LINKÖPINGS UNIVERSITET

Brian Air

TDDD37 Database Technology - theoretical questions

Simon Jakobsson (simja649) and Gustav Hanstorp (gusha433)

23 december 2020

1 Introduction

This is the hand-in for the project in TDDD37 made by Simon Jakobsosn and Gustav Hanstorp. The first sections will be answering the questions, followed by the EER-diagram and schema, finally this will be followed by the code itself.

2 Question 8

a) We put all values through a hash. This way the credit-card number is encoded, and not even people with access to the db can read it.

b)
i. It's more secure, because hackers would have an easier time finding vulnerabilities in the front-end. It is harder to get access to the servers structure.

ii. You're not affected by the different languages in front- and back-end, which in turn makes easier to change, test and apply new code.

iii.It's faster, instead of sending a lot of data back and forth to handle different statements you only do it once.

3 Question 9

a) In session A, add a new reservation.
Done

b) Is this reservation visible in session B? Why? Why not?
No it is not, this is because of the isolation between different transactions principle

What happens if you try to modify the reservation from A in B? Explain what happens and why this happens and how this relates to the concept of isolation of databases.

ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
We guess the reason behind this is that A has a lock. Our other guess would have been that B would have updated A but A couldn't see it. After everything would have been committed both A and B would have seen it.

This relates to the concept of isolation because a transaction should not interfere with another, thus be invisible to each other.

4 Question 10

a) No we did not get an overbooking. This is the cause of our payment method that has 3 if statements to be able too book, if the reservation exists, if there is a contact and if there enough seats. This will not approve two bookings resulting in a overbooking and delete the other reservation(s).

b)

Yes it is possible, if there are no locks on the payment method and if they were to execute at nearly the exact time, bypassing our if statements there could be an overbooking.
c)
We put a sleep argument in the code, before the critical part, in the addPayment function and we came to a conclusion that a overbooking did occur.
d)
We add the following to the testscript:
LOCK TABLES RESERVED WRITE, RESERVATION WRITE, CONTACT WRITE, CREDENTIALS WRITE, BOOKING WRITE,
YEAR READ, WEEKLYSCHEDULE READ, WEEKDAY READ, ROUTE READ;
CALL addPayment(...)
UNLOCK ALL;

This solves the overbooking issue because we create locks on the relevant tables; write if we need to change the table and read if we are just acquiring data from it. This makes it so that only one session at the time can make changes in the relevant tables for payment, and thus, even if it's a sleep, they won't execute at the same time and no overbooking can occur. In our case it will say that there are not enough seats for one of the reservations and instead delete it.

5 Secondary Index

For a secondary index to be effective it would need to be a case where it uses a binary search, while the normal case would use a linear search.
For instance this could be finding which contact is responsible for a which reservation(s) (secondary index based on contact), or finding which ticket number belongs to which passenger(secondary index based on ticket number), etc.

An example:
Looking at the RESERVATION table we have 4 columns, each an INT on 4 bytes, which means 16 bytes in total for each row. Let's say that (the convenient number of) 192 reservations are made every day over the span of 7 years (a year is 365 days).
This mean that we will have a total of 490560 reservations. Assuming that the block size is 512 we get a blocking factor of 32 (512/16).
The amount of blocks needed to store our files are, assuming that there is no wasted space, 15330 (490560/32).
This mean that if we were to use a linear search we would get 7665 (15330/2) cases on average.
Having a secondary index based on contact, we would get the same amount of records, namely 490560 (as we have a non-ordered non-key dense index). Although we would only have 8 bytes of data in each row of the index, 4 for the contact ID and 4 for the pointer.
This means that we would get a blocking factor of 64 (512/8), and assuming there is no wasted space we can see that 7665 (490560/64) blocks are needed to store our data.
Using a binary search we would get 13 (LOG2(7665), rounded up) cases as a worst case.
As we can see, this is a lot faster and easier than not having a secondary index, especially when the amount of data stored increases.
To implement mentioned design we would simply add "KEY 'search_by_contact' ('CONTACT')" after the primary key when we create the CONTACT table.