

a)

EXEC SQL BEGIN DECLARE SECTION;

int modelNumber;

int pricePC;

int ramPC;

float speedPC;

EXEC SQL END DECLARE SECTION;

void lookupPC(){

EXEC SQL SET TRANSACTION READ ONLY ISOLATION READ COMMITED;

EXEC SQL DECLARE cursorPC CURSOR FOR

SELECT model, price FROM PC

WHERE speed =: speedPD AND ram =: ramPC;

// OPEN sætter os til starten af vores cursorPC

EXEC SQL OPEN cursorPC;

EXEC SQL FETCH cursorPC

INTO :modelNumber, :pricePC;

// lig 0 når ingen fejl er opstået.

while (SQLCODE == 0){

printf("Model Number: %i Price: %i", modelNumber, pricePC);

EXEC SQL FETCH cursorPC INTO :modelNumber, :pricePC;

}

EXEC SQL CLOSE cursorPC;

EXEC SQL COMMIT;

}

lookupPC() is a READ ONLY transaction so READ COMMITED is the optimum isolation level for concurrency and no dirty reads are allowed

read only, so no write or update atomicity problems

if system crush happens, truncated and app is required to rerun on system restart

b)

EXEC SQL BEGIN DECLARE SECTION;

int modelNumber;

EXEC SQL END DECLARE SECTION;

void deleteModel() {

EXEC SQL SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

EXEC SQL DELETE FROM Product

WHERE model =: modelNumber;

EXEC SQL DELETE FROM PC

WHERE model = :modelNoumber;

EXEC SQL COMMIT;

}

if we put SERIALIZABLE, no other transaction is able to read one of the two tuples for given model (either from Product or PC) while this transaction was deleting another tuple (dirty read)

and if that other transaction needs to know if both tuples are present or both of them are non present, it can set it's own isolation lvl to SERIALIZABLE

if only one of the tuples is pressent, EXEC SQL SET TRANSACTION READ WRITE ISOLATION LEVEL UNCOMMITED is safe enough

c) EXEC SQL BEGIN DECLARE SECTION;

int modelNumber;

EXEC SQL END DECLARE SECTION;

void decreasePCPrice() {

EXEC SQL SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

EXEC SQL UPDATE PC

SET price = price - 100

WHERE model =: modelNumber;

EXEC SQL COMMIT;

}

again (as in b)), serializable so dirty reads won't happen

d)

EXEC SQL BEGIN DECLARE SECTION;

char pcMaker[1];

int exists = 0;

int modelNumber;

int pricePC;

int ramPC;

int hddPC;

float speedPC;

EXEC SQL END DECLARE SECTION;

void insertPC() {

EXEC SQL SET TRANSACTION ISOLATION READ COMMITED;

EXEC SQL DECLARE cursor CURSOR FOR

SELECT 1 FROM Product P

WHERE P.model =:modelNumber;

EXEC SQL OPEN cursor;

EXEC SQL FETCH cursor1 INTO :exists;

if (exists){

printf ("ERROR: Model number %d already exists in database", modelNumber);

}

else{

EXEC SQL INSERT INTO Product

VALUES (:pcMaker, :modelNumber, 'pc');

EXEC SQL INSERT INTO OC

VALUES (:modelNumber, :speedPC, :ramPC, :hddPC, :pricePC);

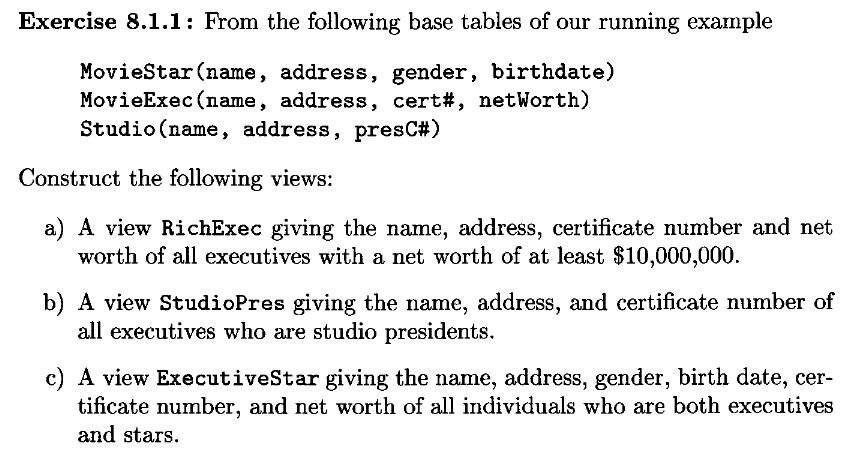
}

EXEC SQL CLOSE cursor;

EXEC SQL COMMIT;

}

read committed is enough as we are only inserting



a)

CREATE VIEW RichExec AS

SELECT \* FROM MovieExec

WHERE netWorth >= 10000000;

b)

CREATE VIEW StudioPres AS

SELECT name, address, certNumber

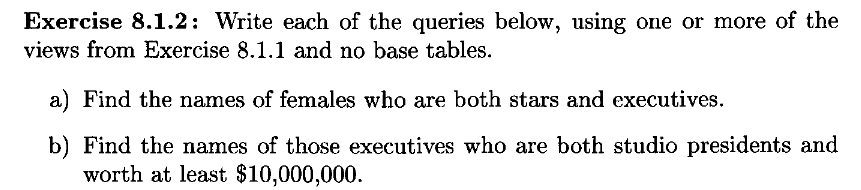
FROM MovieExec INNER JOIN Studio

ON MovieExec.certNumber = Studio.presCertNumber;

c)

CREATE VIEW ExecuteStar AS

SELECT \* FROM MovieExec INNER JOIN MovieStar;



a)

select DISTINCT name

from ExecutiveStar

where gender='f'

b)

select DISTINCT name

from StudioPres

where netWorth >= 10000000

Extra: 8.1.2 C) using two views and one where.

select name

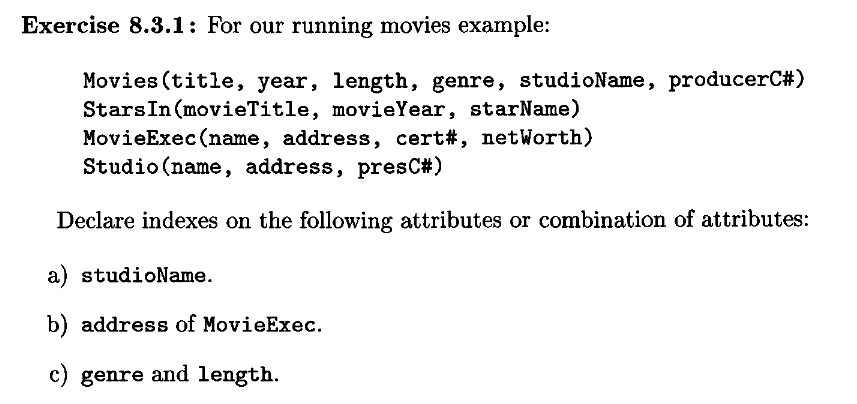
from StudioPres, ExecutiveStar

where networth >= 50000000 AND StudioPres.name = ExecutiveStar.name;

----------------------MODIFYING VIEWS



The views RichEXec and StudioPres are updatable since the attributes are occurrences from only one table, but ExecutiveStar is not updatable since it includes attributes from MovieExec and MovieStar.



a) ioName

CREATE INDEX studioName\_Index ON Movies(studioName);

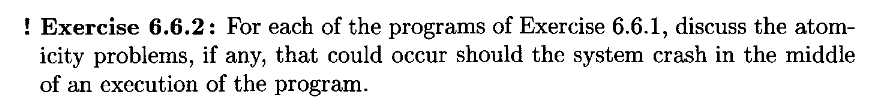
b)

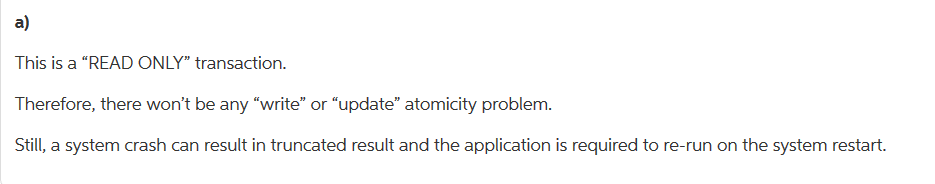
CREATE INDEX movieExecAddress\_Index ON MovieExec(address);

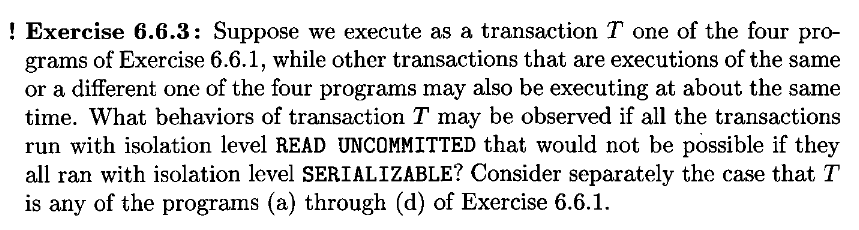
c)

CREATE INDEX genreLength\_Index ON Movies(genre, length);

---------------------------EXTRAS







read uncommitted:

a)

With 2 transactions of a) nothing would occur as a) doesn’t modify data. With b) if the tuple from products were deleted first and an instance of a) ran it could be possible to find a pc which is in the process of being deleted. If c) was made to decrease the cost, then check if the cost was still >0 and then roll back the change if that wasn’t the case we could have cases where a would deliver results of pc’s with negative prices.

b) nor other transactions modify the model numbers so b) wouldn’t read any uncommitted data, except for reading newly put in data by d) when made, but not committed.

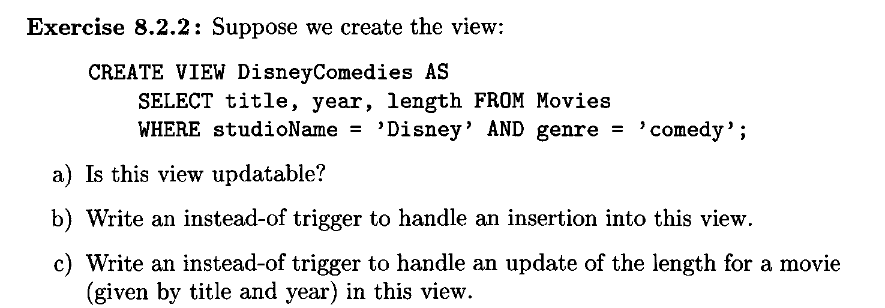
c) same as b).

d)

If d) runs with another instance of itself reading uncommitted data can stop it from adding the same tuple twice. If the first d) doesn’t find a model with the info then it creates one, and the next instance of d) might then read the uncommitted data and then not create a new tuple with the same data.

Serializable:

All the transactions would have to wait for each other. Probably not optimal as a) does not change values and wouldn’t need to lock the table from the other transactions, especially not other a) transactions.



a)

Yes, since the view includes the title and year attributes which are the only NOT NULL attributes in the table (as far as we are told).

b)

CREATE TRIGGER DisneyComedyInsert

INSTEAD OF INSERT ON DisneyComedies

REFERENCING NEW ROW AS NewRow

FOR EACH ROW

INSERT INTO Movies(title, year, length, genre, studioName)

VALUES(NewRow.title, NewRow.year, NewRow.length, 'comedy', 'Disney');

c)

CREATE TRIGGER DisneyComedyUpdate

INSTEAD OF UPDATE ON DisneyComedies

REFERENCING NEW ROW AS NewRow

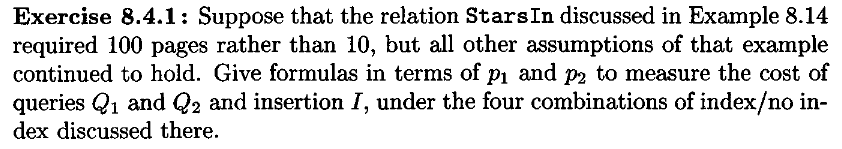
FOR EACH ROW

UPDATE Movies

SET length = NewRow.length

WHERE title = NewRow.title

AND year = NewRow.year;



Example 8.14: Let us consider the relation

StarsIn(movieTitle, movieYear, starName)

suppose that there are three database operations that we somethimes perform on this relation:

query 1(Q1): We look for the title and the year of the movies in which a given star appeared.

That is, we execute the query of the form:

SELECT movieTitle, movieYear

FROM StarsIn

WHERE starName = s;

for some constant s.

query 2(Q2): We look for the stars that appeared in a given movie.

That is, we execute a query of the form:

SELECT starName

FROM StarsIn

WHERE movieTitle = t AND movieYear = y;

for some constants t and y.

query 3(I): We insert a new tuple into StarsIn:

INSERT INTO StarsIn VALUES(t, y, s);

for some constants t, y and s.

Assumptions:

1. StarsIn occupies 10 pages, if we need to examine the entire relation it will cost 10.

2. On average, a star has appeared in 3 movies, and a movie has 3 stars in it.

3. Since the tuples of StarsIn are not likely in a sorted order even if we had an index on starName

or on movieTitle, movieYear it would take 3 disc accesses to retrieve the on average 3 tuples.

If we have no index it will take 10 disk acecsses.

4. One disk access is needed to read a page of the index ecey time we use that index to locate tuples

with a given value for the indexed attributes. If an index page must be modified (in case of an

insertion), then another disk access is needed to write back the modified page.

5. Likewise, in case of an insertion,one disk access is needed to read a page on which the new tuple

will be placed, and another disk access is needed to write back this page. We assume that, even

without an index we can find some page on which an additional tuple will fit, without scanning

the entire file.

cost of operations:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Action | No Index | Star Index | Movie Index | Both Indexes |
|  | 100 | 4 | 100 | 4 |
|  | 100 | 100 | 4 | 4 |
| I | 2 | 4 | 4 | 6 |
| Average |  | 4+96 | 4+96 | 6-2 |

p1 is the fraction of the time Q1 is done, p2 is the fraction of the time Q2 is done.

Exercise 8.4.1:

Suppose that the relation StarsIn discussed in Example 8.14 (above)

required 100 pages instead of 10, but all other assumptions still holds.

give the formulas in terms of p1 and p2 to measure the cost of queries Q1, Q2 and I,

under the four combinations of index/no index.

solution:

Action | No Index Star Index Movie Index Both Indexes

Q1 | 100 4 100 4

Q2 | 100 100 4 4

I | 2 4 4 6

average | 2+98p1+98p2 4+96p2 4+96p1 6-2p1-2p2

The average equation is found by:

usual way to calculate average:

p3 is the fraction of the time I is done,

p3 is given by p3 = (1-p1-p2)

cost of I is x, cost of Q1 is y, cost of Q2 is z

avg = x\*(1-p1-p2) + y\*p1 + z\*p2

= x -x\*p1 -x\*p2 + y\*p1 + z\*p2

= x + (y-x)\*p1 + (z-x)\*p2

No Index:

avg = 2\*(1-p1-p2) + 100\*p1 + 100\*p2

= 2 -2p1 -2p2 + 100\*p1 + 100\*p2

= 2 + 98p1 + 98p2

Star Index:

avg = 4\*(1-p1-p2) + 4\*p1 + 100\*p2

= 4 -4p1 -4p2 + 4\*p1 + 100\*p2

= 4 + 0\*p1 + 96\*p2

Movie Index:

avg = 4\*(1-p1-p2) + 100\*p1 + 4\*p2

= 4 -4p1 -4p2 + 100\*p1 + 4\*p2

= 4 + 96\*p1 + 0\*p2

Both Indexes:

avg = 6\*(1-p1-p2) + 4\*p1 + 4\*p2

= 6 -6p1 -6p2 + 6\*p1 + 6\*p2

= 6 - 2p1 - 2p2

example: only Q1 is ever performed,

No Index - ( 2 + (98 \* 1) + (98 \* 0) ) = 100

Star Index - ( 4 + (96 \* 0) ) = 4

Movie Index - ( 4 + (96 \* 1) ) = 100

Both Indexes - ( 6 - (2 \* 1) - (2 \* 0) ) = 4

example: only Q2 is performed half the time, the other half is insert

No Index - ( 2 + (98 \* 0) + (98 \* 0.5) ) = 51

Star Index - ( 4 + (96 \* 0.5) ) = 52

Movie Index - ( 4 + (96 \* 0) ) = 4

Both Indexes - ( 6 - (2 \* 0) - (2 \* 0.5) ) = 5