• Table of Contents

1 Introduction

In this report a take the role of “IT person” at ReserveWithUs. I have been given a simple application server with a set of schemas and scripts to generate data. The data are hosted on a DB2-server.

My job is to tune the application and DB2 according to various assignments. There are 4 tuning assignments (2-5 in project description) that I will tackle in 4 separate sections.

I will hypothesise on what is a propper action to achieve various tuning-goals, design test-cases and execute execute the experiments. The results will either confirm or deny my hypothesises.

I rely heavily on scripts: db2batch and (especially) bash. I do this for several reason:

- I hope to save myself some time not having to monitor every experiment.
- Reduce the risk of human error. Doing repetitive tasks increases the possibilities of doing something wrong (doing something in the wrong order, typos etc.).
- Making it easy to reproduce and extend the experiments.

In the end of each section I will reflect over what pitfalls, epiphanies and challenges I encountered.

In the final conclusion I will try to pick up the most important parts of this project.

In Appendix A I have put the codes and the data in Appendix B.

2 Description of setup

I have 2 computers to do my experiments on. The first computer is a Giada N10 nettop that I use primarily for XBMC. Most of the time it is not in use and not while I will perform my experiments.

The second computer is at Thinkpad laptop that is my primary computer. So to avoid DB2 wrecking havoc on my system it is installed in a VMware virtual machine. I would have prefered having DB2 running on dedicated hardware, but I had none to spare.

I have chosen to run DB2 and the application-server on different hardware to separate the processes and be able to pinpoint where activity is happening. And to see what kind of impact network communication have on performance.

The choice of VMware Player over Virtual Box is due to the fact that I've previously have done test with Virtual Box on similar setup (64bit Linux host) and found that the performance of DB2 was severely degraded compared both to a Windows host and VMware Player on a 64bit Linux host.

My choice of Linux (Ubuntu) is simply because of my familiarity with it.

In Table 1 through Table 3 I have listed my setup of “hardware”.

VMware Player – DB2	
CPU	2 cores
RAM	3 GB
Harddisk	15 GB dynamically sized / write-cache is off
OS	Kubuntu 12.04 (64 bit)

Table 1: Guest system setup. Write-cache is off.

Lenovo Thinkpad Edge 13” (0217)	
CPU	Intel Core i3 i3-380UM @1.33 GHz
Chipset	Mobile Intel HM55 Express
RAM	8 GB (2 x 4 GB)
Harddrive	500.0 GB HDD / 5400.0 rpm (SATA-300)
OS	Kubuntu 12.04 (64 bit)

Table 2: Host system setup for the VMware virtual machine.

Giada N10	
CPU	Intel Atom N330 @1.6GHz
Chipset	Nvidia Geforce 9400M
RAM	2GB DDR II SO-DIMM
Harddrive	250 GB HDD / 5400.0 rpm (SATA-300)
OS	Xubuntu 11.10 (32 bit)

Table 3: This is the computer where the application will run

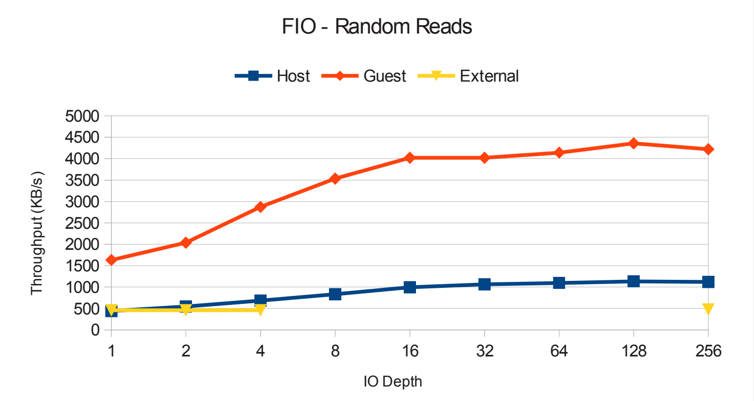
2.1 FIO – Performance of secondary storage

I will do some basic performance test of the secondary storage that DB2 will use in some of my experiments. I will use FIO to test the Thinkpads HDD, the virtual machines virtual HDD and an external HDD that I will attach to the virtual machine. I will test throughput/iops of sequential writes and random reads with various IO-depth.

I start with IO-depth 1 and doubles until I can see the performance level out.

I will do the test on the virtual drive with a clear file-cache on the host in a hope to eliminate measuring host-cache performance instead of direct access by guest.

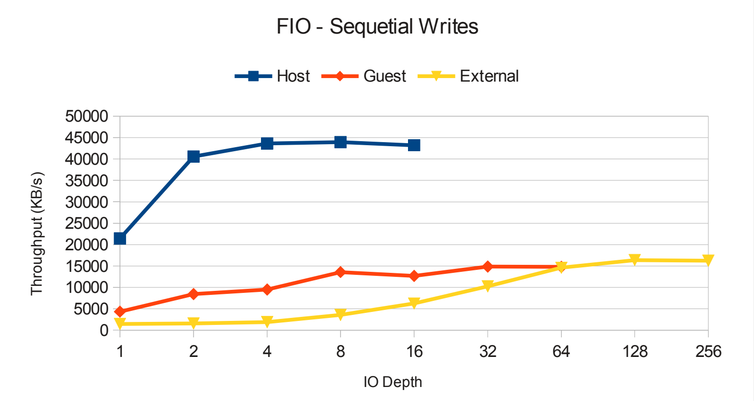
2.1.1 Result of FIO test.



Graph 1: Throughput of random reads of the 3 different secondary storage.

In Graph 1 we can see that the virtual HDD outperforms its host HDD even though I have cleared the hosts file-cache. It shouldn't.

An observation I did while a fio-test was executing was that the throughput increased – the virtual drive got faster and faster. My conclusion is that the virtual drive access the host file-cache that is being filled with the FIO test file. So I am not able to eliminate host-caching while reading.



Graph 2: Result of sequential write to the 3 secondary drives

In Graph 2 we see that the result of the sequential writes are more in line of what I would expect. There is clearly some overhead of writing to the virtual drive (a factor 3-4 times slower than the host). This might partly be because the virtual drive is dynamically allocated and the drive might therefore not be sequential allocated on the host and thus causing random access.

One thing I encountered was that if I forgot to clear the host file-cache the throughput would doubled (6558 iops vs. 3318 iops), so even though write-cache is off in the virtual machine the host does seem to ignore that to some extend.

My external hdd is in most cases slower or as slow at the virtual drive, which limit any performance boost I might hope to achieve by offloading part of the DB2 IO to an external drive.

2.2 Network

The application server is connected to the network through a 100 Mbps ethernet cable, while the DB2 server is connected to through a 802.11g wireless connection via a bridged connection in the virtual machine.

This is a ping-statistics between the two:

```
--- 10.0.0.10 ping statistics ---
25 packets transmitted, 25 received, 0% packet loss, time 24040ms
rtt min/avg/max/mdev = 1.292/2.973/6.220/1.132 ms
```

3 ms roundtrip I think is ok being that close together and over wifi.

3 Index Analysis and Tuning

In this section I will look into benefits and penalty of creating various indexes to serve ReserveWithUs.

I will argue for which indexes and types of indexes (clustered vs. non-clustered etc.) I find appropriate. I will create test-cases based on the presented database workload characteristics, both regarding reading of the database and the (increased) cost of inserting and updating rows into tables. Indexes must be updated as well so creating an index adds an overhead to write-operations.

I will then compare the performance of my choices against the baseline performance. I will then be able to confirm or dismiss whether my choices are appropriate – or at least qualify the pros and cons.

3.1 Selecting indexes

The choice of indexes must address the following workload characteristics:

- Many queries on hotels and rooms are based on country and city
- Many queries are repeated many times (by different users)
- All queries on rooms should contain a date interval
- The sales department loads new deals once a month
- The customers and the hotels grow slowly and regularly (a hundred new customers per month and a couple of new hotels per month)
- There is a lot of activity on the shopping cart (it is required that all information is given when a new item is added to the shopping cart).

I will assume that the “and” in “... are based on country *and* city” is a logic *and* keeping the number of indexes low.

I will create the following indexes:

1. xHOTEL (*country*, *city*, *hotel_id*) – non-clustered and good covering.
2. xROOMS (*hotel_id*, *room_type_id*) – non-clustered
3. xROOM_DATE (*room_type_id*, *single_day_date*) clustered.
4. xCUSTOMER (*customer_id*) – non-clustered.
5. xSHOPPING_CART (customer_id, room_type_id, date_start, date_stop) – non-clustered

3.1.1 xHOTEL (country,city, hotel_id)

With this index it should be faster to acquire information in the HOTEL table based on queries on country *and* city. I choose a *non-clustered* index to keep the overhead of adding new hotels low. Since HOTEL grows “slowly and regularly” - I assume that means continuously over the month – making the index clustered would introduce severe overhead of IOs in reordering the table when writing (updating or inserting) to the table.

I have included *hotel_id* in this index – I could have left it out and made the index denser and faster. *But* by including it I have made a good covering index that would benefit xROOMS. The overheads of including it I believe would be minimal if I assume the number of hotels in each city are limited. The number of extra pages DB2 would have to load would be minimal if any. A default page size of 4k can store a large number of rows of:

30 x varchar (country) + 30 x varchar (city) + 1 int[1] =

30 x 1 byte + 30 x 1 byte + 1 x 4 bytes = 64 bytes.

So you can list 64 hotels (4kb/64bytes) inside a single page (minus some headers etc.) in this index. And on the plus side you avoid some random IO in servicing queries on "... rooms based on country and city". To do this you would first need to get a list of hotel_ids, but if *hotel_id* wasn't included in the index DB2 would have to have to read multiple pages placed randomly in the table and thus random IOs to secondary storage.

3.1.2 xROOMS (hotel_id, room_type_id)

I would have preferred making an index over (*country*, *city*, *room_type_id*) but that would require making a new View or Table and rewriting the application-server to use those.

Instead queries on "... rooms based on country and city" would continue to be a 2 step process. xROOMS would offer an improvement.

If we assume that hotels don't get deleted and we don't delete, update or insert rooms after the initial inserts then a clustered index could be an option because you would just add new rows at the end of the table and hence you don't need to reorder the table and index.

But I don't assume that, so I will settle on a non-clustered index. I assume room_type_id is sufficient information to answer "... rooms based on country and city", though I could have included *room_type* as well. But I don't see any indication in the Workload Characteristics that it should be necessary.

3.1.3 xROOM_DATE (room_type_id, single_day_date)

This would be a clustered index in the database. All queries on rooms "contain(s) a date interval", so having the rows stored in an unordered fashion would require a lot of random read to fetch all relevant pages regarding a date interval from secondary storage.

This is why I choose a clustered index. But it doesn't come without a penalty. When new deals are loaded once a month many new rows are inserted into the table and thus triggering a lot of reordering of the table. There will be no reordering overhead of updating attributes (like *numtaken*) because the they are confined to fixed value.

One way to deal with this could be to deactivate the index while inserting a huge amount of rows.

Since ROOM_DATE plays a big role in the performance of SHOPPING_CART I could have included more attributes in this index. But being a clustered index and SHOPPING_CART needs most of the attributes I see no extra benefit making a Good Covering index.

I am setting the pctfree = 0 when I create the clustered index for testing purposes (ie. worst-case-scenario).

3.1.4 xCUSTOMER (customer_id)

Assuming that no customers gets deleted then a clustered index would seem to be the right choice. Adding new customers wouldn't really add much of an overhead because new rows would just be inserted in the end of the table and index. Unless the accumulated insertion of hundreds of new customer is noticeable.

On the other hand, with the workload characteristics of the database I don't see in which situation a clustered index would be beneficial. So I will keep the DB2 created default – a non-clustered.

1 http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=%2Fcom.ibm.db2.udb.apdv.cli.doc%2Fdoc%2Fr0006844.htm

3.1.5 xSHOPPING_CART (customer_id, room_type_id, date_start, date_stop)

Due to the high activity on the shopping cart I believe a clustered index would introduce a severe overhead of reordering due to large amount of inserts, updates and deletes. Instead I think a non-clustered index on the primary keys is the way to go (customer_id, room_type_id, date_start, date_stop).

The way I have selected my other indexes – xROOM_DATE especially – would serve the operations on SHOPPING_CART at a faster pace. The Application-server needs to retrieve data from ROOM_TYPE, ROOM_DATE and CUSTOMER before checking out items in the SHOPPING_CART.

3.2 Design of test-cases

My *intent* of the experiment design is to test the performance improvements or lack of of the chosen indexes and their penalties compared to the default setup. Since some of my suggested indexes are created by default by DB2 some of my intended experiments will not be doable.

What I want to test is the performance of various queries that fit the workload characteristics. It will be a mix of simple select-queries, inserts and select-queries on those inserts.

I will test the performance of both a cold and warm database (ie. whether or not the tables are to be fetched from secondary storage or are already in RAM). To get a cold database I will shut down DB2 and restart it (db2stop; db2start). And because DB2 is installed on a virtual machine I will have to flush the VM Hosts file-cache as well:

```
#sync; echo 3 > /proc/sys/vm/drop_caches (executed as root).
```

Every experiments are repeated 3 times to get an average that will level out single flukes (sudden unrelated operations on the database server).

All of these tests and procedures are woven into a set of bash-, db2batch and sql-scripts so I can execute them in one batch without supervision and the execution-times are written to log-files. And it is easy to repeat and replicate the experiments. See Appendix 9.1 for code examples.

I will test these queries:

SELECT * FROM HOTEL WHERE country='country1' AND city='city2';	INSERT INTO HOTEL VALUES (50000, 'hotelx', 'streetx', 'cityx', 333, 'statex', 'countryx', 1, 1);	SELECT * FROM HOTEL WHERE country='countryx' AND city='cityx';
SELECT hotel_id FROM HOTEL WHERE country='country1' AND city='city2';		
SELECT room_type_id FROM ROOM_TYPE WHERE hotel_id=25000;	INSERT INTO ROOM_TYPE VALUES (300002, 25000, 'roomtypex');	SELECT room_type_id FROM ROOM_TYPE WHERE hotel_id=25000;
SELECT * FROM ROOM_DATE WHERE room_type_id=219117 and single_day_date between '2013-04-01' and '2013-04-30';	INSERT INTO ROOM_DATE VALUES (5000, '2013-01-05', 2, 0, 99);	SELECT * FROM ROOM_DATE WHERE room_type_id=5000;
SELECT * FROM CUSTOMER WHERE customer_id between 1000 and 1005;	INSERT INTO CUSTOMER VALUES (1000002, 'userx', BLOB('123456'), 'firstnamex', 'lastnamex', 'homestreetx', 'homecityx', 4000, 'homestatex', 'businessstreetx', 'businesscityx', 4001, 'businessstatex', '99999999', '8888888', 'emailx', 'danish');	SELECT * FROM CUSTOMER WHERE customer_id=1000002;

The first column are all multipoint queries to level out the randomness of the data-set. In the second column I insert just a single row – this is just to get an idea of the penalty of inserts. I am aware that a batch insert may have another profile due to logfiles, IOs etc.. The last column it would have been better if they'd been identical to the first column to make them more comparable. But since they are new data its impossible anyway.

Keeping the default non-clustered index in for CUSTOMER and SHOPPING_CART there is no need to run an experiment since the performance of “before” and “after” should be identical.

This concludes phase 1 of the index-experiments. To test the performance of the shopping cart I will do a simple experiment on the Application-server. I will test how fast a checkout is before and after I have created my indexes. I will run this simple script which is just the supplied user2.checkout-script:

```
command=login&customer_id=112
command=add_to_shopping_cart&customer_id=112&date_start=2013-05-01&date_stop=2013-05-08&total_price=150&roo
m_type_id=1&numtaken=2
command=checkout&customer_id=112
command=logout&customer_id=112
```

And I will only be testing on a cold database because getting to a reliable warm state is complicated but possible doable (doing a login and logout first and make a query on the data-range).

I have not explicitly tested how the performance increases/degraded when I insert a batch of new room_date row (loading new deals). I could alter the python-script to generate a new set of deals a month ahead and test how loading of new deals are handled.

Implicitly I test it by inserting a single row into ROOM_DATE table.

3.2.1 Limitation of Cold and Warm databases.

DB2 is installed inside a virtual machine and this introduces some issues on IOs to secondary storage, reading especially. The Linux host keeps a cache of recently accessed files and having 8 GB of memory, this file-cache is quite large and capable of keeping most of the guest OS's virtual drive in RAM. So for me to get to at cold database state I need to not only stop and restart DB2 I also need to flush the host file-cache.

And warm databases have a few issues to. It is hard to tell whether DB2 is accessing a buffer or the host file-cache so we need to consider this when analyzing the experiments done on a warm database.

Writing to secondary storage shouldn't in principle be an issue since write-cache is turned off so the guest OS writes directly to secondary storage. But as my FIO-test have shown (see Chapter 2.1) this isn't entirely correct.

3.3 Encountered issues and Comments

It appears that DB2 creates indexes on it own, making several of my suggested indexes redundant and impossible to create because they already exists. These auto-generated unclustered indexes are created over the tables primary keys. So unless I wish to create an index with the primary keys in alternate order or include attributes that are not primary keys, the index is already created.

It does make sense though. A primary purpose of the default indexes is to make sure that no rows with identical primary keys gets inserted. Without these default indexes DB2 would have to scan the entire table each time a new row is inserted and degrade performance considerable.

Furthermore I am unable to delete these indexes to perform comparisons of performance improvements and penalties of index vs. non-index. DB2 returns this error if I try:

```
“SQL0669N  A system required index cannot be dropped explicitly”
```

xROOM_DATE, xCUSTOMER and xSHOPPING_CART are the affected indexes.

Yet another point is to determine whether these system required indexes are non-clustered or clustered. If I issue this command:

```
$db2 select indextype from syscat.indexes where tabname='customer'
SQL0206N  "CUSTOMER" is not valid in the context where it is used.
SQLSTATE=42703
```

But if I issue this:

```
$ db2 select tabname from syscat.indexes

TABNAME
--------------------------------------------
BOOKED
CUSTOMER
HOTEL
HOTEL
LOGIN
ROOM_DATE
ROOM_TYPE
ROOM_TYPE
SHOPPING_CART
SYSATTRIBUTES
[---snip---]
```

It's clear that the CUSTOMER is a valid tabname. I am apparently missing something here. But I did succeed – I think – in getting the information I wanted. Issuing:

```
$ db2 select indextype from syscat.indexes

INDEXTYPE
---------
REG
REG
REG
REG
REG
REG
REG
REG
REG
[--- snip ---]
```

If the output are listed in the same order all the relevant indexes are “REG” (regular aka non-clustered). There is an “ALTER INDEX index-name CLUSTER” procedure in DB2 that can change and existing index into a clustered index. But it is not available on Linux.

In the end I ended up with a work-around for my xROOM_DATE that I wanted to be clustered. I created the following index followed by a offline reorg:

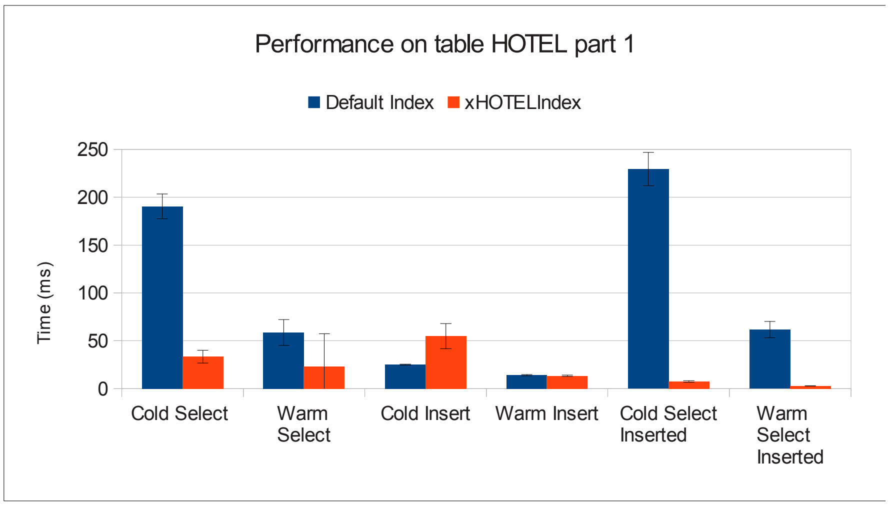
```
$db2 “create index xroom_date on room_date (room_type_id, single_day_date, price) cluster pctfree 0”
$db2 reorg table room_date index xroom_date
```

The end-result is the same, the rows gets organized by room_type_id and single_day_date.

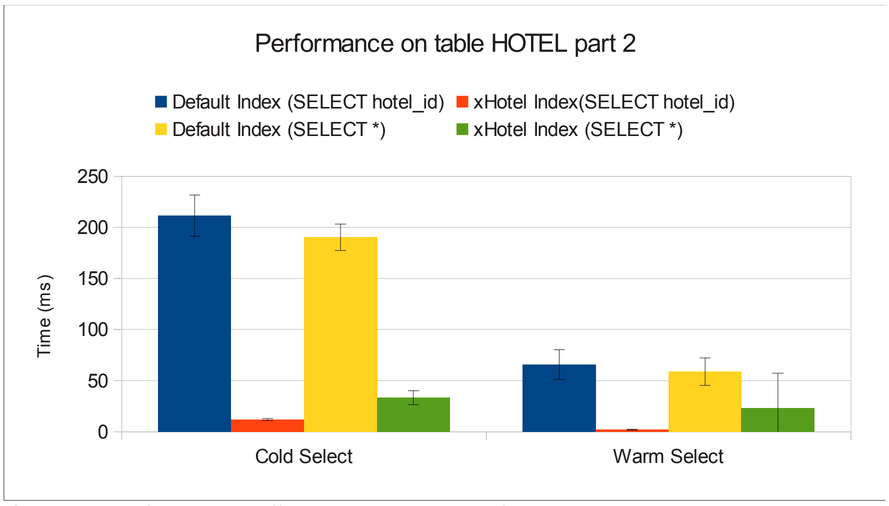
After further research it appears that a better approach would be to create the schemas without primary key constraints, create a clustered index and *then* assign the primary keys. This way I would only end up with just one index.

3.4 Results

These are the results of my experiments.



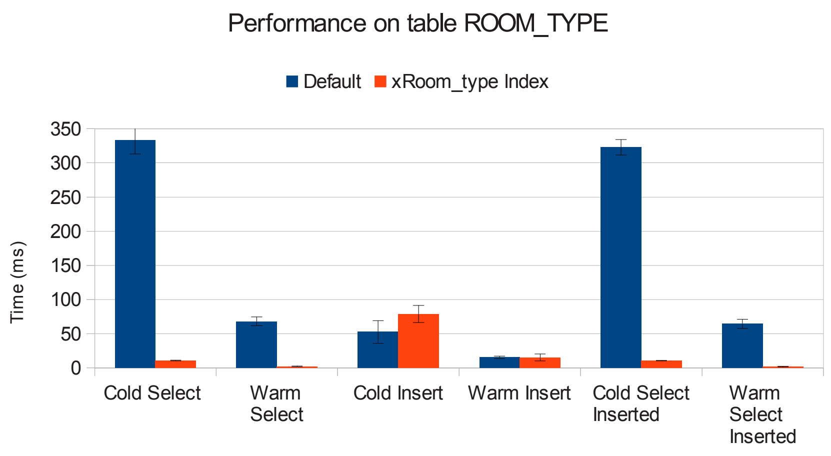
Graph 3: Performance differences between default index and xHotel



Graph 4: Performance differences between default index and xHotel

Graph 3 and Graph 4 shows that there are major improvements across the board by introducing the xHotel index. *Except* when inserting new rows (adding new hotels). Insertion time doubles for a cold database. But the Hotel table only increases by a couple of hotels pr. month so the cost are negligible.

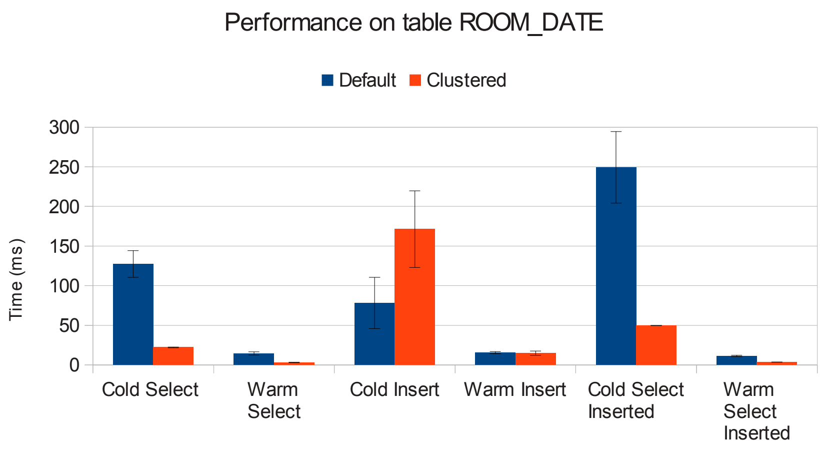
The performance improvements in Graph 3 are: 82%, 61%, **-120%**, 4%, 97% and 96%.



Graph 5: Performance differences between default index and xRoom_type index.

In Graph 5 we see the same pattern though the performance boost is even bigger. And again there is a slight penalty of inserting new rows into this table. A penalty that will only occur when adding new hotels or the rare occasion that an existing hotel gets new room_types (but I will assume that is not the case.

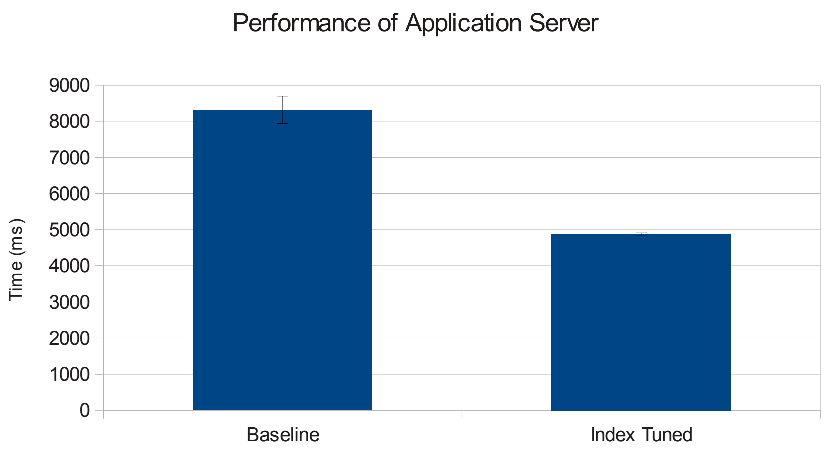
The performance improvements are: 97%, 97%, **-50%**, 2%, 97%.



Graph 6: Performance of default index and clustered index on table Room_date

In Graph 6 we see once again the same pattern. *But* this time around we can't ignore the penalty of inserting new rows. When new deals are inserted once a month this penalty would be accumulated over ~3,5 mio. times, if we assume the number of deals pr. month are roughly the same.

The performance improvements are: 83%, 78%, **-119%**, 5%, 80%, 68%.



Graph 7: Performance of Application Server

Finally, in Graph 7 we see that my index-tuning does benefit the application-server as well. With tuned indexes it takes ~60% of the time to complete a checkout compared to the baseline. So my selection of indexes does something to address the high activity on the shopping cart. A 41% performance improvement.

3.5 Discussion

My choices of indexes offers noticeable performance improvements over the default – some up to 97%! Any penalties for inserting new rows are negligible in the daily operations (the events are few). Except when you need to load new monthly deals.

A back-of-the-envelope calculation suggest (for worst case on cold database) it will take:

~150 ms/event x 3,5 mio event = ~145 hours!

That's unacceptable. But even with a warm database it would take more than *24 hours* if we assume no extra overhead pops up (like logging IOs, buffer fills up etc.) which still makes it unacceptable because that is a big window where DB2 performance in servicing the Application-server is degraded considerable.

I think you will need to use DB2 load command[2] just like we populated the database. *If* we assume no deals gets loaded into existing ones.

I would also design some test-cases, where I would create another month worth of deals and see how the load command performs up against various logging and buffer parameters. But that is beyond the scope of index-tuning, so I will not do that here.

2 http://pic.dhe.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=%2Fcom.ibm.db2.luw.admin.cmd.doc%2Fdoc%2Fr0008305.html

4 Operation on customer table

Once in a while the customer database need to be downloaded, deleted from the database and then reinserted into the database. Everything has to be done as efficient as possible. I identify the following challenges that I need to address:

1. How to download the database into a file.
2. Define a format for the downloaded data.
3. How to upload the customer data again.
4. How to keep customer_id intact (ie. avoid inserting new with old id while process is under way).
5. Should the database be online or offline.

Constraint #4 have not been explicitly expressed but without it the database would be messed up (the tables booked and shopping_cart would refer to the wrong customer). But since the Application-server doesn't support adding of new customers[3] I will assume that insertion of *new* customers are done on the system administrative level and new customers will not be added while this process is being executed.

There are 2 ways to put a database (or table) into a file (constraint #1) – as far as I know:

- Being on Linux (or any POSIX environment) I could just pipe the output like this:

```
db2 "select * from customer" > custom.txt
```

  I will rule out this one immediately though on 2 counts:

  - The output is not easy to work with because it is not delimited in a manageable fashion (fx. comma separated).
  - The customer table includes a blob-type which doesn't come out right.

- DB2 also have an export command[4]. I will use this.

Using DB2 export command the choices of formats (constraint #2) are limited to:

- DEL – delimited plain-text (ASCII)
- IXF – proprietary binary format
- WSF – Work Sheet Format, which is not applicable here and therefor ruled out.

To upload the customer database again (constraint #3) DB2 offers 2 options:

- import
- load

According to the documentation[5] both could be applicable for this purpose with Load possible being the faster option. But my experiments will show.

3 https://code.google.com/p/databasetuning-cases/wiki/ReserveWithUs
4 http://pic.dhe.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=%2Fcom.ibm.db2.luw.admin.cmd.doc%2Fdoc%2Fr0008303.html
5 http://pic.dhe.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=%2Fcom.ibm.db2.luw.admin.dm.doc%2Fdoc%2Fr0004639.html

4.1 Design of test-cases

My set of experiments will consist of 6 parts:

- exporting in DEL-format
- exporting in IXF-format
- importing in DEL-format
- importing in IXF-format
- loading in DEL-format
- loading in IXF-format

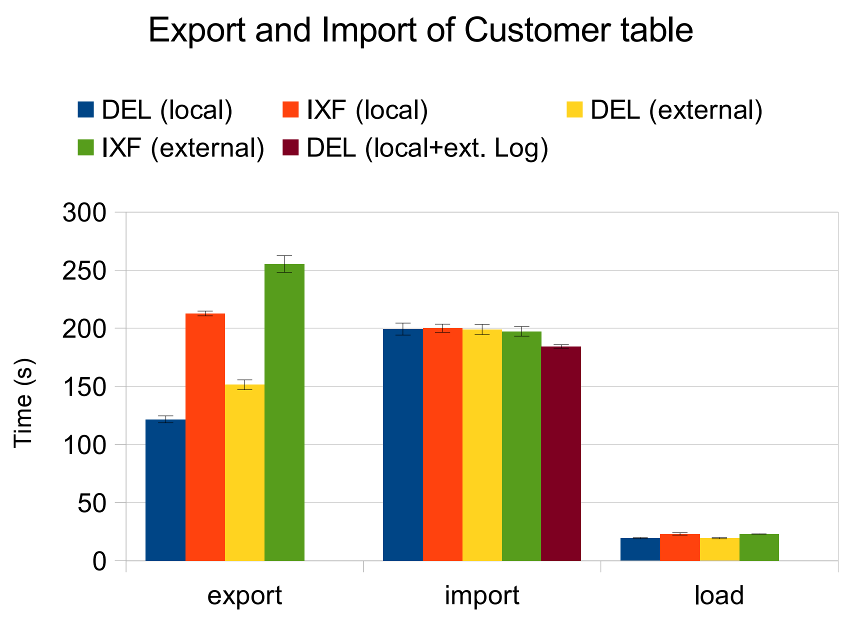
I will time each part and calculate the throughput. And I will set the database in cold state because I think a warm database defies the purpose of doing a “clean slate” of the customer database. As before I will repeat each experiment 3 times.

Finally I will redo the experiments by writing the files to a external storage unit. It is limited what conclusion can be drawn from this because the drives are on different interfaces (virtual SATA vs. usb). But I do expect to see something valuable anyway – especially while loading/importing because there would be a lot of random IOs (reading file, writing logs, writing tables).

4.1.1 Testing logfile-performance.

After running the first set of tests I am curious to what kind of impact it would have on importing if I put the database logfile on an “exclusive” secondary storage. So I will repeat a small subset of my previous experiments: import a DEL-format customer-file located on the local secondary storage while logfile resides on the external storage unit.

4.2 Results



Graph 8: These are the results of exporting the CUSTOMER table and then import or load in back into the CUSTOMER table under various conditions.

I can make 3 conclusions from my experiments:

- Load is a lot faster than import.
- DEL is the faster format.
- The local drive is faster to export the data to.
- My little extra experiment with DB2's logfile did improve import but only by a small margin.

4.3 Solution

While this operation is in process there should be no access to the database, it would mess things up – we want a complete snapshot when exporting and no interfering when uploading. So the database will be offline – or at least the customer-table and all related operations.

This means that we would want the process to be as fast as possible and from the experiments the answer is to export to a DEL-formated file on the virtual drive and use load to insert the data again.

So the process is:

1. Take database offline
2. Export customer-table to a DEL-formatted file on the virtual drive.
3. Drop the customer-table (there is no foreign key restrictions).
4. Create the customer-table.
5. Load the DEL-formated file into the customer-table.
6. Put database back online.

According to my experiments the export and load part takes a total of 141 s and then there is the overhead of dropping table and creating it. I would assume the whole process could be done within a 5 min. window. So we need to find a window with the least activity.

4.4 Encountered issues and comments

While importing the customer data I got an error:

```
SQL3306N  An SQL error "-964" occurred while inserting a row into the table.
SQL0964C  The transaction log for the database is full.  SQLSTATE=57011
SQL3110N  The utility has completed processing.  "164684" rows were read from
the input file.
```

The DB2 logfile got full, so I could do 2 things: increase the size of the logfile or import in smaller chunks. I opted for the latter and set import to commit after every 100.000 rows.

I might as well have opted for a lager logfile, and I would have liked to experiment with that. But because load is so much faster (10×) I don't believe a larger logfile would close the gap between import and load and thus change my recommendations.

5 Checkout() bottleneck

In this part I need to:

1. Establish what is the bottleneck in the application-servers checkout-function.
2. Propose a design for a more efficient checkout-function.
3. Implement the design and verify.

I assume that checkout-function refers to checkout() in the ShoppingCart class.

My hypothesis is that these 4 candidates are the bottlenecks that makes checkout() slow:

1. Database locks.
2. Network roundtrip.
3. Network congestion.
4. Code design and algorithm

5.1 Database locks

Locking *could* add to the checkout-function being slow. If concurrent checkout-threads updates the same rows (and locks them them from the other threads) in ROOM_DATE they will have to wait until the checkout-thread has committed an released its locks.

You could insert another commit right after the third part (update ROOM_DATE). You can relax it even further further by executing commit after each single update. *But* it comes at a cost: there is no (easy) rollback after a commit. And assuming the customer wants “all-or-nothing” in his shopping cart, that policy is risky, because the customer might end up with just a part of his orders instead the “whole package”.

So I wouldn't recommend the most relaxed approach, but executing a commit right after all the updates to ROOM_DATE are finished wouldn't break how the current code works, because there is no check/rollback implemented in the rest of the code anyway.

I will not test this. If I were to test it I would have to set up 2 (preferable more) concurrent clients that execute checkout almost simultaneously, that access the same rows and that have enough items in their shopping cart so they would want to update the same rows within the same timeframe. And then compare before code change and after. This precision would be hard (if possible) to achieve.

But I *can* offer a qualified estimate (a “back-of-the-envelope” calculation) at what kind of performance gain that might be achieved. I we look back at Graph 7, I made a performance test with user2.checkout-script. If we strip out the 4 s delay in checkout() the performance was about 870 ms.

If I assume the following:

- The 4 commands in user2.checkout-script takes an equally amount of time to process.
- The 4 remaining parts in checkout() (I exclude part 2 with the simulated user interaction) takes an equally amount of time to process.

I then get that a lock is hold for ~163 ms and if I remove the locks sooner by executing a commit() right after the update of ROOM_DATE, I gain ~108 ms for other checkout-threads waiting for locks to be released.

How this translates into average performance gain for the checkout-function is really down the frequency of concurrent *and* overlapping database access to ROOM_DATE. If the frequency is high there is certainly something to be gained by adding an extra commit.

5.2 Network roundtrip

The RTT (round-trip time) – the time it takes to send *and* receive a signal over the network. Or the overhead (T_1) of sending a query and receiving the answer over a network. According to my baseline measurement (see Chapter 5) $T_1 \cong 3$ ms.

If we look a checkout() and N is the number of items, then the total overhead of transmitting data over the network is:

$$T_{total} = T_1 + N \times T_1 + N \times T_1 + T_1 = (2N + 2) \times T_1$$

For a single item checkout (N = 1) I get that $T_{total} = 12$ ms. So there is very little incentive to try improve here compared what could be gained by improving on the locks. *But* if my database-server and application-server were located at different geographical location the RTT would increase by a factor 3 or more. Ie. if I ping one of my webhotels at unoeuro.com I get a RRT of approximately 10 ms. Then there might be an incentive to tune.

For large N there would also be an incentive to try to improve if they happen with a high frequency. But there is nothing in the material from ReserveWithUs whether or not that is the case.

But *if* we assume there is an incentive, my solution would be to bundle the queries into groups of queries. The initial select from SHOPPING_CART needs to be handled separately, while the rest could be bundled into 1 string of queries. *But* if we have to deal with locking the update-queries to ROOM_DATE needs to be handled in 1 separate string as well.

The result is that $T_{total} = 3 \times T_1$ no matter how large N is.

I will not run an experiment on this either because T_1 is so small in my setup that any improvement would fall within the standard deviation and in practice be non-existing. I.e. the standard-deviation in user2.checkout is 42 ms, while I would gain only 3 ms.

If I were to run an experiment I would extend user2.checkout to include 10 items in the shopping cart. Then I would execute user2.checkout 3 times for original code and the improved code respectively from a cold database and compare the results.

5.3 Network speed

The DB2 is connected to the network via IEEE 802.11g with a maximum theoretical throughput of 54Mbit/s. In practice it is a lot slower due to overhead from error-correction, interference from neighbours WiFi-routers or other 2,4GHz devices (wireless mouse, keyboards etc.) and sharing the bandwidth with other wifi devices (mobile phones, squeezebox etc.).

If we – for a “back of the envelope” calculation assumes that in practise I only have 25 Mbit/s which translates into:

$$((1000 \text{ ms/s} \times 1 \text{ s}) \ / \ (25 \text{ Mbit/s} / 8 \text{ bits/byte})) / (1024 \text{ kB/MB} * 1024 \text{ B/kB}) = 0{,}0003 \text{ ms/B}$$

The data that are transmitted are wrapped inside at TCP-segment which adds an overhead of about 20 bytes. This TCP-segment is wrapped inside an IP-package that adds another 20 bytes of overhead. So for each transmission you have an overhead of 40 bytes.

So for each reduced transmission I can hope to gain 40 B × 0,0003 ms/B = 0,012 ms. That is even less of an incentive to try to improve things.

But my solution would be the same as above: bundle the queries.

And no experiments either – it would be impossible to measure any improvements which translates into: there are no improvements.

5.4 Code design and algorithm

The code in checkout() is very linear with no nested loop and the likes. I see little incentive to try to remove a single assignment here, an increment there. The benefits would be negligible. So I will not do a algorithm analysis.

Then there is the design. Instead of waiting for user interaction you *could* spawn that (user interaction) into a separate thread and continue like this:

- If lock-conflict isn't an issue you could execute the rest of the code until commit(). Then wait for the spawned thread to return a go and commit() (or rollback() in case of a cancel).
- If lock-conflict *is* an issue prepare the bundled sql-queries. Then wait for a go and transmit and commit. If you were “frisky” you *could* issue the inserts into BOOKED and delete for SHOPPING_CART before updating ROOM_DATE and while waiting for the spawned thread to return a go.

All in all you could execute a large part of the code while waiting and reduce the time spend in checkout() significant. If we return to the “back of the envelope” calculation in Chapter 5.1, I came to the conclusion that 163 ms was spend in the section after the user interaction. If I could make most of that section run I parallel to the user interaction I would save “up to” 163 ms.

But this doesn't come without a cost. I need to implement code that makes communication between threads possible which would add some overhead. Java.util.concurrent supply the tools to do this.

I will not run an experiment on this because I haven't implemented this. But if I had to do the experiment I would repeat the experiment with user2.checkout, before and after, each repeated 3 times.

5.5 Results

Of my 4 candidates for bottlenecks, the 2 of them seems plausible, 1 I would reject and 1 maybe:

- (plausible) lock-conflicts.
- (plausible) code design.
- (maybe) roundtrip.
- (rejected) network speed.

5.6 Solution

With that in mind my solution would be:

- Spawn user interaction into a separate thread (a).
- Prepare the sql-strings while waiting for user interaction.
- Bundle the sql-queries into 2 sql-strings:
    - Update to room_date (b).
    - Insert into booked and delete in shopping_cart (c).
- Execute (b) when message returns from (a) and commit().
- Execute (c) and close connection.

See Appendix 9.3 for mockup-code.

5.7 Encountered issues and comment

This part had me on a detour for some time. I had 3 approaches in mind:

1. Change the queries to something more efficient.
2. Find a faster algorithm for the java-code in checkout().
3. Reduce the overhead of the transmission over the net.

For a long time I thought 1) was the obvious – move the work away from the checkout-function and into the highly optimised DBMS. My thinking was to wrap this SELECT-statement:

```
SELECT A.room_type_id, A.single_day_date, A.numtaken + B.numtaken as numtaken, A.numavail - B.numtaken as
numavail from ROOM_DATE as A, SHOPPING_CART as B WHERE  A.room_type_id=B.room_type_id AND A.single_day_date
BETWEEN B.date_start AND B.date_stop AND B.customer_id = {customer_id};
```

inside a single UPDATE statement, so that UPDATE would take the above join and process it row by row. Something like this:

```
UPDATE ROOM_DATE as R
    SET (R.numtaken, R.numavail) = (SELECT Y.numtaken, Y.numavail
        FROM ( [insert join] ) as Y
        WHERE Y.room_type_id=R.room_type_id AND Y.single_day_date=R.single_day_date )
    WHERE EXISTS ( SELECT 1
        FROM ( [insert join] ) as Y
        WHERE Y.room_type_id=R.room_type_id AND Y.single_day_date=R.single_day_date );
```

I haven't tested if this UPDATE-statement actually works because for one the query started to get overly complicated with repeated nested queries, which looks inefficient (experiments might prove me wrong). Secondly UPDATE doesn't support the kind of operations I had in mind.

But more importantly, after looking over the checkout()-code again this part caught my eye:

```
// Second, validate with user
// This wait cannot be removed - it simulates a user interaction
this.wait(4000);
```

I assume that this simulate an interaction where the items in the shopping cart are presented to the user to accept or cancel. This means that the items needs to be retrieved from SHOPPING_CART which in turn cancels the purpose of moving the work to DB2 with “smarter” queries.

6 Moving shopping_cart

The manager of ReserveWithUs suggest moving the shopping cart from the database to the application server. I am to give the pros and cons of such a move and give my recommendation in the end.

By “shopping cart” I assume it refers to the database part of the shopping cart functionality, ie. storing the shopping cart items.

I also assume that the functionality of the shopping cart have to remain unchanged. The application server have to some extent the functionality needed in the ShoppingCartItem-class. What needs to be added is a way to keep all ShoppingCartItem-objects “alive” until they are explicitly deleted *and* a way to search for ShoppingCartItem-objects. Ways to do this could be by implementing linked-list functionality into the ShoppingCartItem-class (one ShoppingCartItem-object links to the next and so forth). Or something similar (hashmaps, combination etc.).

6.1 Pros of moving the shopping cart

(1) By moving the shopping cart away from the database you save all the roundtrips between database-server and application-server when adding, removing and looking up shopping cart items. In Chapter 5.2 I found that roundtrip had a limited impact on the performance on checkout() due to the close distance between servers and other factor having a larger impact. But dependent on the activity on the shopping cart (the frequency of access) – a small amount can be accumulated to a large amount if it happens frequent enough.

(2) All ShoppingCartItem-objects are stored in RAM unless you implement a method to serialise and store them on secondary storage (which adds an overhead). Manipulating an object in RAM is order of magnitude faster than accessing a remote database-server.

(3) You also avoid having the overhead it is to translate between java-objects and serialised strings that the database-server sends and receives.

(4) Data from the database-server is send when the communication buffer is full or the batch is finished executing. So if we on one hand bundle sql-queries to save cost on roundtrip, the application-server then might have to wait longer before it can start process the results. Having it all on the application-server we avoid that altogether.

6.2 Cons of moving the shopping cart

(1) DB2 is specialised in storing, retrieving and search in a very optimised manner. I doubt that I would ever be able to implement algorithms in java that would measure up with that kind of performance.

(2) All ShoppingCartItem-objects are stored in RAM. In case of an outage or similar, every shopping carts will be lost. ReserverWithUs might decide that this is a risk worth taking – the customer can just start over and shopping cart are not as important as the booked, room_date etc. tables. But generally it is a bad idea, you should have some kind of redundancy – and that means some extra overhead for the application-server in managing that (like storing the ShoppingCartItem-objects on secondary storage).

(3) All ShoppingCartItem-objects are stored in RAM. And they do take up at lot more space compare to a row in the ShoppingCart-table. In addition to storing the data they also implement methods, constructor etc. that takes up valuable RAM. Whether 3 GB is enough is down to the actual size and number of objects, and I have no knowledge of what level of activity there is on the shopping cart. I would have to monitor the activity and calculate the size of an object (once the design is finalised).

(4) All ShoppingCartItem-objects are stored in RAM. As far as I can tell the application-server have no method for dealing with stale/forgotten ShoppingCartItem-objects. Everything is kept until they are explicitly deleted (like in checkout()). With the added size of a ShoppingCartItem-object compared to a row in the database, 3 GB will get maxed out a lot sooner than a 15GB secondary storage.

(5) DB2 offers concurrency checks/locks – the same cannot be said of ShoppingCartItem. If we for example keep ShoppingCartItem as linked list and 2 adjacent ShoppingCartItem gets deleted by 2 different users simultaneously you might get a pointer that points to a wrong object and corrupt the linked list. So you would have to implement some sort of concurrency check and locking mechanism, which adds an overhead to operations.

(6) Clients loose connection or returns at a later time. When reconnecting the application-server needs to retrieve all ShoppingCartItem-objects belonging to that customer. Dependant on what kind of structure (linked list, hashmap etc.) that search can be slow compared to DB2. If reconnection is rare then it might not be an issue, but if it happens often it might.

(7) You will have a larger code-base to maintain.

6.3 Solution

Before starting out with this section my first instinct was it didn't really matter. But now, looking at the pros and cons, in my mind there is no doubt that the best solution is to not move the shopping cart. With a lot of but's.

Many of the pros and cons are very dependant on actual use case. So as the IT person I would have to monitor:

- Size of SHOPPING_CART.
- Frequency of access to SHOPPING_CART.
- Number of stale rows.
- Calculate size of a ShoppingCartItem with added methods.
- The kind of structure will be selected (linked list or something else). Or do some performance test of the various possible solution and see how the stack up against the actual use case.

When all those numbers are available it might be that moving the shopping cart to the application server may be beneficial. But until then I would say the cons outweight the pros.

So my recommendation is to keep things as they are.

7 Conclusion

I have done what I set out to do.

I have found that by using a VM you need to be aware of some of the limitations of the guest-system. Most importantly regarding IO-performance on secondary storage – what performance you are measuring is to a large extend the host file-cache (RAM) rather than actual IOs on secondary storage.

Using the right indexes does give a considerable performance boost over the default indexes on all counts. Except for SHOPPING_CART where I decided to stay with the default. The indexes do come at a cost when inserting rows into the tables, but in most cases it is a small amount that'll happen at a low frequency so the benefits outweight the costs.

There is one exception though – in the case of ROOM_DATE, the cost of inserting is too high – up to 145 hours. So when uploading the monthly deals precaution must be taken to tackle that situation – like using the load-command instead of insert.

The way to handle the downloading, deleting and re-uploading the customer-data is to find a 5 minute window with the least activity and take the database offline, export the customer-data to a DEL-file, drop the CUSTOMER-table and recreate it and then use load to reinsert the customer-data into the database. The other possible solution I found took too long.

As possible bottleneck in checkout() I have identified database locks and code design as likely candidates, with network roundtrip as a possible 3rd (depending on actual use cases). My solutions are 1) to commit sooner and thus releasing the locks sooner and 2) run user-interaction in parallel with 3) preparation of the bundled sql-queries. I have not been able to come up with workable test-cases, so I have relied on an analytic approach alone.

What I have encountered while working with this project is that there is always one more experiment you'd like to do – while experiments answers questions they tend to create new ones as well. So the lesson I take from that is to know when you have enough data to give an answer that is sufficient.

Another point is that some of my solutions and recommendation are highly dependant on actual use-cases, which are unavailable. So I had to make some assumptions and “covering all bases”.

8 Litterature

Sasha, Dennis & Bonnet, Philippe, *“Database Tuning – Principles, Experiments and Troubleshooting Techniques”*, 2003.

IBM, *“Database administration”*, 2013, http://pic.dhe.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=%2Fcom.ibm.db2.luw.admin.cmd.doc%2Fdoc%2Fr0008305.html

9 Appendix A – code

9.1 Scripts used for experimenting with indexes

9.1.1 flush_cache.bash

```
#!/bin/bash
flagfile="flushnow.txt"
endloop=0
sleeptime=2s

while [ $endloop == 0 ];
do
    sleep $sleeptime
    if [ -f $flagfile ];
    then
        sync
        echo 3 > /proc/sys/vm/drop_caches
        echo "remove flagfile"
        rm $flagfile
    fi
done
```

9.1.2 run_experiment1.bash

```
#!/bin/bash
#Initialize experiments
#Drop all tables and reinsert them
db2 connect to tuning
db2 -tf drop_tables.sql
db2 -tf init_tables.sql
cd loaddb
db2 -tf load.sql
cd ..
db2 disconnect tuning

#How many repeats of each experiments
nrexp=3

#Baseline
for exp in "xroom_date" "xhotel1" "xhotel2" "xroom_type" "xroom_date" "xcustomer" #"xshopping_cart"
do
    logfile="logfiles/$exp.log"
    echo $logfile
    echo "************">>$logfile
    echo "* BASELINE *">>$logfile
    echo "************">>$logfile
    bash common.bash $exp $nrexp
done

#Create index
logfile="logfiles/index.log"
db2 connect to tuning
echo "create index"
db2batch -a db2inst1/tuning -d tuning -f create_index.sql >>$logfile
echo "running runstats"
db2 reorg table room_date index xroom_date
db2 runstats on table hotel and indexes all;
db2 runstats on table room_type and indexes all;
db2 runstats on table room_date and indexes all;
db2 runstats on table customer and indexes all;
db2 runstats on table shopping_cart and indexes all;
db2 disconnect tuning
db2stop
db2start
```

```
#Run experiments
for exp in "xroom_date" "xhotel1" "xhotel2" "xroom_type" "xroom_date" "xcustomer" #"xshopping_cart"
do
    logfile="logfiles/$exp.log"
    echo "**************">>$logfile
    echo "* EXPERIMENT *">>$logfile
    echo "**************">>$logfile
    bash common.bash $exp $nrexp
done
```

9.1.3 common.bash

```
#!/bin/bash

logfile="logfiles/$1.log"
queryfile="queries/select_$1.sql"
insertfile="inserts/insert_$1.sql"
deletefile="inserts/delete_$1.sql"
readinsert="queries/select_inserted_$1.sql"
flagfile="flushnow.txt"
sleeptime=2s

nrexp=$2

echo "********************">>$logfile
echo "*  $1 ">>$logfile
echo "********************">>$logfile
echo "-- COLD SELECT QUERY --">>$logfile
for (( i=1; i<=$nrexp; i++))
do
    #read -p  "Flush cache on host and press [Enter]."
    touch $flagfile
    while [ -f $flagfile ];
    do
        sleep $sleeptime
    done
    db2stop
    db2start
    db2batch -a db2inst1/tuning -d tuning -f $queryfile | grep "* Total Time" >>$logfile
done

echo "-- WARM SELECT QUERY --">>$logfile
for (( i=1; i<=$nrexp; i++))
do
    db2batch -a db2inst1/tuning -d tuning -f $queryfile | grep "* Total Time" >>$logfile
done

if [ -f $insertfile ];
then
    echo "-- COLD INSERT --">>$logfile
    for (( i=1; i<=$nrexp; i++))
    do
        touch $flagfile
        while [ -f $flagfile ];
        do
            sleep $sleeptime
        done
        db2stop
        db2start
        db2batch -a db2inst1/tuning -d tuning -f $insertfile | grep "* Total Time" >>$logfile
        db2batch -a db2inst1/tuning -d tuning -f $deletefile
    done
    echo "-- WARM INSERT --">>$logfile
    for (( i=1; i<=$nrexp; i++))
    do
        db2batch -a db2inst1/tuning -d tuning -f $insertfile | grep "* Total Time" >>$logfile
        db2batch -a db2inst1/tuning -d tuning -f $deletefile
    done
    echo "-- COLD READ INSERT --">>$logfile
    db2batch -a db2inst1/tuning -d tuning -f $insertfile
    for (( i=1; i<=$nrexp; i++))
    do
        touch $flagfile
        while [ -f $flagfile ];
        do
            sleep $sleeptime
        done
        db2stop
        db2start
        db2batch -a db2inst1/tuning -d tuning -f $readinsert | grep "* Total Time" >>$logfile
    done
    echo "-- WARM READ INSERT --">>$logfile
    for (( i=1; i<=$nrexp; i++))
    do
        db2batch -a db2inst1/tuning -d tuning -f $readinsert | grep "* Total Time" >>$logfile
    done
    db2batch -a db2inst1/tuning -d tuning -f $deletefile
fi
echo "* CYCLE END *">>$logfile
```

9.1.4 Test index from client side

```
#!/bin/bash

nrexp=3
logfile="logfiles/appserver.log"
flagfile="flushnow.txt"
sleeptime=2s

function testappserver {
    nc 10.0.0.10 4001 < user2.checkout
}

function inittables {
    #Drop all tables and reinsert them
    db2 connect to tuning
    db2 -tf drop_tables.sql
    db2 -tf init_tables.sql
    cd loaddb
    db2 -tf load.sql
    cd ..
    db2 disconnect tuning
}

function createindex {
    db2 connect to tuning
    db2batch -a db2inst1/tuning -d tuning -f create_index.sql
    db2 reorg table room_date index xroom_date
    db2 runstats on table hotel and indexes all;
    db2 runstats on table room_type and indexes all;
    db2 runstats on table room_date and indexes all;
    db2 runstats on table customer and indexes all;
    db2 runstats on table shopping_cart and indexes all;
    db2 disconnect tuning
}

for exp in "Baseline" "Clustered"
do
    echo "*** $exp ***">>$logfile
    for (( i=1; i<=$nrexp; i++))
    do
        inittables
        if [ $exp == "Clustered" ];
        then
            createindex
        fi
        db2stop
        touch $flagfile
        while [ -f $flagfile ];
        do
            sleep $sleeptime
        done
        db2start
        nanotime="$(date +%s%N)"
        testappserver
        nanotime="$(($(date +%s%N)-nanotime))"
        mstime="$((nanotime/1000000))"
        echo "Running Time:$mstime">>$logfile
    done
done
```

9.2 Scripts used to experiment with customer table

9.2.1 run_customer.bash

```
#!/bin/bash

home="/home/db2inst1/"
external="/media/bigdata/"
nrexp=3
delformat="del"
ixfformat="ixf"
logfile="logfiles/customerImportExport.log"
messagefile="/home/db2inst1/msg.log"
sleeptime=5
flagfile="flushnow.txt"

function inittables {
    #Drop all tables and reinsert them
    echo "Init tables"
    db2 connect to tuning
    db2 -tf drop_tables.sql
    db2 -tf init_tables.sql
    cd loaddb
    db2 -tf load.sql
    cd ..
    db2 disconnect tuning
}

function resettable {
    echo "reset CUSTOMER"
    db2 connect to tuning
    db2 "drop table customer"
    db2 "create table CUSTOMER (  customer_id int not null,  username varchar(30 ),  password blob,
first_name varchar(30),  last_name varchar(30),   home_street varchar(100),  home_city varchar(30),
home_zip_code int,  home_state varchar(30),  business_street varchar(100),  business_city
varchar(30),  business_zip_code int,  business_state varchar(30),  home_phone varchar(10),
business_phone varchar(10),  email varchar(20),  language varchar(12),  PRIMARY KEY (customer_id))"
    db2 disconnect tuning
}

function coldstate {
    echo "coldstate"
    db2stop
    touch $flagfile
    while [ -f $flagfile ];
    do
        sleep $sleeptime
    done
    db2start
}

function exportdb {
    echo "exportdb $1 $2"
    filename="$1customer.$2"
    echo $filename
    db2 connect to tuning
    db2 "export to $filename of $2 messages $messagefile select * from customer"
    db2 disconnect tuning
}
```

```
function importdb {
    echo "importdb $1 $2"
    filename="$1customer.$2"
    echo $filename
    db2 connect to tuning
    db2 "import from $filename of $2 method P(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17) commitcount
100000 insert into customer"
    db2 disconnect tuning
}

function loaddb {
    echo "loaddb $1 $2"
    filename="$1customer.$2"
    echo $filename
    db2 connect to tuning
    db2 "load from $filename of $2 method P(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17) insert into
customer"
    db2 disconnect tuning
}

inittables
echo "*** New Cycle ***">$logfile
echo "*** New Cycle ***">$messagefile

for storage in $home $external
do
    for format in $delformat $ixfformat
    do
        for process in "exportdb" "importdb" "loaddb"
        do
            echo "** $storage - $format - $process **">>$logfile
            for (( i=1; i<=$nrexp; i++))
            do
                echo $process
                if [ "$process" == "exportdb" ];
                then
                    coldstate
                    nanotime="$(date +%s%N)"
                    exportdb $storage $format
                    nanotime="$(($(date +%s%N)-nanotime))"
                    secondstime="$((nanotime/1000000000))"
                    echo "$(date +'%D %T') Running Time:$secondstime">>$logfile
                elif [ "$process" == "importdb" ]
                then
                    resettable
                    coldstate
                    nanotime="$(date +%s%N)"
                    importdb $storage $format
                    nanotime="$(($(date +%s%N)-nanotime))"
                    secondstime="$((nanotime/1000000000))"
                    echo "$(date +'%D %T') Running Time:$secondstime">>$logfile
                else
                    resettable
                    coldstate
                    nanotime="$(date +%s%N)"
                    loaddb $storage $format
                    nanotime="$(($(date +%s%N)-nanotime))"
                    secondstime="$((nanotime/1000000000))"
                    echo "$(date +'%D %T') Running Time:$secondstime">>$logfile
                fi
            done
        done
    done
done
```

9.3 Suggested checkout() function

```
public synchronized void checkout() throws SQLException, InterruptedException {
    // First, get all items for this customer_id and create connection
    Connection con = this.session.open(); // might generate an Error - remains unchecked
    con.setAutoCommit(false);
    String sql_stmt = DB2SQLStatements.shopping_cart_getAll(this.customer_id);
    Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
    ResultSet res = stmt.executeQuery(sql_stmt);
    // Need cardinality of result set to initialize array
    res.last();
    int rowcount = res.getRow();
    res.beforeFirst();
    ShoppingCartItem[] resArray = new ShoppingCartItem[rowcount];
    // Materialize result set into array
    int i = 0; // index over rows
    while (res.next()) {
        // retrieve and materialize a row
        resArray[i] = new ShoppingCartItem();
        resArray[i].setCustomer_id(res.getInt("customer_id"));
        resArray[i].setDate_start(res.getDate("date_start"));
        resArray[i].setDate_stop(res.getDate("date_stop"));
        resArray[i].setRoom_type_id(res.getInt("room_type_id"));
        resArray[i].setNumtaken(res.getInt("numtaken"));
        resArray[i].setTotal_price(res.getInt("total_price"));
        i ++;
    }

    // Second, validate with user
    // This wait cannot be removed - it simulates a user interaction
    spawnNewThreadForUserInteraction();

    //Third prepare sql-statements
    String sql_update = "";
    String sql_therest = "";
    for (int j = 0; j < rowcount; j++) {
        sql_update = sql_update + DB2SQLStatements.room_date_update(resArray[j]) + ";";
        System.out.println(j);
        sql_therest = sql_therest + DB2SQLStatements.booked_insert(resArray[j], 2) + ";"; // status
is confirmed
    }
    sql_therest = sql_therest + DB2SQLStatements.shopping_cart_deleteAll(this.customer_id);

    //Fourth wait for userinteraction to complete
    waitForUserInteractionFeedback();

    //Fifth execute sql_update and commit()
    try {
        stmt = con.createStatement();
        stmt.executeUpdate(sql_update);
    }
    catch (SQLException e) {
        e.printStackTrace();
        con.rollback();
    }
    con.commit();

    //Sixth execute the rest and close.
    stmt = con.createStatement();
    stmt.executeUpdate(sql_therest);
    this.session.close(con);
}
```

10 Appendix B – data

10.1 FIO test

Throughput									
IO depth	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**
Host	437	545	681	834	996	1062	1098	1135	1119
Guest	1634	2039	2871	3534	4024	4022	4143	4356	4221
External	459	456	457						477

Table 4: FIO test of random reads in KB/s.

Throughput									
IO depth	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**
Host	21437	40563	43628	43976	43229				
Guest	4351	8433	9501	13590	12698	14863	14819		
External	1484	1585	1904	3579	6238	10247	14594	16399	16261

10.2 Index analysis and tuning

	Default Index	**σ**	**xHOTEL Index**	**σ**
Cold Select	190,43	12,92	33,38	6,81
Warm Select	58,70	13,50	23,02	34,43
Cold Insert	24,97	0,55	54,89	13,04
Warm Insert	13,97	0,60	13,35	0,78
Cold Select Inserted	229,50	17,43	7,39	0,89
Warm Select Inserted	61,60	8,63	2,76	0,12

Table 6: Experiments on HOTEL part 1, time in ms.

	Cold Select	σ	Warm Select	σ
Default Index (SELECT hotel_id)	211	20	66	15
xHotel Index(SELECT hotel_id)	12	1	2	1
Default Index (SELECT *)	190	13	59	14
xHotel Index (SELECT *)	33	7	23	34

Table 7: Experiments on HOTEL part 2, time in ms.

	Default	σ	Clustered	σ
Cold Select	127	17	22	0
Warm Select	15	2	3	0
Cold Insert	78	32	171	48
Warm Insert	16	1	15	3
Cold Select Inserted	249	45	50	0
Warm Select Inserted	11	1	4	0

Table 8: Experiments on ROOM_DATE, time in ms.

	Default	σ	xRoom_type Index	σ
Cold Select	333	20	11	1
Warm Select	68	7	2	0
Cold Insert	53	16	79	12
Warm Insert	16	2	15	5
Cold Select Inserted	323	11	10	0
Warm Select Inserted	65	7	2	1

	Average	σ
Baseline	8315	379
Index Tuned	4870	42

Table 10: Experiments on application server, time in ms.

10.3 Export and import of customer-data

	DEL (local)	σ	IXF (local)	σ	DEL (ext)	σ	IXF (ext)	σ	DEL (del + ExtLog)	σ
Export	122	3	213	2	151	4	255	7	NA	NA
Import	199	5	200	4	199	4	197	4	184	2
Load	19	1	23	1	19	1	23	0	NA	NA

Table 11: Performance of export, import and load. Time in s.