well be situations, however, where there is no forewarning. The DBA who receives a call from some irate user saying their report is taking ages to complete needs to be able to review many possible causes very quickly:

System-Wide What else is happening? Is there a large batch job running now which doesn't usually run? Is a tablespace offline because it is being backed-up or moved to a new device? Is the network functioning normally?

Application-Specific Can we see which SQL is causing the problem? It may not be that belonging to the report in question… it could be that the report is waiting because some rows have been locked by another user. Is the report SQL suitably tuned? Does the client process any of the returned dataset, and if so, what else is happening at the client end?

11.4.2 Benchmarking

Especially when it comes time to decide upon which RDBMS to purchase, people begin to ask questions like: which is the best database for performance? But we have seen throughout this chapter that database performance is a slippery thing. The simple act of inserting data to the most finely tuned database tends to worsen its performance.

The truth of the matter is that all the databases currently on the market have their own strengths and weaknesses and some perform better in some ways, whilst others perform better in other ways. That isn't, however, to say that we simply use a pin to decide—nor allow ourselves to be beguiled by the smoothest sales pitch!

A series of realistic tests is a sensible approach, using examples of each possible alternative database to run some realistic processes against the sort of data your application will be collecting. This is a sensible approach if you have the time, and the skills available to set up suitable test scenarios. However, this isn't always possible and you may need to rely on data gathered by others.

It almost goes without saying that test results provided by vendors are to be treated with a degree of scepticism. What is really needed is a fair, standard set of tests against a known dataset. This is what is used in benchmarking.

Probably the most well known impartial agency for this sort of thing is the Transaction Processing Council (TPC). On their website (http://www.tpc.org/) there are the results of a number of different tests against different types of data. Their mission is declared as:

> The TPC is a non-profit corporation founded to define transaction processing and database benchmarks and to disseminate objective, verifiable TPC performance data to the industry.

An output from one of their tests at the time of writing is given in Fig. 11.6.

11.4.3 Another Perspective

As we saw in Chap. 5 we have, in more recent times, been re-examining some of the assumptions we have worked with since the advent of the client/server RDBMS. Indeed, it is sensible for any IT professional to constantly review the current appropriateness of a technology. This is true of database tuning too.

In their paper, Rethinking Cost and Performance of Database Systems, Florescu and Kossmann (2009) argue that the traditional question, which they saw as:

> "Given a set of machines, try to minimize the response time of each request."

should be reshaped to be:

> "Given a response time goal for each request, try to minimize the number of machines (i.e., cost in $)."

Of course minimising cost has always been a major objective for any DBA, but recasting the tuning question to provide this focus helps us recognise what our priorities need to be. This is especially useful in the era of cheap computing grids and

Rank	Company	System	Performance (tpmC)	Price/tpmC	Watts/KtpmC	System Availability	Database
1	ORACLE	SPARC SuperCluster with T3-4 Servers	30,249,688	1.01 USD	NR	06/01/11	Oracle Database 11g R2 Enterprise Edition w/RAC w/Partitioning
2	IBM	IBM Power 780 Server Model 9179-MHB	10,366,254	1.38 USD	NR	10/13/10	IBM DB2 9.7
3	ORACLE	Sun SPARC Enterprise T5440 Server Cluster	7,646,486	2.36 USD	NR	03/19/10	Oracle Database 11g Enterprise Edition w/RAC w/Partitioning
4	IBM	IBM Power 595 Server Model 9119-FHA	6,085,166	2.81 USD	NR	12/10/08	IBM DB2 9.5
***	BULL	Bull Escala PL6460R	6,085,166	2.81 USD	NR	12/15/08	IBM DB2 9.5
5	ORACLE	Sun Server X2-8	5,055,888	.89 USD	NR	07/10/12	Oracle Database 11g R2 Enterprise Edition w/Partitioning
6	hp invent	HP Integrity Superdome-Itanium2/1.6GHz/24MB iL3	4,092,799	2.93 USD	NR	08/06/07	Oracle Database 10g R2 Enterprise Edition w/Partitioning
7	IBM	IBM System p5 595	4,033,378	2.97 USD	NR	01/22/07	IBM DB2 9
8	IBM	IBM eServer p5 595	3,210,540	5.07 USD	NR	05/14/05	IBM DB2 UDB 8.2

Fig. 11.6 TCP test results

the ultimate flexibility offered by Cloud provisioning. Indeed, in the era of cloud database perhaps their question could be reworked again to:

"Given a response time goal for each request, try to minimize the cost (in $) of provision."

11.5 Tools

11.5.1 Tuning and Performance Tools

In this section we have a tutorial exploring some of the tools that are available to help tuning the database. We are using Oracle as an example. In order for this to work you need to have admin rights on an Oracle 10g or 11g instance with the sample HR schema installed (which is there by default unless you ask not to have it).

In order to give ourselves some meaningful data to experiment with we will also use a further table called EmpHours which should have about 15 Mb of data in it and will therefore make timings improvements more noticeable than with smaller tables. Details about how to create the table and generate these rows are in at the end of the chapter.

11.5.1.1 SQLPLUS Tools

For years Oracle has shipped with a command line tool called SQLPLUS. It is part of Oracle whatever the Operating System which means that a DBA who masters it can ply his or her trade on any platform.

It isn't a friendly environment, being invented before the GUI environments became the norm, but it is very functional, and has some built-in tools that can assist with basic performance tuning.

Timing When users say things like; "that sql report took a long time to run" there are a number of responses a DBA could give—some of which are not polite! But the good DBA will know that what they need is hard, cold measurement. In this case, measurement of the time it takes to get the data back to the user.

From your Oracle working directory, log in as HR and select all the rows from the Departments table where Department_id is less than 100 (Fig. 11.7).

How quickly did that answer come back? Chances are it will have been so quick you might say it responded instantaneously. We can find out how long the query took by using the built-in facility within SQLPLUS to time events. If we issue the ***set timing on*** command and then re-run the query we now get some information. Don't worry that you don't get any message back after issuing the command. You will get an error if you issue a bad command (Fig. 11.8).

Yes, almost instantaneous: 0.01 of a second! But, we didn't really stretch the optimiser with that query, so let's try something with a join in it that runs against our big table: EmpHours. The ***set timing o***n will remain the default action now whilst the session lasts, or you issue the ***set timing off*** command.

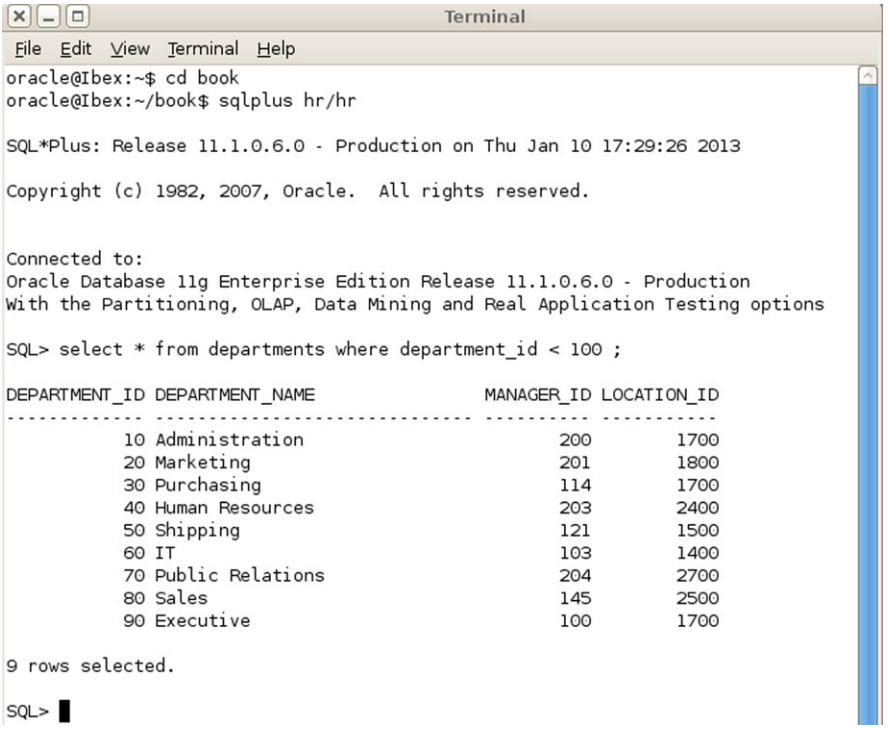


Fig. 11.7 SQL query where department_id < 100

Try running this query:

```
select a.employee_id, a.last_name, b.work_date, b.hours
from employees a, emphours b
where b.emp_id < 105 AND b.emp_id = a.employee_id AND b.fee_earning
= 'Y';
```

Use a text editor to type this in, or copy it in the save it in your working folder and call it something like q1.sql

On the author's server this takes over 3 seconds (Fig. 11.9).

Now run the same query immediately again (Fig. 11.10).

It may not have looked any quicker, but it is almost half a second quicker. Was yours the same? I would guess it would be. If you remember we said reading from disk was a bottleneck. The first time the query ran Oracle will have had to gather the data from the disk. The results would be kept in RAM until they were flushed out by other data. So the second time we asked the question Oracle could get the answer from memory, saving disk access time. **The rule then must be to always**

Fig. 11.8 Turning timing on

```
SQL> set timing on
SQL> select * from departments where department_id < 100 ;

DEPARTMENT_ID DEPARTMENT_NAME                MANAGER_ID LOCATION_ID
------------- ------------------------------ ---------- -----------
           10 Administration                        200        1700
           20 Marketing                             201        1800
           30 Purchasing                            114        1700
           40 Human Resources                       203        2400
           50 Shipping                              121        1500
           60 IT                                    103        1400
           70 Public Relations                      204        2700
           80 Sales                                 145        2500
           90 Executive                             100        1700

9 rows selected.

Elapsed: 00:00:00.01
SQL>
```

Fig. 11.9 Test output showing elapsed time

```
EMPLOYEE_ID LAST_NAME                  WORK_DATE      HOURS
----------- -------------------------- --------- ----------
        104 Ernst                      04-JAN-13         27
        104 Ernst                      08-JAN-13         25
        104 Ernst                      10-JAN-13         45

21464 rows selected.

Elapsed: 00:00:03.21
SQL>
```

Fig. 11.10 Second run—usually will be quicker

```
EMPLOYEE_ID LAST_NAME                  WORK_DATE      HOURS
----------- -------------------------- --------- ----------
        104 Ernst                      04-JAN-13         27
        104 Ernst                      08-JAN-13         25
        104 Ernst                      10-JAN-13         45

21464 rows selected.

Elapsed: 00:00:02.74
SQL>
```

run tests several times to get an average, and always exclude the first time from that calculation.

Autotrace Statistics Let's now use another SQLPLUS tool: ***set autotrace on***.

In order to use this as HR we will need to create the PLUSTRACE role as SYSDBA and then grant its privileges to HR. Here are those steps to be carried out as SYSDBA (Fig. 11.11).

Now, after moving back to your working directory and logging on to SQLPLUS as HR, we can issue the command; **set autotrace on.** As this generates a report that is wider than the default line width we also need to issue the **set linesize 200** command to allow for the output to format nicely.

Now re-run the q1 query and see what we get. There is a lot of extra information now appended to the end of the output. We will examine what the key items are, but first, just to emphasise the point about disk reads being slow, just note the statistics section below which was run after restarting Oracle (and therefore emptying the buffers) (Fig. 11.12).

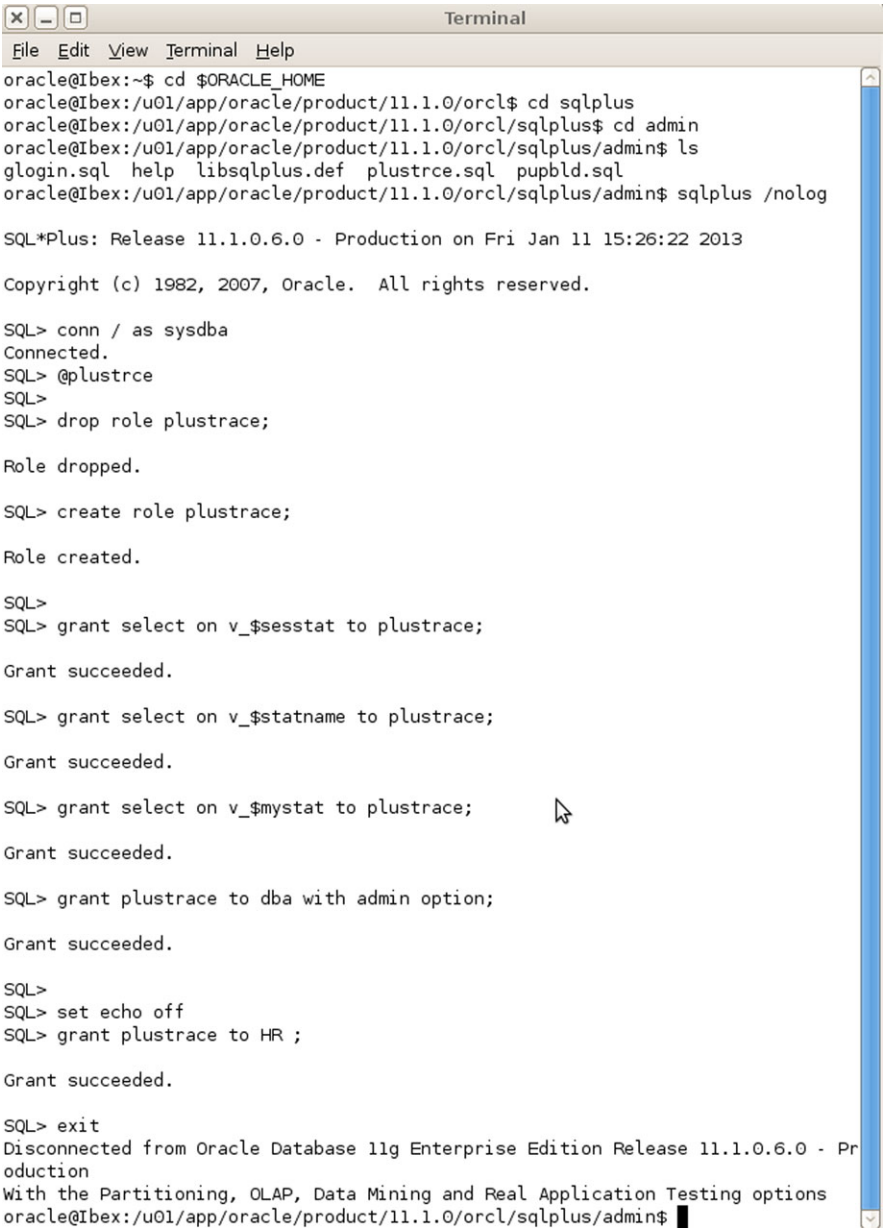


Fig. 11.11 Setting up autotrace

Now, unless you restarted your instance, what you would be more likely to get is shown in Fig. 11.13.

Fig. 11.12 Autotrace first time through

```
Statistics
----------------------------------------------------------
        447  recursive calls
          0  db block gets
       3221  consistent gets
        175  physical reads
          0  redo size
     604747  bytes sent via SQL*Net to client
      16150  bytes received via SQL*Net from client
       1432  SQL*Net roundtrips to/from client
         10  sorts (memory)
          0  sorts (disk)
      21464  rows processed

SQL>
```

Fig. 11.13 Autotrace with no physical reads

```
Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
       3050  consistent gets
          0  physical reads
          0  redo size
     604542  bytes sent via SQL*Net to client
      16150  bytes received via SQL*Net from client
       1432  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      21464  rows processed

SQL>
```

Note that one line of the Statistics report shows how many **physical reads** there have been. This tells you how many blocks Oracle has had to read from disk in order to be able to answer the query. We can see that the first time we needed 175 block to be read from disk, but the second time we needed none, since all rows were already in the buffer (i.e. in RAM).

When looking for tuning problems, looking to reduce disk reads is an obvious place to start. The point here, however, is that the first time you run a query is not a sensible time to look at tuning data since, in the normal course of events, an active production system is likely to be able to service some, or all, data requests from memory. To reiterate the rule:

> always run tests several times to get an average, and always exclude the first time from that calculation.

Now we will review the other useful information presented when we turn on Autotrace. Firstly, continuing in the Statistics area we have, we can particularly note:

- **recursive calls**: these are internal SQL statements that Oracle needs to issue to service the request. There isn't much you can do about this. Note, in the example above, that the first time through this number was a lot higher. This is probably because of the extra parsing work that Oracle will have had to carry out

- **consistent gets**: this is probably the most useful item to keep an eye on when tuning. It is the total number of blocks Oracle needed to read from disk and memory to service the query. The bigger the number, in general, the slower the query.
- **sorts**: Sometimes you can't avoid having data sorted by Oracle, but sorts can slow the query considerably. You could try rewriting queries to reduce sorts if you know the client will be processing the data. In any case we should aim for memory sorts rather than disk sorts.

We are now going to be playing around with the same query to demonstrate the effect of changes. However, we need to remember our rule about never trusting the first run of a query when we make changes to queries since the second time the revised query runs is likely to be faster than the first time.

In this case let us add an ORDER BY to the query so that our output is sorted by hours worked and then by date worked. Try saving this as q2.sql and then run it twice:

```
select a.employee_id, a.last_name, b.work_date, b.hours
from employees a, emphours b
where b.emp_id < 105 AND b.emp_id = a.employee_id AND b.fee_earning
  = 'Y'
ORDER BY b.hours, b.work_date;
```

The second time, this ran on the author's server the run time returned to just a little longer at 2.82 seconds as compared with 2.22 for q1. The thing that has changed in the Statistics section is the Sorts (memory) which is now 1. Because the rows were already in RAM from the previous query, Oracle could carry out the sort in one operation, all in memory—hence the only slight performance hit of 0.6 of a second.

The query generates many rows which makes comparing Statistics difficult so, since we know what the query output looks like, we can turn off query row output by issuing **set autotrace traceonly**. Once you have issued that command, before running the next query, run q2 one more time so we have the information we need to compare with the next query. But now look at the time to run. In the author's case it has shrunk from 2.22 secs to a mere 0.22. Why is this? Because writing the results to the screen takes time. The traceonly results are the times for the answers to be generated rather than delivered and this is fairer since writing output will take different times on different platforms.

To give SQL more rows to work with we can now remove the b.emp_id predicate to see what effect that has. Try saving this as q3.sql and then run it twice (see Fig. 11.14).

```
SQL> @q3

313015 rows selected.

Elapsed: 00:00:02.02

Execution Plan
----------------------------------------------------------
Plan hash value: 2217401640

-------------------------------------------------------------------------------------------
| Id  | Operation           | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |   315K|  8625K|       |  2851   (2)| 00:00:35 |
|   1 |  SORT ORDER BY      |           |   315K|  8625K|    24M|  2851   (2)| 00:00:35 |
|*  2 |   HASH JOIN         |           |   315K|  8625K|       |   389   (3)| 00:00:05 |
|   3 |    TABLE ACCESS FULL| EMPLOYEES |   107 |  1284 |       |     3   (0)| 00:00:01 |
|*  4 |    TABLE ACCESS FULL| EMPHOURS  |   315K|  4929K|       |   383   (3)| 00:00:05 |
-------------------------------------------------------------------------------------------

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

   2 - access("B"."EMP_ID"="A"."EMPLOYEE_ID")
   4 - filter("B"."FEE_EARNING"='Y')


Statistics
----------------------------------------------------------
         0  recursive calls
         0  db block gets
      1332  consistent gets
         0  physical reads
         0  redo size
  10134177  bytes sent via SQL*Net to client
    229957  bytes received via SQL*Net from client
     20869  SQL*Net roundtrips to/from client
         1  sorts (memory)
         0  sorts (disk)
    313015  rows processed

SQL>
```

Fig. 11.14 Screenshots of the output for q3.sql

```
select a.employee_id, a.last_name, b.work_date, b.hours
from employees a, emphours b
where b.emp_id = a.employee_id AND b.fee_earning = 'Y'
ORDER BY b.hours, b.work_date;
```

As usual, the first run through would probably be slower. On the author's server the output from the second run looked like in Fig. 11.14.

We can note that we now have 313015 rows returned as compared to 21464 in q2. So the report is returning about 14 times more rows. And then if we compare **bytes sent via SQL*Net to client** 10134177 for q3 and 604542 for q2 we see that we are sending nearly 17 times more data. However, look at the comparison in runtime: 2.02 for q3 and 0.22 for q2. The response is only 9 times slower.

```
SQL> @q4

no rows selected

Elapsed: 00:00:00.05

Execution Plan
----------------------------------------------------------
Plan hash value: 2217401640

-------------------------------------------------------------------------------------------
| Id  | Operation           | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           | 59282 |  1620K|       |   851   (2)| 00:00:11 |
|   1 |  SORT ORDER BY      |           | 59282 |  1620K|  4664K|   851   (2)| 00:00:11 |
|*  2 |   HASH JOIN         |           | 59282 |  1620K|       |   387   (3)| 00:00:05 |
|   3 |    TABLE ACCESS FULL| EMPLOYEES |   107 |  1284 |       |     3   (0)| 00:00:01 |
|*  4 |    TABLE ACCESS FULL| EMPHOURS  | 59282 |   926K|       |   383   (3)| 00:00:05 |
-------------------------------------------------------------------------------------------

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

   2 - access("B"."EMP_ID"="A"."EMPLOYEE_ID")
   4 - filter("B"."FEE_EARNING"='Q')


Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       1332  consistent gets
          0  physical reads
          0  redo size
        460  bytes sent via SQL*Net to client
        409  bytes received via SQL*Net from client
          1  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          0  rows processed

SQL>
```

Fig. 11.15 Output from q4.sql demonstrating overheads

So another useful thing to remember is that the volume of data is not the only factor in the speed of a query, although it will plainly have an effect. There are some processes that will need to happen only once, or a few times only, in the overall task, regardless of the number of rows being returned. Parsing, for example, is not dependant upon likely row count, and it must happen for every query. These "overheads" can be demonstrated by creating q4 as shown in Fig. 11.15. Note, there are no rows with a fee-earning value of 'Q'.

```
select a.employee_id, a.last_name, b.work_date, b.hours
from employees a, emphours b
where b.emp_id = a.employee_id AND b.fee_earning = 'Q'
ORDER BY b.hours, b.work_date;
```

```
Execution Plan
----------------------------------------------------------
Plan hash value: 700587718

-------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |   483 | 12075 |    73   (2)| 00:00:01 |
|   1 |  SORT ORDER BY                |               |   483 | 12075 |    73   (2)| 00:00:01 |
|   2 |   NESTED LOOPS                |               |       |       |            |          |
|   3 |    NESTED LOOPS               |               |   483 | 12075 |    72   (0)| 00:00:01 |
|   4 |     TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     5 |    60 |     2   (0)| 00:00:01 |
|*  5 |      INDEX RANGE SCAN         | EMP_EMP_ID_PK |     5 |       |     1   (0)| 00:00:01 |
|*  6 |     INDEX RANGE SCAN          | SYS_C0010205  |   191 |       |    13   (0)| 00:00:01 |
|*  7 |    TABLE ACCESS BY INDEX ROWID | EMPHOURS      |    96 |  1248 |    14   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

Fig. 11.16 Execution plan for q2.sql

Despite not returning a single row this process took 0.05 of a second and needed to read 1332 blocks.

Autotrace Execution Plan The other significant section in the autotrace outputs is the execution plan. We will use q2 and q3 again so either find the output, or run them again.

Execution Plan for Q2 is shown in Fig. 11.16.

The first thing to mention about these plans is that you need to read them from the inside out. The Operation column tells you what Oracle will be doing to deliver the required dataset. The name tells you any objects Oracle will be using. It will help to note that SYS_C0010205 is the system generated index to support the primary key in EMPHOURS, whilst EMP_EMP_ID_PK is the index used to support the primary key in EMPLOYEES.

The columns to the right of Name are all estimates that Oracle generates, based upon the statistics in the Data Dictionary. Cost is how Oracle Cost-Based Optimiser decides which plan to use since lowest cost will typically also be most efficient in terms of resource usage. The cost estimates the IO, CPU, and network resources that will be used to answer the query.

Start reading this plan with the one that has the most indentation as this will probably be the first one to be executed. (If two statements have the same level of indentation read top down.)

In this case we have an Index Range Scan using the EMP_EMP_ID_PK index. This means that Oracle will read that index to get the ROWIDS for what it estimates is 5 rows. (It is right, isn't it? Ids 100–104 meet the query WHERE b.emp_id < 105).

Operation ID 4 and 6 are equally nested, so next comes the earlier operation; **Table Access by Index Rowid**. Oracle uses the ROWID to read the block(s) that contain the Employee data it needs for the query. It estimates that will be around 60 bytes.

Next comes the **Index Range Scan** on the EMPHOURS primary key index. It will use the emp_id number discovered in operation 4 to lookup the ROWID from the index of all the rows in EMPHOURS that have this emp_id. Finally Oracle can access EMPHOURS and return the rows selected by Operation 6.

```
Execution Plan
----------------------------------------------------------
Plan hash value: 2217401640

-------------------------------------------------------------------------------------------
| Id  | Operation           | Name      | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |   217K|  4874K|       |  1879   (2)| 00:00:23 |
|   1 |  SORT ORDER BY      |           |   217K|  4874K|    15M|  1879   (2)| 00:00:23 |
|*  2 |   HASH JOIN         |           |   217K|  4874K|       |   388   (3)| 00:00:05 |
|   3 |    TABLE ACCESS FULL| EMPLOYEES |   107 |  1070 |       |     3   (0)| 00:00:01 |
|*  4 |    TABLE ACCESS FULL| EMPHOURS  |   217K|  2755K|       |   383   (3)| 00:00:05 |
-------------------------------------------------------------------------------------------
```

Fig. 11.17 Revised execution plan output

These operations are in nested loops. First all the rows from Employees are returned and then EMPHOURS index information each row is returned from Employees. Once completed Oracle can sort the data in memory.

Operation 0 is the finished output. Note how poor the estimated number of rows is. Oracle doesn't know how many EMPHOURS rows are likely to be returned for each EMPLOYEE row.

To see the optimiser use an alternative approach to getting similar data, let's now run q3—which removes the emp_id < 105 predicate and generates many more rows (Fig. 11.17).

Because we are returning more than 15 % of the rows from both tables Oracle now decides that it will not use indexes. Instead it will read all the rows from the two tables straight into memory in an operation called a Full Table Scan (here described as TABLE ACCESS FULL). Thus two sets of rows are then joined on the join key.

Two common means of joining are:

- **Merge Join**: Outputs from both scans are sorted by the join key and then merged together
- **Hash Join**: A hash join iterates through the rows of the smaller table and performs a hash algorithm on the columns for the joined columns and then stores the result. It then iterates through the rows of the other table performing the same hashing algorithm on the joined columns. It then compares with the first result and if they match it returns the row.

Once the rows are output Oracle sorts. Because it has estimated how many rows will be output it also suggests it will need 15 Mb of Temporary Space to perform the sort. Note that the estimated row count is a lot more accurate this time.

What we sometimes need to do when testing is to run several scripts and then want to compare the outputs, saving them for future reference. We can use the SQLPLUS Spool command to send output to a file. We also need to close the file when we have finished recording. Before we run this test script it is a good idea to make sure that the data dictionary has up-to-date statistics. To force this you can issue this command:

```
analyze table emphours compute statistics;
```

We shall now create such a script, run an SQL query twice, then create an index and then run the same query twice again to see what difference it makes to the Execution plan. The query returns employees who have recorded more than 40 hours which was non-fee-paying in the previous 30 days.

Save the follow to q6.sql and then run it, after making sure you understand what the script is doing.

```
set timing on
set autotrace traceonly
set linesize 200
spool test.txt

drop index hrs_idx;
select a.employee_id, a.last_name, b.work_date, b.hours from employees a,
emphours b where b.emp_id = a.employee_id AND b.fee_earning = 'N'
AND hours > 40 AND work_date > (SYSDATE -30);
select a.employee_id, a.last_name, b.work_date, b.hours from employees a,
emphours b where b.emp_id = a.employee_id AND b.fee_earning = 'N'
AND hours > 40 AND work_date > (SYSDATE -30);

create index hrs_idx on EMPHOURS(hours);

select a.employee_id, a.last_name, b.work_date, b.hours from employees a,
emphours b where b.emp_id = a.employee_id AND b.fee_earning = 'N'
AND hours > 40 AND work_date > (SYSDATE -30);
select a.employee_id, a.last_name, b.work_date, b.hours from employees a,
emphours b where b.emp_id = a.employee_id AND b.fee_earning = 'N'
AND hours > 40 AND work_date > (SYSDATE -30);

spool off
```

You can now read the results by opening the file **test.txt** in your working directory.

Our developers have said that they think that there will be several queries, like this one, which use the hours field to select and filter on. So, they have suggested, we should have an index on that column. Sounds reasonable?

Well if we look at the result we can see this isn't such a good idea. With or without an index on the Hours column Oracle uses the same Execution Plan (Fig. 11.18).

In other words, the Primary Key is used (at Id3) to filter on date first and there is no need for Oracle to use the new index.

The bad news is that having that index sitting there doing nothing is costing the system performance for every insert and update since it needlessly continues to maintain the index. So not a good idea after all.

Since we are filtering on the Y/N field you could try the same script again, but this time create a bitmap index on the fee_earning column:

```
Execution Plan
----------------------------------------------------------
Plan hash value: 1656800177

-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |    22 |   506 |   114   (1)| 00:00:02 |
|*  1 |  HASH JOIN                   |              |    22 |   506 |   114   (1)| 00:00:02 |
|*  2 |   TABLE ACCESS BY INDEX ROWID| EMPHOURS     |    22 |   286 |   110   (0)| 00:00:02 |
|*  3 |    INDEX SKIP SCAN           | SYS_C0010205 |    65 |       |   109   (0)| 00:00:02 |
|   4 |   TABLE ACCESS FULL          | EMPLOYEES    |   107 |  1070 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

Fig. 11.18 Execution plan showing the index is not used

create bitmap index fee_bit_idx on EMPHOURS(fee_earning);

What did this last experiment tell you?

We now know indexes aren't always a good thing. So when would we use them? Here is advice from the Oracle Tuning Guide:

> (You should...) index keys that have high selectivity. The selectivity of an index is the percentage of rows in a table having the same value for the indexed key. An index's selectivity is optimal if few rows have the same value.

The problem with indexing hours is that it has low selectivity. Prove this by running the following query:

```
select hours, count(hours) from emphours group by hours order by hours;
```

Actually, if we had bothered to understand our data better, we could have guessed that 30 different values which were randomly allocated for 400000+ rows are likely to generate many rows with the same value.

The reverse case, when we have low cardinality data, calls out for a bitmap index, but only really if the data is non-volatile.

To see when Oracle might use an index, have a look at this slightly reworked query from q6.sql above:

```
select a.employee_id, a.last_name, b.work_date, b.hours from employees a,
emphours b where b.emp_id = a.employee_id AND b.fee_earning = 'N'
AND hours > 40 AND work_date > (SYSDATE -30) AND a.last_name =
'Kumar';
```

```
-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |              |     2 |    46 |     5   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                 |              |       |       |            |          |
|   2 |   NESTED LOOPS                |              |     2 |    46 |     5   (0)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |     1 |    10 |     2   (0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN          | EMP_NAME_IX  |     1 |       |     1   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN           | SYS_C0010205 |     6 |       |     2   (0)| 00:00:01 |
|*  6 |   TABLE ACCESS BY INDEX ROWID | EMPHOURS     |     2 |    26 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Fig. 11.19 Execution plan using an index to get at a single row

We are now asking for the same information for only one particular employee. The Execution plan now uses indexes only—no table scans (Fig. 11.19).

To prove how useful bitmap indexes can be, let us assume the **emphours** table is a read-only table within a data warehouse. You have to write a report that lists all rows where no fee has been earned and yet the employee has claimed 50 hours.

The query might now look like this:

```
select a.employee_id, a.last_name, b.work_date, b.hours from employees a,
emphours b where b.emp_id = a.employee_id AND b.fee_earning = 'N'
AND hours = 50;
```

We can note that two columns which are in the WHERE clause are low cardinality, and the data is not volatile, so bitmap indexes might help. Placing this within our test template in the same way that we did with **q6.sql**, we get:

```
set timing on
set autotrace traceonly
set linesize 200
spool test2.txt

drop index hrs_bit_idx;
drop index fee_bit_idx;

select a.employee_id, a.last_name, b.work_date, b.hours from employees a,
emphours b where b.emp_id = a.employee_id AND b.fee_earning = 'N'
AND hours = 50;
select a.employee_id, a.last_name, b.work_date, b.hours from employees a,
emphours b where b.emp_id = a.employee_id AND b.fee_earning = 'N'
AND hours = 50;
```

```
Elapsed: 00:00:00.08

Execution Plan
----------------------------------------------------------
Plan hash value: 1594556531

--------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |  7001 |   157K|   387   (3)| 00:00:05 |
|*  1 |  HASH JOIN         |           |  7001 |   157K|   387   (3)| 00:00:05 |
|   2 |   TABLE ACCESS FULL| EMPLOYEES |   107 |  1070 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| EMPHOURS  |  7001 | 91013 |   383   (3)| 00:00:05 |
--------------------------------------------------------------------------------
```

Fig. 11.20 Without Bitmap

```
Elapsed: 00:00:00.04

Execution Plan
----------------------------------------------------------
Plan hash value: 1594230613

-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |             |  7001 |   157K|   315   (1)| 00:00:04 |
|*  1 |  HASH JOIN                    |             |  7001 |   157K|   315   (1)| 00:00:04 |
|   2 |   TABLE ACCESS FULL           | EMPLOYEES   |   107 |  1070 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS BY INDEX ROWID | EMPHOURS    |  7001 | 91013 |   311   (0)| 00:00:04 |
|   4 |    BITMAP CONVERSION TO ROWIDS|             |       |       |            |          |
|   5 |     BITMAP AND                |             |       |       |            |          |
|*  6 |      BITMAP INDEX SINGLE VALUE| HRS_BIT_IDX |       |       |            |          |
|*  7 |      BITMAP INDEX SINGLE VALUE| FEE_BIT_IDX |       |       |            |          |
-----------------------------------------------------------------------------------------------
```

Fig. 11.21 With Bitmap indexes

```
create bitmap index hrs_bit_idx on EMPHOURS(hours);
create bitmap index fee_bit_idx on EMPHOURS(fee_earning);

select a.employee_id, a.last_name, b.work_date, b.hours from employees a,
emphours b where b.emp_id = a.employee_id AND b.fee_earning = 'N'
AND hours = 50;
select a.employee_id, a.last_name, b.work_date, b.hours from employees a,
emphours b where b.emp_id = a.employee_id AND b.fee_earning = 'N'
AND hours = 50;

spool off
```

Try this and see what the effect is. On the author's server the difference was striking in that the query time was halved (Figs. 11.20, 11.21).

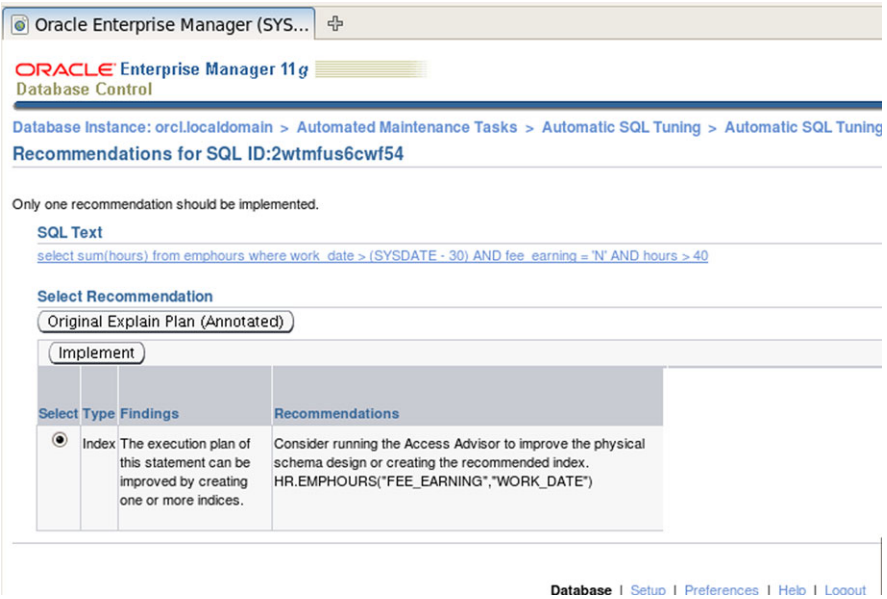


Fig. 11.22 Example Enterprise Manager tuning recommendation

11.5.2 Using the Built-in Advisers

The testing suggested above is fine during system builds, especially if we have realistic data to test against. However, once an optimised system is in place. It may well cause problems in the future due to changes in data. Assuming we want to be proactive and not just wait for complaints from users, how does a DBA establish which SQL might be worth investigating? Logged on as a DBA we can see what SQL has run recently using a DBA-only view called V$SQL.

```
select sql_text from v$sql;
```

Unfortunately this can list all sorts of system generated SQL and in an operational production system there may be thousands of different SQL statements being run at any time. It is possible to join other v$ views to restrict the rows returned to those that might be of interest, but the easiest solution is to use the Enterprise Manager's advisors.

If you have Enterprise Manager open, move to Advisor Central and then select Automated Maintenance Tasks, and then run the Automatic SQL Tuning and finally review the Automatic SQL Tuning Result Details. On the author's server there was a recommendation for an index to support one of the queries that has recently run (Fig. 11.22).

11.5.3 Over to You!

11.5.3.1 Scenario One

We have been asked to create a report that counts the number of entries in EMPHOURS where no fee was earned. It should have a count for Hours <30, one for 30–40, and one for >40.

This could be tackled by having two UNIONS join three queries, or by using the CASE statement. You need to establish which is quicker.

NOTE: You should still have your bitmap index on the two columns from earlier for this to work well.

11.5.3.2 Scenario Two

Your developers are suggesting that some reports would benefit from there being a bitmap index on the fee_earning column. You need to get an understanding of how much slower your database would become in writing new records in a write-intensive period if you were to implement a bitmap index on the fee_earning table.

As a suggestion, slightly rewrite the emphours creation and insertion script to:

1. Create an identical table, called emphours2
2. Time the INSERTEMPHOURS process
3. drop the table
4. create it again, this time with an index
5. time INSERTEMPHOURS again

If you put this all in one script you might find it useful to name your timers so they are easy to read when you are looking at your spooled outputs. You do this in sqlplus like this:

```
Timing start InsertsNoIdx
do some processing ....
Timing stop InsertsNoIdx
```

Is the insertion process any longer with an index?

11.6 Summary

In this chapter we have reviewed the impact each of the tiers in any database system can have on overall performance. We have seen that CPU usage and Disk I/o are important, but that so are some of the less physical aspects of a system, such as its design, how the indexes are used, and which types of indexes and tables are used. We have seen that performance problems can be hard to diagnose because of the complexity of the tiers. Moreover, even if we do find the cause and resolve the problem, there is no guarantee that it will remain solved because, as we add data to our system, the data distribution may make some of our previous design assumptions invalid.

11.7 Review Questions

The answers to these questions can be found in the text of this chapter.

- What is the difference between Throughput and Workload?
- Which physical element of a database system is generally the slowest
- Why might a DBA decide not to update the CBO statistics often?
- What effect does a B-tree index have on inserts and updates?
- Why will a query which returns 20 % of a table's rows probably not use an index, and what will it use?
- Name at least two alternative types of index, other than B-Tree, and explain when they might best be used

11.8 Group Work Research Activities

*These activities require you to research beyond the contents of the book and can be tackle*d *individually or as a discussion group.*

Discussion Topic 1 *Indexes are a nuisance in a volatile OLTP system which manages many writes per second.* Discuss this assertion. You should review the benefits and disadvantages of different types of indexes in doing so.

Discussion Topic 2 Normalisation is often seen as a required process in the design of a database. Explain why this is so, and then go on to discuss scenarios when no normalisation is required, or indeed, when data should be denormalised.

Appendix: Creation Scripts and Hints

To create the EMPHOURS table and add the rows, save this code to a file in your Oracle working directory and call it something like CreateEmpHours.sql. Then Log in as hr/hr and run the script. This generates 400000+ rows and so it will take up to a minute or so to complete.

```
drop table emphours;

create table emphours ( emp_id number,
         work_date date,
         hours number,
         fee_earning varchar(1),
         FOREIGN KEY (emp_id) REFERENCES HR.EMPLOYEES
         (EMPLOYEE_ID),
         PRIMARY KEY (emp_id, work_date) );

create or replace
```

```
PROCEDURE INSERTEMPHOURS AS
BEGIN

  DECLARE
  eid employees.employee_id%TYPE;
  hdate employees.hire_date%TYPE;
  fee emphours.fee_earning%TYPE;
  howrs emphours.hours%TYPE;
  rnd Integer;

  CURSOR c1 IS
    SELECT EMPLOYEE_ID, hire_date FROM EMPLOYEES;

BEGIN
  OPEN c1;
  -- loop through each of Employee rows
  LOOP
    FETCH c1 INTO eid, hdate;
    EXIT WHEN c1%NOTFOUND;

    -- loop through every day since they started to sysdate
    LOOP
      EXIT WHEN hdate > sysdate;
      hdate := hdate +1;
      -- check if weekend
      IF to_char(hdate, 'D') > 1 AND to_char(hdate, 'D') < 7
        THEN
          -- make fee_earning a random Y or N but with more Y
          values
          rnd := DBMS_RANDOM.value(low => 1, high => 10);
          IF rnd > 7 THEN
            fee := 'N';
          ELSE
            fee := 'Y';
          END IF;
          -- generate a random number of hours works between 20
          and 50 howrs := round(DBMS_RANDOM.value(low => 20, high
          => 50));
          INSERT INTO emphours (emp_id, work_date, hours,
          fee_earning)
            values (eid, hdate, howrs, fee );
      END IF;
    END LOOP;
  END LOOP;
  CLOSE c1;
END;
END INSERTEMPHOURS;
/

begin
  INSERTEMPHOURS;
end;
/
```

```
SQL> @createEmpHours
Table dropped.

Table created.

Procedure created.

PL/SQL procedure successfully completed.

SQL> select count(*) from emphours;

  COUNT(*)
---------
    434068
```

A.1: Hints on the Over to You Section

A.1.1: Scenario One

set timing on
set autotrace on
set linesize 200
spool test19.txt

set timing on
SELECT COUNT (*)
FROM emphours
WHERE fee_earning = 'N' AND hours < 30
UNION
SELECT COUNT (*)
FROM emphours
WHERE fee_earning = 'N' AND hours BETWEEN 30 AND 40
UNION
SELECT COUNT (*)
FROM emphours
WHERE fee_earning = 'N' AND hours > 40;

SELECT COUNT (case when fee_earning = 'N' AND hours < 30 THEN 1 ELSE null END) lessthan30,
COUNT (case when fee_earning = 'N' AND hours BETWEEN 30 AND 40 THEN 1
ELSE null END) betwix3040,
COUNT (case when fee_earning = 'N' AND hours > 40 THEN 1 ELSE null END) gt40

```
From EMPHOURS;

spool off
```

The CASE fails to use the index and so is much slower.

A.1.2: Scenario Two

```
drop table emphours2;

create table emphours2 (emp_id number,
                work_date date,
                hours number,
                fee_earning varchar(1),
                FOREIGN KEY (emp_id) REFERENCES HR.EMPLOYEES
                (EMPLOYEE_ID),
                PRIMARY KEY (emp_id, work_date));

create or replace
PROCEDURE INSERTEMPHOURS AS
BEGIN

   DECLARE
   eid employees.employee_id%TYPE;
   hdate employees.hire_date%TYPE;
   fee emphours2.fee_earning%TYPE;
   howrs emphours2.hours%TYPE;
   rnd Integer;

   CURSOR c1 IS
      SELECT EMPLOYEE_ID, hire_date FROM EMPLOYEES;

BEGIN
   OPEN c1;
   – loop through each of Employee rows
   LOOP
      FETCH c1 INTO eid, hdate;
      EXIT WHEN c1%NOTFOUND;

      – loop through every day since they started to sysdate
      LOOP
         EXIT WHEN hdate > sysdate;
         hdate := hdate + 1;
         – check if weekend
         IF to_char(hdate, 'D') > 1 AND to_char(hdate, 'D') < 7
            THEN
            – make fee_earning a random Y or N but with more Y values
            rnd := DBMS_RANDOM.value(low => 1, high => 10);
```

```
            IF rnd > 7 THEN
               fee := 'N';
            ELSE
               fee := 'Y';
            END IF;
            – generate a random number of hours works between 20 and 50
            howrs := round(DBMS_RANDOM.value(low => 20, high => 50));
            INSERT INTO emphours2 (emp_id, work_date, hours, fee_earning)
               values (eid, hdate, howrs, fee );
         END IF;

      END LOOP;
   END LOOP;
   CLOSE c1;
END;
END INSERTEMPHOURS;
/

Timing start InsertsNoIdx
begin
   INSERTEMPHOURS;
end;
/

Timing stop InsertsNoIdx

– now drop the table and recreate with an index

drop table emphours2;

create table emphours2 (emp_id number,
                work_date date,
                hours number,
                fee_earning varchar(1),
                FOREIGN KEY (emp_id) REFERENCES HR.EMPLOYEES
                (EMPLOYEE_ID),
                PRIMARY KEY (emp_id, work_date));

create bitmap index fee_bit_idx on Emphours2 (fee_earning);

Timing start InsertsWithIdx
begin
   INSERTEMPHOURS;
end;
/
Timing stop InsertsWithIdx

spool off
```

References

Florescu D, Kossmann D (2009) Rethinking cost and performance of database systems. SIGMOD Rec 38(1):43–48. 2009