

Relational Database Index Design

***Prerequisite for
Index Design Workshop***

Tapio Lahdenmaki
April 2004

Copyright Tapio Lahdenmaki, DBTech Pro

CHAPTER 1: INTRODUCTION.....	4
INADEQUATE INDEXING.....	4
THE IMPACT OF HARDWARE EVOLUTION	5
<i>Volatile Columns Should Not Be Indexed – True Or False?</i>	7
<i>Example</i>	7
<i>Disk Drive Utilisation</i>	8
SYSTEMATIC INDEX DESIGN.....	9
CHAPTER 2.....	13
TABLE AND INDEX ORGANIZATION	13
INTRODUCTION.....	13
INDEX AND TABLE PAGES	14
INDEX ROWS	14
THE INDEX STRUCTURE.....	14
TABLE ROWS.....	15
BUFFER POOLS AND DISK I/OS	16
<i>Reads from the DBMS Buffer Pool</i>	16
<i>Random I/O from Disk Drives</i>	17
<i>Reads from the Disk Server Cache</i>	18
<i>Sequential Reads from Disk Drives</i>	19
<i>Skip-sequential Reads from Disk Drives</i>	20
<i>Synchronous and Asynchronous I/Os</i>	22
HARDWARE SPECIFICS	23
DBMS SPECIFICS	25
<i>Pages</i>	25
<i>Table Clustering</i>	25
<i>Index Rows</i>	26
<i>Table rows</i>	26
<i>Index-Only Tables</i>	27
<i>Page Adjacency</i>	27
<i>Alternatives to B-tree indexes</i>	29
<i>The Many Meanings of Cluster</i>	30
CHAPTER 3.....	31
SQL PROCESSING	31
INTRODUCTION.....	31
PREDICATES	32
OPTIMIZERS AND ACCESS PATHS	32
<i>Index Slices and Matching Columns</i>	33
<i>Index Screening and Screening Columns</i>	34
<i>Access Path Terminology</i>	35
<i>Monitoring the Optimizer</i>	36
<i>Helping the Optimizer (Statistics)</i>	36
<i>Helping the Optimizer (Number of FETCH Calls)</i>	37
<i>When the Access Path is Chosen</i>	38
FILTER FACTORS	39
<i>Filter Factors for Compound Predicates</i>	40
<i>The Impact of Filter Factors on Index Design</i>	43
MATERIALIZING THE RESULT ROWS	45
<i>Cursor Review</i>	46
<i>Alternative 1: FETCH Call Materializes One Result Row</i>	47
<i>Alternative 2: Early Materialization</i>	47
<i>What Every Database Designer Should Remember</i>	48
EXERCISE 1.....	49

EXERCISE 2.....	50
EXERCISE 3.....	50

Chapter 1: Introduction

Inadequate Indexing

For many years, inadequate indexing has been the most common cause of performance disappointments. The most widespread problem appears to be that indexes do not have sufficient columns to support all the predicates of a WHERE clause. Frequently, there are not enough indexes on a table; some SELECTs may have no useful index; sometimes an index has the right columns but in the wrong order.

It is relatively easy to improve the indexing of a relational database, because no program changes are required. However, a change to a production system always carries some risk. Furthermore, while a new index is being created, update programs may experience long waits because they are not able to update a table being scanned for a **CREATE INDEX**. For these reasons, and of course to achieve acceptable performance from the first production day of a new application, indexing should be in fairly good shape *before* production starts. Indexing should then be finalized soon after cutover, without the need for numerous experiments.

Database indexes have been around for decades, so why is the average quality of indexing still so poor?

One reason is perhaps because many people assume that, with the huge processing and storage capacity now available, it is no longer necessary to worry about the performance of seemingly simple SQL. Another reason may be that few people even think about the issue at all. Even then, for those who do, the fault can often be laid at the door of numerous relational database textbooks and educational courses. Browsing through the library of relational DBMS books will quite possibly lead to the following assessment:

- The *Index Design* topics are short, perhaps only a few pages.
- The negative side effects of indexes are emphasized; indexes consume disk space and they make inserts, updates and deletes slower.
- Index design guidelines are vague and sometimes questionable. Some writers recommend one to index all restrictive columns. Others claim that index design is an art that can only be mastered through trial and error.
- Little or no attempt is made to provide a simple but effective approach to the whole process of index design.

Many of these warnings about the cost of indexes are a legacy from the 1980s when storage, both disk and semi-conductor, was *significantly* more expensive than it is today.

The Impact of Hardware Evolution

Even recent books suggest that only the *root page* of a B-tree index will normally stay in central storage. This was an appropriate assumption 20 years ago, when central storage was typically so small that the database buffer pool could contain only a few hundred pages, perhaps less than a megabyte. Today, the size of the database buffer pools may be hundreds of thousands of pages, one gigabyte or more; the read caches of disk servers are typically even larger - 64 gigabytes for instance. Although databases have grown as disk storage has become cheaper, it is now realistic to assume that *all the non-leaf pages* of a B-tree index will usually remain in central storage or the read cache. Only the leaf pages will normally need to be read from a disk drive; this of course makes index maintenance much faster.

The assumption “*only root pages stay in central storage*” leads to many obsolete and dangerous recommendations..

The index shown in Figure 1.1 corresponds to a 100 million row table. There are 100 million index rows with an average length of 100 bytes. Taking the distributed free space into account, there are 35 index rows per leaf page. If the DBMS does not truncate the index keys in the non-leaf pages, the number of index entries in these pages is also 35.

The probable distribution of these pages as shown in Figure 1.1, together with their size, can be deduced as follows:

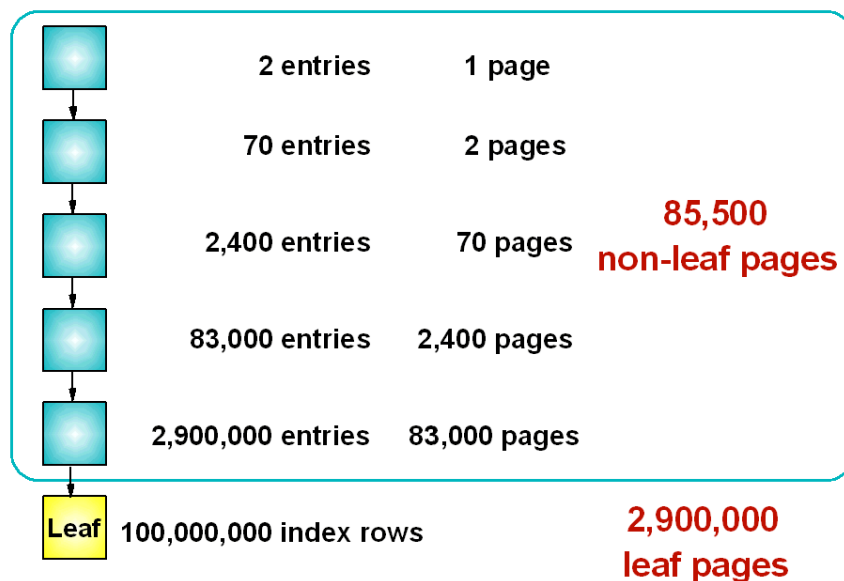


Figure 1.1 A large index with six levels

- The *index in total* holds about 3,000,000 4K pages which requires **12 GB** of disk space.
- The total size of the *leaf pages* is 2,900,000 x 4K which is almost **12 GB**. It is reasonable to assume that these will normally be read from a disk drive (10 ms).
- The size of the *next level* is 83,000 x 4K which is **332 MB**; to a large extent these pages will stay in the read cache (perhaps 64 GB in size) of the disk server, if not in the database buffer pool in central storage (say 4 GB for index pages).
- the *upper levels*, roughly 2,500 x 4K = **10 MB**, will almost certainly remain in the database buffer pool.

Accessing *any* of these 100,000,000 index rows in this six level index then takes about **11 milliseconds**, 10 for the leaf page and 1 for the other index pages.

Volatile Columns Should Not Be Indexed – True Or False?

Index rows are held in key sequence, so when one of the columns is updated, the DBMS *may* have to move the corresponding row from its old position in the index to its new position, to maintain this sequence. This new position may be in the *same* leaf page in which case only the one page is affected. However, particularly if the modified key is the first or only column, the new index row may have to be moved to a *different* leaf page; the DBMS must then update two leaf pages. *Twenty years ago*, this might have required *six* random disk reads if the index had four levels; three for the original, two non-leaf and one leaf, together with a further three for the new. When a random disk read took 30 milliseconds, moving **one** index row could add $6 \times 30 \text{ ms} = \mathbf{180 \text{ milliseconds}}$ to the response time of the update transaction. It is hardly surprising that volatile columns were seldom indexed.

These days when *three levels* of a four level index, the non-leaf pages, stay in central storage and a random read from a disk drive takes 10 milliseconds, the corresponding time becomes $2 \times 10 \text{ ms} = \mathbf{20 \text{ milliseconds}}$. Furthermore, many indexes are multi-column indexes, called *compound* or *composite* indexes, which often contain columns that make the index key unique. When a volatile column is the last column of such an index, updating this volatile column *never* causes a move to another leaf page; consequently, with current disks, updating the volatile column adds only **10 milliseconds** to the response time of the update transaction.

Example

A few years ago, the DBAs of a well tuned DB2 installation having an average local response time of *0.2 seconds*, started transaction level exception monitoring. Immediately, they noticed that a simple browsing transaction regularly took more than *30 seconds*; the longest observed local response time was *a couple of minutes*. They quickly traced the problem to inadequate indexing on a two million row table. Two problems were diagnosed:

- A volatile column **STATUS**, updated up to twice a second, was absent from the index although it was an *obvious essential requirement*. **STATUS** was ANDed to five other predicates in the WHERE clause.
- An ORDER BY required a sort of the result rows.

These two index design decisions had been made consciously, based on widely used recommendations. The Column STATUS was much more volatile than most of the other columns in this installation. This is why the DBAs had not dared to include it in the index. They were also afraid that an extra index, which would have eliminated the sort, would have caused problems with INSERT performance, because the insert rate to this table was relatively high. They were particularly worried about the *load* on the disk drive.

Following the realisation of the extent of the problem caused by these two issues, rough estimates of the index overhead were made, and they decided to create an additional index containing the five columns, together with STATUS at the end. This new index solved both problems. The longest observed response time went down *from a couple of minutes to less than a second*. The update and insert transactions were not compromised and the disk drive containing the new index was not overloaded.

Disk Drive Utilisation

Disk drive load and the required speed of INSERTs, UPDATEs and DELETEs still set an upper limit to the number of indexes on a table. However, this ceiling is much higher than it was 20 years ago. A reasonable request for a new index should not be rejected intuitively. With current disks, an indexed volatile column may become an issue only if the column is updated perhaps *more than 10 times a second*. Such columns are not very common.

With current disks, a random read keeps one drive busy for about **6 milliseconds** (seek = 4 ms, half a rotation = 2 ms at 15,000 rpm). Random writes also keep a drive busy for 6 milliseconds if there is no redundant data for fault tolerance (discussed later), otherwise 2 x 6 milliseconds or 2 x 12 milliseconds depending on the implementation.

Let us assume an index is striped over seven drives in a RAID 5 array; if an index column is *updated 10 times per second* and if the index row moves to another leaf page each time (the worst case), each of the seven drives will be busy as a result of an index update for

$$(2 \times 10 \times (6 \text{ ms} + 24 \text{ ms})) / 7 = 86 \text{ milliseconds per second.}$$

The utilization of each drive in the array (*drive busy*) goes up by **9%** (86/1,000). If the *volatile* column is the *last column* of a multi-column index which, without the last column, is *unique*, drive busy increases by **4.5%** (43/1,000). A disk drive may be considered to be overloaded in an environment with response critical operational transactions, if drive busy exceeds **25%**. With drive busy at 25%, the average drive queuing time will be a couple of milliseconds.

This calculation is a theoretical worst case, because we assumed that every random update to an index requires one drive read and two drive writes. This is true only if the index is large compared to the read cache (64 GB, say) and write cache (2 GB, say) and if the leaf page access pattern is truly random. Because these assumptions are often pessimistic, the actual *drive busy* values tend to be lower. In addition, the largest indexes may be striped to dozens of drives. Therefore, drive load is not likely to become a serious issue with current RAID 5 disks unless an index column is updated *more than 100 times a second*.

Systematic Index Design

The first attempts towards an index design method originate from the 1960s. At that time, textbooks recommended a matrix for predicting how often each field (column) is read and updated, and how often the records (rows) containing these fields are inserted and deleted. This led to a list of columns to be indexed. The indexes were generally assumed to have only a single column and the objective was to minimize the number of disk I/Os during peak time. It is amazing that this approach is still being mentioned in recent books although a few, somewhat more realistic writers, do admit that the matrix should only cover the most common transactions.

This *column activity matrix approach* may explain the column oriented thinking that can be found even in recent textbooks and database courses, such as “**consider** indexing columns with these properties” and “**avoid** indexing columns with those properties”.

In the 1980s, the *column oriented approach* began to lose ground to a *response oriented approach*. Enlightened DBAs started to realise that the objective of indexing should be to make *all* database calls fast enough, given the hardware capacity constraints. The pseudo relational DBMS of IBM S/38 (later AS/400, then the iSeries) was the vanguard of this attitude. It automatically built a good index for each database call. This worked well with simple applications. Today, many products propose indexes for each SQL call, but indexes are not created automatically, apart from primary key indexes and, sometimes, foreign key indexes.

As applications became more complex and databases much larger, the importance and complexity of index design became obvious. Ambitious projects were undertaken to develop tools for automating the design process. The basic idea was to collect a sample of production workload and then generate a set of index candidates for the SELECT statements in the workload. Simple evaluation formulae or a cost based optimizer would then be used to decide which indexes were the most valuable. This sort of product has become available over the last few years, but their usage has spread rather slower than expected.

Systematic index design consists of two processes as shown in Figure 1.2.

- 1 **Detect SELECT statements that are too slow due to inadequate indexing**

Worst input: Variable values leading to the longest elapsed time
- 2 **Design indexes that make all SELECT statements fast enough**

Table maintenance (INSERT, UPDATE, DELETE) must be fast enough as well

Figure 1.2 Systematic Index Design

Firstly, it is necessary to find the SELECTs that are, or will be, too slow with the current indexes, at least with the worst input; for example “*the largest customer*”, or “*the oldest date*”. *Secondly*, indexes have to be designed to make the slow SELECTs fast enough without making other SQL calls noticeably slower. Neither of these tasks is trivial.

The first attempts to detect inadequate indexing at *design time* were based on hopelessly complex prediction formulae, sometimes simplified versions of those used by cost based optimizers. Replacing calculators with programs and graphical user interfaces did not greatly reduce the effort. Later, extremely simple formulae, like the **QUBE**, developed in IBM Finland in the late 1980s, or a simple estimation of the number of random I/Os were found useful in real projects. The **Basic Question** proposed by Ari Hovi was the next and probably the ultimate step in this process. .

Methods for improving indexes *after* production cutover developed significantly in the 1990s. Advanced monitoring software forms a necessary base to do this, but an intelligent way to utilize the massive amounts of measurement data is also essential. The business model used by Joe Luukkonen is well known in Canada; he offered a tuning service complete with a guarantee that if performance did not improve by at least 10% following a week of tuning, he would not send a bill!

This second task of systematic index design went unrecognised for a long time. The SELECTs found in textbooks and course material were so unrealistically simple that the best

index was usually obvious. Experience with real applications has taught, however, that even harmless looking SELECTs, particularly joins, often have a huge number of reasonable indexing alternatives. Estimating each alternative requires far too much effort, and measurements even more so. On the other hand, even experienced database designers have made numerous mistakes when relying on intuition to design indexes.

This is why there is a need for an algorithm to design the best possible index for a given SELECT. The concepts of a three star index and the related index candidates have proved helpful.

Chapter 2

Table and Index Organization

Introduction

Before we are in a position to discuss the index design process, we need to understand how indexes and tables are organized and used. Much of this of course, will depend on the individual relational DBMS, however these all rely on broadly similar general structures and principles, albeit using very different terminology in the process.

In this chapter we will consider the fundamental structures of the relational objects in use; we will then discuss the performance related issues of their use, such as the role of buffer pools, disks and disk servers, and how they are used to make the data available to the SQL process.

Once we are familiar with these fundamental ideas, we will be in a position, in the next chapter, to consider the way these relational objects are processed to satisfy SQL calls.

Index and Table Pages

Index and table rows are grouped together in **pages**; these are often 4k in size, this being a rather convenient size to use for most purposes, but other page sizes may be used. Fortunately, as far as index design is concerned, this is not an important consideration other than that the page size will determine the number of index and table rows in each page and the number of pages involved. To cater for new rows being added to tables and indexes, a certain proportion of each page may be left free when they are loaded or reorganized. This will be considered later.

Buffer pools and I/O activity (discussed later) are based on pages; for example, an entire page will be read from disk into a buffer pool. This means that *several rows*, not just one, are read into the buffer pool with a single I/O. We will also see that *several pages* may be read into the pool by just one I/O.

Index Rows

An index row is a *useful concept* when evaluating access paths. For a *unique* index, such as the primary key index CNO on table CUST, it is equivalent to an index entry in the leaf page (see Figure 2.1); the column values are copied from the table to the index, and a pointer to the table row added. Usually, the table *page number* forms a part of this pointer, something that should be kept in mind for a later time. For a *non-unique* index, such as the index CITY on table CUST, the index rows for a particular index value should be visualised as *individual* index entries, each having the same CITY value, but followed by a different pointer value. What is *actually stored* in a non-unique index is, in most cases, one CITY value followed by several pointers. The reason why it is *useful* to visualise these as individual index entries will become clear later.

The Index Structure

The non-leaf pages always contain a (possibly truncated) key value, the highest key together with a pointer, to a page at the next lower level as shown in Figure 2.1. Several index levels may be built up in this way, until there is only a single page, called the *root page*, at the top of the index structure. This type of index is called a B-Tree index (a balanced tree index), because the same number of non-leaf pages are required to find each index row.

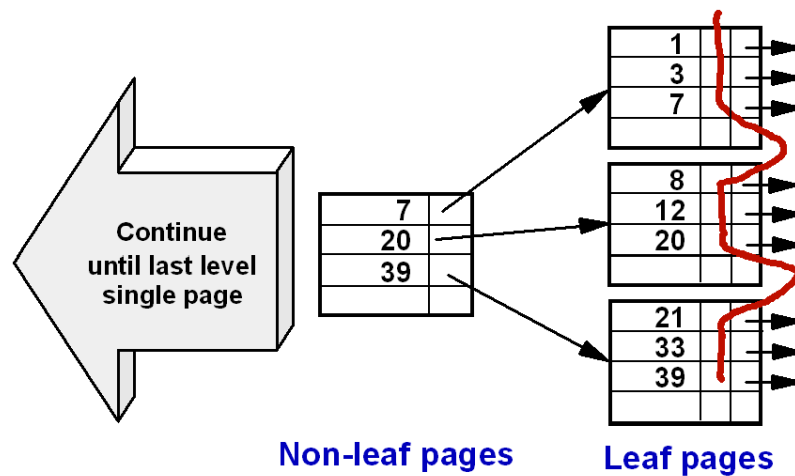


Figure 2.1 A very small index

Table Rows

Each index row shown in Figure 2.1 points to a corresponding row in the table; the pointer usually identifies the page in which the row resides together with some means of identifying its position within the page. Each table row contains some control information to define the row and to enable the DBMS to handle insertions and deletions, together with the columns themselves.

The sequence in which the rows are positioned in the table, as a result of a table load or row inserts, *may* be defined so as to be the same as that of *one* of its indexes. In this case, as the index rows are processed, one after another in key sequence, so the corresponding table rows will be processed, one after another in the same sequence. Both index and table are then accessed in a sequential manner, which, as we will see shortly, is a very efficient process.

Obviously, *only one of the indexes* can be defined to determine the sequence of the table rows in this way. If the table is being accessed via *any other* index, as the index rows are processed, one after another in key sequence, the corresponding rows will *not* be held in the table in the same sequence. For example, the first index row may point to page 17, the next index row to page 2, the next to page 85 and so forth. Now, although the index is still being processed sequentially, and efficiently, the table is being processed randomly, and much less efficiently.

Buffer Pools and Disk I/Os

One of the primary objectives of relational database management systems is to ensure that data from tables and indexes is readily available when required. To enable this objective to be achieved as far as possible buffer pools, held in processor storage, are used to minimise disk activity. Each DBMS may have several pools according to the type, table or index, and the page size. Each pool will be large enough to hold many pages, perhaps hundreds of thousands of them. The buffer pool managers will attempt to ensure that frequently used data remains in the pool to avoid the necessity of additional reads from disk. How effective this is will be extremely important with respect to the performance of SQL statements, and so will be equally important for the purposes of this book. We will return to this subject on many occasions where the need arises. For now we must simply be aware of the *relative costs* involved of accessing index or table rows from pages which may or may not be stored in the buffer pools.

Reads from the DBMS Buffer Pool

If an index or table page is found in the buffer pool, the only cost involved is that of the processing of the index or table rows. This is highly dependent on whether the row is rejected or accepted by the DBMS, the former incurring very little processing, the latter incurring much more as we will see in due course.

Random I/O from Disk Drives

Figure 2.2 shows the enormous cost involved of having to wait for a page to be read into the buffer pool from a disk drive

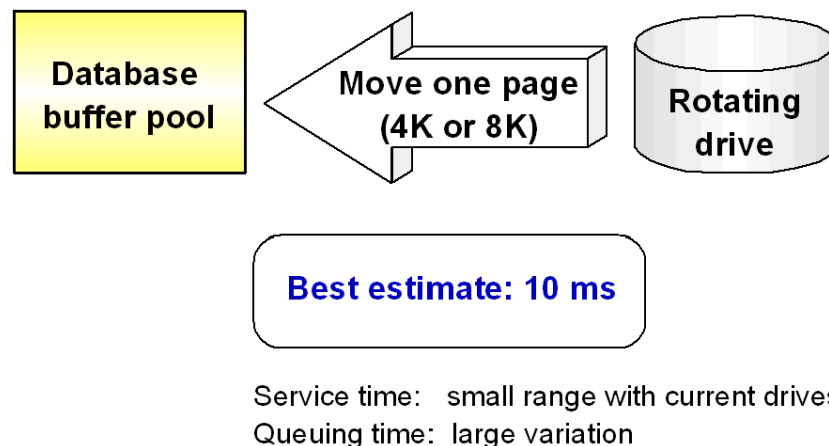
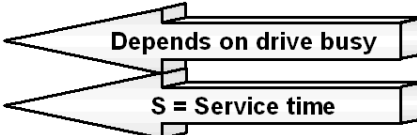


Figure 2.2 *Random I/O from disk drive - 1*

Again, we must remember that a page will contain several rows; we may be interested in all of these rows, just a few of them or even only a single row - the cost will be the same, roughly 10 ms. If the disk drives are heavily used, this figure might be considerably increased as a result of having to wait for the disk to become available. In computing terms, 10 ms is an eternity.

Figure 2.3 breaks the 10 ms down into its constituent components. From this we can see that we are assuming the disk would actually be busy for about 6 out of the 10 ms. The transfer time of roughly 1 ms refers to the movement of the page from the disk server cache into the database buffer pool. The other 3 ms is an estimate of the queuing time that might arise, based on disk activity of say, 50 reads per second. These sort of figures would equally apply to directly attached drives; all the figures will of course vary somewhat, but we simply need to keep in mind a rough, but not unreasonable, figure of 10 ms.

Queuing (Q)	3 ms	
Seek	4 ms	
Half a rotation	2 ms	
Transfer	1 ms	
Total I/O time	10 ms	

One random read keeps a drive busy for 6 ms

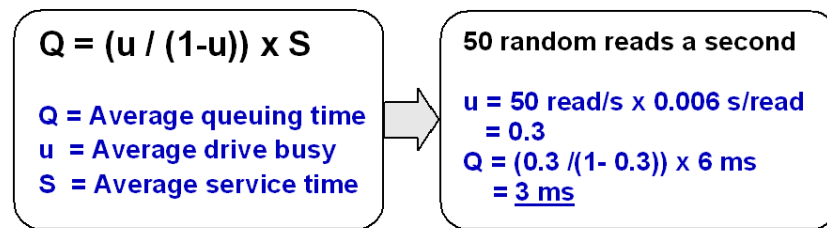


Figure 2.3 Random I/O from disk drive – 2

Reads from the Disk Server Cache

Fortunately, disk servers in use today provide their own storage (or cache) in order to reduce this huge cost in terms of elapsed time. Figure 2.4 shows the read of a single table or index page (again equivalent to reading a number of table or index rows) from the cache of the disk server. Just as with the buffer pools, the disk server is trying to hold frequently used data in storage (cache) rather than incurring the heavy disk read cost. If the page required by the DBMS is not in the buffer pool, a read is issued to the disk server who will check to see if it is in the server cache and only perform the real disk read if it is not found there. Even if a real read access is required, these are performed somewhat faster because of factors such as streaming which will be discussed at a more appropriate time. All that we need to understand at this time is that the figure of 10 ms may be considerably reduced to a figure as low as 1 ms if the page is found in the disk server read cache.

In summary then, the ideal place for an index or table page to be when it is requested is in the database buffer pool. If it is not there, the next best place for it to be is in the disk server read cache. If it is in neither of these, a slow read from disk will be necessary, perhaps involving a long wait for the device to become available.

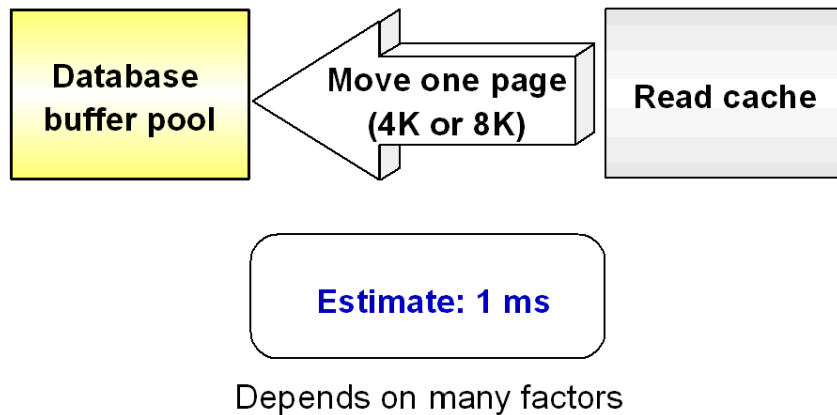


Figure 2.4 *Read from disk server cache*

Sequential Reads from Disk Drives

So far, we have only considered reading a single index or table page into the buffer pool. There will be many occasions when we actually want to read several pages into the pool and process the rows in sequence. Figure 2.5 shows the four occasions when this will apply. The DBMS will be aware that several index or table pages should be read sequentially and will identify those which are not already in the buffer pool. It will then issue multiple-page I/O requests, where the number of pages in each request will be determined by the DBMS; only those pages not already in the buffer pool will be read because those that are already in the pool may contain updated data that has not yet been written back to disk.

There are two very important advantages of reading the pages sequentially in this way:

- reading several pages together means that the time per page will be reduced; with current disk servers, the value may be as low as 0.1 ms for 4K pages.
- because the DBMS knows in advance which pages will be required, the reads can be performed before the pages are actually required; this is called prefetch.

The terms *index slice* and *clustering index* referred to in Figure 2.5 will be addressed shortly.

- Full table scan
- Full index scan
- Index slice scan
- Scan table rows via clustering index

Prefetch possible even if
pages not adjacent

Estimate: As low as 0.1 ms per 4K page

Service time: large range...should be measured

Figure 2.5 Sequential reads from disk drives

Skip-sequential Reads from Disk Drives

There are occasions when a set of pages may be read sequentially even though the pages themselves are not consecutive. This way of accessing data is called *skip-sequential*. One example where skip-sequential read might take place is shown in Figures 2.6 and 2.7, where the former *does not* use it, while the latter *does* use it. We will consider this example only briefly at this time, leaving a fuller description until later. In both figures, the result rows are required in the *sequence according to column B* but the table rows are assumed to be *physically ordered by any column other than column B*, perhaps according to its primary or foreign key.

If the DBMS was to use each index row in turn to access the corresponding table row, as shown in Figure 2.6, the table rows would indeed be accessed in the correct order, column B. The table rows though, would unfortunately have to be read by random read I/Os; the numbers 1 to 6 shown in Figure 2.6 indicate the sequence of events. This could take 10 ms for each page, ignoring disk server read cache. We have also seen that this is not only 10 ms for *each page* but could also be 10 ms for each *row* because a random read is being requested due to the need to read a particular *row* into the buffer pool.

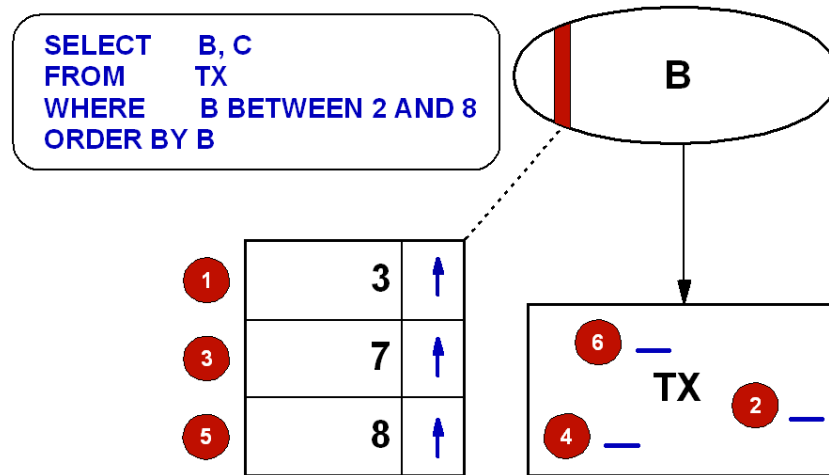


Figure 2.6 Traditional index scan

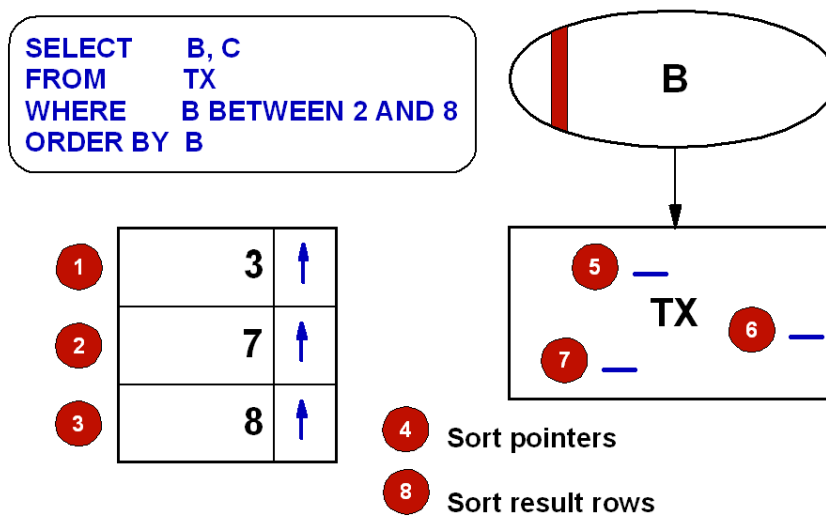


Figure 2.7 Skip-sequential read

To avoid these long random reads, it may be possible for the DBMS to access *all* the index entries that satisfy the WHERE clause, shown in Figure 2.7 as numbers 1, 2 and 3, *before* any

access is made to the table. The pointers obtained from these qualifying index rows could then be sorted into table page sequence, number 4 in the figure, so that the DBMS will then know which pages will need to be read into the pool. This list of pages, being in sequence, can then be used to perform a skip-sequential scan for the pages corresponding to numbers 5, 6 and 7 in the figure, although in reality the result set could of course be much larger. This is precisely what DB2 for z/OS does when it uses a feature called *list prefetch*.

In addition to the faster read times resulting from the sequential access, several result rows might reside in the same page; all of them would then be obtained as the page is moved to the buffer pool. Using random reads, this could have meant *several* reads for the *same* page.

Skip-sequential then may be a very useful feature, but what sort of performance figure can we attribute to it. Unfortunately, every time the process takes place we will have a different result set, the position of the rows within the pages will be different, and the distance between the pages will be different. The performance costs, and indeed the benefits, will therefore differ considerably. How we come to terms with this problem will have to be left until a later time.

Before we leave skip-sequential, however, it might be worth considering one final point, namely the order of the result set, which is to be based on column B. Accessing the table rows as we access the index rows, as in Figure 2.6, in column B order, means that the sequence is provided automatically. Accessing the table rows *after* having sorted the index pointers in order to use the more efficient skip-sequential technique has destroyed this sequence; the result set will have to be sorted, number 8 in Figure 2.7, before it can be returned to the user or the program - again, much more about this in due course.

Synchronous and Asynchronous I/Os

Having discussed these different access techniques, it will be appropriate now to ensure we fully appreciate one final consideration, synchronous and asynchronous I/Os as shown in Figure 2.8.

The term *synchronous* I/O infers that while the I/O is taking place, the DBMS is not able to continue any further; it is forced to wait until the I/O has completed. With a synchronous read, for example, we have to identify the row required (shown as “C” to represent the first portion of CPU time in the figure), access the page and process the row (shown as the second portion of CPU time), each stage waiting until the previous stage completes.

Asynchronous reads on the other hand, are being performed in advance while a *previous* set of pages are being processed; there may be a considerable overlap between the processing and I/O time; ideally the asynchronous I/O will complete before the pages are actually required for processing. Each group of pages being prefetched and processed in the figure is shown in a different colour; note that a synchronous read kick-starts the whole prefetch activity before the green set is prefetched.

Most database writes are performed asynchronously such that they should have little effect on performance. The main impact they do have is to increase the load on the disk environment, which in turn may affect the performance of the read I/Os.

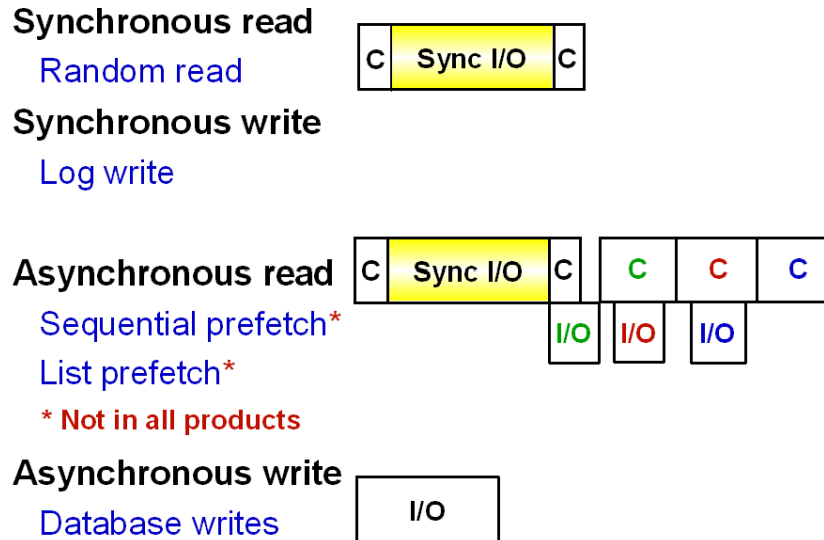


Figure 2.8 Synchronous and asynchronous I/O

Hardware Specifics

At the time of writing, the disk drives used in database servers do not vary much with regard to their performance characteristics. They run at 10,000 or 15,000 rotations per minute and the average seek time is 3 or 4 milliseconds. Our suggested estimate for an average random read from a disk drive (10 ms) - including drive queuing and the transfer time from the server cache to the pool - is applicable for *all* current disk subsystems.

The time for a sequential read, on the other hand, varies according to the configuration. It depends not only on the bandwidth of the connection (and eventual contention), but also on the degree of parallelism that takes place. RAID striping provides potential for parallel read-ahead for a single thread. It is *strongly* recommended that the sequential read speed in an environment is measured before using our suggested figure of 0.1 ms per 4K page.

The sequential read may be measured by executing a singleton SELECT with a predicate that is never true. The predicate column should not be indexed - we want the optimizer to choose a full table scan. As all the rows are rejected, the CPU time will be less than the I/O time,

much less if we have long rows. If the table is large enough (at least 100,000 pages) and if the table pages are not in the pool, the I/O time will then be close to the elapsed time of the program divided by the number of pages in the table. With a fast disk server and striping, the expected time for scanning a table with 100,000 4K pages will be about 10 seconds.

In addition to the I/O time estimates, the cost of disk space and central storage influence index design.

Local disk drives provide physical data storage without the additional function provided by disk servers (such as fault tolerance, read and write cache, striping and so forth), for a very low price.

Disk servers are computers with several processors and a large amount of central storage. The most advanced disk servers are fault tolerant: all essential components are duplicated, and the software supports a fast transfer of operations to a spare unit. A high-performance fault tolerant disk server with a few terabytes may cost two million euros. The cost per gigabyte, then, is in the order of 500 euros (purchase price) or 50 euros per month (outsourced hardware).

Both local disks and disk servers employ industry-standard disk drives. The largest drives lead to the lowest cost per gigabyte; for example, a 145 GB drive costs much less than eight 18 GB drives. Unfortunately, they also imply much longer queuing times than smaller drives with a given access density (I/Os per gigabyte per second).

The cost of main storage has been reduced dramatically over the last few years as well. A gigabyte of RAM for Intel servers (Windows servers) now costs about 500 euros while the price for RISC and mainframe servers (UNIX, LINUX, z/OS) is in the order of 10,000 euros per gigabyte. With 32-bit addressing, the maximum size of a database buffer pool might be a gigabyte (with Windows servers, for example), and a few gigabytes for mainframes which have several address spaces for multiple buffer pools. Over the next few years, 64-bit addressing, which permits **much** larger buffer pools, will probably become the norm. If the price for central storage (RAM) keeps falling, database buffer pools of 100 gigabytes or more will then be common.

The price for the read cache of disk servers is comparable to that of RISC server central storage. The main reason for buying a 64GB read cache instead of 64GB of server central storage is the inability of 32-bit software to exploit 64 GB for buffer pools.

In our examples, we are using the following cost assumptions:

CPU time	5,000 euros per hour
Central storage	1,000 euros per month
Disk space	50 euros per month

These are possible current values for outsourced mainframe installations. Each designer should, of course, ascertain their own values, which may be much lower than the above.

DBMS Specifics

Pages

The size of the table pages sets an upper limit to the length of table rows. Normally, a table row must fit in one table page; an index row must fit in one leaf page. If the average length of the rows in a table is more than one third of the page size, space utilization suffers. Only one row with 2,100 bytes fits in a 4K page, for instance. The problem of unusable space is more pronounced with indexes. As new index rows *must* be placed in a leaf page according to the index key value, the leaf pages of many indexes should have free space for a few index rows, after load and reorganization. Therefore, index rows that are longer than 20% of the leaf page may result in poor space utilization and frequent leaf page splits. We have much more to say about this in Chapter 11.

With current disks, one rotation takes 4 ms (15,000 rpm) or 6 ms (10,000 rpm). Thus, the time for a random read is roughly the same for 2K, 4K, and 8K page sizes. Larger sizes add a couple of milliseconds to the transfer time. It is essential, however, that the stripe size on RAID disks is large enough for one page. Otherwise, more than one disk drive may have to be accessed to read a single page.

In most environments today, sequential processing brings several pages into the buffer pool with one I/O operation – several pages may be transferred with one rotation, for instance. The page size does not then make a big difference in the performance of sequential reads.

SQL Server 2000 uses a single page size for both tables and indexes: 8K. The maximum length of an index row is 900 bytes.

Oracle uses the term **block** instead of **page**. The allowed values for BLOCK_SIZE are 2K, 4K, 8K and 16K, but some operating systems may limit this choice. The maximum length of an index row is 40% of BLOCK_SIZE. In the interests of simplicity, we trust Oracle readers will forgive us if we use the term *page* throughout this book.

DB2 for z/OS supports 4K, 8K, 16K, and 32K pages for tables but only 4K pages for indexes. The maximum length for index rows is 254 bytes in V7 but this becomes 2,000 bytes in V8.

DB2 for LUW allows page sizes of 4K, 8K, 16K, and 32K for both tables and indexes. The upper limit for the index row length is 1024 bytes.

Table Clustering

Normally a table page contains rows from a single table only. Oracle provides an option to interleave rows from several related tables; this is similar to storing a hierarchical IMS database record with several segment types. An insurance policy, for instance, may have

rows in 5 tables. The policy number would be the primary key in one table and a foreign key in the other four tables. When all the rows relating to a policy are interleaved in one table, they might all fit in one page; the number of table I/Os required to read all the data for one policy will then be only one, whereas it would otherwise have been five. On the other hand, as older readers may remember, interleaving rows from many tables may create problems in other areas.

Index Rows

The maximum number of columns in an index varies across the current DBMSs: SQL Server 16, Oracle 32, DB2 for z/OS 64, and DB2 for LUW 16.

Indexing variable length columns have limitations in some products. If only fixed length index rows are supported, the DBMS may pad an index column to the maximum length. As variable length columns are becoming more common (because of JAVA, for instance) - even in environments in which they were rarely used in the past - support for variable length index columns (and index rows) is now the norm in the latest releases. DB2 for z/OS, for instance, has full support for variable length index columns in V8.

Normally, all columns copied to an index form the index key, which determines the order of the index entries. In unique indexes, an index entry is the same as an index row; in non-unique indexes, there is an entry for each distinct value of the index key – the pointer chain is normally ordered by the address of the table row. DB2 for LUW, for instance, allows non-key columns at the end of an index row.

Table rows

We have already seen that some DBMSs, for instance DB2 for z/OS, DB2 for LUW, support a clustering index, which affects the placement of inserted table rows. The objective is to keep the order of the table rows as close as possible to the order of the rows in the clustering index. If there is no clustering index, the inserted table rows are placed in the last page of the table or to any table page that has enough free space.

Some DBMSs, for example Oracle and DB2 for OS/400, do not support a clustering index that influences the choice of table page for an inserted table row. However with any DBMS, the table rows can be maintained in the required order by reorganizing the table frequently; by reading the rows via a particular index (the index which determines the required order) before the reload or by sorting the unloaded rows before the reload. The second alternative is often faster.

Index-Only Tables

If the rows in a table are not too long, it may be desirable to copy *all* the columns into an index to make *SELECT*s faster. The table is then somewhat redundant. Some DBMSs have the option of avoiding the need for the table. The leaf pages of one of the indexes then effectively contain the table rows.

In Oracle, this option is called an index-organized table and the index containing the table rows is called the primary index. In SQL Server, the table rows are stored in an index created with the option *CLUSTERED*. In both cases, the other indexes (called secondary indexes in Oracle and unclustered indexes in SQL Server), point to the index that contains the table rows.

The obvious advantage of index-only tables is a saving in disk space. In addition, *INSERT*s, *UPDATE*s and *DELETE*s are a little faster, because there is one less page to modify.

There are, however, disadvantages relating to the other indexes. If these point to the table row using a direct pointer (containing the leaf page number), a leaf page split in the primary (clustered) index causes a large number of disk I/Os for the other indexes. Any update to the primary index key that moves the index row, forces the DBMS to update the index rows pointing to the displaced index row. This is why SQL Server, for instance now uses the *key* of the primary index as the pointer to the clustered index. This eliminates the leaf page split overhead, but the unclustered indexes become larger if the clustered index has a long key itself. Furthermore, any access via a non-clustered index goes through two sets of non-leaf pages; firstly, those of the unclustered index and then those of the clustered index. This overhead is not a major issue as long as the non-leaf pages stay in the buffer pool.

The techniques presented in this book apply equally well to index-only tables, although the diagrams always show the presence of the table. If index-only tables are being used, the primary (clustered) table should be considered as a clustering index that is fat for all SELECTs. This last statement may not become clear until Chapter 4 has been considered. The order of the index rows is determined by the index key. The other columns are non-key columns.

Note that the primary (clustered) index does *not* have to be the *primary key index*. However, to reduce pointer maintenance, it is a common practice to choose an index whose key is not updated, such as a primary or a candidate key index. In most indexes (the non-key column option will be discussed later), all index columns make up the key, so it may be difficult to find other indexes in which *no* key column is updated.

Page Adjacency

Are the logically adjacent pages (such as leaf page 1 and leaf page 2) physically adjacent on disk? Sequential read would be very fast if they are (refer to Figure 2.9).

In some older DBMSs, such as SQL/DS and the early versions of SQL Server, the pages of an index or table could be spread all over a large data set. The only difference in the

performance of random and sequential read was then due to the fact that a number of logically adjacent *rows* resided in the same page (Level 1 in Figure 2.9). Reading the next page required a random I/O. If there are ten rows per page and a random I/O takes 10 ms, the I/O time for a sequential read is then 1 ms per row.

SQL Server allocates space for indexes and tables in chunks of 8 8K pages. DB2 for z/OS allocates space in “*extents*”. An extent may consist of hundreds of cylinders; all pages of a medium-size index or table are often in one extent. The logically adjacent pages are then physically next to each other. In Oracle (and several other systems) the placement of pages depends on file options chosen.

Three levels:

Read one page, get many rows

Read one track, get many pages

Disk server reads ahead from drives to read cache

- **Level 1 automatic**

If 10 rows per 4K page, then I/O time = 1 ms per row

- **Level 2 support by DBMS or disk subsystem**

May reduce sequential I/O time per row to 0.1 ms

- **Level 3 support by Disk Server**

May reduce sequential I/O time per row to 0.01 ms

Figure 2.9 *Page adjacency*

Many databases are now stored on RAID 5 or RAID 10 disks. RAID5 provides **striping** with **redundancy**. RAID 10, actually RAID 1 + RAID 0, provides **striping** with **mirroring**).

The terms redundancy and mirroring are defined in the glossary. RAID *striping* means storing the first stripe of a table or index (32K, for instance) on drive 1, the second stripe on drive 2, and so on. This obviously balances the load on a set of drives, but how does it affect sequential performance? Surprisingly, the effect may be positive.

Let us consider a full table scan where the table pages are striped over 7 drives. The disk server may now *read ahead from seven drives in parallel*. When the DBMS asks for the next set of pages, they are likely to be already in the read cache of the disk server. This combination of prefetch activity may bring the I/O time down to 0.1 ms per 4K page (Level 3 in Figure 2.9). The 0.1 ms figure is achievable with fast channels (2 GB/s fibre) and a disk server that is able to detect that a data set is being processed sequentially.

Alternatives to B-tree indexes

Bitmap indexes

Bitmap indexes consist of a bitmap (bit vector) for each distinct column value. Each bitmap has one bit for every row in the table. The bit is on if the related row has the value represented by the bitmap.

Bitmap indexes make it feasible to perform queries with *complex and unpredictable* compound predicates against a large table. This is because ANDing and ORing (covered in Chapters 6 and 10) bit map indexes is very fast, even when there are hundreds of millions of table rows. The corresponding operation with B-tree indexes requires collecting a large number of pointers and sorting large pointer sets.

On the other hand a B-tree index, containing the appropriate columns, eliminates table access. This is important, because random I/Os to a large table are very slow (about 10 ms). With a bitmap index, the table rows *must* be accessed unless the SELECT list contains only COUNTs. Therefore, the total execution time using a bitmap index may be *much longer* than with a tailored, (fat) B-tree index.

Bitmap indexes should be used when the following conditions are true:

1. The table has a large number of rows (at least 10 million).
2. The number of possible predicate combinations is so large that designing adequate B-tree indexes is not feasible.
3. The simple predicates have a high filter factor (considered in Chapter 3), but the compound predicate (WHERE clause) has a low filter factor – or the SELECT list contains COUNTs only.
4. The updates are batched (no lock contention).

Hashing

Hashing - or randomizing - is a fast way to retrieve a single table row whose primary key value is known. When the row is stored, the table page is chosen by using a randomiser, which converts the primary key value into a page number between 1 and N. If that page is already full, the row is placed in another page, chained to the home page. When a SELECT ...WHERE PK = :PK is issued, the randomiser is used again to determine the home page number. The row is either found in that page or by following the chain that starts in that page.

Randomizers were commonly used in non-relational DBMSs like IMS and IDMS. When the area size (N) was right - corresponding to about 70% space utilization, the number of I/Os to retrieve a record could have been as low as 1.1, which was very low compared to an index (a three-level index at that time could require 2 I/Os - plus a third to access the record itself).

However, the space utilizations required constant monitoring and adjustments. When many records were added, the overflow chains grew and the number of I/Os increased dramatically. Furthermore, range predicates tended to result in random I/Os, because the records had been randomised.

Oracle, for instance, supports hashing as an alternative to indexes.

The Many Meanings of Cluster

Cluster (the verb) is a term that is widely used throughout relational literature. It is also a source of much confusion, because its meaning varies from product to product.

In **DB2** (z/OS, LUW, VM and VSE), a clustering index refers to an index which defines the home page for a table row being inserted. An index is clustered if there is a high correlation between the order of the index rows and the table rows. A table can have only one clustering index but, at a given point in time, *several* indexes may be clustered. The **CLUSTERRATIO** of an index is a measure of the correlation between the order of the index rows and the table rows. It is used by the optimizer to estimate I/O times.

DB2 tables normally have a clustering index.

In **SQL Server**, the index that stores the table rows is called *clustered*; a clustered index is only defined if an index-only table is required. The other indexes (SQL Server term: “*non-clustered indexes*”) point to the “*clustered index*”.

In **Oracle**, the word cluster is used for the option to interleave table rows (clustered tables). *It has nothing to do with the clustering index* that we have taken to determine the sequence of the table rows.

DB2 for LUW V8 has an option called **multi-clustered indexes**, whereby the table is divided into compartments; the clustering index defines the home page in a compartment.

Important

In the diagrams throughout this book, **C** is used to *mark the index that defines the home page for a table row that is being inserted* (clustering index in DB2). In our calculations, *the table rows are assumed to be in that same order*. For a product that does not support a clustering index in this sense, the order of the table rows is determined when reorganizing and reloading the table.

CHAPTER 3

SQL Processing

Introduction

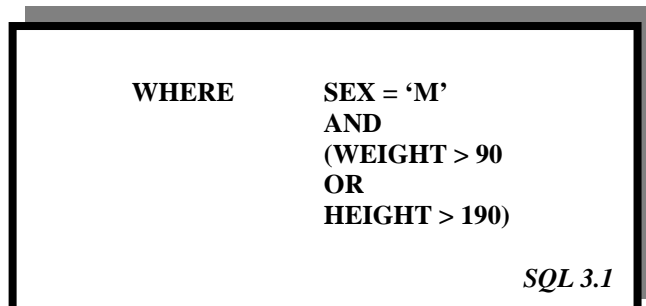
We now have some understanding of the structure of tables and indexes; we also understand how these objects relate to buffer pools, disks and disk servers, and how the latter are used to make the data available to the SQL process. We are now in a position to consider the processing of the SQL calls.

As before, much will depend on the individual relational DBMS being used; here again, there are many similarities but there are also a number of differences. We will describe the underlying processes first in general terms in order to avoid confusing the basic issues with DBMS specifics; the latter will be addressed as appropriate. Considerably more detail of these processes will be provided throughout the book, at a time when it is more convenient to do so.

Remember that the glossary provided at the end of the book summarises all the terms used throughout this chapter.

Predicates

A WHERE clause consists of one or more *predicates* (search arguments). Three *simple predicates* are shown in SQL 3.1. These are



```
WHERE      SEX = 'M'  
          AND  
          (WEIGHT > 90  
          OR  
          HEIGHT > 190)  
  
SQL 3.1
```

:

- SEX = 'M'
- WEIGHT > 90
- HEIGHT > 190

They can also be considered as two *compound predicates*:

- WEIGHT > 90 OR HEIGHT > 190
- SEX = 'M' AND (WEIGHT > 90 OR HEIGHT > 190)

Predicates are the primary starting points for index design. When an index supports *all* the predicates of a SELECT statement, the optimizer is likely to build an efficient access path.

Comment

In his numerous books, Chris Date uses the term *predicate* reluctantly (Reference 4):

“We follow conventional database usage here in referring to conditions as *predicates*. Strictly speaking, however, this usage is incorrect. A more accurate term would be *conditional expression* or *truth-valued expression*.”

Optimizers and Access Paths

One of the long standing advantages of relational databases has been that data is requested with little or no thought for the way in which the data is to be accessed. This decision is made by a component of the DBMS called the *optimizer*. These have, and probably always will,

vary widely across different relational systems, but they all try to access the data in the most effective way possible, using statistics stored by the system collected on the data. The optimizer is of course at the heart of SQL processing and is also central to this book as well.

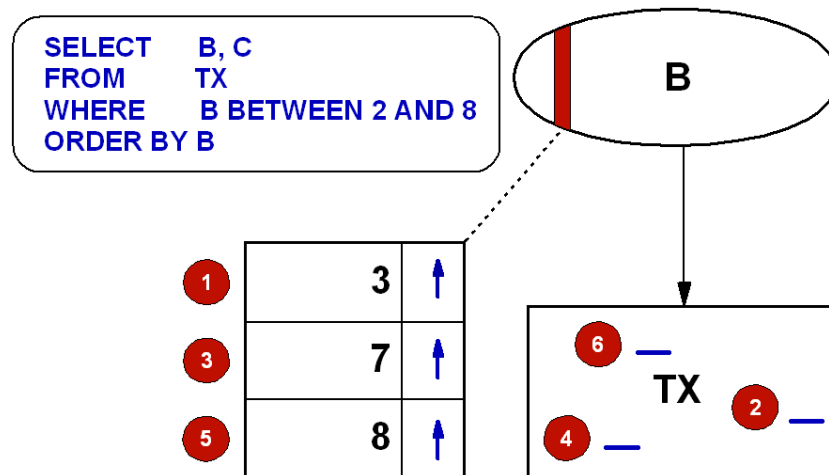


Figure 3.1 Traditional index scan

Before an SQL statement can be executed, the optimizer must first decide how to access the data; which index, if any, should be used, how the index should be used, should sequential or skip-sequential reads be used and so forth; all of this information is embodied in the *access path*. For the example we used in Chapter 2 and shown again in Figure 3.1, the access path would define a sequential scan of a portion of the index together with a synchronous read for each table page required.

Index Slices and Matching Columns

Thus a *thin slice* of the index, depicted in Figure 3.1 by the thin bar in the index B, will be sequentially scanned, and for each index entry having a value between 2 and 8 the corresponding row will be accessed from the table using a synchronous read unless the page is already in the buffer pool. The cost of the access path is clearly going to depend on the thickness of the slice which in turn will depend on the range predicate; a thicker slice will of course require more index pages to be read sequentially and more index rows will have to be processed. The real increase in cost though is going to come from an increase in the number of *synchronous reads* to the table at a cost of perhaps 10 ms per page. Similarly, a thinner slice will certainly reduce the index costs but again the major saving will be due to fewer synchronous reads to the table. The size of the index slice may be very important for this reason.

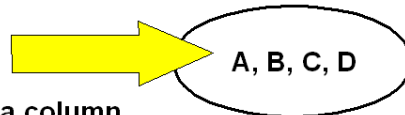
The term *index slice* is *not* one that is used by any of the relational database systems; these all use their own terminology but we feel that an *index slice* is a much more descriptive term and one that we can use regardless of DBMS. Another way that is often used to describe an index slice is to define the number of *matching columns* used in the processing of the index. In Figure 3.1 we have only a single matching column, B. This single matching column in fact defines the thickness of our slice. If there had been a second column, both in the WHERE clause and in the index, such that the two columns together were able to define an even thinner slice of the index, we would have 2 matching columns. Less index processing would be required, but of even greater importance, fewer synchronous reads to the table would be required.

Index Screening and Screening Columns

Sometimes a column may indeed be in both the WHERE clause and in the index, but for one reason or another it is not able to *further* define the index slice - this is a very complex area and one that will have to be addressed continually throughout this book; suffice it to say at this time, that not all columns in the index are able to define the index slice. Such columns, however, may still be able to reduce the number of synchronous reads to the table, and so play the more important part. We will call these columns *screening columns*, as indeed do some relational database systems, because this is exactly what they do. They avoid the necessity of accessing the table rows because they are able to determine that it is not necessary to do so by their very presence in the index.

Without going into too much detail at this stage, we can at least provide an initial simple understanding of how one can determine whether predicates may participate in this matching and screening process.

Examine index columns from leading to trailing



1. In the WHERE clause, does a column have at least one *simple enough predicate* referring to it?

If yes, the column is an **M column**.
If not, this column and the remaining columns are not **M columns**.

```
WHERE  A = :A
        AND
        B > :B
        AND
        C = :C
```

2. If the predicate is a *range predicate*, the remaining columns are not **M columns**.
3. Any column after the last **M column** is an **S column** if there is a *simple enough predicate* referring to that column.

Figure 3.2 Predicting matching and screening columns

Figure 3.2 shows a WHERE clause consisting of three predicates, the columns of each being part of the index shown; column D, the last column in the index, is not present in a predicate.

We must take the first index column A. This column is present in an equal predicate, the simplest predicate of all. This is obviously a matching column and will be used to define the index slice.

Next we take the second index column, B. This too, like the BETWEEN predicate we saw in Figure 3.1, is simple enough to be a matching column and again be used to define the index slice.

Item 2 in Figure 3.2 now indicates that because B is a *range* predicate, the remaining column C cannot participate in the matching process - it *cannot* further define the index slice. Item 3 goes on to say, however, that column C *can* avoid unnecessary table accesses because it can participate in the screening process. In effect Column C is *almost* as important as columns A and B; the index slice scanned will just be a little thicker.

The most difficult issue is what constitutes a simple or a difficult predicate; this depends very much on the DBMS and we can safely leave this until Chapter 6.

To summarise, the WHERE clause shown in Figure 3.2 has two matching columns, A and B, which define the index slice used. In addition it has one other column, C, which will be used for screening; thus the table will only be accessed when it is *known* that a row satisfies *all three* predicates. If the predicate on column B had been omitted, we would have a thicker index slice with only one matching column, A, but column C would still be used for screening. If the predicate on column B had been an equal predicate, *all three columns* would be used for matching, resulting in a very thin index slice. Finally, if the predicate on column A had been omitted, the index slice used would be the *whole index*, with columns B and C both used for screening.

Access Path Terminology

Unfortunately, the terminology used to describe access paths is far from standardized.

Matching predicates are sometimes called *range delimiting* predicates. If a predicate is simple enough for an optimizer in the sense that it is a *matching* predicate when a suitable index is available, it is called indexable (DB2 for z/OS) or sargable (SQL Server, DB2 for VM and VSE). The opposite term is non-indexable or non-sargable. Oracle books use the term *index suppression* when they discuss predicates that are too difficult for matching.

SQL Server uses the term *table lookup* for an access path that uses an index but also reads table rows. This is the opposite of index only. The obvious way to eliminate the table accesses is to add the missing columns to the index. Many SQL Server books call an index a *covering index* when it makes an index only access path possible for a SELECT call. SELECT statements that use a covering index are sometimes called *covering SELECTs*.

In DB2 for z/OS the predicates that are too complex for the optimizer in the sense that index *screening* is not enabled are called Stage 2 predicates. The opposite term is Stage 1. Many product guides do not discuss index screening at all. Hopefully this means that the products do index screening whenever it is logically possible; before making this assumption, however, it could well be worthwhile performing a simple experiment such as the following:

A table having four columns is created with 100,000 rows; the first three columns A, B and C, each have 100 different values. Column D is added to ensure that the table is accessed; it should be long enough to allow only one table row per page. An index (A, B, C) is created and **SELECT D FROM TABLE WHERE A = :A AND C = :C** is run to determine the number of *table rows* accessed. We are assuming that *all* the relevant rows are FETCHed and that the index is in fact used; with only one row per page and consequently having 100,000 pages in the table, it is unlikely that it wouldn't be. If the observed number of table pages accessed is close to 10 ($0.01 \times 0.01 \times 100,000$; the first 0.01 for matching, the second for screening), index screening must have been used for predicate **C = :C**. If it is closer to 1,000 ($0.01 \times 100,000$; only the first 0.01, used for matching) then it hasn't. This experiment can then be repeated at any time with complex predicates to determine whether they are too difficult for the optimizer.

Chapter 6 discusses in some detail difficult and very difficult predicates, together with the implications regarding matching and screening.

Monitoring the Optimizer

When a slow SQL call is identified, the first suspect is often the optimizer; perhaps it chose the wrong access path. A facility is available in relational DBMSs to explain the decision made by the optimizer. This facility is called EXPLAIN or SHOW PLAN and is discussed in Chapter 7. All references to EXPLAIN from now on should be taken to apply to both terms.

Helping the Optimizer (Statistics)

The optimizer may have made a wrong choice because the statistics it uses as a basis for the making cost estimates were inadequate; perhaps the options chosen when the statistics were gathered were too limited, or the statistics are out-of-date. These statistics are usually gathered on request, for example by running a special program called RUNSTATS in DB2 for z/OS.

The default statistics gathered normally include basic information, such as the number of table rows and pages per table, the number of leaf pages, the cluster ratio per index and the number of distinct values for some columns or column combinations (termed cardinality) as well as the highest and lowest values (or the second highest and the second lowest) for some columns. Other optional statistics provide more information about the value distribution of columns and column combinations, such as the N most common values together with the number of rows for each value; Oracle and DB2 for LUW, for instance, may also allow value

distributions to be collected as histograms (N% of rows occur in a user defined number of key ranges).

Helping the Optimizer (Number of FETCH Calls)

When cost based optimizers estimate the cost of alternative access paths, they assume that *all* the result rows are required (FETCHed), unless they are informed otherwise. If the whole result set is *not* going to be required, we can indicate that we are only interested in the first N result rows. This is done in SQL Server by adding:

```
OPTIONS (FAST n)
```

at the end of the SELECT statement.

With Oracle, an access path hint is used:

```
SELECT/*+ FIRST_ROWS(n)*/
```

(the (n) option is only available with Oracle 9i and above).

The syntax for DB2 for z/OS is:

```
OPTIMIZE FOR n ROWS
```

Examples are shown in SQL 3.2 S, O and D below.

```
DECLARE LASTINV CURSOR FOR
SELECT      INO, IDATE, IEUR
FROM        INVOICE
WHERE       CNO = :CNO
ORDER BY    INO DESC
OPTIONS (FAST 1)
```

SQL 3.2 S

```
DECLARE LASTINV CURSOR FOR
SELECT /*+ FIRST_ROWS(1)*/
      INO, IDATE, IEUR
FROM    INVOICE
WHERE   CNO = :CNO
ORDER BY INO DESC
```

SQL 3.2 O

```
DECLARE LASTINV CURSOR FOR  
SELECT      INO, IDATE, IEUR  
FROM        INVOICE  
WHERE       CNO = :CNO  
ORDER BY    INO DESC  
OPTIMIZE FOR 1 ROW
```

SQL 3.2 D

When the Access Path is Chosen

The only additional optimizer based question we need to understand at this stage is the extremely important one posed in Figure 3.3. It should be clear that choosing the access path *every time* an SQL statement is executed will consume much more processing power than doing it perhaps only once, when *the application is developed*; the cost of the access path selection process is not insignificant for cost based optimizers.

Far less obvious, although sometimes far more important, is that choosing the access path *every time* an SQL statement is executed, may give the optimizer a much better chance of choosing the *best* access path. This is because actual values, rather than program or host variables, are then used. **WHERE SALARY > :SALARY** is not nearly as transparent as **WHERE SALARY > 1,000,000**.

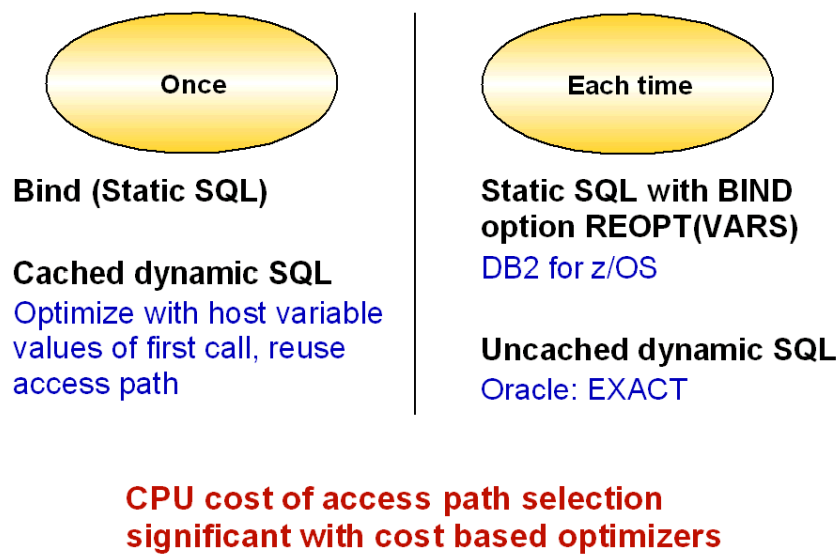


Figure 3.3 When does the optimizer select the access path

The items shown in Figure 3.3 indicate how optimizers allow this choice to be made; these again are discussed later.

Filter Factors

The *filter factor* specifies the selectivity of a predicate - what proportion of the source table rows satisfy the condition expressed by the predicate. It is dependent on the distribution of the column values. Thus, the filter factor of predicate **SEX = 'F'** increases whenever a female customer is added to the customer table.

The predicate **CITY = :CITY** has an *average filter factor* (1 / the number of distinct values of CITY) as well as a *value specific filter factor* (**CITY = 'HELSINKI'**). The difference between these two is critical, both for index design and for access path selection. Figure 3.4 shows an example, where a filter factor of 0.2% for the CITY column in a one million row customer table, predicts the size of the result set will be 2,000 rows.

When evaluating the adequacy of an index, *worst case filter factors* are more important than average filter factors; “*worst case*” relates to the *worst input*, that is the input which results in the longest elapsed time with a given index.

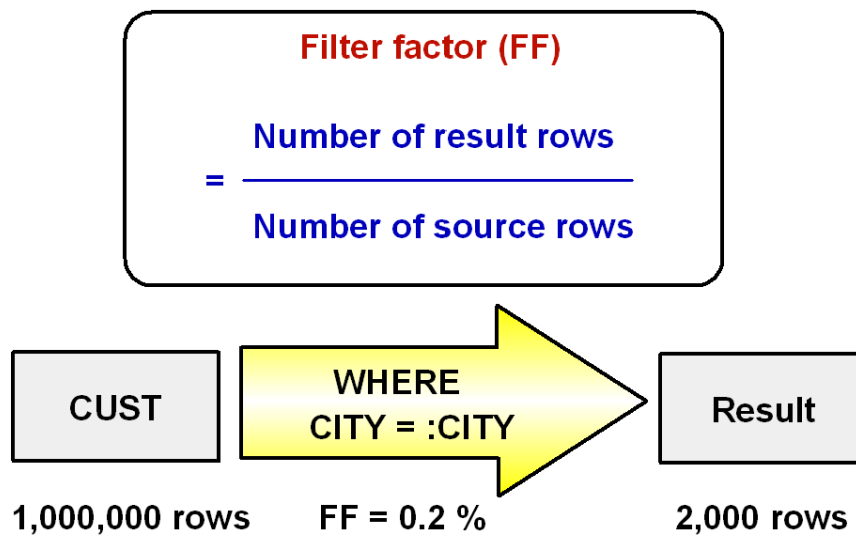


Figure 3.4 *Filter factor is a property of a predicate*

Filter Factors for Compound Predicates

The filter factor for a *compound predicate* can be derived from the filter factors of the simple predicates, only if the distribution of the values of the predicate columns are not statistically correlated. Consider, for example, the filter factor for **CITY = :CITY AND LNAME = :LNAME**. If there is no statistical correlation between the columns CITY and LNAME, the filter factor of the compound predicate will be equal to the *product* of the filter factors for **CITY = :CITY** and **LNAME = :LNAME**. If column CITY has 500 distinct values and column LNAME 10,000 distinct values, the filter factor for the compound predicate will be $1/500 \times 1/10,000$ or $1/5,000,000$. This implies that the column combination CITY, LNAME has 5,000,000 distinct values. In most CUSTOMER tables, however, the two columns *are* correlated. The proportion of Andersens is much higher in Copenhagen than in London, and there could even be cities in England that don't have a single customer with a common Danish last name. Therefore, the filter factor of this compound predicate is probably much lower than the product of the two simple predicates.

Columns CITY and BD (birthday, mmdd) are probably not related. Therefore, the filter factor of the compound predicate can be estimated by a process of multiplication as shown in Figure 3.5.

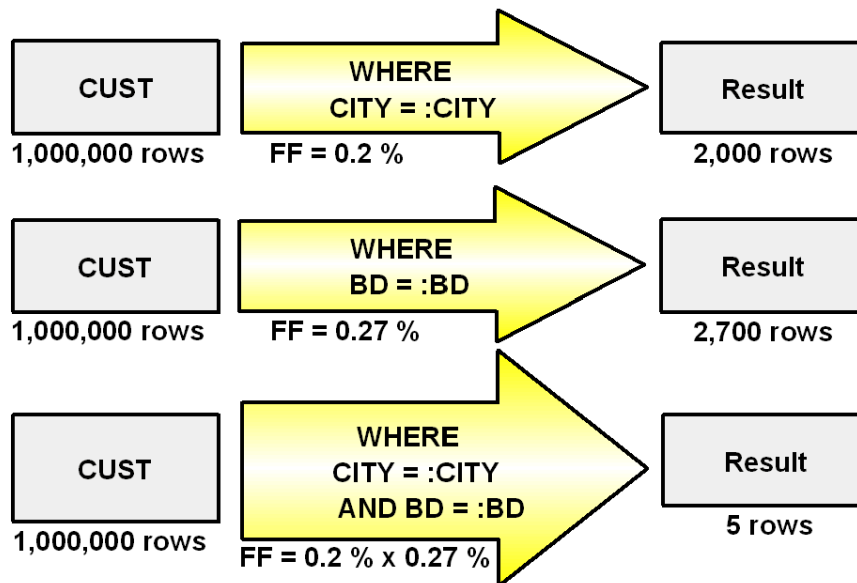


Figure 3.5 No correlation

An extreme example of column correlation is the table CAR, shown in Figure 3.6, with columns MAKE and MODEL. The filter factors for predicates **MAKE = :MAKE** and **MODEL = :MODEL** may be 1/100 and 1/1,000 respectively. The filter factor for **MAKE = :MAKE AND MODEL = :MODEL** is much less than 1/100,000, perhaps as low as 1/1,000 because, for example, to the best of our knowledge, only Ford made Model Ts.

When designing indexes, one should *estimate* the filter factor for a compound predicate *as a whole* rather than basing it on zero correlation. Fortunately, the estimation of the *worst* case filter factor is normally straightforward. If the *largest* filter factors for **CITY = :CITY** and **LNAME = :LNAME** are 10% and 1% respectively, the *largest* filter factor for **CITY = :CITY AND LNAME = :LNAME** is indeed $0.1 \times 0.01 = 0.001 = 0.1\%$ because it relates to the LNAME distribution of a *specific* CITY.

Optimizers also have to estimate the filter factors before they can estimate the costs for alternative access paths. When they were less sophisticated than they are today, an incorrect filter factor estimate for compound predicates was one of the most common reasons for inappropriate access path choices. Many current optimizers have options for counting or sampling the *cardinality* (the number of distinct values) of *index column combinations*

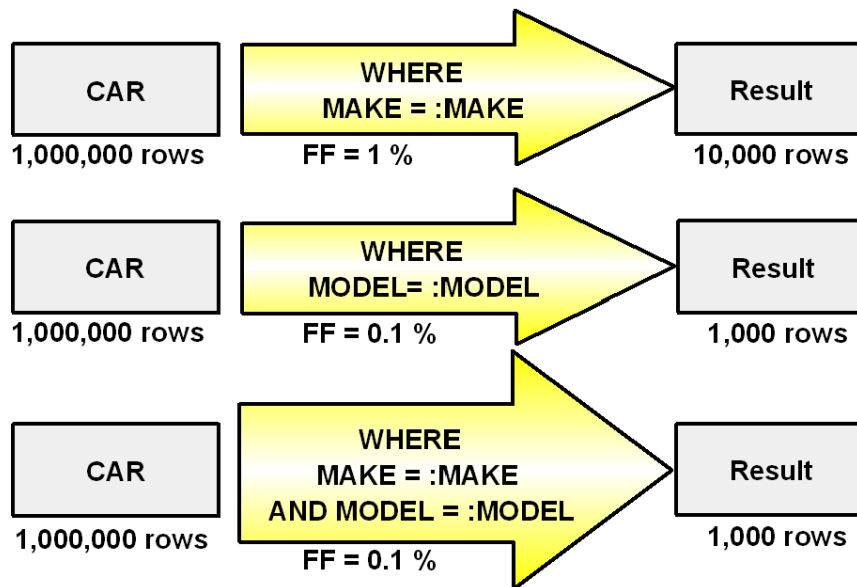


Figure 3.6 Strong correlation

Comment

A similar idea, sometimes discussed in relational literature, is the *selectivity ratio*. Vipul Minocha (Reference 3) defines this in the following way:

Selectivity ratio = (100 x (Total number of rows uniquely identified by the key) / Total number of rows in the table)).

If table CUST has one million rows and 200,000 of them have the value HELSINKI in column CITY, the selectivity ratio of index CITY is $100 \times 200,000 / 1,000,000 = 20\%$ for HELSINKI, equal to the filter factor of the predicate `CITY = 'HELSINKI'`. The index would therefore have poor selectivity, especially for the value HELSINKI, so CITY would not be a good index.

Selectivity is a somewhat confusing concept because it is a property of an index (and the whole index key); but the picture becomes even more confusing as a quotation from Reference 1 nicely warns: *Various definitions of selectivity from vendor manuals or other texts are imprecise or confusing: "the number of rows in the table divided by the number of distinct values" (Oracle); "the ratio of duplicate key values in an index" (Sybase). The phrase "high selectivity" means either "a large selectivity number" or "a low selectivity number" depending on who is saying it.*

We do not wish to add to this confusion; the *filter factor* is a more useful concept for index design, and an essential one for performance prediction. It will be widely used throughout this book.

The Impact of Filter Factors on Index Design

The thickness of the *index slice* that must be scanned is of importance to the performance of an access path. With current hardware, the most important measure of thickness is the number of index rows that must be scanned; the *total* number of index rows multiplied by *the filter factor of the matching compound predicate*. A *matching predicate*, by definition, participates in defining the slice; where to enter, where to exit. The number of index rows is the same as the number of table rows, with the exception that Oracle does not create index rows for NULL values. Consider the following query (*SQL 3.3*):

SELECT	PRICE, COLOUR, DEALERNO
FROM	CAR
WHERE	MAKE = :MAKE
	AND
	MODEL = :MODEL
ORDER BY	PRICE

SQL 3.3

Both simple predicates will be matching predicates if the index in Figure 3.7 is used. If the filter factor of the compound predicate is 0.1%, the index slice accessed will be 0.1% of the whole index. Columns MAKE and MODEL are *matching columns*. The index in Figure 3.7 appears to be fairly appropriate for the SELECT shown in *SQL 3.3*, although it is far from being the *best possible* index; at least the index slice to be scanned is rather thin.

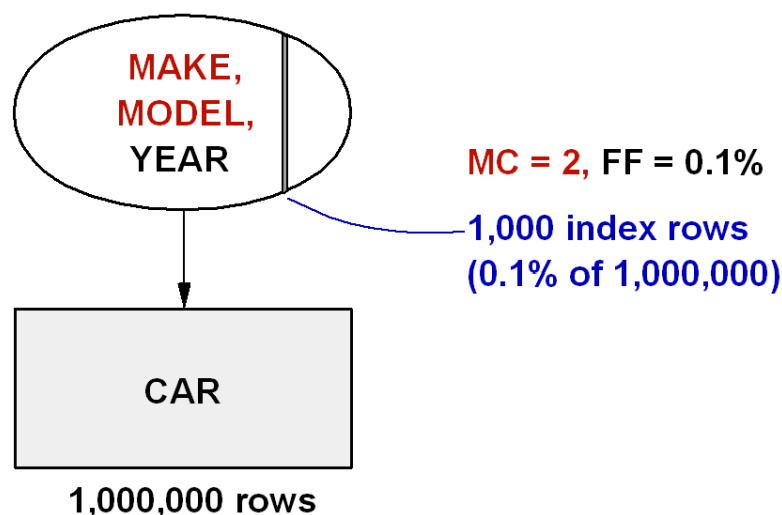


Figure 3.7 *Two matching columns - a thin slice*

The index is not as good, however, for the SELECT shown in *SQL 3.4*, because we have only *one* matching column. This is shown in Figure 3.8.

SELECT	PRICE, COLOUR, DEALERNO
FROM	AUTO
WHERE	MAKE = :MAKE
	AND
	YEAR = :YEAR

SQL 3.4

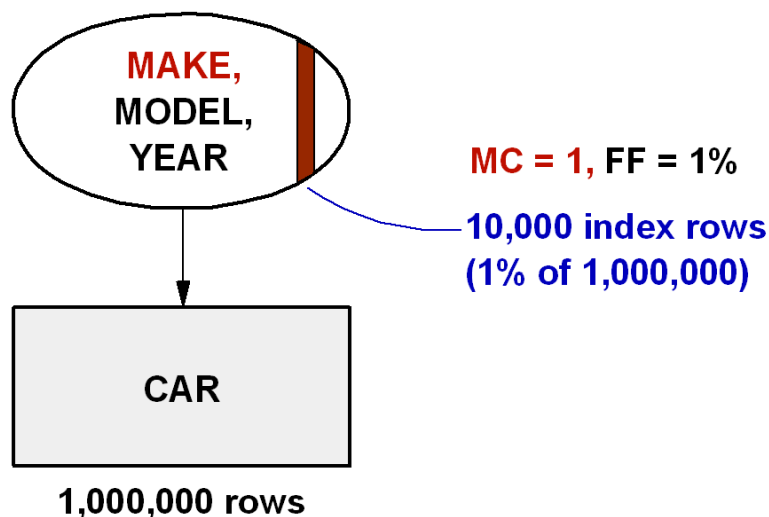


Figure 3.8 *One matching column - a thicker slice*

If index (SEX, HEIGHT, WEIGHT, CNO) is chosen to search for large men using the SELECT shown in *SQL 3.5*, there will be *only one* matching predicate, SEX, the filter factor of which is normally 0.5. The reason why only one predicate participates in the matching process will be discussed later.

Consequently, if there are 1,000,000 rows in table CUST, the index slice to be scanned will have $0.5 \times 1,000,000$ rows = **500,000 rows**, a very thick slice!

Some textbooks recommend that index columns should be ordered according to *descending cardinality* (the number of distinct values). Taken literally, this advice would lead to absurd indexes, such as (CNO, HEIGHT, WEIGHT, SEX), but in certain situations - assuming the order of the columns doesn't adversely affect the performance of SELECT statements (for example, with **WHERE A = :A AND B = :B**, indexes (A, B) and (B, A) would be equivalent) nor

that of updates - this is a reasonable recommendation. It increases the probability that the index will be useful for *other* SELECTs.

```

SELECT      LNAME, FNAME, CNO
FROM        CUST
WHERE       SEX = 'M'
           AND
           (WEIGHT > 90
           OR
           HEIGHT > 190)
ORDER BY    LNAME, FNAME

```

SQL 3.5

Materializing the Result Rows

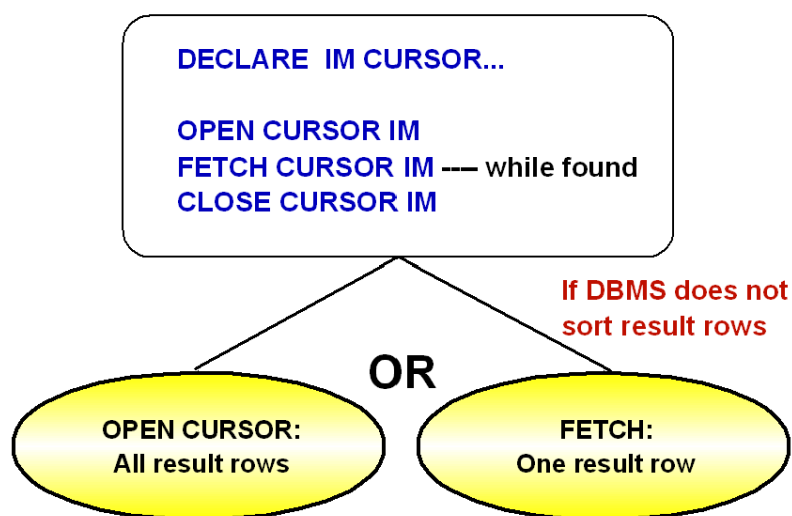


Figure 3.9 *Result row materialization*

Materializing the result rows means performing the *database accesses* required to *build the result set*. In the best case, this simply requires a row to be moved from the database buffer pool to the program. In the worst case, the DBMS will request a large number of disk reads.

When a single row is retrieved with a *singleton* SELECT, there is no choice but to materialize the result row when the SELECT call is executed. On the other hand, when a

cursor is used, which is necessary if there may be several result rows, there are *two* alternatives as shown in Figure 3.9:

1. the DBMS materializes the *whole* result table at OPEN CURSOR (or at least at the first FETCH).
2. each FETCH materializes one result row.

We will discuss each of these possibilities in turn.

Cursor Review

A FETCH call moves one row of the result table defined by the DECLARE CURSOR statement to the application program. When DECLARE CURSOR contains *host variables*, at execution time the application program moves the actual values into the host variables before the OPEN CURSOR call. If the application program creates several result tables using the same cursor, a CLOSE CURSOR call must first be issued. New values may be moved into the host variables and the cursor is reopened with an OPEN CURSOR call.

The CLOSE CURSOR call releases any locks that may have been taken by the last FETCH call, assuming the customary isolation level (ANSI SQL-92: READ COMMITTED, Level 1, for example, CURSOR STABILITY with DB2). If the program does not issue CLOSE CURSOR or if a stronger isolation level is specified, the DBMS releases the locks at commit point.

Query tools generate DECLARE CURSOR, OPEN CURSOR, a FETCH loop and CLOSE CURSOR from the SELECT statement.

SQL application programming courses have been known to claim that OPEN CURSOR *always* creates the result table, defined by DECLARE CURSOR with the values moved into the host variables, at the time of the OPEN CURSOR call. Fortunately, this statement is *not true*. To avoid unnecessary work, the DBMS materializes result rows *as late as possible*. The chosen materialization time may also affect the *contents* of the result: if the DBMS chooses *early materialization*, the FETCH calls retrieve result rows from a temporary table; the DBMS does not update this temporary table when the database is updated.

To illustrate the significance of the time of materialization, we will use the program shown in **SQL 3.6** which reads a customer's *last* invoice.

The response to the query *is always* a *single* row. The program will issue only one FETCH call, hence the request to use the most efficient access path for 1 row. The actual syntax is DBMS specific as we saw earlier; whenever we want to use this function throughout this book, we will use this “generic form” to avoid the necessity of providing multiple variations of the code. Despite making the 1 row request, what does the DBMS actually read?

1. the *whole* result table as defined by the cursor (*all* the customer's invoices)?

2. a *single* row relating to the required invoice?

```

DECLARE LASTINV CURSOR FOR
SELECT      INO, IDATE, IEUR
FROM        INVOICE
WHERE       CNO = :CNO
ORDER BY    INO DESC
WE WANT 1 ROW PLEASE

```

```

OPEN CURSOR LASTINV
FETCH CURSOR LASTINV
CLOSE CURSOR LASTINV

```

SQL 3.6

From the performance point of view, this may be a *critical question*. We will consider the two possible alternatives in turn.

Alternative 1: FETCH Call Materializes One Result Row

The DBMS chooses this desirable alternative if

- there *is no* sequence requirement (ORDER BY, GROUP BY etc) *or*
- there *is* a sequence requirement but the following conditions are *both* true:
 - there is an index which is able to provide the *result rows* in the ORDER BY sequence, for example (CNO, IDATE DESC).
 - The optimizer decides to use this index in the traditional way, that is, the first qualifying index row and its table row are accessed, followed by the second qualifying index row and its table row, and so on.

If the optimizer assumes that the program will fetch *the whole result set* it may select the wrong access path - perhaps a full table scan.

Alternative 2: Early Materialization

By far the most common reason for this choice is that a sort of the result rows is necessary; this will be reported by EXPLAIN or its equivalent. There are a few special cases, partly DBMS specific, in which the whole result is materialized even though the result rows are *not* sorted. These cases may also be detected by the EXPLAIN output.

In our example, the DBMS would have to sort the result rows if there isn't an index starting with either the columns CNO and IDATE or with just the column IDATE. If the DBMS cannot read an index backwards, the index column IDATE would have to be specified as descending: IDATE DESC.

If early materialization is chosen, some DBMSs materialize the result table at OPEN CURSOR, others at the first FETCH. It is only necessary to know this if reading an SQL trace. Otherwise it does not make a significant difference; it seems unlikely that a program would issue an OPEN CURSOR without issuing any FETCH calls to it.

What Every Database Designer Should Remember

*Sorting results rows implies that the whole result table is materialized even if only the **first** result row is fetched.*

Exercise 1

A - Design the best possible indexing for the following query

```
SELECT      LNAME, FNAME, CNO  
FROM        CUST  
WHERE       SEX = 'M'  
            AND  
            HEIGHT > 190  
ORDER BY    LNAME, FNAME
```

SQL 3.7

B - Design the best possible indexing for the following query

```
SELECT      LNAME, FNAME, CNO  
FROM        CUST  
WHERE       SEX = 'M'  
            AND  
            (WEIGHT > 90  
            OR  
            HEIGHT > 190)  
ORDER BY    LNAME, FNAME
```

SQL 3.8

Exercise 2

Evaluate the indexes you designed for 1A and 1B:

- How many matching columns (columns that define the index slice)?
- Does the DBMS have to sort the result rows before the first FETCH?
- Does the DBMS have to read table rows?

Exercise 3

Search the web and database books for advice: How to derive a good index for the SELECTs in Exercise 1.

access path, 33
 hint, 37
access pattern, 8
asynchronous read, 22
automating, 9
balanced tree index, 14
basic question, 10
bit vector, 29
bitmap index, 29

block, 25
browsing, 7
B-tree index, 5, 29
buffer pool, 5, 16
candidate key index, 27
cardinality, 41, 44
central storage, 5
cluster, 30
clustered, 30

Index

- clustered index, 27
- clustering index, 26
- column
 - fixed length, 26
 - restrictive, 4
 - variable length, 26
 - volatile, 7
- column correlation, 41
- control information, 15
- cost, 24
 - access selection, 38
 - assumptions, 24
 - CPU time, 24
 - disk servers, 24
 - disk space, 24
 - main storage, 24
 - storage, 24
- covering index, 35
- CPU time, 24
- cursor, 46
- DB2, 7, 22, 25, 26, 28, 30, 35, 36, 46
- disk
 - load, 7
- disk drive, 23
- disk server, 5
 - cache, 17
- disk storage, 5
- disorganization, 6
- drive queuing, 8, 23
- education, 4
- exception monitoring, 7
- EXPLAIN, 36, 47
- extent, 28
- fault tolerance, 24
- filter factor, 39
- FIRST_ROWS(n), 37
- frequently used data, 18
- guidelines, 4
- hardware capacity, 9
- hashing, 29
- histogram, 37
- home page, 30
- inadequate indexing, 4, 7, 10
- index
 - backwards, 48
 - candidates, 11
 - composite, 7
 - design method, 9
 - levels, 14
 - maintenance, 5
 - non-unique, 14
 - row, 14
 - unique, 14
- index slice**, 43
- index suppression, 35
- index-organized table, 27
- insert
 - rate, 7
- Intel servers, 24
- interleave, 25
- join, 11
- key
 - foreign, 9
 - modified, 7
 - primary, 9
 - sequence, 7
- leaf page, 5
- leaf page split, 27
- list prefetch, 22
- local disk drives, 24
- local response time, 7
- mainframe servers, 24
- matching columns, 34
- materialization, 45
 - early, 47
- matrix, 9
- mirroring, 28
- misconceptions, 5
- monitoring software, 10
- multi-clustered indexes, 30
- myths, 5
- non-clustered *indexes*, 30
- non-indexable, 35
- non-leaf page, 5
- non-sargable, 35
- OPTIMIZE FOR n ROWS, 37
- optimizer, 32
 - cost based, 10
- OPTIONS (FAST n), 37
- Oracle, 25, 26, 28, 30, 35, 36, 37, 43
- page, 14, 25
 - adjacency, 27
 - number, 14
 - size, 25
- predicate, 32
 - compound, 32, 40
 - difficult**, 35
 - matching, 35
 - really difficult**, 36
 - simple**, 32
 - stage 1, 36
 - stage 2, 36
- prediction formulae, 10
- prefetch, 19
- production, 4
 - cutover, 4, 10
 - workload, 9
- QUBE**, 10
- RAID 10, 28
- RAID 5, 28
- RAM, 24
- random read, 8, 20, 23
- random write, 8
- randomiser, 29
- read cache, 5
- redundancy, 28
- redundant data, 8
- reorganization, 26

Index

response critical, 8
RISC, 24
root page, 5
row, 25
RUNSTATS, 36
sargable, 35
screening, 34
seek time, 23
selectivity ratio, 42
sequential read, 23, 27
server
 cache, 18
SHOW PLAN, 36
sixty four bit, 24
skip-sequential, 20
SQL Server, 25, 27, 30, 37
statistics, 33
striping, 24, 28
synchronous read, 22
systematic index design, 10
table
 lookup, 35
textbooks, 4
thick slice, 33
thin slice, 33
three star index, 11
UNIX, 24
Windows servers, 24