Database Administration and Tuning
Mini-Project 2 - Report

Henrique Rocha - 68621
Ludijor Barros - 68626
Fábio Martins - 71073

May 26, 2015

1 Transaction Isolation Levels

Consider a relational database for information related to cheese products, with the following tables:

CHEESE(cheeseID, type, producer, calories, proteins)
REGION(regionID, name, country)
PRODUCTION (productionID, cheeseID, season, amount)
PROVENANCE (productionID, regionID)

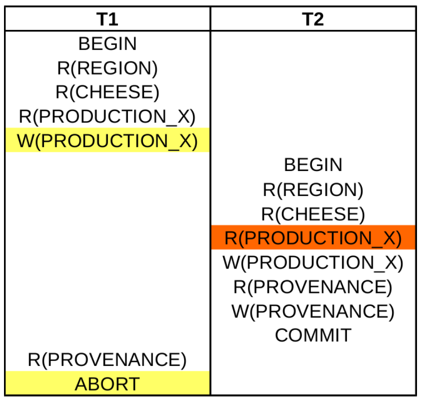
Assume that the following three stored procedures can run concurrently in a given application that is supported by the relational database:

1. **insert_cheese**: creates new records in the CHEESE table, for new cheeses that are produced. This procedure uses only the CHEESE table.
2. **update_production**: inserts new records, or modifies existing PRODUCTION and PROVENANCE records. This procedure writes to the PRODUCTION and PROVENANCE tables, updates the tuples in the PROVENANCE table, and may be reading from the REGION and CHEESE tables.
3. **delete_region**: deletes a given region from the REGION table.

1.1

Give a scenario that leads to a possible dirty read in the concurrent execution of operations from this group of stored procedures, or explain why a dirty read cannot happen in this group of stored procedures.

Consider two transactions both using **update_production** stored procedure over the same PRODUCTION tuples.



T1	T2
BEGIN	
R(REGION)	
R(CHEESE)	
R(PRODUCTION_X)	
W(PRODUCTION_X)	
	BEGIN
	R(REGION)
	R(CHEESE)
	R(PRODUCTION_X)
	W(PRODUCTION_X)
	R(PROVENANCE)
	W(PROVENANCE)
	COMMIT
R(PROVENANCE)	
ABORT	

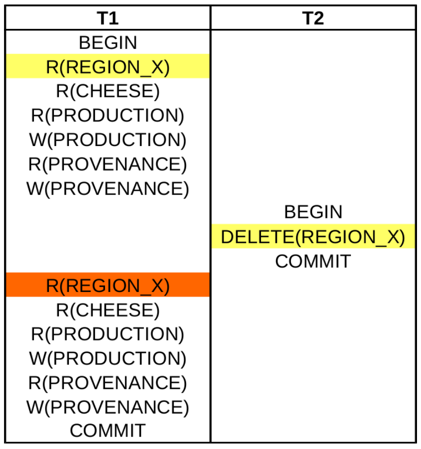
Figure 1: An example of dirty read

T2 commits and T1 aborts after, rolling back its changes, the PRODUCTION tuples that T2 read from T1 are dirty reads because they reflect changes of an aborted transaction (T1).

1.2

Give a scenario that leads to a possible non-repeatable read in the concurrent execution of operations from this group of stored procedures, or explain why a non-repeatable read cannot happen in this group of stored procedures.

Consider that transaction one executes two **update_production** stored procedures over the same REGION tuple, the second transaction deletes this REGION tuple.



T1	T2
BEGIN	
R(REGION_X)	
R(CHEESE)	
R(PRODUCTION)	
W(PRODUCTION)	
R(PROVENANCE)	
W(PROVENANCE)	
	BEGIN
	DELETE(REGION_X)
	COMMIT
R(REGION_X)	
R(CHEESE)	
R(PRODUCTION)	
W(PRODUCTION)	
R(PROVENANCE)	
W(PROVENANCE)	
COMMIT	

Figure 2: An example a non-repeatable read

The second time the REGION tuple is read, it doesn't exists, leading to a non-repeatable read.

1.3

Give an example of a possible overwriting of uncommitted data in the concurrent execution of operations from this group of stored procedures, or explain why a phantom read cannot happen in this group of stored procedures.

Consider two transactions both using **update_production** stored procedure over the same PRODUCTION tuples.

T1	T2
BEGIN	
R(REGION)	
R(CHEESE)	
R(PRODUCTION_X)	
W(PRODUCTION_X)	
	BEGIN
	R(REGION)
	R(CHEESE)
	R(PRODUCTION_X)
	W(PRODUCTION_X)
	R(PROVENANCE)
	W(PROVENANCE)
	COMMIT
R(PROVENANCE)	
W(PROVENANCE)	
COMMIT	

Figure 3: An example overwriting of uncommitted data

T2 commits and T1 PRODUCTION tuple value is lost.

1.4

Indicate what transaction isolation level would you use for executing each of the three procedures above, and why? For each procedure you should use the least restricted transaction isolation level that ensures correctness.

1. **insert_cheese**: read uncommited, because one either serializable or not, if two transactions insert the same cheese, one of them must fail with a violation of the primary key constraint.
2. **update_production**: repeatable read, if a read over the same PRODUCTION or PROVENANCE tuples occurs, the second transaction should wait for the first to commit.
3. **delete_region**: read commited, because if an update_production is happening the delete should wait for it to commit (update_production has read a region)

2 Concurrency Control

3 Recovery System

Consider the following simplified representation for the log records that correspond to a given execution, and suppose the ARIES algorithm is followed by the recovery system:

LSN	Type	Transaction	Page
10	Update	T1	P1
20	Update		P2
30	Begin_checkpoint	-	-
40	End_checkpoint	-	-
50	Commit	T1	
60	Update	T3	P3
70	Commit	T2	-
80	Update	T3	P2
90	Update	T3	P5
	CRASH!!!		

Show the contents of the log, the dirty page table, and the active transactions table after (a) the analysis phase, (b) the redo phase, and (c) the undo phase.

3.1 Analysis Phase

LSN	Type	Transaction	Page
50	Commit	T1	
60	Update	T3	P3
70	Commit	T2	-
80	Update	T3	P2
90	Update	T3	P5

Table 1: Log

Transaction ID	Last LSN
T3	90

Table 2: Active Transactions Table

Page ID	Recovery LSN
P3	60
P5	90

Table 3: Dirty Pages Table

3.2 Redo Phase

LSN	Type	Transaction	Page
50	Commit	T1	
60	Update	T3	P3

Table 4: Log

Transaction ID	Last LSN
T3	60

Table 5: Active Transactions Table

Page ID	Recovery LSN
P3	60

Table 6: Dirty Pages Table

3.3 Undo Phase

LSN	Type	Transaction	Page
50	Commit	T1	
60	Update	T3	P3

Table 7: Log

Transaction ID	Last LSN
T3	60

Table 8: Active Transactions Table

Page ID	Recovery LSN
P3	60

Table 9: Dirty Pages Table

4 Schema and Index Tuning

5 Query Tuning

For each of the following queries, identify one possible reason why an optimizer might not find a good execution plan. Rewrite the query so that a good plan is likely to be found. Any available indexes, or known constraints, are listed before each query.

5.1

An index is available on the calories attribute:

```sql
SELECT type
FROM CHEESE
WHERE calories * 100 < 800
```

The optimizer might not find a good execution plan because the index over calories times 100 might not exist, a good solution is to divide 800 by 100.

```sql
SELECT type
FROM CHEESE
WHERE calories < 8
```

It is important that the index is a b+ tree so the search for cheeses with calories lower then 8 is sequential after the first tree traversal.

5.2

An index is available on the calories attribute:

```sql
SELECT type
FROM CHEESE
WHERE calories<90 AND calories>40
```

5.3

A B+ Tree index is available on the calories attribute:

```sql
SELECT type
FROM CHEESE
WHERE calories=90 OR calories=40
```

5.4

No index is available:

```sql
SELECT DISTINCT CheeseId, type
FROM CHEESE
```

5.5

No index is available:

```sql
SELECT AVG (proteins)
FROM CHEESE
GROUP BY producer
HAVING type = "Alverca"
```

6 Database Tuning