Advanced Data Management
Mandatory Assignments 1-3

Anders Bjerg Pedersen, 070183

April 12, 2010

Contents

1. Assignment 1

1.1. Introduction

In this assignment we are to perform experiments on a DB2v9.7 server running on Amazon Web Services to see the effects of using two different isolation levels (CS and RR), as well as examining the impact of the new "currently committed semantics" implemented in version 9.7 of DB2. We conduct various experiments to test the performance and correctness of the various isolation levels and semantics and check with the DB2 documentation to test our hypothesis.

1.2. Part 1: Threads and isolation levels

The experiment consists of testing whether or not there is a time difference between the CS and RR isolation levels. See hypothesis and experiment setup below on how we do this.

1.2.1. Isolation levels in DB2

We shall first have a quick look at the two relevant isolations levels of DB2: repeatable read (RR) and cursor stabillity (CS).

Repeatable Read (RR)

When using this isolation level, the DBMS puts locks on all rows affected by a given transaction and holds them until it has been committed. Locks are not only held on the returned rows, but on *all* rows affected (read, updated, deleted or inserted). Phantom reads are impossible under this isolation level and the RR isolation level is considered serializable.

Cursor Stabillity (CS)

When using CS as isolation level, locks are held on those rows that the "cursor" (i.e. the row currently being read/updated/inserted) points to. All other locks are released, as is the one on the cursor when it moves on in the table. When changing a row, though, the lock is held until commit time. Phantom reads *are* possible using this isolation level.

1.2.2. Hypothesis

Our hypothesis builds mostly on the documentation of DB2 and the way it describes the effects of the two isolation levels. One would expect the RR isolation level to be more time-consuming than CS, due to the fact that it attains a substantially larger amount of locks on our table. However, it is not clear to what extent this increased amount of locks actually reflects on the mean time of each run. With regards to the number of threads, we should be able to show that the mean time of the runs decreases, as we add more swap threads to the pool[1]. Another

[1] This is mainly because the Python script times the entire process and not just the sum query.

point here is that Amazon runs their instances on an Intel Xeon E5430 quad core processor[2]. This means that when going up to 4 threads, we should see drastic reductions in execution time. When using more than 4 threads we should begin to see a slow increase in execution time again because of added thread overhead and more threads interfering with the sum query.

Also, we would expect the RR isolation level to deliver perfect results, as it prevents phantom reads which are the main problem in the queries running in our setup (RR is actually serialisable). CS, on the other hand, we expect not to deliver correct sums in every run. In the CS case we also expect to see substantial fluctuations with the number of swap threads - the more threads that can interfere with the sum query in shorter time, the more incorrect sums.

1.2.3. Setup and experiments

All experiments have been conducted on the supplied AWS instance (ami-4b4ea222) on Amazon Web Services, running DB2 version 9.7.1. We have used standard parameters for the AMI (no EBS, standard security group, small size) and have otherwise followed the experiment guide on the course homepage. All experiments have been run 50 times for the given number of threads (see Appendix A for those statistics) and 5 times when trying all number of threads from 1 to 50.[3] All runs have used 100 swaps. Note also that we have decreased the value of `time.sleep(1)` to `time.sleep(0.1)` (100 milliseconds) in order to be able to keep runtime within reasonable limits. One could argue whether or not this command is relevant at all, and indeed removing it yields interesting results. More on this later. Also, with a setting of 1 second only the first batch of threads will actually influence the sum query, which – when left uninterrupted – finishes within 0.2 seconds.

First we have a look at the two isolation levels CS and RR. We plot the mean time per run as a function of the number of threads used. Benchmarks are shown in Figure 1.1. For the 50-runs, 1-5-10-25-50-thread experiment, please refer to Appendix A.

Next we have a look at the sums generated by the query in each run. We count the number of sums in 50 runs equal to the result of a sum query run before executing the scripts and calculate the ratio. We do this for 1, 5, 10, 25 and 50 threads. Benchmarks are shown in Figure 1.2.

1.2.4. Observations and discussion

We begin by discussing the benchmarks on the mean time of the runs. We initially get a substantial reduction when increasing the number of threads from only 1 to about 10. When going above 10 threads, the mean increases again. This is coherent with our hypothesis, as it is predictable that when we reach a certain number of threads, more of them will interfere with the sum query and thread overhead will begin to have an effect. Additional experiments on a local setup using DB2 version 9.5.2 have shown similar results. Also, we see that there is

[2] On Amazon use: `less /proc/cpuinfo`.
[3] As this would have taken forever, and 5 runs still give a fairly good statistical approximation.

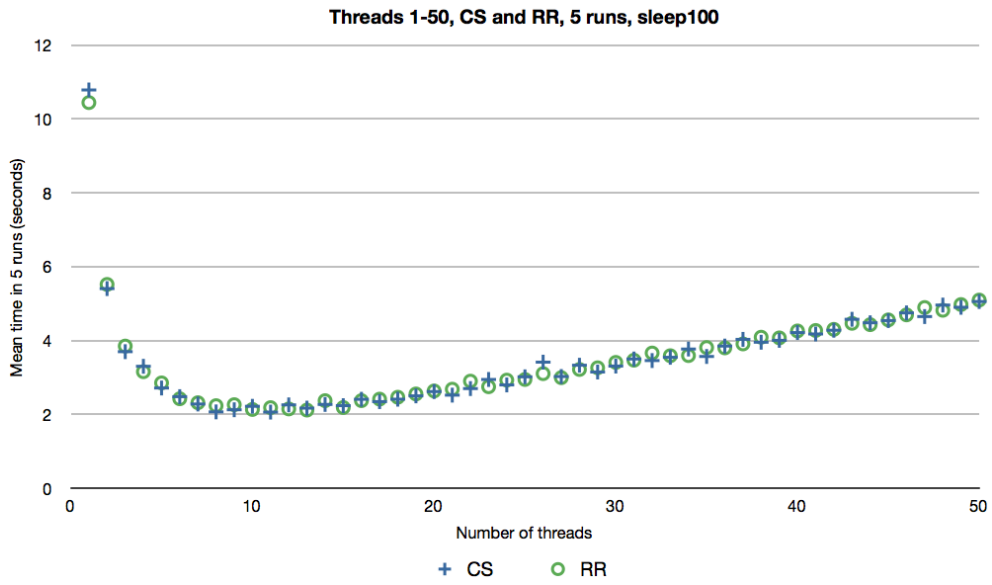

Figure 1.1.: AWS times for isolation levels CS and RR with sleep of 100 ms

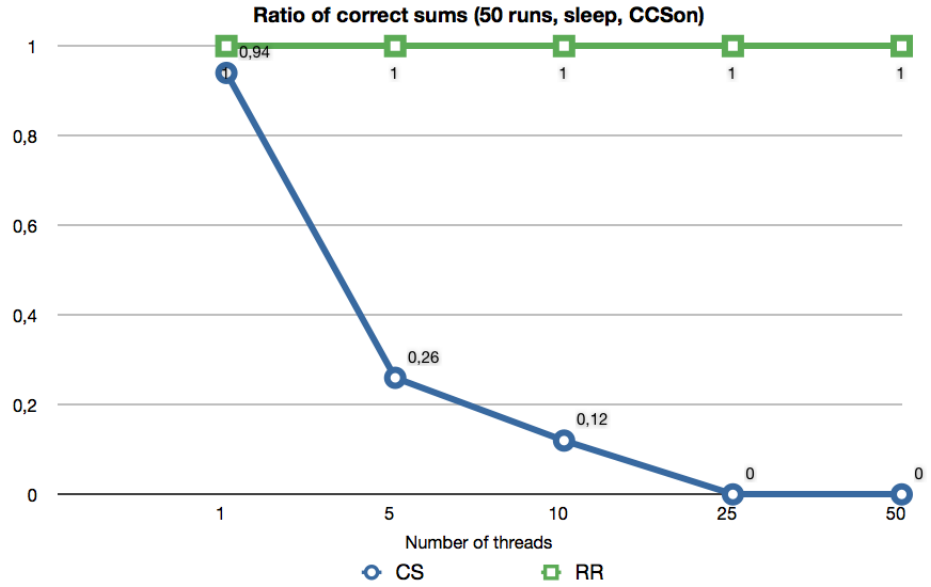

Figure 1.2.: AWS sum ratios with sleep of 100 ms

actually a negligible difference between CS and RR, contrary to our stated hypothesis.[4] Also, we see that mean execution time is approximately cut in half when doubling the number of threads in the beginning. When reaching 8-10 threads, however, the time increases again, albeit slowly. Again, this is probably due to the quad core processor executing the threads. Once

[4]Interestingly, when completely eliminating the `time.sleep`, mean execution time is linearly growing from 1 and upwards. See Figure A.3 in the appendix.

we reach more than two threads per core, thread overhead and exchange becomes a factor. So, mean execution time must also depend on something other than isolation levels or the number of threads. The number of threads can very well be adjusted to minimise execution time on a daily basis in a company; more threads do not necessarily imply gains in overall performance.

With regards to the number of correct sums, the results are pretty much as predicted: RR delivers a perfect 1:1 ratio of correct sums, whereas CS fares much worse. However, the ratio is unexpectedly low and compared to the negligible difference in execution time, in this case there would be no reason whatsoever to choose less than RR for one's isolation level. We also notice that the ratio of correctness for CS drops, as the number of threads increases. The more threads, the more interference with the sum query.

1.3. Part 2: Currently committed semantics

The "currently committed semantics" (CCS), as introduced in DB2v9.7, defines a new standard for how the CS isolation level is handled. The parameter `cur_commit` handles this behaviour and is enabled by default. Using currently committed semantics ensures that the DBM can circumvent waiting for most deadlocks and lock timeouts by using the currently committed value of an affected row instead of waiting for another process to finish modifying it. In other words, the DBM can choose not to wait and use the previously committed value instead. In our experiment this means that when the sum query tries to access a row that is currently being updated by a swap thread, the DBM reads the committed (old) value from the log instead. In this section we shall investigate whether or not there is an impact on performance and correctness when disabling CCS.

1.3.1. Hypothesis

Disabling CCS will force the DBM to wait for locks instead of using the already committed version of the data. We therefore expect the mean time per run to increase as compared to the default CS isolation with CCS enabled.

Regarding the ratio of correct sums, we would initially expect it to decrease, as the sum query is now forced to wait for the update threads, instead of being able to read from the log directly. However, the sum query will finish at some point, probably before the remaining threads have executed, hereby enforcing a time penalty as the number of threads exceed some threshold.

1.3.2. Setup and experiments

We turn off CCS on the AWS instance with the following command:

```
> db2 update db cfg using cur_commit disabled
```

We then reconnect to the database and perform the same experiments as in Part 1, with a 10 ms sleep factor set in the `writes.py` script.[5] As CCS only affects the CS isolation level, we do not repeat the experiments for the RR isolation level (except for the ratio of correct sums for reference). Results can be seen in Figure 1.3 and Figure 1.4. The 50-run experiment can be seen in Figure A.4 in the appendix.

[5]Note that this is changes from the default 1000 ms. There is no apparent need for such a long sleep delay.

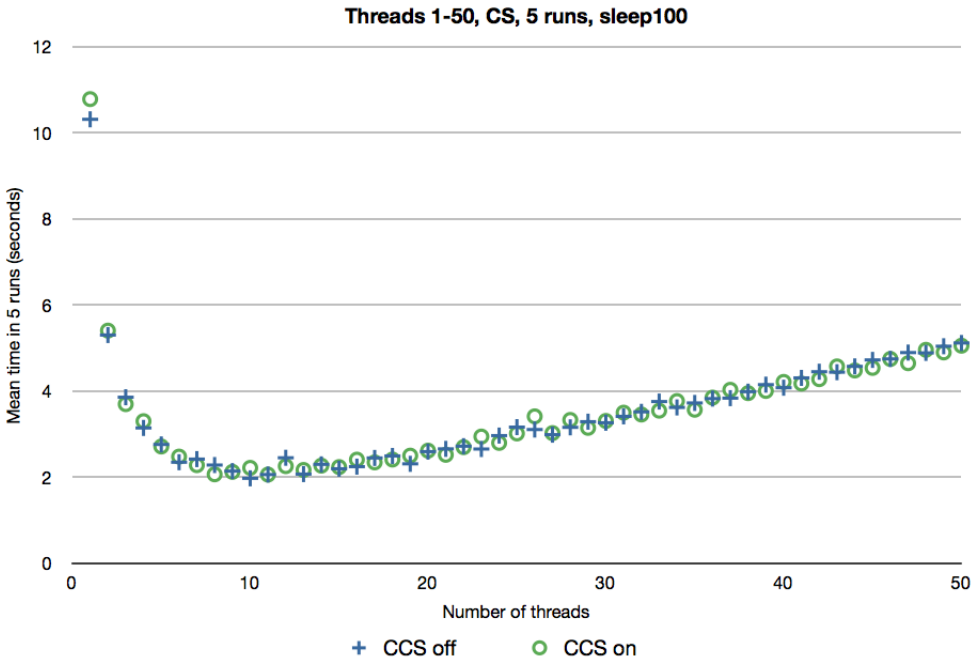

Figure 1.3.: AWS times for isolation level CS with CCS enabled/disabled.

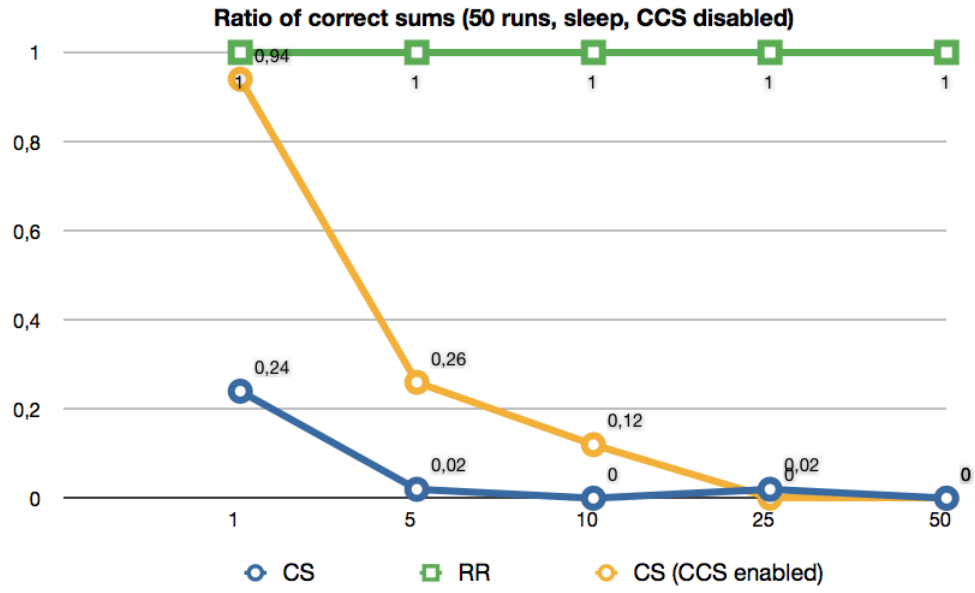

Figure 1.4.: AWS sum ratios for isolation level CS with CCS disabled.

1.3.3. Observations and discussion

With respect to the ratio of correct sums for the CS isolation level, we see that in this case it fares much worse than with CCS enabled. With the exception of using only one thread, the

ratio drops to around 0-2% correct sums with CCS disabled. Here the swap threads have a much greater chance of interfering with the sum query, as this is now forced to wait for the swap threads.

We do not, however, see an increase in execution time when turning off CCS (see Figure 1.3). This probably means that in our case the waiting time for accessing the old (committed) data from the log is roughly the same as the time spent for the sum query waiting for the swap threads to release their locks.

1.4. Conclusions

In this assignment we have investigated the effects of different isolation levels in DB2v9.7. We have looked at performance measurements as well as measurements of correctness in a long sum query, running whilst being interfered by a varying number of threads executing swaps in the same table.

The results have shown a negligible difference in execution time for the two isolation levels CS and RR, even when disabling currently committed semantics (CCS). One should, however, be very careful when choosing the number of threads in the pool available to the DBMS, as an increasing number of threads does not necessarily guarantee better performance. There is a huge difference in the ratio of correct sums: RR gets a perfect score in every scenario, CS gets a bad score with CCS enabled, and a close-to-zero score with CCS disabled.

We therefore conclude that there is no impact on performance when using these different isolation levels, however there is a great impact on the correctness of the transactions, especially when disabling CCS.

2. Assignment 2

2.1. Introduction

The purpose of this assignment is to investigate performance fluctuations when varying certain parameters and settings regarding logging, locking and indexing in the DB2 database system. We conduct various experiments to measure the impact of different parameters on execution time and try to identify those parameters that have a real impact on performance.

All experiments have been conducted on a local instance of DB2 version 9.5.2 for Mac, running on a 2.4 GHz Intel Core 2 Duo machine with 4GB of RAM and a 320GB S-ATA 5400rpm disk. Many of the results have been verified on an AWS instance as well.

For reference purposes we benchmark the host system to see what amount of throughput it is theoretically capable of. Our system has the following theoretical performance:[1]

Main test	**Subtest**	**Result**
Memory	Fill	8730.84 MB/sec
Memory	Copy	4185.86 MB/sec
Disk	Sequential Uncached Write [4K blocks]	47.52 MB/sec
Disk	Sequential Uncached Read [4K blocks]	23.17 MB/sec
Disk	Random Uncached Write [4K blocks]	1.06 MB/sec
Disk	Random Uncached Read [4K blocks]	0.45 MB/sec

So, as our 100.000 tuples take up approximately 1.5MB of disk space, we should be able to insert or update these in no time at all, depending on where on the disk we write, and if the writes are sequential or random. Let's see if we can actually achieve this performance.

2.2. Part 1: Writes

Overall the goal of this assignment is to tune the DB2 database to deliver the best possible performance. Therefore the first thing we do is to adjust the usual parameters given to `writes.py`. We are also asked to state what the initial performance of a "fresh" out-of-the-box system is. We test insertion of 100.000 tuples into the `accounts` table, varying the number of threads (1-50), the number of transactions (1 or N), and the isolation level. Some results can be seen in Figure 2.1. From Figure 2.1 we see that using 3 threads yields optimal performance when running under isolation level CS. Other isolation levels show almost completely similar results. We therefore choose to use 3 threads and isolation level CS for the remainder of these experiments (unless otherwise explicitly stated). Our benchmark execution time is somewhere

[1] Measured using Xbench 1.3.

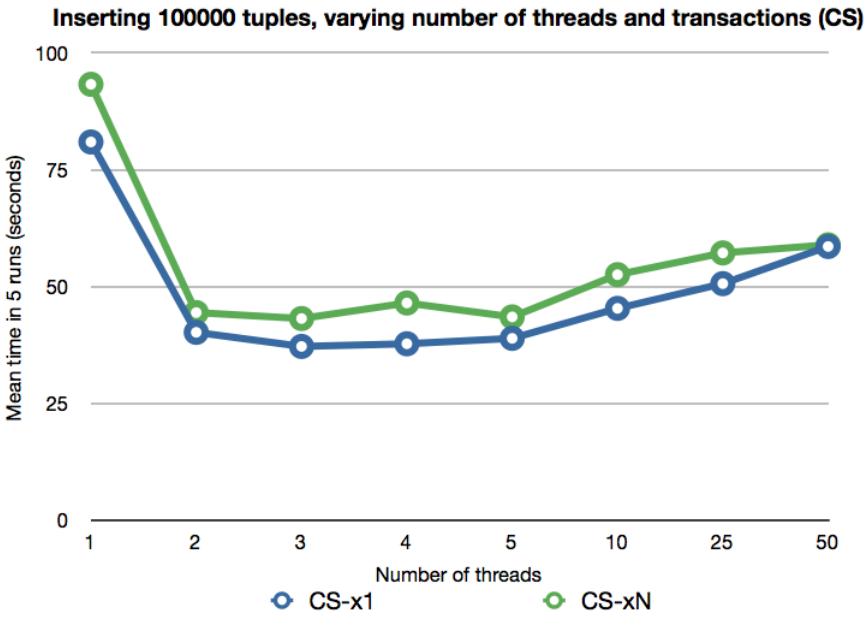

Figure 2.1.: Varying number of threads and transactions, CS.

around 40 seconds per run.

In this first part of the assignment we look at the impact of certain parameters on the write performance of the DB2 system. Specifically we look at tuning log operations to speed up inserts and tuning locking mechanisms to speed up updates.

2.2.1. (A): Logging

The log plays a very important role in tuning the performance of insert operations. Whenever an update (insert/update) is committed to the database, a corresponding record of the event is stored in the log – making it possible to roll back transactions and recover from crashes. For this assignment we cannot change the disk on which the log is stored, but fortunately DB2 offers a wealth of parameters that have an influence on logging.

Tuning parameters

We now briefly describe the parameters that are to be tuned to improve insertion time on our DBMS. Descriptions are mainly from the DB2 documentation.

Parameter	Name	Description	value [range]
LOGBUFSZ	Log buffer size	Adjusts the size of the log buffer. A large buffer can accommodate more changes to the database before a write to the actual log file is necessary.	8 $[4-4096]$
LOGFILSIZ	Log file size	This parameter adjust the size of the log files. When using small log files the system must more often switch to new log files, creating an overhead when issuing large transactions that only have few commits. Such transactions will quickly fill up a small log file.	1000 $[4-1048572]$
MINCOMMIT	Number of commits to group	This parameter allows you to delay the writing of log records to disk until a minimum number of commits have been performed, helping reduce the database manager overhead associated with writing log records.	1 $[1-25]$
SOFTMAX	Soft checkpoint interval	The SOFTMAX parameter controls the interval with which the DBMS writes dirty pages to disk. This is done by specifying how many percent of a primary log file is needed to recover from a crash at any point in time.	100 $[1-100 * logprimary]$

We set each of these parameters by the command

```
> db2 update db cfg using <parameter> <value>
```

For each parameter variation we do 5 runs with 100.000 inserts using 3 threads and the CS isolation level. As some of the parameters do not take effect until the next connection to the database, we make sure to force a connect reset followed by a new database connection when updating these parameters. Results of the experiments can be seen in Figures 2.2, 2.3, 2.4, and 2.5. Also, results have been verified on the AWS instance.

Observations and discussion

One immediate observation is that none of the tested parameters seem to have a substantial effect on the runtime of the insertion query. This is either due to experiment flaws or other reasons. We will assume no experiment flaws, as the experiments have been run on three different DB2 instances, using two different versions of DB2 and three different operating systems and hardware. Also, the experiments have been run with exactly the same parameters and data. When running the experiments, `db2top` shows no bottlenecks and indicates that possibly only one commit is done in each run – whatever the settings.

We therefore look to other reasons for the parameters not having any effect. We would have expected the LOGBUFSZ and LOGFILSIZ parameters to have given some effect on the results,

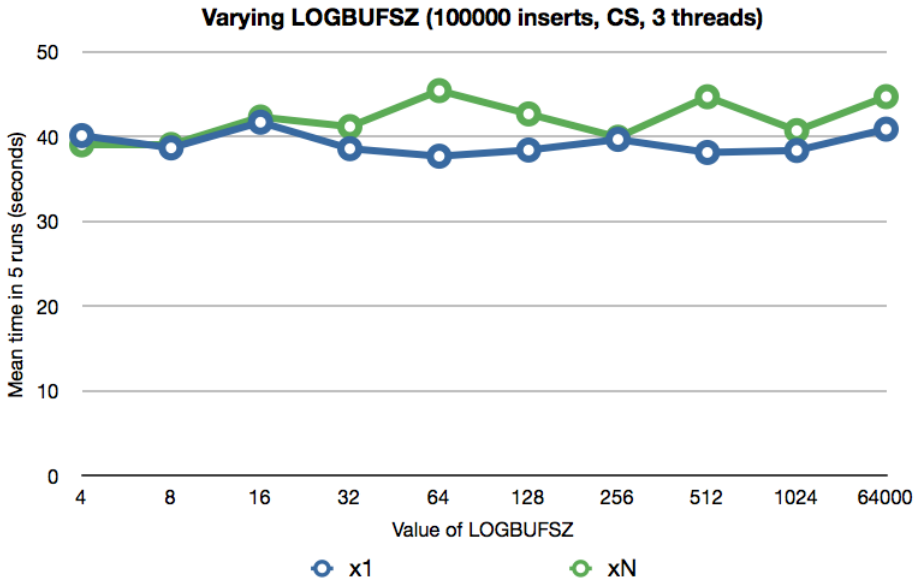

Figure 2.2.: Varying the LOGBUFSZ parameter.

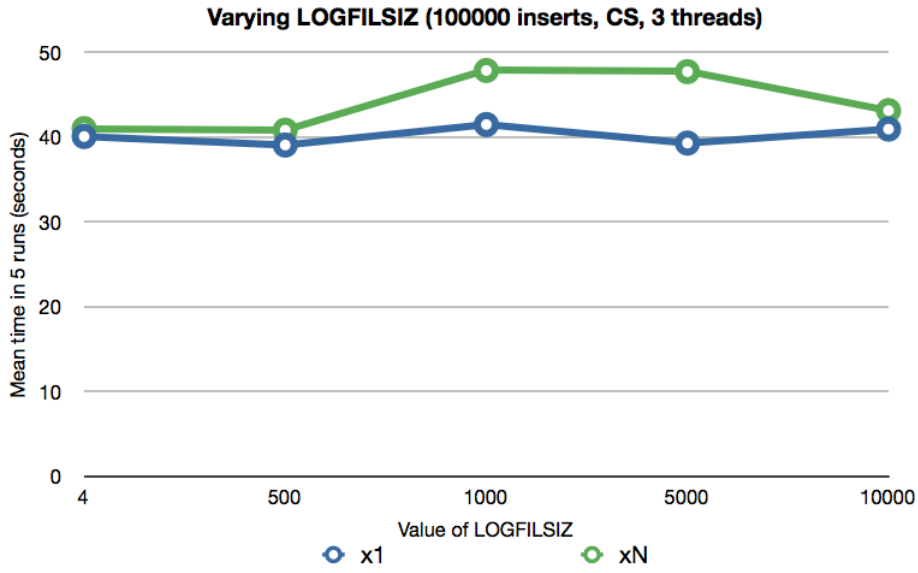

Figure 2.3.: Varying the LOGFILSIZ parameter.

but it seems that the default setup of DB2 handles the insertions well; no matter the size and number of log files. Also, one should take into consideration the fact that the data amount of our query is only 1.5MB. This isn't much and we would probably see more interesting results if using a table that contained more attributes and data per tuple.

It also seems as though the script only does one commit of data, although specifying the `-xN` parameter. This could be caused by an error in the `writes.py` script (although we have

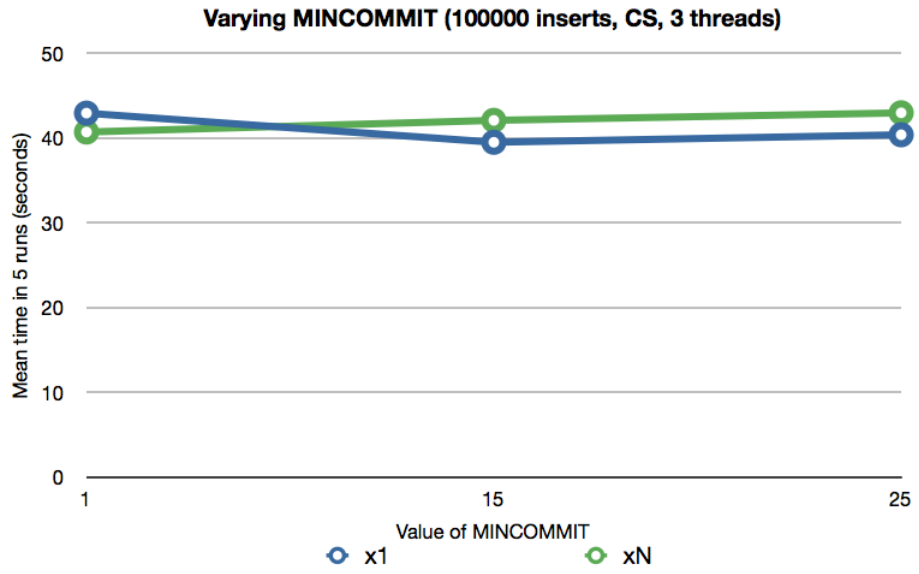

Figure 2.4.: Varying the MINCOMMIT parameter.

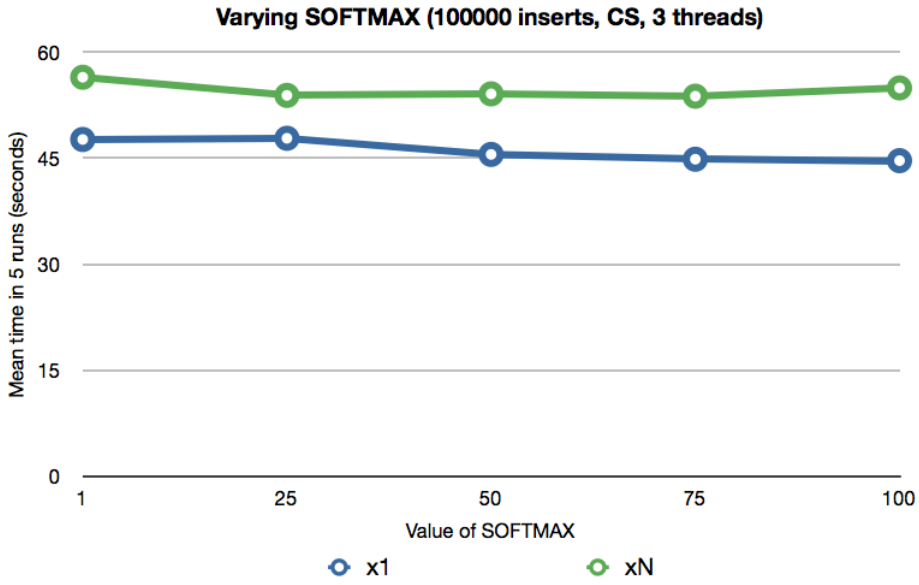

Figure 2.5.: Varying the SOFTMAX parameter.

not investigated this further).

2.2.2. (B): Locking

In this section we have a look at how different locking mechanisms affect many updates on a table. Specifically we look at *row level locking* versus *table level locking*. Row level locking means that when performing an update only the rows affected by the update are locked from

other transactions modifying it. In a table level locking environment, whenever performing an update, the entire table is locked; effectively preventing other transactions from modifying the table.

In our case the update transaction uses only one commit and thereafter the table lock is released. Also, we use only one thread and one repetition, having to reenable the table locks in between each run. We issue the following command before executing `writes.py` to enforce table level locking:[2]

```
> db2 "alter table accounts locksize table"
```

Our goal is to recreate Figure 2.8 from [1]. In order to do this we perform inserts and updates of 100.000 tuples in the `accounts` table using `writes.py` with both row level locking and table level locking. We repeat the experiments 5 times and quote mean execution time ratio of row versus table level locking.

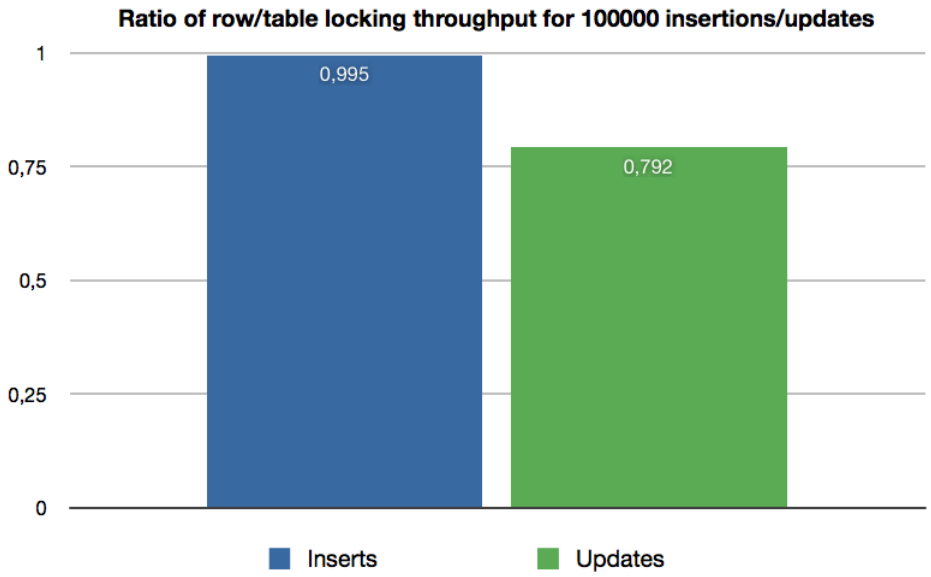

Figure 2.6.: Throughput (transactions per second) ratio of row versus table locking.

Observations and discussion

Results can be seen in Figure 2.6. We observe that the ratio is very close to 1 when performing inserts. These are therefore unaffected by a potential switch to table locking by the DBMS. When performing updates, however, the ratio drops as the times for performing updates with table locks take substantially longer. This has to do with the fact that DB2 uses logical logging (i.e. logging the logic that caused the update instead of logging before or after values). This has an impact on the logging overhead when doing updates as opposed to inserts: the logical logging overhead now surpasses the locking overhead, which corresponds perfectly with (although not as marked as) the results in [1].

[2] Note that we do *not* use `db2 +c "lock table accounts in exclusive mode"` as this seems to halt execution by creating one or more deadlocks (which can be seen as a bottleneck in `db2top`). This may also be an interesting thing to investigate, however not the purpose of this experiment.

2.3. Part 2: Reads

In this section we have a look at how different parameters and index types have an influence on the speed with which we can read data from the database. First we do a simple benchmark as to how fast we can scan one million tuples from the `employee` table. We then try to tune three different parameters to reduce the time needed.

In parts (A), (B), and (C) we look at how different types of indexes and queries influence read speeds.

2.3.1. A note on obtaining cold buffers

To enforce a cold buffer in sections (B), (C), and (D) we perform the following operations (truncated for explanatory purposes):

```
> db2 connect reset
> db2stop
> Copy a 1.2GB file to clear the file system cache
> db2start
> db2 connect to tuning
> db2 flush package cache dynamic (flushes the query cache)
> db2pdcfg -db tuning -flushbp (flushes the buffer pages)
> python writes.py ...
```

2.3.2. (A): Tuning table scans

We will now take look at how fast we can get DB2 to do full scans of the `employee` table. We begin by a benchmark run with standard configuration and then try to modify the schema to obtain better speeds. Again, our system benchmark indicates that these reads should be executed in no time at all (see the introduction). The initial benchmark table scan can be seen in Figure 2.7. We note that scanning the 1.000.000 tuples in `employees` takes approximately 27 seconds. This, of course, compares miserably to what the host system itself is capable of. Let's try to improve...

Tuning parameters

When tuning reads, some of the most important changes to make is varying keys on the schema and adding indexes. The last option we shall investigate even further in parts (B), (C), and (D). Our query is very short:[3]

```
1    select max(ssnum) from employees;
```

There are not as many parameters for adjusting read speeds in DB2 as there are for adjusting write speeds. The fact that we are using a warm buffer for this particular section suggests maybe tuning the size of the database buffer, but our main focus shall lie on keys and indexes, as these are the top-notch parameters for achieving better read speeds. Not having done any schema changes in Part 1, in this part we focus our attention on trying the following variations:

[3] We have added the aggregate to enforce a full scan every time.

Adding a primary key on `ssnum`

Description	Expected effect
The schema is declared without any primary or foreign keys. If we make the (relatively obvious) assumption that social security numbers are unique in the table, we can modify the schema to make the attribute `ssnum` the primary key.	As such, the DBMS will usually try to guess on some attribute being key and create an index on it. In our case, when we force `ssnum` to be the primary key, we also hint very much to the DBM which attribute to index. We do not, however, have any control whatsoever over which *type* of index the DBMS creates. For this particular query, though, we should see some improvement by hinting structure to the DBMS. Also, having generated the data with `gentable.py`'s last parameter set to 1, the loaded data is prepped for having a primary key on `ssnum`.

Adding clustering index on `ssnum`

Description	Expected effect
Another approach, which doesn't make any assumptions about the distribution and uniqueness of the `ssnum` attribute is adding an index on it. The first one to try is a clustering index, which pairs together records with matching or same values of `ssnum`. We use a slightly modified version of the one handed out, using `ssnum` instead of `hundreds2`.	For this particular query containing an aggregate on the attribute indexed, the DBMS should not need to access the data pages at all. All the information needed is contained in the index and even grouped (clustered) together. Therefore we should see this variation outperform the two first variations, especially if the index is completely contained in memory.

Adding nonclustering index on `ssnum`

Description	Expected effect
We add a nonclustering index on `ssnum` to speed up the reads, again using a slightly modified version of the one handed out with the assignment.	Adding a nonclustering index on `ssnum` should show similar results to those obtained by adding the clustering index as described above. However, there may be a slightly bigger overhead from index searching, as the desired values are not necessarily grouped together, but there should still be no need for disk access.

Adding covering index on `ssnum`

Description	Expected effect
We add a covering index on `ssnum` and `name` to speed up the reads.	The speedup using a covering (or composite) index is harder to predict. We would expect it to show results similar to those using the nonclustering index, perhaps even slightly worse.

We conduct each experiment with 200 queries and measure mean time in 10 runs on a warm buffer. The throughput ratio of the best performing experiment is set to 1.0 and the others are calculated accordingly. Results can be seen in Figure 2.7.

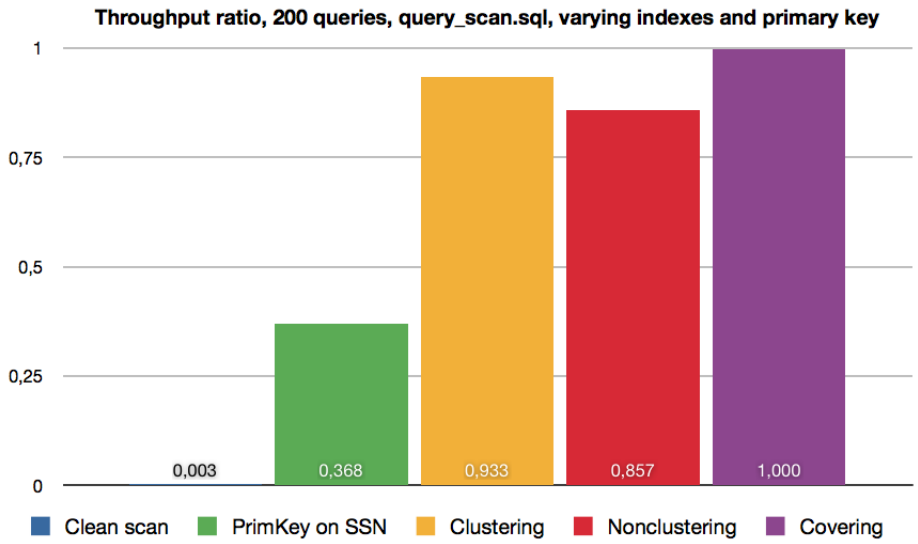

Figure 2.7.: Throughput ratio of of scan, primary key, and different index types.

Observations and discussion

Results can be seen in Figure 2.7. We observe that most of the results are as expected: we see a miserable performance when using the standard scan without keys or indexes. When adding a primary key on `ssnum`, we see a drastic increase in performance, and even more so when adding an index on `ssnum` instead. The different indexes fare almost equally well, with a slight advantage for the covering index (which is slightly odd when keeping in mind that it is actually a nonclustering index with an extra attribute). This may root in DB2 handling these two types of indexes in different ways – something which we cannot investigate here.

Overall, though, we see that adding an index on the `ssnum` attribute helps improve read speeds by up to a factor of 300.

2.3.3. (B): Clustering indexes

In this section we look at the effect of clustering indexes, with specific attention put on multipoint queries. Again we use the `employee` table with 1.000.000 tuples, use 200 instances of the `query_multipoint.sql` query, and measure mean execution time m_{mean} in 10 runs. The throughput $t_{clustered}$ of the clustered scan is set to have ratio 1.0, and the other throughput ratios are calculated as $r_{other} = \frac{200}{m_{mean} \cdot t_{clustered}}$. Our goal is to recreate Figure 3.6 from [1].

Hypothesis

Using a clustering index on the `hundreds2` attribute should show a substantial benefit compared to a non-clustered one, which again would be beneficial over not having an index at all.

Using the clustering index causes records with matching or nearby values of the `hundreds2` attribute to be easily accessible when doing table scans. The non-clustering index should fare somewhat worse, although still better (faster at lookups) than having none at all.

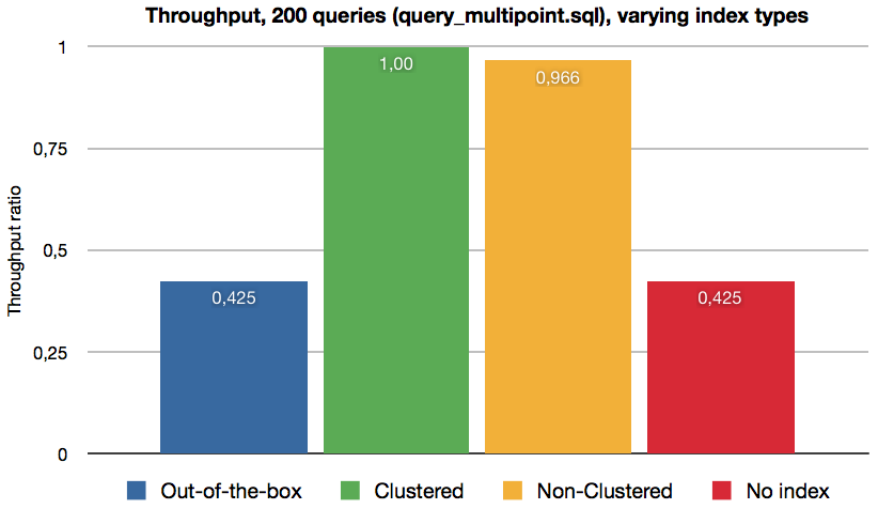

Figure 2.8.: Throughput ratio with focus on clustering indexes.

Observations and discussion

Results can be seen in Figure 2.8. The "Out-of-the-box" benchmark is the same as the "No index" result. We see that the clustering index fares the best, compared to having no index at all. The nonclustering index, however, performs close to that of the clustering index, indicating that either something has gone wrong (in the positive sense), or a lot has happened in DB2 since version 7.1 that the book used, compared to version 9.5.2 that we have used. Also, our multipoint query takes out 10.000 records (1%) out of the 1.000.000 in the table instead of the 100 records (0.01%) in the book - this may also have an impact on the performance of the indexes. Other than this our results are pretty much in correspondence with our hypothesis.

2.3.4. (C): Covering indexes

In this section we focus our attention on various covering indexes, again using the `query_multipoint.sql` query. Our goal is to recreate Figure 3.11 from [1]. We again use `reads.py` with 200 queries and measure mean time in 10 runs using a cold cache.

Hypothesis

Following the results from [1] we expect to see the good covering index fare best with our multipoint query, followed by the clustering index, the nonclustering index and finally the table scan (no index).[4] The reason for the composite (good covering) index to fare good

[4] As we have not been given a prefix match query for this assignment, we have used the multipoint query instead. This may have an impact on performance, as a prefix match query possibly will be able to utilise a covering index better than a multipoint query.

is the fact that it packs records with the same values of `hundreds2` together, whereas the nonclustering index, for instance, packs them more or less randomly.

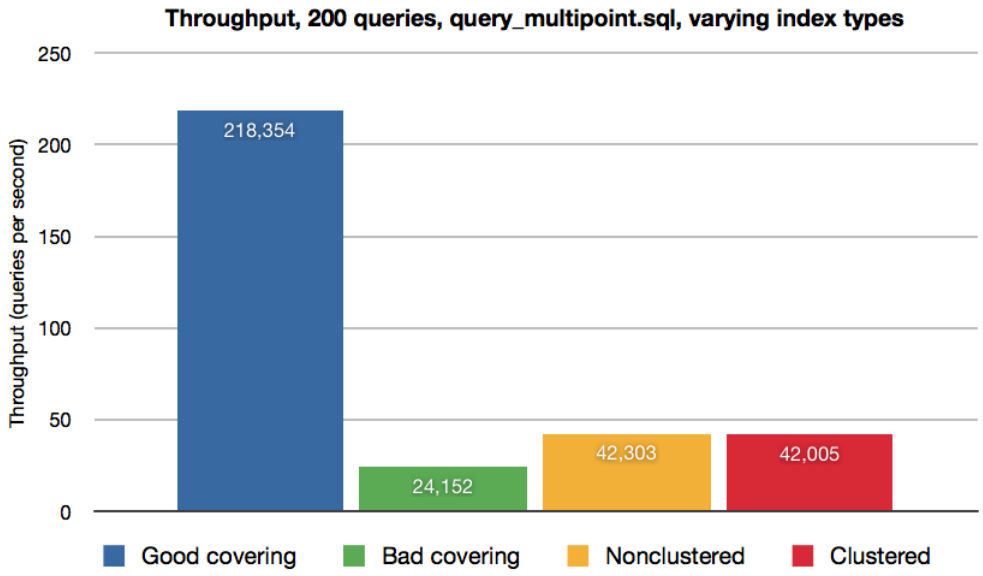

Figure 2.9.: Throughput with focus on covering indexes.

Observations and discussion

Results can be seen in Figure 2.9. We observe that overall we get similar results to the figure in [1], although the difference between the nonclustering and clustering indexes is hardly noticeable. The clustering index should do far better and be on par with the covering index. This may have to do with not using a prefix query, otherwise we have no real explanation for this. Our hypothesis is pretty much correct.

2.3.5. (D): Selectivity

In this section we look at how selectivity (i.e. the percentage of the table returned by a query) affects throughput for a nonclustering index as opposed to a regular table scan. In the experiment we use `query_range.sql` to vary the percentage of data returned by the query. We do this by changing the value of the constant `p` in `hundreds2 <= p` in the query. A value of $p = 5$ returns 5% of the data in the 1.000.000 records in the table. We conduct this experiment with both a cold and a warm buffer. Again, our goal is to recreate a figure from the book - in this case Figure 3.12.

Hypothesis

As stated in the book: when records of the table are stored consecutively on the disk, at some point a full table scan will fare better than a nonclustering index. In the book this turning point is reached at approximately 15% selectivity. We expect to see similar results in our own experiments, except when using a warm cache. In this case disk seeks should not be necessary

at all, since the entire table should be able to fit in memory. Here the overhead of the index will play a more important role, but not one as significant as using a cold buffer, where we force disk reads, if the required data is not in memory.

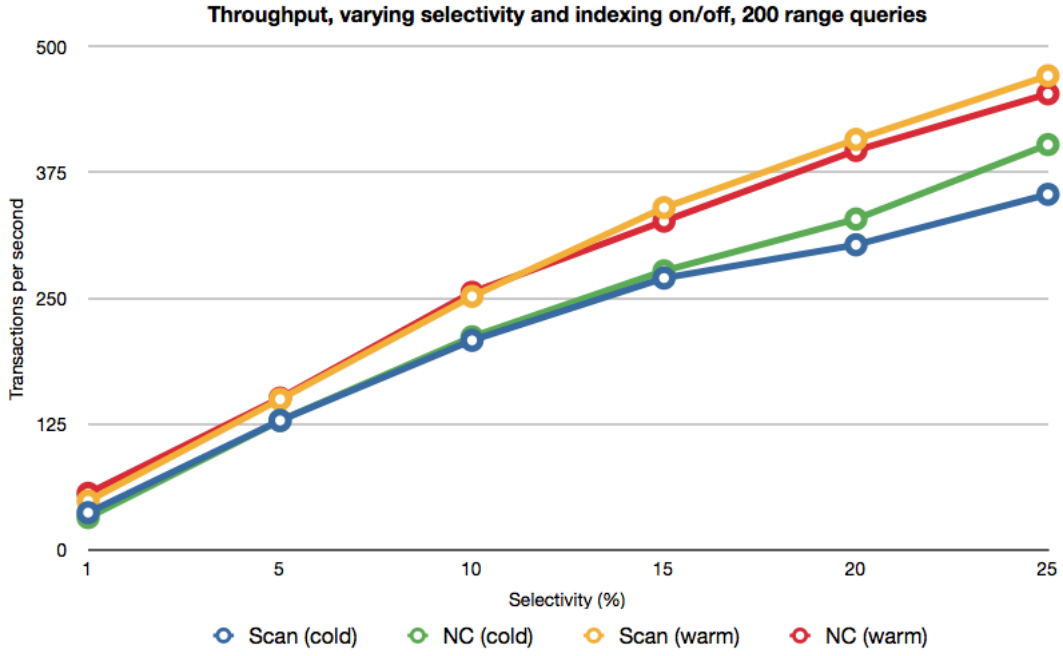

Figure 2.10.: Throughput when varying selectivity and index on/off.

Observations and discussion

Results can be seen in Figure 2.10. We see results somewhat vaguely similar to the figure in [1]: there is a turning point of approximately 10% selectivity, after which the scan fares better than the index. Also, not surprisingly the warm buffer fares better than the cold one, although not as marked as expected. Also, the difference when using low selectivity and a warm buffer is close to zero. The clear difference from Figure 3.12 in [1] is the fact that when increasing selectivity, the throughput *also* increases. We have not been able to find a reasonable explanation for this in the textbook or the DB2 documentation.

From the results we also observe – as expected – that the difference between index and scan is more marked when utilising a cold buffer and a large selectivity. Here the overhead of one or more (most likely only one) disk seeks has a greater impact than that of the index overhead and therefore the index wins over the scan. So, apart from the curves going up instead of down, the results confirm (albeit vaguely) our hypothesis.

2.4. Part 3: Problem Log

Numerous problems have been encountered while doing this assignment:

- When tuning parameters, many of the efforts have yielded uninteresting results: nothing happened when adjusting the parameters. This is extremely frustrating, as you would

expect that tuning a database (which this course is all about!) should have an effect – otherwise the course doesn't make sense. This may lie in not conducting the experiments properly, but first of all we are not the only ones with this problem, and secondly we have tested on three different instances and hardware platforms.

- Finding tuning knobs for tuning the reads is rather hard. It was fairly straightforward finding some for tuning inserts and the logging, but the obvious parameters for tuning reads are keys and indexes.
- Most of the experiments had already been conducted when someone found a potential bug in the `reads.py` script (the missing "conn"). This *must* not happen one week before the due date of the assignments! There has been no time for redoing the experiments and we do not know, whether this would have yielded completely different results.
- There was no prefix match query to help the experiment towards resembling the figures in the book.
- It took very long time to figure out how to get a cold buffer for the reads. A more thorough guide on this particular matter would have been helpful. Also, writing the shell scripts for running with a cold buffer took a long time, after having figured out how to obtain one.

3. Assignment 3

3.1. Introduction

In this assignment we take a look at two important factors in designing modern database systems: query tuning and row compression. In the first part we are given a query to optimise. We do this using different techniques presented in the course and use the `db2expln` tool to monitor the estimated costs of executing the query.

In the second part we investigate the potential performance and storage benefits derived from using the row compression feature in DB2. We specifically look at how row compression can reduce storage requirements, as well as potentially speed up read and write operations, if the overhead of the compression algorithm is not larger than the overhead of performing disk operations.

All experiments have been conducted on a local instance of DB2 version 9.5.2 for OSX, running on a 2.4 GHz Intel Core 2 Duo machine with 4GB of RAM and a 320GB S-ATA 5400rpm disk. None of the experiments have suffered from lack of disk space or memory.

3.2. Part 1: Query tuning

Our starting point is the following query on the `employees` table:

```
1 select distinct e1.ssnum, e1.name
2 from employees e1, employees e2
3 where e1.hundreds1 > e2.hundreds1 or e1.hundreds2 < e2.hundreds2
4       and sqrt(bigint(e2.lat - e1.lat)*bigint(e2.lat - e1.lat) +
5        bigint(e2.long - e1.long)*bigint(e2.long - e1.long)) < 10000
```

The `hundreds1` attribute describes the attendance rate of the employee, `hundreds2` describes the sickness rate, and the `lat` and `long` attribute describe the position (in latitudes and longitudes) of the employee. Thus, this query selects the social security numbers and names of those employees who have a higher attendance or lower sickness rate than their neighbouring colleagues (where neighbouring is defined as being within a 10km radius). We shall now try to optimise this query in order to get it to run faster.

PLEASE NOTE:
After having done the below experiments, it came to our attention that the query was missing parenthesis around the `OR` condition, causing it to return very different results. As we did not have time do redo the experiments, they have been conducted on the above (incorrect) query.

3.2.1. (A): Baseline performance

We begin with loading a fresh copy of the `employees` table, without any indexes or primary keys. We then update the database statistics using the following command:

```
> db2 reorgchk update statistics on table <username>.employees
```

With updated statistics we can use the `db2expln` tool to estimate the costs of the above query (`original.sql`):

```
> db2expln -f original.sql -d tuning -t -g
```

The last parameter enables a graph of the optimiser plan, making it easier to see the actual work done by the DBMS when executing the query. From the output of this command we can extract the estimated cost and cardinality of the query:

```
[...]
Estimated Cost = 1044677,312500
Estimated Cardinality = 1000000,000000
[...]
```

The value of "Estimated Cost" is measured in "timerons": an abstract unit of measure that indicates the cost for the DBMS to execute the query (in terms of CPU time, disk access, etc.). We shall try to minimise the value of this cost.

With respect to the cardinality, the DBMS indicates to us that almost all rows are returned. This makes good sense, due to the fact that one of the clauses on `hundreds1` or `hundreds2` will very frequently be true, as the attributes are uniformly distributed.[1]

3.2.2. (B): Possible performance issues

When analysing the query in its original form, we get the following optimiser plan:

```
Optimizer Plan:

          Rows
        Operator
          (ID)
          Cost

           1e+06
          RETURN
           ( 1)
        1,04468e+06
           |
           1e+06
          TBSCAN
           ( 2)
        1,04468e+06
           |
           1e+06
          SORT
           ( 3)
        1,03732e+06
```

[1] Executing the query returns 999.898 tuples, so the estimate is actually quite precise.

```
          |
        1e+06
        NLJOIN
         ( 4)
        983690
       /      \-\
    1e+06        *
   TBSCAN        |
    ( 5)       1e+06
   12172,5   Table:
     |       ANDERS
    1e+06    EMPLOYEES
  Table:
  ANDERS
  EMPLOYEES
```

We see that the dominating cost factors are the `NLJOIN` (nested loop join) and the following `SORT`. Also, we observe that the optimiser deals with all of the one million rows in every step of the process. In the following section we shall investigate how to reduce the cost of these operations or get the query optimiser to choose other equivalent means of execution.

Initially, when only looking at the query, the following possible performance issues can be identified:

- There is a potential $O(n^2)$ performance bottleneck if the query processor should choose to do a Cartesian product on `employees` with itself. This would result in $1.000.000^2$ tuples in the result table. As one would expect, it seems that the query optimiser chooses not to do so and performs a join instead.
- There are no indexes or primary keys present to help the DBMS perform the two table scans and the sort in the optimiser plan.
- The `DISTINCT` operator forces a sort on the result set. In our case, the data has been generated with `gentable.py` set to use `ssnum` as a key, although the schema does not declare it as such. Therefore the `DISTINCT` should in theory be unnecessary, if we choose to set a primary key on `ssnum`.
- There is no need to declare the variables for calculating distances as `bigint`'s. This may cause a memory overhead when doing the many calculations needed.

3.2.3. (C): Actions to take

There are several different actions to take in order to decrease the estimated cost of the query. These include:

- In the schema, the `ssnum` attribute has not been declared a primary key although the data has been generated according to this (which also makes perfect sense in the real world). Adding a primary key on `ssnum` might hint to the optimiser that it can save the `DISTINCT` operator, as the result tuples are already distinct.

- Adding various indexes may help speed up or even eliminate some of the table scans.
- Rewriting the query to perform better. However, despite intense efforts we have not been able to realise, how this may be done. There are no nested subqueries, no unnecessary `DISTINCT`s (see above), no temporary tables (other than the sort table the optimiser creates), no materialised views, no `ORDER BY`, no nothing. We have also tried using various different joins on the conditions in the query, but all having no effect.
- Remove the `bigint`'s from the query.

3.2.4. (D): Quantifying experiments

Setting `ssnum` as primary key

We change the schema in `init.sql` to set `ssnum` as primary key. This should in theory eliminate the need to sort the table before returning the relevant tuples. Luckily, we see from the new optimiser plan that the cost decreases and the cardinality remains the same. The `SORT` and `TBSCAN` also disappear:

```
[...]
Estimated Cost = 983690,500000
Estimated Cardinality = 1000000,000000
[...]
Optimizer Plan:

        Rows
      Operator
        (ID)
        Cost

       1e+06
       RETURN
        ( 1)
       983690
         |
       1e+06
       NLJOIN
        ( 2)
       983690
      /      \-\
   1e+06        *
  TBSCAN       |
   ( 3)       1e+06
  12172,5   Table:
    |       ANDERS
   1e+06    EMPLOYEES
 Table:
 ANDERS
 EMPLOYEES
```

Creating indexes

We create three index for the `employees` table:

```
1 create index ssnumcl  on employees (ssnum) cluster pctfree 0;
2 create index myindex1 on employees (hundreds1,hundreds2);
3 create index myindex2 on employees (lat, long);
```

After updating statistics, we get exactly the same estimated costs as with not having the indexes present. If we, alongside these indexes, set `ssnum` as primary key, we get the same cost as just setting the primary key.

Removing the `bigint`'s from the query

We simply remove the four `bigint` declarations from the query. Doing this, the query execution plan remains the same, but the overall cost and cardinality of the query from `db2expln` now becomes:

```
[...]
Estimated Cost = 1042598,562500
Estimated Cardinality = 1000000,000000
[...]
```

which is actually a slight reduction.

Quantifying experiments and hypothesis

In order to justify the proposed enhancements we conduct experiments to see, whether the costs predicted by the `db2expln` utility are actually correct on real-life tests. We use the `reads.py` script from assignment 2 using only one query (we leave it up to the reader to guess which one), and repeat the experiment 10 times to get more precise results and to prevent outliers from cluttering conclusions. We have tested the following four scenarios:

- The baseline performance of the query. We expect this baseline performance to take some time.
- Adding indexes to the schema. We add the indexes mentioned above to the schema. These indexes have been chosen to help the DBMS do the join on these attributes.[2]
- Setting a primary key on the `ssnum` attribute. This should in theory prevent the sort in the optimiser plan, and maybe even eliminating the `DISTINCT` operator.
- Removing the `bigint` declarations should (according to `db2expln`) give us a slight reduction in execution time, most likely due to the calculations in the query requiring less memory overhead.

[2]Numerous different indexes have been tried, but they all seem to fare on par with these two.

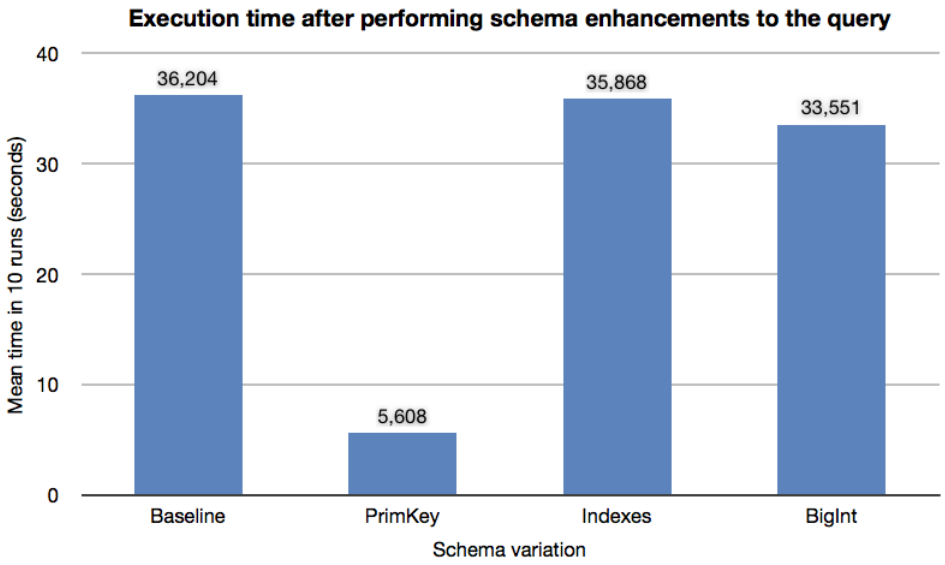

Figure 3.1.: Varying schema enhancements.

Observations and discussion

Results can be seen in Figure 3.1. The baseline performance of the query is slow, as expected: around 36 seconds to retrieve the 999.898 tuples. If we add indexes, the query improves only by a fraction. This is quite interesting, as we would have expected to see drastic improvements when adding relevant indexes. Finally, we also see that removing the `bigint` declarations have a small, but yet noticeable improvement on execution time, as expected.

It is not obvious how to explain this behaviour, especially not keeping in mind the results of the query when adding a primary key on `ssnum`. This apparently boosts the execution of the query from around 36 seconds to only 5.6 seconds, which is quite an improvement. The primary key added will help execute table scans in general, but the effect of this schema addition is surprising and unexplainable.

The behaviour of these enhancements is puzzling and unexpected and we would be quick to put it down to flaws in the experiment. We are not able to see where these flaws might occur, though.[3]

We have not tried combining the above modifications, but we believe that this will improve further on the execution.

3.3. Part 2: Compression

In this part we will take a look at how row compression in DB2 possibly can improve performance of I/O intensive operations as well as storage requirements. In section (A) we describe the potential benefits and drawbacks of using row compression, and in section (B) we conduct

[3]Except maybe for working on the wrong query...

experiments that try to prove or disprove the points stated in section (A).

3.3.1. (A): Advantages and drawbacks

Row compression works by identifying multiple instances of the same data and using this to compress these instances in the table to save space. The translation between the compressed and uncompressed versions of the data occurs using a dictionary taking up little space and stored elsewhere in the table (usually also always in-memory). Using compression therefore has some obvious advantages:

- Depending on the type and organisation of the data in the table, the documentation states that savings of 50% or more can be achieved by using compression. This is of course achieved by compressing similar data in the table.
- The above holds also for memory usage, as the data is also loaded compressed into memory.
- When data takes up less space on a disk, the I/O needed to fetch an amount of tuples from disk decreases, as the compressed data of course takes up less space and therefore can be loaded faster.

There are, however, also some possible drawbacks from using row compression:

- If the data is of a such nature that almost no tuples contain the same data, compression algorithms will not find any work to do. In this case a CPU overhead may occur, when a compression algorithm desperately scans a table for something to do.
- When compressing and decompressing data, there is of course a CPU overhead of doing the work. Whether or not this overhead actually has an influence on query execution time depends on the circumstances (size and contents of table, nature of queries, etc.).

3.3.2. (B): Quantifying experiments

In this section we will conduct experiments to see whether or not we can benefit from using row compression on a DB2 database instance. The row compression feature of DB2 is described briefly in [2]. We conduct the experiments on the same `employees` table as used in Part 1.

First we have a look at storage requirements and how row compression may be able to save disk space and memory usage. We issue the following command to get the physical size (in 4KB pages) of the `employees` table:

```
> db2pd -db tuning -tcbstat | grep '  EMPLOYEES'
```

This tells us that the `employees` table takes up 11.913 pages, equal to 47.652KB (somewhere around 46.5MB). We then enable compression using the `alter table` command, followed by a forced reorganisation of the table:

```
> db2 alter table employees compress yes
> db2 reorg table employees
```

After enabling compression, the table has now shrunk in size to 7.407 pages, equal to 29.628KB. We have therefore achieved a storage space saving of about 39%, which is pretty good considering the contents of the `employees` table, where not an awful lot of data seems to be candidate for compression.

So, in terms of storage requirements, the compression feature of DB2 seems to live up to its promises. We will now try to issue requests to the compressed database in order to measure the performance when accessing and updating compressed data.

Setup and experiments

We test five different operations on the `employees` table, all with and without compression turned on: inserts (100.000 tuples), updates (100.000 tuples), reads with multipoint queries, reads with range queries, and reads with full table scans. We use the following general setup and make some slight modifications for the `writes.py` script to function with the `employees` table:

- We run the `writes.py` script with the following arguments (for updates):
  `-t1 -r1 -iCS -wupdateN -x1 -n100000 -a2 -a0 -semployeespec`.
  This way we use the attributes `lat` and `ssnum` for the updates.
- We modify the `updateN.sql` query to:
  `update employees set lat = ? where ssnum = ?`
  to match the above `writes.py` parameters.
- We modify the `insertN.sql` query to:
  `insert into employees values (?, ?, ?, ?, ?, ?)`
  in order to accommodate the extra attributes of the `employees` table.
- In `query_range.sql`, we select 10% of the data.
- For all the read queries we measure mean execution time for 200 queries.
- We utilise only one thread for both reads and writes.

Observations and discussion

Results can be seen in Figure 3.2. Initially we observe that when performing inserts or updates, there is no noticeable difference from using compression or not. So what can one conclude on these observations? Well, apparently there is no noticeable overhead from the *compression* algorithm when inserting new rows or writing data when committing updates – or at least the compression algorithm overhead combined with the smaller amount of data to be written to disk matches the time it takes to write the uncompressed data to disk.

When we look at reads, however, we observe some real differences. In both the multipoint, scan, and range queries, the results when using row compression are about twice as slow as when not using it. This suggests to us that the *decompression* algorithm used may not be efficient enough to keep up with regular disk reads. Not surprisingly, in [2] the author only shows results from storage savings experiments. There are no results on the actual performance of

Figure 3.2.: Different operations with and without compression.

the algorithms in terms of query response time.

If we were to make recommendations as database tuners based on the results above, we would recommend using row compression in an environment with large databases, that have few data requests and many updates. This way a lot of storage space could be saved and performance would be almost unaffected. However, with the prices on disk storage these days the incentive isn't that large. We would *not* recommend using it in read-intensive environments where fast access to data is required.

3.4. Part 3: Problem log

Several problems have occurred during the course of solving this assignment:

- Four days before the due date of the assignment, we find out that the query in Part 1 is incorrect. There has been no time to re-analyse the query and redo the experiments, so basically Part 1 tells us very little on query tuning. Not good...!
- When analysing the query (correct or not) it has been very troublesome finding ways to improve it, other than primary keys (which ought to have been there from the beginning) and indexes. There are no nested subqueries, no unnecessary `DISTINCT`s, no use of temporary tables or views, etc. So most of the examples in [1] are useless for this query.

Posts on forum
I have posted 21 posts on the forum (3 questions, 9 answers, and 9 lines of code).

Bibliography

[1] P. Bonnet, D. Shasha: *"Database Tuning – Principles, Experiments and Troubleshooting Techniques"*, Morgan Kaufmann Publishers (2003).

[2] R. Ahuja: *"Introducing DB2 9, Part 1: Data compression in DB2 9"*, http://www.ibm.com/developerworks/data/library/techarticle/dm-0605ahuja/index.html (2006).

A. Supplementary results

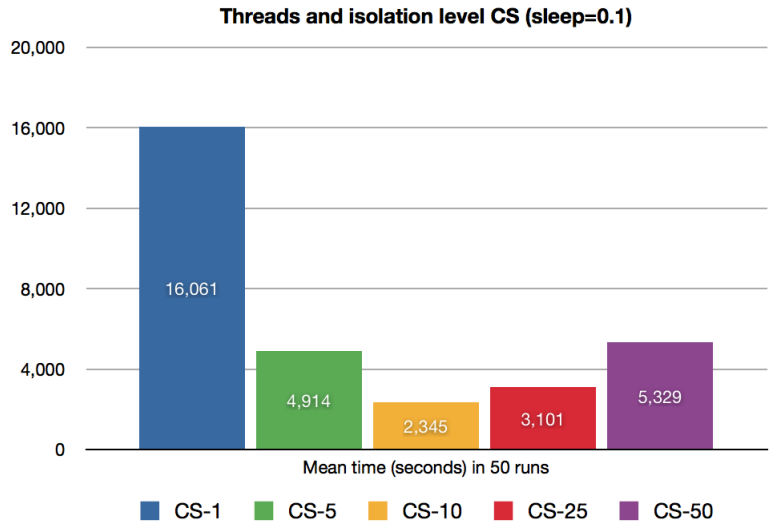

Figure A.1.: AWS times for isolation level CS with sleep of 100 ms (50 runs)

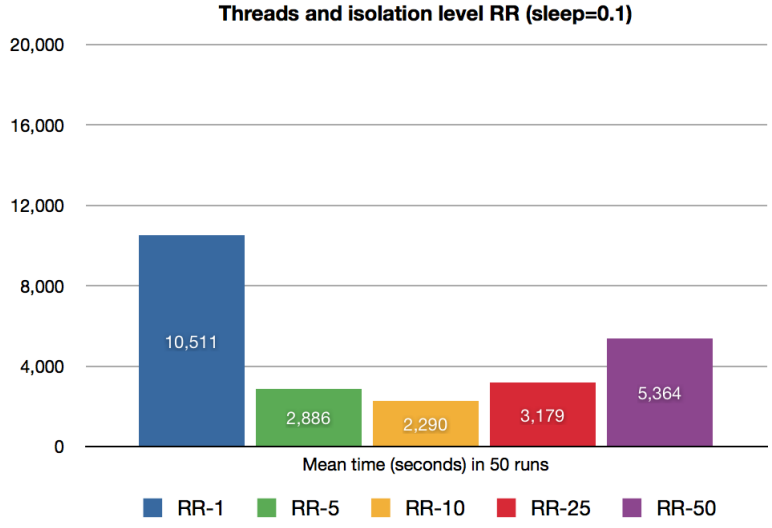

Figure A.2.: AWS times for isolation level RR with sleep of 100 ms (50 runs)

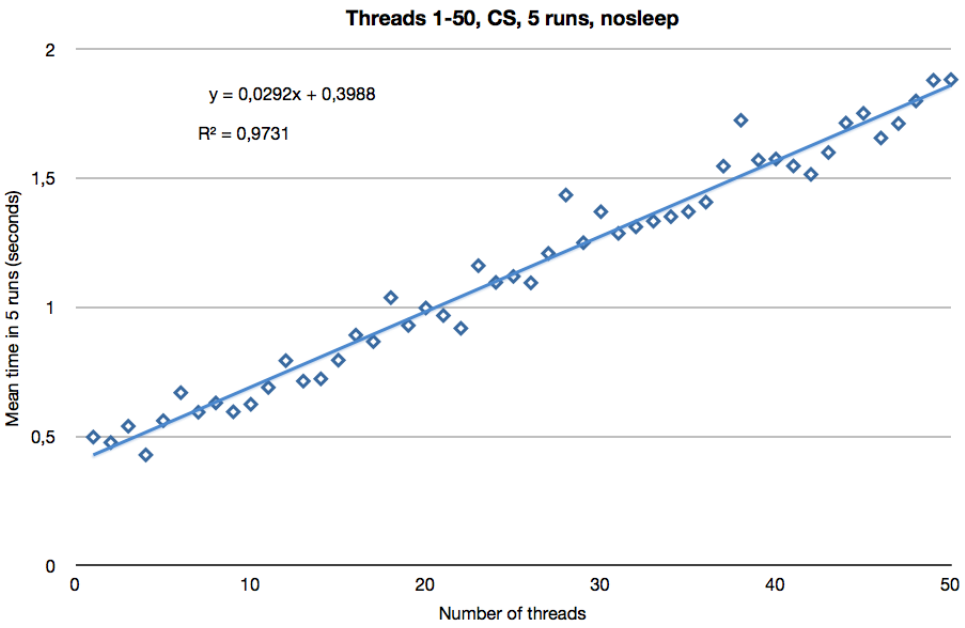

Figure A.3.: AWS times for isolation level CS with no sleep

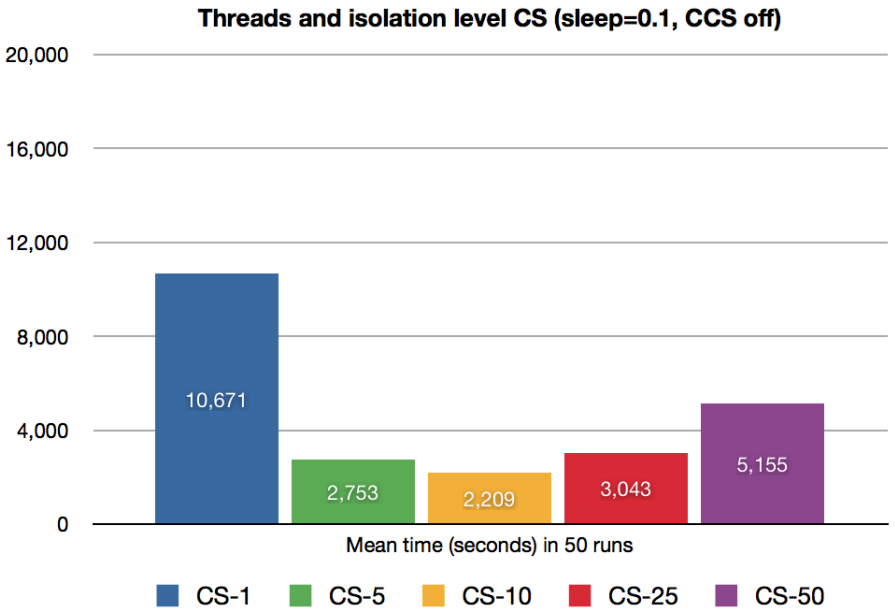

Figure A.4.: AWS times for isolation level CS with CCS turned off (50 runs).