Homework #8
CS486
Prof. Charles Winstead
CRN: 40876

Jiacheng Zhao
Jordan Witte
Resheet Schultz
Due: 04-13-2019

Exercise 1

In this section, you are to use two browser processes, phpPgAdmin w dbclass, and the sailors database, to demonstrate transaction inconsistency between two concurrent transactions (read-write, write-read, or write-write) when transaction management isn't used. You will need to set up two different SQL popup windows, so use separate browser processes (like Chrome and Edge) and not multiple tabs within the same instance.

A) *Create a table that you can use to record internal transaction data. You can't create this in the sailor db, so you'll have to create your own scheme for it. (I called my table my.myboat). It should include a column for a label for the transaction, a timestamp for the time of the transaction, and columns to hold data from the boat table (bid, bname, color).*

```
CREATE TABLE public.myboat (label TEXT, timestamp DATE, bid INT, bname TEXT, color TEXT);
```

B) *Create two query sets to use in your experiment. Query Set 1 will read data from the boat table and insert it into your boat transaction table twice, with a pause between the inserts. Query Set 2 should modify some data in the orginal boat table that you read in Query Set 1. (Yes, you can change the color or name for some boat ID). (Note, you can use* `SELECT pg_sleep(10);` *in Query Set 1 to extend the time it takes to execute)*

Query set 1:

```
INSERT INTO public.myboat(label, timestamp, bid, bname, color)
SELECT 'test', CURRENT_TIMESTAMP, bid, bname, color
FROM boats B
WHERE B.bid=101;
SELECT pg_sleep(10);
INSERT INTO public.myboat(label, timestamp, bid, bname, color)
SELECT 'test', CURRENT_TIMESTAMP, bid, bname, color
FROM boats B
WHERE B.bid=102;
```

Query set 2:

```
UPDATE boats
SET color='hot-pink'
WHERE bid=101 OR bid=102;
```

C) *Execute Query Set 1, and while it's running, execute Query Set 2. Show that Query 1 records inconsistent data to boat transaction table while query set 1 executes. Explain what kind of inconsistency it is.*

label	timestamp	bid	bname	color
test	2019-03-13	101	Ilake	blue
test	2019-03-13	102	Interlake	hot-pink

This happened because query 2 changed some data before the second `INSERT INTO` could retrieve that data. This is a concurrency issue because the whole query set wasn't locked at the start but (presumably) only at the individual command level.

D) *Use* `SELECT current_setting('transaction_isolation')` *in the SQL window of Query Set 1 to determine the default transaction setting of phpPgAdmin. What are the implications of this as the default transaction setting?*

The transaction_isolation setting is read committed. This means that locking occurs at the level of each command. The `UPDATE` command is scheduled to happen between the two `INSERT INTO` commands.

E) *In Query Set 1, use a combination of* `BEGIN TRANSACTION; SET TRANSACTION xxx; COMMIT;` *to prevent the inconsistency of the data. Show the new Query Set 1, and the results of rerunning the experiment in* ***C*** *to show that the inconsistent data is eliminated.*

New query set 1:

```
BEGIN TRANSACTION;
INSERT INTO public.myboat(label, timestamp, bid, bname, color)
SELECT 'test', CURRENT_TIMESTAMP, bid, bname, color
FROM boats B
WHERE B.bid=101;
SELECT pg_sleep(10);
INSERT INTO public.myboat(label, timestamp, bid, bname, color)
SELECT 'test', CURRENT_TIMESTAMP, bid, bname, color
FROM boats B
WHERE B.bid=102;
COMMIT;
```

Resulting data:

label	timestamp	bid	bname	color
test	2019-03-13	101	Ilake	hot-pink
test	2019-03-13	102	Interlake	hot-pink

Exercise 2

Consider a concurrency control manager that uses strict two phase locking that schedules three transactions. Each transaction begins with its first read operation, and commits with the Co statement.

```
T1 : R1(A), R1(B), W1(A), W1(B), Co1
T2 : R2(B), W2(B), R2(C), W2(C), Co2
T3 : R3(C), W3(C), R3(A), W3(A), Co3
```

Schedule 1:

```
R2(B), W2(B), R3(C), W3(C), R3(A), W3(A), Co3, R2(C), W2(C), Co2, R1(A), R1(B),
W1(A), W1(B), Co1
```

Schedule 2:

```
R2(B), W2(B), R3(C), W3(C), R1(A), R1(B), W1(A), W1(B), Co1, R2(C), W2(C), Co2,
R3(A), W3(A), Co3
```

A) *Is Schedule 1 conflict-serializable? If yes, indicate a serialization order. Support your answer by building a precedence graph.*

Schedule 1 is not conflict-serializable because the precedence graph is cyclical.

$$\begin{tikzcd}
 & T1 \arrow[dl, leftrightarrow] \arrow[dd, leftrightarrow] \\
T2 \arrow[dr, leftrightarrow] & \\
 & T3
\end{tikzcd}$$

B) *Is Schedule 1 possible under a 2PL protocol? Demonstrate why or why not.*

No. 2PL guarantees serializability. Since Schedule 1 is non-conflict-serializable, it does not qualify as 2PL.

C) *Is Schedule 2 conflict-serializable? If yes, indicate a serialization order. Support your answer by building a precedence graph.*

Schedule 2 is not conflict-serializable because the precedence graph is cyclical.

$$\begin{tikzcd} & T1 \arrow[dl] \\ T3 \arrow[dr] & \\ & T2 \arrow[uu] \end{tikzcd}$$

D) *Is Schedule 2 possible under a 2PL protocol? Demonstrate why or why not.*

No. 2PL guarantees serializability. Since Schedule 2 is non-conflict-serializable, it does not qualify as 2PL.

Exercise 3

Suppose we are using undo/redo recovery, and there is a table: `hero(HName Group Strength)`. *Below are various sequences of actions that begin from the following state. For each one, say whether afterwards the copy of R5 on disk should be undone, redone, left as is, or can't be fixed.*

1. Transaction T1 has updated data record R5 to change the Group of Hero "Beast" from "X-Men" to "Avengers". The record has changed in memory, but not written to disk.
2. A log record L1 has been created, but it hasn't yet been written to disk.
3. T1 has not yet committed.
4. Currently on disk: `R5:<Beast, X-Men, 50>`
5. Currently in memory: `R5:<Beast, Avengers, 50>`,
   `L1:[T1, old:<Beast, X-Men, 50>, new: <Beast, Avengers, 50>]`

A)

1. *L1 written to disk*
2. *T1 commits*
3. *crash*

This results in `R5:<Beast, Avengers, 50>`. It should be left as is because this is the desired state.

B)

1. *L1 written to disk*
2. *R5 written to disk*
3. *crash*

This results in `R5:<Beast, X-Men, 50>`. This should be redone because T1 didn't commit and thus shouldn't have changed the data.

C)

1. *L1 written to disk*
2. *R5 written to disk*
3. *T1 commits*
4. *crash*

This results in `R5:<Beast, X-Men, 50>`. The R5 write needs to be undone because T1 committed and this final state is wrong.

D)

1. *R5 written to disk*
2. *crash*

   This results in `R5:<Beast, X-Men, 50>`. The transaction should be redone because it didn't commit.

E)

1. *L1 written to disk*
2. *R5 commits*
3. *T1 rolls back*

   This results in `R5:<Beast, X-Men, 50>`. Nothing should be done because T1 was supposed to have been undone.