CARNEGIE MELLON UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
15-445/645 – DATABASE SYSTEMS (SPRING 2023)
PROF. CHARLIE GARROD

Homework #5 (by Chi Zhang) – Solutions

Due: **Friday Apr 21, 2023 @ 11:59pm**

IMPORTANT:

- **Upload this PDF** with your answers to **Gradescope by 11:59pm on Friday Apr 21, 2023**.
- **Plagiarism**: Homework may be discussed with other students, but all homework is to be completed **individually**.
- **You have to use this PDF for all of your answers.**

For your information:

- Graded out of **100** points; **4** questions total

Revision : 2023/04/25 11:29

Question	Points	Score
Isolation Level Done Right	28	
Recovering from a Crash	36	
WAL Gets Replicated	20	
Scaling Distributed Joins	16	
Total:	100	

Question 1: Isolation Level Done Right .... [28 points]

The CMU-DB group decides to build a new database system called RusTub. When designing the concurrency control mechanism, the group decides to implement row-level locking without table-level hierarchy locks. They feel the ANSI SQL isolation levels are too restrictive because all isolation levels need to hold X locks until commit, and decide to implement two isolation levels with the following locking protocols:

- Isolation Level 1 (I1):
  - S locks are acquired on read operations and released on commit / abort. (This is the same as the ANSI SQL repeatable read.)
  - X locks are acquired on write operations and released on commit / abort. (This is the same as the ANSI SQL repeatable read.)
- Isolation Level 2 (I2):
  - S locks are acquired on read operations and immediately released.
  - X locks are acquired on write operations and immediately released.

Read, write, commit and abort in RusTub are atomic. You do not need to consider scan filter pushdown optimizations.

(a) Assume there are two transactions running concurrently: Read-only transaction T1 runs in I1. Write-only transaction T2 runs in I2. You need to consider the case for inserting new data, deletion, and updates. Assume all transactions eventually commit.

i. **[3 points]** Is it possible to have dirty reads in T1?
[✓ **Yes**]
☐ No

Solution: If dirty read means ”read things not committed at the time of execution”: T2 writes and releases the lock, now T1 can read it.
If dirty read means ”read things not eventually committed”, even if we assume all txns commit in this question, there will be dirty reads. For example, T2 writes A=1; T1 reads A=1, writes B=A and commits; T2 writes A=2 and commits. Now T1 sets B based on a non-existent value of A.

ii. **[3 points]** Is it possible to have non-repeatable reads in T1?
☐ Yes
[✓ **No**]

Solution: For all reads, T1 will hold the R lock, and no one can modify it.

iii. **[2 points]** Is it possible to have phantom reads in T1?
[✓ **Yes**]
☐ No

Solution: T1 cannot take lock on non-existent tuples, and therefore will be phantom reads.

Question 1 continues…

(b) Assume there are two transactions running concurrently: Write-only transaction T1 runs in I1. Read-only transaction T2 runs in I2. You need to consider the case for inserting new data, deletion, and updates. Assume all transactions eventually commit.

i. **[3 points]** Is it possible to have dirty reads in T2?
☐ Yes
■ **No**

Solution: T1 will always hold the lock until commit, so T2 will never see dirty reads.

ii. **[3 points]** Is it possible to have non-repeatable reads in T2?
■ **Yes**
☐ No

Solution: T2 reads A, T1 modifies A and commits, now T2 can read the new value.

iii. **[2 points]** Is it possible to have phantom reads in T2?
■ **Yes**
☐ No

Solution: This is the same as in Q1.3.

(c) Assume there are two transactions T1 and T2 running concurrently. You need to consider insertions, deletions, and updates. Transactions may abort.

i. **[3 points]** Assume T1 and T2 both run in I1. Is it possible to have lost updates?
☐ Yes
■ **No**

Solution: Only one txn can hold X lock to a tuple.

ii. **[3 points]** Assume T1 and T2 both run in I1. Is it possible to have cascading aborts?
☐ Yes
■ **No**

Solution: If T1 writes to a tuple, T2 cannot read it, so whether T2 will abort or not is irrelevant to T1.

iii. **[3 points]** Assume T1 runs in I1 and T2 runs in I2. Is it possible to have lost updates?
■ **Yes**
☐ No

Solution: T2 writes to a, T1 writes to a, T1 commits, T2 commits, and the update of T2 is lost.

iv. **[3 points]** Assume T1 runs in I1 and T2 runs in I2. Is it possible to have cascading aborts?
■ **Yes**
☐ No

Question 1 continues…

Solution: T1 can read dirty data from T2, and therefore if T2 aborts, T1 will need to abort.

Question 2: Recovering from a Crash .... [36 points]

RusTub uses ARIES recovery with fuzzy checkpoints. It also has a background thread that may arbitrarily flush a dirty bufferpool page to disk at any time.

For this question, assume objects `A`, `B`, `C` reside in three different pages `A`, `B`, `C`, respectively.

LSN	WAL Record
1	`<T1, BEGIN>`
2	`<T2, BEGIN>`
3	`<T1, UPDATE, prev=1, A, 100→120>`
4	`<T1, COMMIT, prev=3>`
5	`<T2, UPDATE, prev=2, C, 100→120>`
6	`<T1, TXN-END>`
7	`<CHECKPOINT BEGIN>`
8	`<T3 BEGIN>`
9	`<T2, UPDATE, prev=5, A, 120→130>`
10	`<T3, UPDATE, prev=8, B, 100→120>`
11	`<CHECKPOINT END, ATT={2}, DPT={A}>`
12	`<T2, COMMIT, prev=9>`
13	`<T3, UPDATE, prev=10, B, 120→130>`
14	`<CHECKPOINT BEGIN>`
15	`<T3, ABORT, prev=13>`
16	`<T3, CLR, prev=15, B, 130→120, undoNext=10>`
17	`<CHECKPOINT END ATT={?} DPT={A}>`

Figure 1: WAL

(a) Suppose the system crashes and, when it recovers, the WAL contains the first 6 records (up to `<T1, TXN-END>`). Of the object states below, which states are possibly stored on disk before recovery starts? Select all that apply.

i. **[4 points]** ■ **A=100** ■ **A=120**

ii. **[4 points]** ■ **C=100** ■ **C=120**

Solution: Because the background thread may flush a dirty page to disk at any time and because there is no information about the database state from a checkpoint record, both on-disk pages can be in either the old or new state. None of the logged operations require buffer pool pages to be flushed to disk.

(b) Suppose the system crashes, and, when it recovers, the WAL contains the first 11 records (up to `<CHECKPOINT END, ATT={2}, DPT={A}>`). Of the object states below, which states are possibly stored on disk before recovery starts? Select all that apply.

i. **[4 points]** ■ **A=100** ■ **A=120** ■ **A=130**

ii. **[4 points]** ■ **B=100** ■ **B=120** ☐ B=130

iii. **[4 points]** ☐ C=100 ■ **C=120** ☐ C=130

Question 2 continues...

Solution: Because A is logged as dirty in the checkpoint, it must have been dirty at some point during the checkpoint; we don't know, though, whether it has never been flushed or whether either (or both) the writes have been flushed at some point. Page B was not logged as dirty is the checkpoint record, but that just means there was some point during the checkpoint that it was not dirty. It might have been flushed by the end of the checkpoint, but it's possible that it was clean at the start of the checkpoint and is still dirty (with the old value on disk) at the end. Because the checkpoint record for the dirty page table contains only page A, C must have been flushed sometime between its write (LSN 5) and the checkpoint end. This guarantees that the new value (120) is on disk by the end of the checkpoint. Pages on disk will not contain any updates later than the last flushed log record.

(c) **[4 points]** Select all transactions that are guaranteed to be in the ATT in record 17.
☐ T1 ☐ T2 [✓ **T3**] ☐ None of them

Solution: We know that T3 is guaranteed to be in the ATT checkpoint record because it was active at the start of the checkpoint and is still active (although in the middle of being aborted) at the end of the checkpoint. It seems likely that T2 would be in the ATT checkpoint record because we haven't seen a `TXN-END` record for it, but the logging protocol does not require the write and flush of the `TXN-END` record before the transaction is removed from the ATT. It's therefore possible that T2 has ended and been removed from the ATT, but its `TXN-END` has not yet been logged.

(d) **[4 points]** The database restarts and finds all log records up to LSN 17 in the WAL. Which pages should the analysis phase select to be redone? Select all that apply.
[✓ **A**] [✓ **B**] ☐ C ☐ None of them

Solution: We simply redo all pages that are in the reconstructed DPT, which contains B and A after the analysis phase.

(e) **[4 points]** The database restarts and finds all log records up to LSN 17 in the WAL. Select all transactions that should be undone during recovery.
☐ T1 ☐ T2 [✓ **T3**] ☐ None of them

Solution: All transactions not committed should undo.

(f) **[4 points]** How many new CLR records will be appended to the WAL after the database fully recovers?
☐ 0 [✓ **1**] ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6

Solution: Only `T3` will produce one more CLR record.

Homework #5 continues…

Question 3: WAL Gets Replicated .... [20 points]

Consider a DBMS using active-passive, primary-replica replication. All read-write transactions go to the primary node (NODE A), while read-only transactions are routed to the replica (NODE B). You can assume that the DBMS has "instant" fail-over and primary elections. That is, there is no time gap between when the primary goes down and when the replica gets promoted as the new primary. For example, if NODE A goes down at timestamp ① then NODE B will be elected the new primary at ②.

The primary node replicates physical logs to the replicas. The DBMS runs 3 transactions on the table `t` with schema `(key, x)`.

1. Transaction 1: `UPDATE t SET x = x + 10 WHERE key = 'A';`
2. Transaction 2: `UPDATE t SET x = x + 20 WHERE key = 'B';`
3. Transaction 3: `UPDATE t SET x = x + 30 WHERE key = 'C';`

The table originally contains 6 rows:

RID	key	x
1	A	1
2	A	2
3	B	1
4	B	2
5	C	1
6	C	2

Its transaction recovery log contains log records of the following form:

`<txnId, RID, beforeValue, afterValue>`

LSN	**WAL Record**
1	`<T1 BEGIN>`
2	`<T1, 1, (A, 1), (A, 11)>`
3	`<T2 BEGIN>`
4	`<T2, 3, (B, 1), (B, 21)>`
5	`<T1, 2, (A, 2), (A, 12)>`
6	`<T1 COMMIT>`
7	`<T2, 4, (B, 2), (B, 22)>`
8	`<T3 BEGIN>`
9	`<T3, 5, (C, 1), (C, 31)>`
10	`<T3, 6, (C, 2), (C, 32)>`
11	`<T2 ABORT>`

Figure 2: WAL

On the replica node, we run the following transaction T4:

time	operation
①	`BEGIN;`
②	`SELECT * FROM t WHERE key = 'A';`
③	`SELECT * FROM t WHERE key = 'B';`
④	`SELECT * FROM t WHERE key = 'C';`
⑤	`COMMIT;`

(a) Transaction #4 – NODE A

Figure 3: Transaction T4 executing in the DBMS.

(a) Assume that the DBMS is using *asynchronous* replication with *continuous* log streaming (i.e., the primary node sends log records to the replica in the background after the transaction executes them). Assume the database is using MVCC so that transaction always sees a consistent snapshot when the transaction begins.

i. **[5 points]** If T4 is running under `READ COMMITTED`, what are the possible outcomes for T4 for keys A and B? Select all that are possible.

■ **(A, 1), (A, 2), (B, 1), (B, 2)**
□ (A, 1), (A, 2), (B, 21), (B, 22)
■ **(A, 11), (A, 12), (B, 1), (B, 2)**
□ (A, 11), (A, 12), (B, 21), (B, 22)

Solution: T4 either sees T1 commits or not depending on how many log records have been replicated.

If T4 is running under the `READ UNCOMMITTED` isolation level, what are the possible outcomes for keys A and B? Select all that are possible.

i. **[5 points]** For A,

■ **(A, 1), (A, 2)**
□ (A, 1), (A, 12)
■ **(A, 11), (A, 2)**
■ **(A, 11), (A, 12)**

ii. **[5 points]** For B,

■ **(B, 1), (B, 2)**
□ (B, 1), (B, 22)
■ **(B, 21), (B, 2)**
■ **(B, 21), (B, 22)**

Solution: Because the database processes log messages in order, T4 will see any changes made before a given time, but it will not see updates out of order.

(b) **[5 points]** Assume the primary goes down and the replica is elected the new primary. A client was running T1, **committed T1**, and now runs T4 in `READ UNCOMMITTED` isolation level *in the same session*. what are the possible outcomes for its `SELECT` query at for key

A? Select all that are possible.

■ **(A, 1), (A, 2)**
■ **(A, 11), (A, 2)**
□ (A, 1), (A, 12)
■ **(A, 11), (A, 12)**
□ None of the above

Solution: Even if in the same session, when the client connects to the replica, some progress might be lost because we are using asynchronous replication.

Question 4: Scaling Distributed Joins........................[16 points]

The CMUDB group is working on a brand new **shared-storage** distributed database system called BusTub**. They are developing the distributed query engine.

Given the following schema:

```
CREATE TABLE t1(PARTITION KEY v1 int UNIQUE, v2 int);
CREATE TABLE t2(PARTITION KEY v3 int UNIQUE, v4 int);
CREATE TABLE t3(PARTITION KEY v5 int UNIQUE, v6 int);
CREATE TABLE t4(PARTITION KEY v7 int UNIQUE, v8 INT UNIQUE, v9 INT);
```

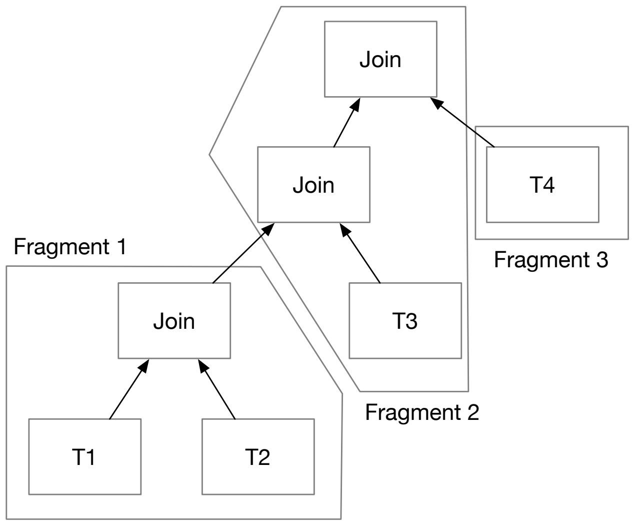
The database system partitions the tables by key range. That is to say, each node in the system manages rows of the table within a non-overlapping range of keys.

Given the following query:

```
SELECT * FROM ((t1 INNER JOIN t2 ON v1 = v3)
INNER JOIN t3 ON v3 = v5)
WHERE v5 IN (SELECT v8 FROM t4);
```

Note that the `IN` clause can be planned as a semi join. For the first two questions, we assume the query optimizer chooses hash join instead of semi join; they produce exactly the same result in this case because key v8 is unique. Assume there is no column-pruning optimization, which means all table scans will need to scan all columns, and all joins will need to produce all columns (except semi join).

The query plan is as follows:



A few assumptions:

1. There are 3 nodes in the system.
2. t1 contains 3000 rows and v1 has all values across 0-2999.

Question 4 continues...

3. t2 contains 3000 rows and v3 has all values across 0-2999.
4. t3 contains 2400 rows, with v5 and v6 values uniformly distributed across 0-2999.
5. t4 contains 1200 rows, with v7 and v8 values uniformly distributed across 0-2999.
6. All joins produce a number of rows equal to *min*{*cardinality of left child, cardinality of right child*}.

(a) **[5 points]** Which are correct schedules for this query? Select all that apply. (Note: A query can be executed only when all required data are available on the same node.)

■ **A)**

```
      ScanT1.v1 ScanT2.v3 ScanT3.v5 ScanT4.v7
Node1    0- 999    0- 999    0-1999         /
Node2 1000-1999 1000-1999 2000-2999         /
Node3 2000-2999 2000-2999         /    0-2999
```

■ **B)**

```
      ScanT1.v1 ScanT2.v3 ScanT3.v5 ScanT4.v7
Node1    0- 999    0- 999 1000-1999    0- 999
Node2 1000-1999 1000-1999 2000-2999 1000-1999
Node3 2000-2999 2000-2999    0- 999 2000-2999
```

☐ C)

```
      ScanT1.v1 ScanT2.v3 ScanT3.v5 ScanT4.v7
Node1    0- 999 1000-1999    0-1999         /
Node2 1000-1999 2000-2999 2000-2999         /
Node3 2000-2999    0- 999         /    0-2999
```

☐ D)

```
      ScanT1.v1 ScanT2.v3 ScanT3.v5 ScanT4.v7
Node1    0- 999 1000-1999 1000-1999    0- 999
Node2 1000-1999 2000-2999 2000-2999 1000-1999
Node3 2000-2999    0- 999    0- 999 2000-2999
```

Solution: Data within a single fragment should be in the same range, otherwise it cannot be joined. Data between different fragments may be in different ranges, which will result in data communication between nodes.

(b) **[6 points]** Using this schedule, how much data is expected to be transferred over the network?

```
      ScanT1.v1 ScanT2.v3 ScanT3.v5 ScanT4.v7
Node1    0- 999    0- 999    0- 999         /
Node2 1000-1999 1000-1999 1000-1999         /
```

Question 4 continues. . .

```
Node3 2000-2999 2000-2999 2000-2999    0-2999
```

Hint: Calculate the answer by summing up all the expected (**rows** * **columns**). You will need to consider the data transferred from the shared storage to the compute nodes. The compute nodes will need to download the data within the scheduled range from the shared storage to perform the table scan.

☐ <21000 ☐ 21000-21999 [✓ **22000-22999**] ☐ 23000-23999 ☐ >=24000

Solution: Table scan: $3000 \times 2 \times 2 + 2400 \times 2 + 1200 \times 3 = 20400$.
Hash join for t1, t2, t3 can all be done locally.
Join t3 and t4: need to shuffle t4 on v8 (not v7). $1200 \times 3 \times \frac{2}{3} = 2400$.

(c) **[5 points]** The BusTub** developers decide to implement the semi-hash-join executor in the system. Using the schedule from question (b), how much data is expected to be transferred over the network? **Exclude table scan cost this time**.

Hint: Semi join works by *sending* the key column from the left child to the executor running on the node that contains the data of the right child, and then *retrieves* a column of existing keys.

☐ <2000 [✓ **2000-2999**] ☐ 3000-3999 ☐ 4000-5999 ☐ 6000-8999 ☐ >=9000

Solution: Fragment 3 sends all the keys to node 3, and node 3 sends back the keys that are in t4.
$(2400 + 1200) \times \frac{2}{3}$.