Proactive Index Design using QUBE
Courtesy of Tapio Lahdenmäki

Lauri Pietarinen
Relational Consulting

Session Code: F04
Nov 09, 2010 8:30 – 9:30 AM | Platform: z/OS, LUW
The QUBE-formula was developed by Tapio Lahdenmäki and others at IBM-Finland during the 80’s and 90’s to quickly estimate the cost of a query using a given access path. It has its roots in the 70’s in the world of estimating DL/I-queries. It was originally created for z/OS DB2 but has been successfully used with DB2 LUW, Oracle and SQLServer.

**Quick** – The formula is very simple and quick to apply

**Upper Bound** – The formula makes some assumptions that may be pessimistic – hence there will be “false positives”, queries flagged as problems

**Estimate** – No formula can give an exact cost for a query, not even the complex one used by the cost based optimizer of DB2! Because of the simplicity of QUBE the estimate will tend to be rather crude, and should be seen as a quick first step

This presentation will first quickly go over some basic indexing principles and terminology. Then, using a simplified real world case from an insurance company we will calculate and compare QUBE-estimates for a given query using different indexing scenarios, including fat- and semifat indexes. We will tackle some common indexing myths and address insert- and update-concerns of indexes. Finally we will discuss the question “Who is responsible for the designing of indexes?”

This presentation will also introduce a way of visualizing indexes that can be applied to every day work whether one uses QUBE or not.

Tapio Lahdenmäki was involved with database performance since 1975 -- in IBM until 2002, then as an independent consultant. He retired last year. In IBM, he was the main developer of several DB2 for z/OS performance
We introduce four basic ways to build the result table for this query to search for a given customers recent invoices:

(1) Scan the whole table, find the qualifying rows
All 1,000,000 table rows touched, sequential read

(2) Read an index slice (up to 10,000 entries) to find the table rows with given CNO
As index rows are ordered by CNO, reading an index slice is sequential read. If table rows are clustered by CNO, a table slice is read sequentially, otherwise the table is accessed randomly.

(3) Read an index slice (up to 300 entries) to find table rows with given CNO and recent IDATE
Index rows are ordered by CNO and IDATE. If table rows are ordered by CNO and IDATE, a table slice is read sequentially, otherwise, random access to table

(4) Read an index slice (up to 300 entries, sequential read), no table access

The number of touches is the basic measure for the cost of an access path
One touch = the DBMS reads an index row (an index entry) or a table row
The T values on the visual relate to the worst input: the biggest customer (1% of invoices) and the lowest allowed IDATE value (3% of invoices)
The index entries on the leaf pages are stored in index key sequence (assuming the index is in good shape). This is why reading an index slice (such as COL BETWEEN 2 AND 10) is very fast; it is sequential read.

**Note the chain connecting the index entries in index key sequence -- this is why sort for ORDER BY is avoided with an appropriate index, even after many inserts**

Note also, the empty slot in each index page: this corresponds to a 25% freespace-value for the index.

The number of non-leaf pages is often 1 or 2 % of the number of leaf pages (the DBMS does not need to store the whole key on non-leaf pages)

Today, with RAMs of several gigabytes, the non-leaf pages of active indexes tend to stay in the buffer pool

Realistic assumption: When a program needs a leaf page for the first time, it must read it from disk. Once in pool, a leaf page stays there for a minute or so after each reference

**SELECT...WHERE COL BETWEEN 2 AND 10 --- Read two adjacent leaf pages from disk**
Recommended Mental Image

It is good to think of the B-Tree index as a table: as many rows (index rows) as in the table the index is pointing to, perfect order by index key.

The non-leaf page accesses to find the right leaf page are ignored and so is the processing to find the beginning of the index slice.

The concept of a **touch** is based on this mental image

Scanning an index slice takes one random touch and N sequential touches, where N is the number of rows in the index slice (when the DBMS finds a row with COL > 10, it knows it's time to exit)

On the visual, the table rows are not in the same sequence as the index rows; therefore, all table touches are random

If the table rows were clustered by COL, a table slice would be read: one random touch and two sequential touches in this case

**The number of touches = TR + TS**

* A touch is sequential if it reads the next row according to the mental image
* A touch is random if it is not sequential
Case Study: Request Tracking

Our running example is a request tracking application for an insurance company.

- There are about 1,000 requests entered into the system each day.
- There are 100 branch offices, the largest one covers 3% of the requests
- For the life cycle of each request there are on average 5 status changes
- Each request has a dead line
- When the request is closed (status=9) the dead line is updated to 31.12.2099
- 99% of all requests are closed, 1% open (status 1-8)

The application has been in production for a couple of years and the response times where excellent when this application was taken into production.

Table REQUEST keeps growing; old rows are not deleted.

Some users are now unhappy with the response times. It sometimes takes more than 10 seconds to get the first screen.

With outsourced hardware, the billing is mainly based on CPU seconds. The CPU time consumed by this application has become significant although the number of transactions is much lower than that of the main applications.

Note: CPU-billing costs is mainly a problem with z/OS DB2, not so with LUW DB2
The end users check their work queues several times a day. The first transaction opens this cursor and reads all result rows with a FETCH loop.

The largest branch office may have 300 result rows (open requests): 0.01 x 0.03 x 1,000,000

There are 20 result rows per screen

The user often moves from the first screen (most urgent requests) to another transaction

The filter factor of a predicate is the number of qualifying rows divided by the number of source rows

FF=0% → no rows qualify, predicate is false for all rows
FF=100% → all rows qualify, predicate is true for all rows

The filter factor for predicate STATUS<9 is 1% because 1% of the REQUEST rows satisfy that predicate.

The biggest Branch Office (BO) has 30,000 REQUEST rows, thus the filter factor for predicate BO = :BO is up to 3%
Which One is Faster?

Assume the optimizer chooses Alternative 1

A common panic reaction: Make the optimizer some how choose Alternative 2 – by falsifying statistics, for example -- because full table scan has a bad reputation

The numbers on the slide relate to the worst input (the biggest branch office).

Would the end users be more happy with Alternative 2?
To answer this question, simply counting the touches (one million vs. 60,000) is not enough; the difference between random and sequential touches must be quantified as well. This will be done in the following slides.
Sequential read from disk has become dramatically faster in the last 20 years.

1. Thanks to the increase in magnetic storage density, a physical track contains much more data than before.
2. As drives have become smaller, the rotations per minute has gone up to as high as 15,000 RPM. One rotation (4 ms) may bring more than 100 4K pages into a drive cache.
3. RAID striping -- first set of pages to drive 1, the second set to drive 2, etc. -- enables reading ahead from several drives in parallel.
4. Data paths are becoming faster.
   The two numbers for CPU time are rough upper-bound estimates (up to 5 us, up to 50 us).

As CPU time and I/O time overlap in sequential read, long sequential scans -- such as a full table scan -- may be CPU-bound or close to it. Therefore, they may cause significant CPU queuing to other concurrent processes, especially if there are only one or two processors.
Even with large buffer pools and read caches, random touches to a large table are likely, in most cases, to require reads from disk drives because the total size of files behind the read cache is normally more than a terabyte (1000 GB).

Because of increasing drive queuing, a random read from drive may take more than 10 ms (next slides)

This is why 10 ms per random touch is still a good rule of thumb (a weighted average of fast reads from cache and slow reads from disk drive) -- at least for fairly large indexes and tables with truly random processing

CPU time per random touch may be as high as 50 us because, by definition, CPU cache hit ratio is low.

It is interesting to note, that while sequential read speed has increased dramatically in the last 30 years, random read speed has barely changed. The value used in the 70's for random read was 30 ms!

For more information on the differences between random and sequential read see article/presentation by Patterson: *Latency Lags Bandwith*
With the fastest currently available disk drives (15,000 RPM), the average random read keeps the drive busy for about 6 milliseconds.

Queuing theory tells us that when a drive is busy for 30% of the time (with many users issuing random queries), the average drive queuing time is a few milliseconds.

Much higher drive busy levels are becoming common as drives are getting larger but not faster (next slide).

**Note:** Flash-drive technology is changing this equation in the future. Flash-drives are still fairly expensive and there are some reliability issues waiting to be solved. Since it will take some time for most or all of the data to be stored in flash-memory, there is also a question of using Flash drives for the right purpose (i.e. getting the most benefit out of it).
As the price per gigabyte is lower with large drives, we may have to learn to live with long drive queuing times.

For instance, replacing two 145 GB disk drives with individual 30% busy times with one 300 GB disk drive will triple queuing time and hence increase average random read from 10ms to 16ms (other factors remaining constant)!

Thus, minimizing the number of random touches -- with better indexes -- is increasingly important.

There seems to be a constant struggle between SAN-managers and DBA’s on allocating disks for databases. Disk is often seen as a "bulk" resource by the SAN-managers, but the number of individual rotating spindles is what counts.
Quick Upper Bound Estimate (QUBE)

\[
\begin{align*}
ET &= TR \times 10 \text{ ms} + TS \times 0.01 \text{ ms} + F \times 0.1 \text{ ms} \\
CPU &= TR \times 50 \text{ us} + TS \times 5 \text{ us} + F \times 50 \text{ us}
\end{align*}
\]

\begin{align*}
ET &= \text{Elapsed time (SQL)} \\
CPU &= \text{CPU time (SQL)} \\
TR &= \text{Number of random touches} \\
TS &= \text{Number of sequential touches} \\
F &= \text{Number of rows returned to program (Fetches)}
\end{align*}

When index alternatives are compared, it is time-consuming to estimate the number of leaf pages in each index candidate. QUBE (0.01 ms per sequential touch) is on the safe side if the rows are not exceptionally long.

The purpose of QUBE is to evaluate current indexes and index candidates: Is a given index good enough for a given SELECT, even with the worst input?

The worst input = the predicate column values that result in the longest ET. In our case study, the worst input is the BO of the biggest branch office (FF = 3%)

**Upper-bound** implies a conservative estimate; in most cases the actual ET is less than QUBE, mainly because random touches do not always result in a read from disk drive -- hot spots tend to stay in the RAM pool or in the disk cache, for instance (in other words, random touches are not always truly random).

**Note:** The sort cost is not included because the sort time is less than the retrieval time (0.01 s per sequential touch) with large sort pools typical today -- even if the number of rows to be sorted is, say, one million.

**Note:** The cost of optimizing, which might be a factor in dynamic SQL is not considered in QUBE. In practice a heavily used dynamic query should use bind-variables, so the pre-optimized query would be found in the dynamic statement cache.
This is the current access path for the query (table scan).

Response time is too long -- note that this is the average response with the worst input; some transactions take much longer because of random variation in queuing times.

CPU time is very high for an operational transaction (expensive, causes CPU queuing)

**Note:** We have slightly reformatted the QUBE-formula for convenience.

\[
ET = ( TR + TS/1000 + F/100 ) \times 10 \text{ ms} \\
CPU = ( TR + TS/10 + F ) / 20
\]
Let us assume that the table rows are **not** in BO sequence (BO is not a clustering index).

Then, the I/O time to read 30,000 table rows (worst input) is far too long.

CPU time is a bit shorter than with alternative 1 but still much too high.

It would be a big mistake to force the optimizer to choose this access path.
Now the table rows are assumed to be exactly in the same order as the rows in index BO, i.e. index BO is the clustering index.
Index (STATUS, BO) is much better than index BO because table touches occur only when the whole WHERE clause is true.

**Basic Question (BQ)** is the simplest possible recommendation for index design. It can be used already when a SELECT is being written. It reveals many index problems early -- but not all of them.

Index BO does not pass the BQ test
Index (STATUS,BO) does pass, so it is, by definition, a semi-fat index for the given WHERE clause
This is a better index:
(1) Two matching columns, thinner index slice
(2) Replaces index BO

Why two matching columns?
The index rows that relate to result rows are next to each other
In other words, no index rows in the index slice are rejected
This was not the case with semi-fat index (STATUS,BO)
QUBE for Semi-Fat Index – Your Turn!

Now is your turn to try out QUBE!

Calculate the ET and CPU for the biggest branch office using the given index (BO,STATUS).

Is this index good enough even for the biggest branch office (300 result rows, 300 FETCHes)?

**Note:** The table rows are not clustered by BO, STATUS

MC = The number of matching columns
SC = The number of screening columns
IXONLY = Index only access: yes/no
SORT = DBMS performs a sort: yes/no
TR = Random touch
TS = Sequential touch

**Note:** One can think of matching columns (MC) defining the thickness of the index slice and screening columns (SC) eliminating rows from that slice

You will find the correct answer on the third last page of this presentation
Still Too Long – What Next?

The problem: 300 random table touches

\[300 \times 10 \text{ ms} = 3 \text{ s}\]

- 20 FETCHes
  - 20 table touches?

- Fat index
  - No table touches

Two alternatives (if table row clustering is out of the question):

(1) The application program builds only a screen at a time. This makes sense if each screen can be built with 20 table touches.

(2) Eliminate table touches by adding columns to the index (fat index, covering index, index only access path)
When Do Touches Take Place?

If the result rows do not come from the database in ORDER BY sequence, the DBMS must read and sort all result rows before the first FETCH.
Assume the program is modified so that it issues max 20 FETCHes before sending a screenful of result rows to the user.

The result index rows, not necessarily all index rows, must be in ORDER BY -sequence -- that is why the index may start with BO (BO = :BO)

For efficient repositioning, the index (and the result rows) should be unique. Hence the added index- and ORDER BY- column RPK.

**Note:** Repositioning (or fetching next set of 20 rows) is not covered in this presentation
Is the last alternative good enough? Probably not, if the touches are truly random.

We will explore one more alternative in the next slide.
Fat Index with Sort

A fat index is easy to implement

No changes to the program: FETCH all result rows

Sorting 300 rows is not a significant cost

Is this index adequate even for the worst input?

MC = Number of M columns
SC = Number of S columns
IXONLY = Index only (Y/N) -- Y if no table touches
SORT = Sort in access path (Y/N)
Worst-Input Estimates, with Fat Index

<table>
<thead>
<tr>
<th></th>
<th>Elapsed time (ET)</th>
<th>CPU time</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>No index</td>
<td>10 s</td>
<td>5 s</td>
<td></td>
</tr>
<tr>
<td>BO (non-C)</td>
<td>300 s</td>
<td>2 s</td>
<td></td>
</tr>
<tr>
<td>BO (C)</td>
<td>1 s</td>
<td>0.3 s</td>
<td></td>
</tr>
<tr>
<td>BO, STATUS</td>
<td>3 s</td>
<td>0.03 s</td>
<td></td>
</tr>
<tr>
<td>BO, DL, RPK, STATUS</td>
<td>0.2 s</td>
<td>0.002 s</td>
<td>First screen Modify ppm</td>
</tr>
<tr>
<td>BO, STATUS, RPK...</td>
<td>0.04 s</td>
<td>0.02 s</td>
<td></td>
</tr>
</tbody>
</table>

The fat-index alternative would seem like the best one in this case. Added benefit: no program modifications!

Actually, not a new index, just a modified “BO”-index.
What is the cost of “fattening” index “BO”?

Issues to be considered include
- Disk space
- Added RAM usage of non-leaf pages
- Effect on INSERT-, UPDATE- and DELETE-performance
- Extra effort needed for index reorg
An example:
Delete 100,000 table rows from a large table with 10 indexes
100,000 x 10 ms x 10 = 10,000 s = 3 hours
A possible solution: Run several delete jobs in parallel
A column added to an existing index (via drop/create, naturally) will not incur any additional cost on delete.

Adequate free space (and using a larger page size) will help mitigate the effect of a longer key.
So, Too Expensive?

Compared to the dramatic effect the new index has on the original request-tracking query, the cost of the fattened index is indeed reasonable.

Note also, that the improved efficiency of the request tracking query will more that compensate for the added I/O and CPU-load on index maintenance.
There are lots of “old wives’ tales” to be found in various database performance books on sale that might have been relevant in the 80’s when disk space was expensive and the difference between random and sequential read was not as dramatic as it is today.

The random touches caused by INSERT/UPDATE/DELETE are normally the biggest potential issues limiting the number of indexes.

From the elapsed time point of view, the number of indexes is critical particularly if one transaction -- or a massive batch job -- adds or deletes many table rows

With QUBE, the effect of N indexes is easy to predict
Many tables tolerate 10 indexes
Index reorg requirements vary a lot -- our example was a tough one in this respect
Index BO was not Adequate for this Select

```
SELECT DL, STATUS, RPK, CNO, C1, C2
FROM REQUEST
WHERE STATUS < 9
    AND BO = :BO
ORDER BY DL
```

Who should have seen this?

When?

Time for discussion. Who should be responsible for getting index design right?
Do your application programmers understand indexing?
Does your DBA understand the application?
Do they talk to each other?
Are they allowed to talk to each other?
Or maybe it’s the testers who should have thought of this?
Etc...
This is the answer to the “QUBE quiz” presented earlier in this presentation.
Indexing seems to be a "dark art" and there is a tendency by the inexperienced to delegate it to tools or totally ignore it. But if we don’t understand the principles behind indexing we will be captive of outdated "best practices" and "rules of thumbs".

QUBE can help make sense of the basic principles behind indexing and it can be used to make an **educated guess** on the effect of an indexing solution before the fact. It can also be helpful in understanding and analyzing existing performance problems.

QUBE is naturally just a method among others and will not replace monitoring, explain or other existing tools or methods.
Lauri Pietarinen
Relational Consulting
lauri.pietarinen@relational-consulting.com
F04
Proactive Index Design using QUBE