Assignment 4: Concurrency and OLAP

Dennis Thinh Tan Nguyen, Nicklas Doolewerdt Mariager Johansen, Pernille Lous,
Thor Valentin Aakjr Olesen, William Diedrichsen Marstrand

6. december 2015

Indhold

1 Problem 1: Serializability and 2PL

1.1 Yes/No Questions

Questions

1. All serial transactions are both conflict serializable and view serializable.
2. For any schedule, if it is view serializable, then it must be conict serializable.
3. Under 2PL protocol, there can be schedules that are not serial.
4. Any transaction produced by 2PL must be conict serializable.
5. Strict 2PL guarantees no deadlock.

Answers

1. Yes
2. No
3. Yes
4. No
5. No

1.2 Serializability

Time	T_1	T_2	T_3
1			$R(A)$
2			$W(A)$
3	$R(A)$		
4	$W(A)$		
5		$R(B)$	
6		$W(B)$	
7	$R(C)$		
8	$W(C)$		
9			$R(C)$
10			$W(C)$
11			$R(B)$
12			$W(B)$

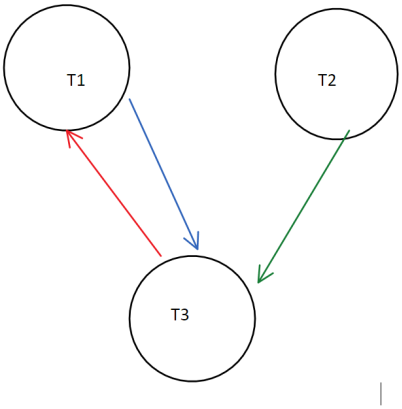
Figur 1: Serializability Schedule, S1

Questions and given schedule

1. Is this schedule serial?
2. Give the dependency graph of this schedule.
3. Is this schedule conflict serializable?
4. If you answer yes to the previous question, provide the equivalent serial schedule. If you answer no, briey explain why.
5. Could this schedule have been produced by 2PL?

Answer 1 No, because transactions are started before running transactions are completed. For instance transaction T_2 starts before T_1 has ended.

Answer 2



Answer 3 No, because a schedule is conflict serializable if and only if the graph is acyclic. However, the graph contains a cycle between T1 and T3.

Answer 4 Not answered since no was started in the answer above.

Answer 5 Yes it could have been produced by 2PL, because it is View Serializable. It is View Serializable because it is View Equivalent to the following schedule *S2*:

S2

Time	T1	T2	T3
1			R(A)
2			W(A)
3	R(A)		
4	W(A)		
5	R(C)		
6	W(C)		
7		R(B)	
8		W(B)	
9			R(C)
10			W(C)
11			R(B)
12			W(B)

Figur 2: Serializability Schedule, S2

and as such it upholds the 2PL Protocol's guarantee of serializability.

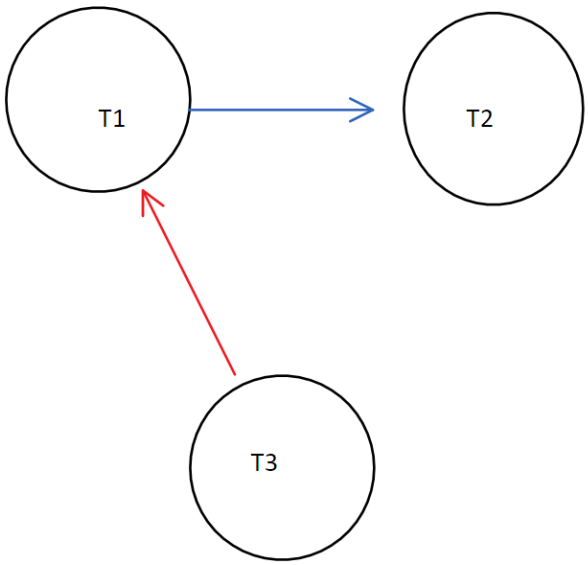
2 Problem 2: Deadlock Detection

2.1 Determine which lock request will be granted, blocked by the lock manager (LM)

Time	T1	T2	T3	LM
1	S(D)			G
2	S(A)			G
3		S(A)		G
4		X(B)		G
5	X(C)			G
6			S(C)	B
7	S(B)			B

Figur 3: Table showing how LM is handling lock requests.

2.2 wait-for graph for the lock requests in the table in section 2.1 showed in Figur: 3



Figur 4:

Figur 5: Wait-for graph of LM

2.3 Determine whether there exists a deadlock in the lock requests showed in the table in section 2.1 (Figur 3) and briefly explain why

There are no deadlock since the wait-for graph (Figur 5) is acyclic.

3 Problem 2: Deadlock prevention

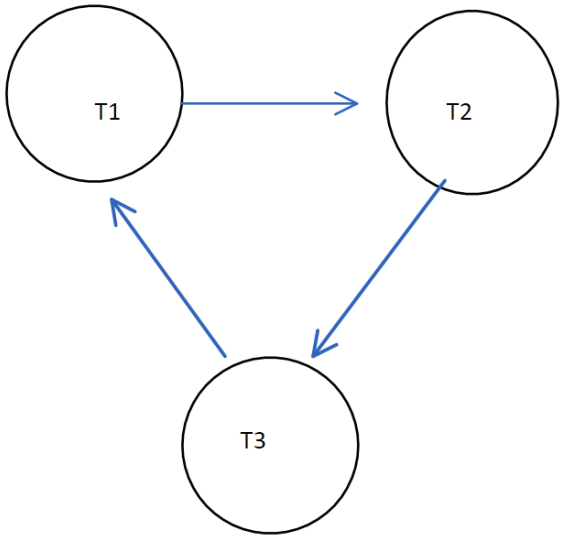
3.1 Determine which lock request will be granted, blocked or aborted by the lock manager 1 (LM1)

Time	T1	T2	T3	LM1	LM2	LM3
1	S(D)			G		
2			X(B)	G		
3	S(A)			G		
4		S(C)		G		
5	X(C)			B		
6		X(B)		B		
7			X(A)	B		

Figur 6: Table showing how LM1 is handling lock requests.

3.2 Give the wait-for graph for the lock request in the table (Figur 6). Give one reason why LM1 Results in a deadlock

Since the graph (Figur 8) contains a cycle in such a way that T1, T2, T3 is waiting for each other, this results in a deadlock



Figur 7:

Figur 8:

3.3 Deadlock prevention with LM2

Please note that we have created a table (Figur 9) that illustrates the task of section 3.3 and section 3.4.

- **LM2 with Wait-Die policy.**
- S(D) on T1 is granted.
- X(B) on T3 is granted
- S(A) on T1 is granted
- S(C) on T2 is granted
- X(C) on T1 is blocked
- X(B) on T2 is blocked
- X(A) on T3 is aborted

3.4 Deadlock prevention with LM3

- **LM2 with Wound-wait policy.**
- S(D) on T1 is granted.
- X(B) on T3 is granted
- S(A) on T1 is granted
- S(C) on T2 is granted
- Abort S(C) on T2
- Abort X(B) on T3
- X(A) on T3 is blocked

Table depicting lock requst handling of LM1, LM2 and LM3 The table (Figur: 9) presentates how LM1, LM2 and LM3 handle locks differently. This table is created from the information based on section 3.1, section 3.3 and section 3.4.

Time	T1	T2	T3	LM1	LM2	LM3
1	S(D)			G	G	G
2			X(B)	G	G	G
3	S(A)			G	G	G
4		S(C)		G	G	G
5	X(C)			B	B	A T2
6		X(B)		B	B	A T3
7			X(A)	B	A	B

Figur 9: This is table is a visualization on LM1, LM2 and LM3.

4 Practical part: Data Cleaning

To clean up the data in the **MovieLens** database we have performed data cleaning and checks on all tables containing foreign keys including **movieGenre**, **rating** and **user**. We have done the following:

1) Checked if genres and movies mutually reference each other correctly.
2) Removed all data lacking proper foreign key references.
    2.1) Deleted Users with no proper zip code references.
    2.2) Deleted Ratings lacking proper User references.
3) Modified user ages to better reflect actual average ages.
4) Checked if all user occupations are referenced properly.

4.1 Example 1: movieGenre

We have performed the following queries to check and clean up the data:
Check: all ids in genre are properly referenced in movieGenre:

```
SELECT id
FROM genre
WHERE NOT id
IN (SELECT genreId FROM movieGenre);
-- Result is empty set
```

Conclusion: we can assume that all movies have a genre and does not require further cleanup.

Check: all genres bound to movies actually reference a genre:

```
SELECT *
FROM movieGenre
WHERE NOT genreId
IN (SELECT id FROM genre);
-- Result is empty set
```

Check: all movies bound to genres actually reference a movie:

```
SELECT *
FROM movieGenre
WHERE NOT movieId
IN (SELECT id FROM movie);
-- Result is empty set
```

Check: all movie records exist and their foreign keys are referenced properly:

```
SELECT *
FROM movieGenre
WHERE movieId LIKE NULL;
-- Result is empty set
```

Check: all genre records exist and their foreign keys are referenced properly:

```
SELECT *
FROM movieGenre
WHERE genreId LIKE NULL;
```

4.2 Example 2: user and zip code

Some tuples in **user** do not reference tuples in **zipcode**. We have concluded that the data becomes cleaner by deleting these **user** records completely. However, one could also just delete the **zipcode** record and give the **User** a default zip code. Arguably, the zip code may be important for demographic research. In this case, the majority of the zip codes should be kept. The following query has been used to delete 63 **user** records lacking proper foreign key values in **zipcode**:

```
DELETE FROM user
WHERE zip NOT IN
(SELECT zip FROM zipcode);
-- Result: deletes 63 user records
```

Comment: note that you can see the actual records by using the "SELECT FROM"expression instead of the "DELETE FROM". Also, if we delete these users, we should also reference check the rating table containing potentially deleted userIds (see example 3).

4.3 Example 3: rating(userId)

The below query checks if all userIds in **ratings** actually reference a userId in **user**. Note that if we delete users from the prior zip query in example 2, the result may change and give bad data.

```
SELECT *
FROM rating
WHERE NOT userId IN
(SELECT id from user);
-- Returns 63 records
```

Problem: the 63 previously deleted users made 10360 reviews, which are no longer referenced according to the following query:

```
SELECT *
FROM rating
WHERE NOT userId IN
(SELECT id from user);
-- Returns 10360 rows in 45 seconds
```

We deleted these rating records that do not properly reference **user**:

```
DELETE
FROM RATING
WHERE userId NOT IN
(SELECT id FROM user);
```

We also checked if all ratings associated with a movie actually reference a movie in the **movie** table:

```
SELECT *
FROM rating
WHERE NOT movieId IN
(SELECT id FROM movie);
```

Indexing We have applied indexing on rating and user to dramatically increase the performance of the above queries that would otherwise take too long time:

```
CREATE INDEX ratingUserIndex
ON rating(userId);

CREATE INDEX UserIdIndex
ON user(id);
```

The indexes made our queries take only seconds as opposed to running in an unbounded time limit.

4.4 Example 4: user(age)

We have modified the age intervals so that they correspond better to the average age of assumed reviewers. By way of example, the age of 18 could be changed to 16 assuming the youngest non edge case reviewers are 14 years old. To encapsulate the average of a given group interval we calculate the difference of the interval and use it as the stored value instead. By way of example, the interval [24-32] has a difference of 9 with an average of 4,5. Thus, the stored value will be 25 + 4 = 29,5 or 30 rounded up.

We found the existing age intervals with the following query:

```
SELECT DISTINCT age
FROM user;
```

Giving the following age intervals:

- Age 1: 1-18
- Age 18: 18-24
- Age 25: 25-34
- Age 35: 35-44
- Age 45: 45-49
- Age 50: 50-55
- Age 56: 56+

We have decided to change the upper limit of the first interval to 16 since it is unlikely that any non edge case users are below the age of 14 (average between 14 and 18). Thus, "Age 1"is renamed to "Age 16". The updated intervals are as following:

- Age 1: 16 implies that 1 is updated to 16.
- Age 18: 21
- Age 25: 30
- Age 35: 40
- Age 45: 47
- Age 50: 53
- Age 56: 61 (assuming oldest non case edge user is 66 years old)

These updates are reflected in the database with the following queries:

```
-- Set age 1 to 16
UPDATE USER
SET age = 16
WHERe age = 1;

-- Update 1085 records from age 18 to 21
UPDATE user
SET age = 21
WHERE age = 18;
```

```
-- Update 2076 records from age 25 to 30
UPDATE user
SET age = 30
WHERE age = 25;

-- Update 1182 records from age 35 to 40
UPDATE user
SET age = 40
WHERE age = 35;

-- Update 545 records from age 45 to 47
UPDATE user
SET age = 47
WHERE age = 45;

-- Update 495 records from age 50 to 53
UPDATE user
SET age = 53
WHERE age = 50;

-- Update 377 records from age 56 to 61
UPDATE user
SET age = 61
WHERE age = 56;
```

Thus, the age intervals now better reflect the average age of reviewers.

4.5 Example 5: occupation

We made a final check on the **occupation** table to se if user occupations are referenced properly in the occupation table:

```
SELECT *
FROM user
WHERE NOT occupation IN
(SELECT id FROM occupation);
-- Result is empty set
```

5 Practical part: Data Enriching

To add more information to **MovieLens**, we have imported the data from the .xlsx file **MedianZIP** containing the median and mean income for a given zip code, as well as the population of the area described by the zip code. Practically, we have done this by exporting the .xlsx file to the .csv file format, which we could then use to import the data into a table in **MovieLens**.

5.1 Importing the .xlsx to a table

1) Export file to .csv:
    a. Removed header (defined when creating table)
    b. Removed dots completely due to European encoding
    c. Removed double semicolon and used new line to separate records
2) Create a table to hold average income for each zip:

```
CREATE TABLE AverageIncomeFromZip(Zip int, Median double, Pop int);
```

3) Import file to table:

```
LOAD DATA LOCAL INFILE 'filename';

CREATE TABLE MedianZip (Zip INT, Median INT, Mean INT, Pop INT);

LOAD DATA LOCAL INFILE '%PATH_TO_CSV%'
INTO TABLE MedianZip
FIELDS TERMINATED BY ';'
LINES TERMINATED BY '\n' (Zip, Median, Mean, Pop);
```

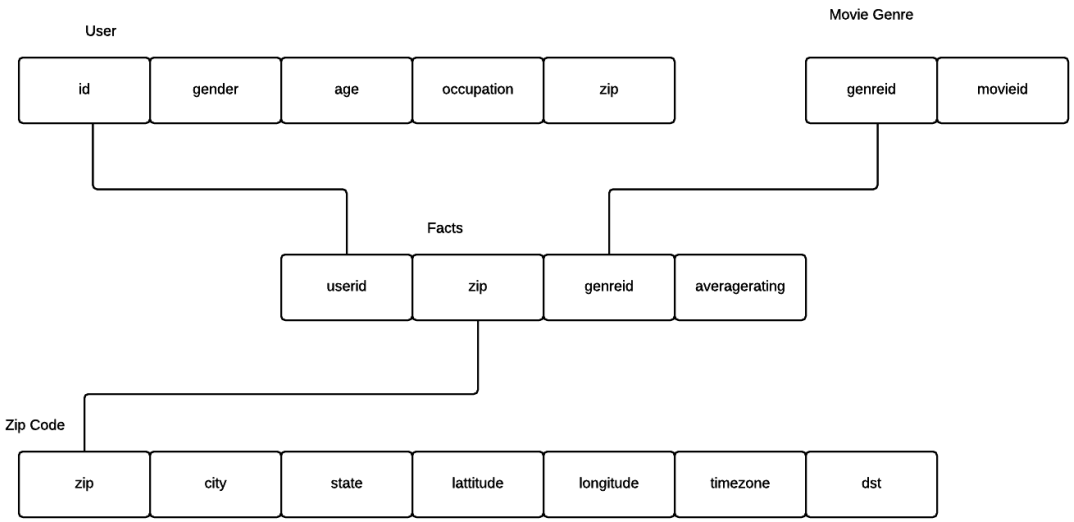
As a result, we have created a table holding information about the average household income as a form of data enrichment. One could also enrich the **zipcode** table by first adding the required attributes such as Median, Mean and Pop and then import the csv file directly to this table.

6 Practical part: Relational OLAP model

6.1 Questions for OLAP

- Which movie genre has highest rating?
- Which movie has most ratings?
- Which user has made most ratings?
- Which movie genre contains most movies?
- How old is the eldest user, who has rated more than five movies?

6.2 Star Schema



Figur 10: Star Schema shows the relations.