Assignment 2 - Database Tuning F2012

Theo Andersen - tha@itu.dk

April 24, 2012

1 Introduction

This paper describes the findings of the experiments done to fulfill the three parts of Assignment 2 for the course: SDT1 - Database Tuning F2012.

Experiments were done using the *STD1_ V2* Virtual box image provided by Philippe, and was executed on a Mac-book Pro Intel Core i5 2.4GHz (dual core) with 4 GB ram. 1GB of ram was given to the virtual machine and it was allowed to use the two cores on the processor. The hard-drive is a standard Mac hard-drive - Hitachi 5400RPM 320GB. The client python script was executed from the same virtual machine as the DB2 database server was running on.

2 Part 1 - Writes

2.1 A - Measuring write performance

How fast can you insert 10000 tuples in your out-of-the-box system?

To test this, i used the scripts provided to create 100000 tuples, and used the writes.py to insert them into the db. The execution time was (as with the last assignment) different depending on the number of threads. Running the python script and inserting the 100000 tuples i got a best-response time at 7,1 seconds.

Is that good compared to the disc/file system throughput?

Using the tool DD and a benchmark tool called Bonnie++ i measured the write throughput and found it to be around 41 MB/sec.

Now the 100000 tuples was at 1.5 MB in the accounts.data file that the gentable.py file had produced. So this would mean that the io write throughput must have been roughly around 1.5mb/7sec = 0.2 MB/Sec which is terrible in comparison. Further more the data that was inserted must be smaller because this file is text-encoded.

The explanation is that a database isn't just a bucket of data. The data saved has format and constraints, and the database has to be able to provide consistency, concurrency and recovery options etc.. Further more that database doesn't even write the data to disc to begin with, but writes to buffer and the log and then writes the data asynchronously, to be able to schedule the actual data write as efficient as possible. So the delay we are seeing here is probably not disc throughput contention, but more to do with the other operations of the database.

Modifying the table-space parameters

I considered the following table-space parameters for modification:

Page Size Might have an effect, if the process of identifying which page to insert to has some overhead (so that larger pages make many inserts faster).

Extent Size Only have an effect when there is more than one container, and per default there is only one.

Prefetch Size This only affects how much is read on reads, this wont affect writes.

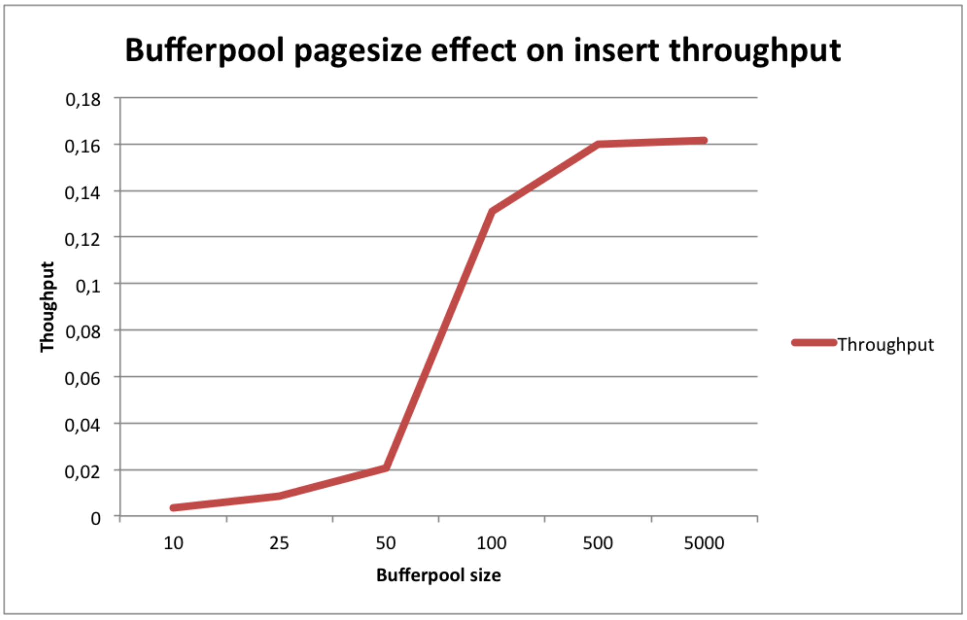
Overhead and transfer rate This is used by the optimizer to calculate costs to create the right execution plan, but then wont really affect our simple inserts.

Buffer-pool size Size of the buffer-pool used by the table-space. This might very well affect insert performance.

I have chosen to concentrate on experimenting with the page size of the table-space and the size of the buffer-pool.

To change the page size i had to first create a new buffer pool with the page size, and then create a new table-space with the page size and referencing the previously created buffer pool. The default page size is 4k, so i selected the one most different from that being the largest at 32k. Dropping the Accounts table and then recreating it in the new table-space i tried running the writes again. This gave the same performance at around 7 seconds, which seems like that the size of the pages doesn't really affect write performance after all.

Adjusting the size of the buffer-pool size (number of pages that can be stored in the buffer-pool) and running the inserts again, produced the following graph.



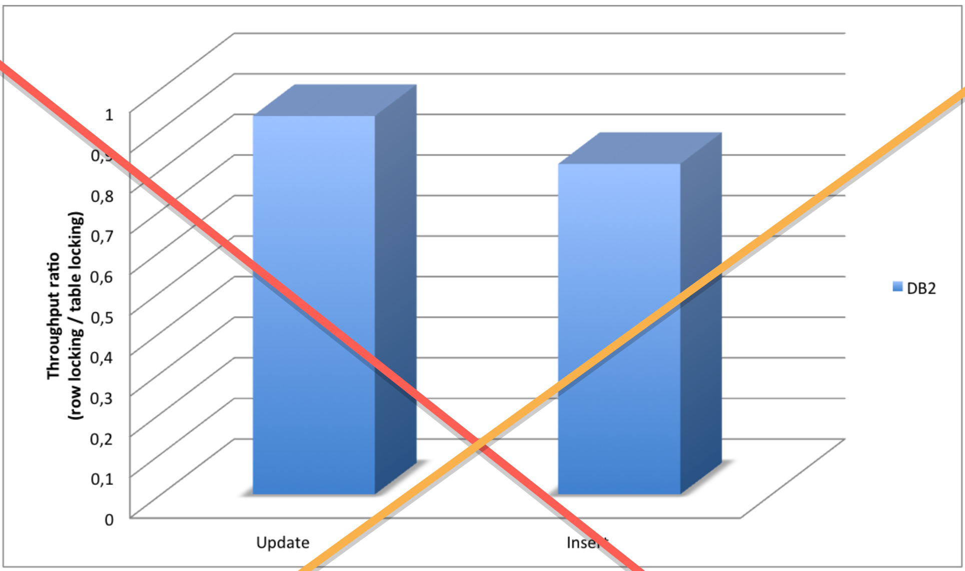
The buffer pool size has a lot to say about insert performance. When a row is inserted, data is put in the buffer and the operation is logged. This way the database system can defer the actual IO's to disc to another time which is more optimal (disc io being slow). But if an insert operation doesn't have enough space in the buffer pool, it will be forced to either swap some other least-used data out of the buffer pool, or write the data to disc right away. We can see from the graph that the throughput increases as we increase the buffer pool-size, and evens out once we reach a level where the buffer pool is larger than the pages to insert.

~~2.2 B - Table versus Row locking~~

No exam question on this

~~**(iii) Another interpretation (the one from the book) is to analyze the overhead of taking locks for insertions or updates [running inserts and updates separately with a single thread]**~~

~~The experiment was done by running the inserts and updates separately with a single thread. That meant two experiments for update and for insert, with both row and table locking. As posted on the blog i forced the table lock size to row or table before running the scripts, and used the '-l' property on the writes.py script for table-locking, to ensure correct locking granularity. I would expect the experiments to perform as in the book, where the update ratio should be lower because db2 uses logical logging, which would make the recovery overhead low and the locking overhead perceptible.~~



~~The response times from my experiments did though not look like the book, with a throughput ratio of updates also close to one. Actually in my experiments the insert ratio was lower than with the updates. Maybe the newer version of db2 has some differences to the one used in the book (v7) that makes a difference, or maybe the recovery overhead for my setup doesn't make the locking overhead as perceptible as it did in the book.~~

3 Part 2 - Reads

3.1 A - Measuring read performance

How fast can you scan a million tuple in your out-of-the-box system?

The fastest that i could scan 1000000 tuples was 6,7 seconds. With the generated 3,7mb data file this gives a throughput at around 5,5 mb/sec. Running the default disc benchmark tool in Ubuntu i found that the average read throughput is around 200mb/sec. As with the writes, the throughput is much less than plain io reads from the file system. This can again be explained by the extra functionality a database ensures for you with its data, making it slower than basic IO.

Modifying the table-space parameters

The following table-space-parameters were considered for experimentation:

Page Size This might have an effect, as the database fetches data in pages so it might affect read performance.

Extent Size This only has an effect when there is more than one container, and per default there is only one. I wont look into this.

Prefetch Size This affects how much is prefetched when doing IO reads, so this may very well affect read performance.

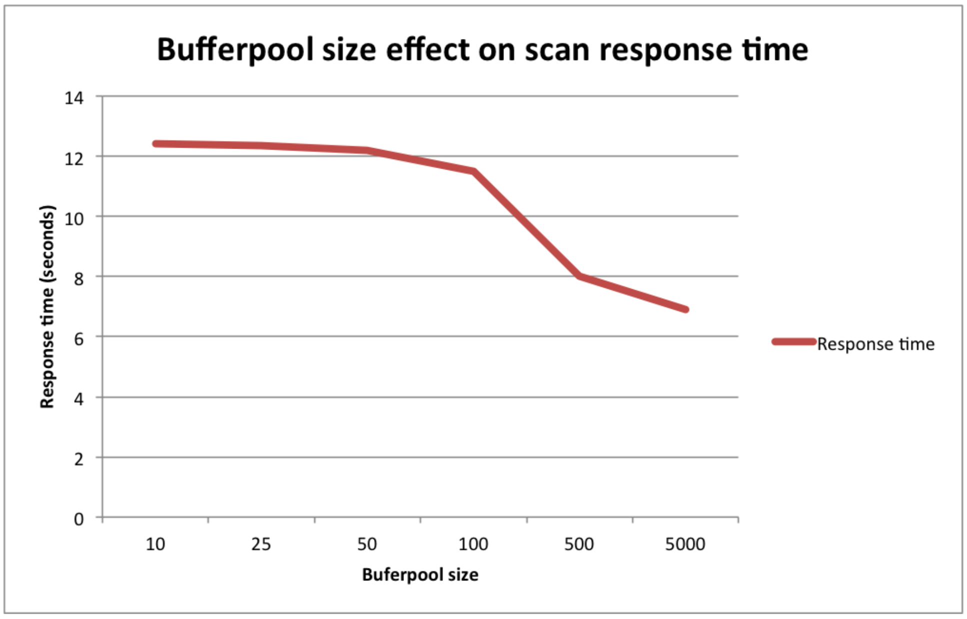
Overhead and transfer rate This is used by the optimizer to calculate costs to create the right execution plan. Everything that affects the execution plan may affect read performance, but this might not be much when this experiment only concerns a basic scan.

Buffer pool size Size of the buffer-pool used by the table-space. May impact performance as it determines how much can be buffered.

Changing the pagesize to 32k (as with the inserts experiment) had an effect on the response time. The scan responded in just under 6 seconds which is faster than with the normal pagesize. So making the page size larger meant that we had could fetch a smaller amount of pages, which seemed to make the reads faster.

Altering the prefetchsize of the tablesspace also had a big impact on performance. Setting the prefetchsize down to 4 decreased the performance drastically to 14 seconds. When the prefetch size was increased back up to 50 pages the response time speeded up to about 8 seconds again. The prefetchsize decides how many pages the database manager can prefetch when it decides to, which helps the database perform less expensive io's to do the same work. Forcing this size down to something small makes the database perform more io's, instead of a few possibly more sequential ones.

Changing the transfer rate and overhead on the tablespace didn't seem to do much with my scan response times. This could be because the overhead and scan affects the query optimizer, helping it to choose execution plans and being that i only have a very simple setup and are doing a simple scan, there might not be that many different execution plans to choose from, which will make it choose the same no matter how i set the overhead and transfer rate.

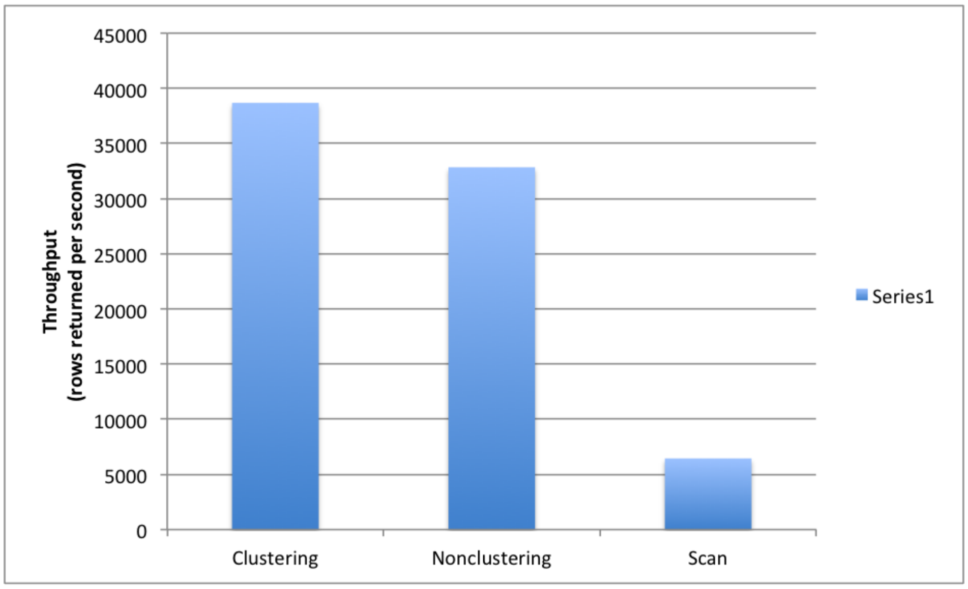


As can be seen from the graph, the size of the buffer pool also affects the read performance. The performance went from 12 seconds down to about 7 seconds, when increasing the bufferpool size. If i had tried even bigger buffer pool sizes, the graph would probably have evened out when the buffer could hold all the data scanned.

3.2 B - Clustering Index

The result of my experiments with clustering, non clustering and scans on a cold multipoint query is in the following diagram.

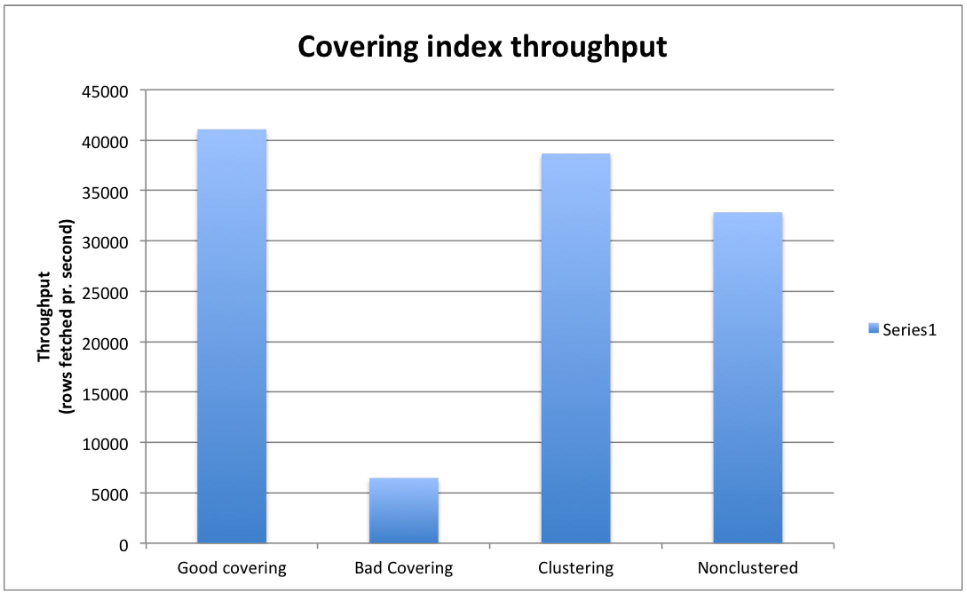
I would expect the clustered index to be the fastest, as it makes the data sorted by the hundreds2 key, which will make the data from the multipoint query be closer to each other physical on disc, which makes the io's quicker. The non-clustered will still be fast, as the index will tell which io's to fetch, even though these will be random. Lastly i think the scan will be slowest.



It seems that my hypothesis was correct. Fastest was the clustered index, and then the non clustered a bit slower, and at last the scan.

3.3 C - Covering index

In this experiment i used the multipoint query to experiment with the non clustered, clustered and both covering indexes. The hypothesis is that a covered index (which contains the keys used in the multipoint query, and in the right order) will perform about the same or maybe better than the clustered index.

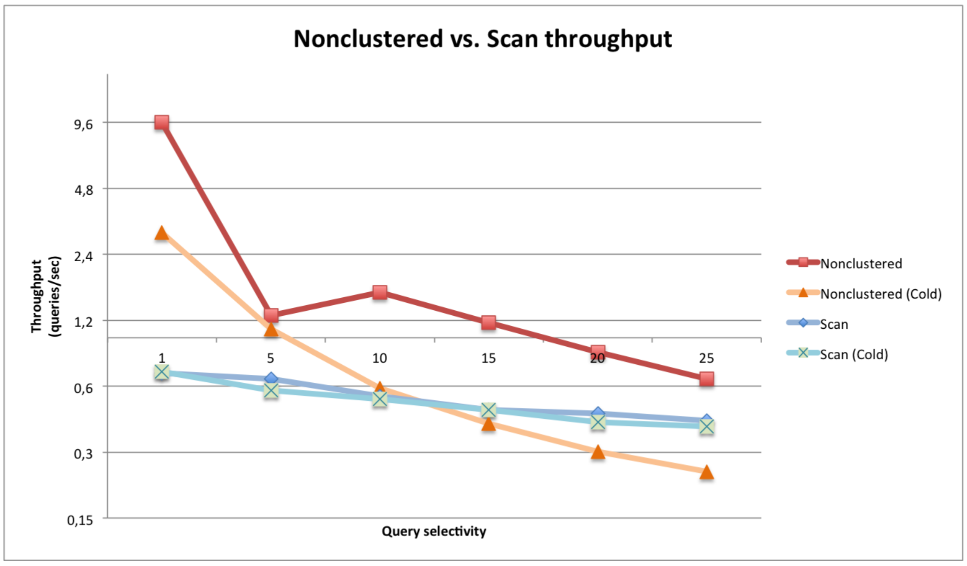


As it can be seen, the covered index was truly the fastest, with the clustered index right behind. It's interesting to see that the covered index with the wrong order of keys is so far behind the other three indexes, because the wrong ordering means that the index isn't being used efficiently enough.

3.4 D - Non clustering vs. Scan

The graph of the experiments of non clustered and scan's in both cold and warm cache setup is shown in diagram.

The hypothesis from the book is that the scan will outperform the non clustered index when the selectivity gets high enough. I am not sure how this will be for the warm/cold versions, other than I'm confident that the warm versions will be faster than the cold.



From the diagram it's clear that the non clustered warm experiment is the fastest one, with the cold version being lower in overall throughput but slowing down at the same rate. The cold and warm scans are though curiously close to each other. Even though the warm version are a tiny bit faster than the cold scan, it seems that the scan doesn't have as much to benefit from a warm database and file buffer than the index does.

It is worthwhile to notice that the non clustered indexes, have a particular high throughput with the selectivity of 1. This is also where the range-query effectively becomes a multipoint query, and thus also don't have the penalty of having to have many io's. It is particularly fast in the warm version, as it must have the right pages loaded in memory at that point.

It seems that the scans reach the performance of the cold non clustered index at around selectivity of 10, This means that for a selectivity higher than 10% a scan is actually faster than a cold non-clustered query on my setup. For the warm non clustered index, it seems that the graphs would reach each other at around 35% (out of my experiment graph), but is it fair to assume that a query on a production system will always be warm?. This underlines the point that it is not necessarily faster to use an index, if the query selectivity is high. Here a scan will actually probably be faster.

4 Part 3 - log of problems

4.1 Problems measuring throughput from scripts

I had a problem defining the measurement of throughput from the writes.py script when all i had was the response-time. I also had the file with the data to be written, but the data was in text-format. I ended up just using these as the basis for the throughput even though, because is was the best i had, and the benchmarked throughput was anyways much faster than the script throughput.

4.2 Problems with indexes not seeming to work

I had a lot of problems getting the nonclustered and clustered indexes to work on my setup. After doing all the experiments i found that my approach of force stopping db2 (db2stop force) seemed to be the culprit. When force stopping db2, a clustered index would stop to perform. Disconnecting the connection to the database and stopping the database without forcing it didn't do this, and the indexes then still worked.

Have i had the time i would have investigated this further to see why the indexes didn't perform in this case. I used the time to run most of the experiments again and get more sane results.

4.3 Ensuring a cold cache - How i cleared the buffers

In several questions a requirement was to test with a cold cache. To do this i ran my experiments with only one run at a time, and did the following to ensure that the buffers were cleared.

First i stopped and restarted db2 so that the database buffers would be empty. Second i tried using the suggestion in the assignment to unmount and remount the harddrive to clear the filesystem cache (using umount and mount). This proved hard because i couldnt find a way to remount the right devices when unmounting, resolving in being unable to open new terminals. I then found another script (which is also referred in the assignment 2 blog from last years Database Tuning course), which would clear the filesystem cache on ubuntu (*"sync"* to write/read all data out of the buffer, and *"echo 3 > /prop/sys/vm/drop_caches"* to clear the file system cache).

These two steps were performed before each run in the assignments that requires a cold cache.