CSCI 1270 HOMEWORK 5

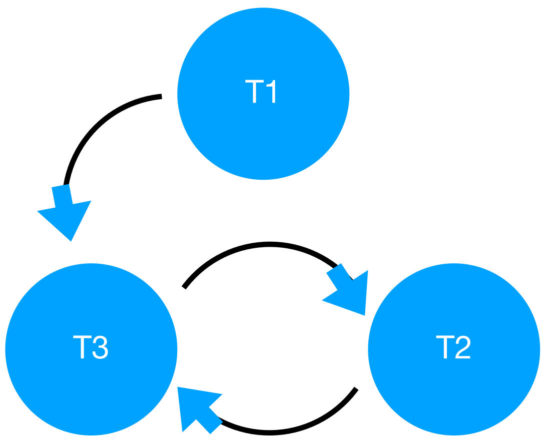
1.
ACID is an acronym to remember the database transaction policies necessary to enforce to ensure a relabel database:
Atomicity - transactions must be executed wholly, or not executed at all
Consistency - transactions must adhere to database constraints.
Isolation - transactions must share information about what they are doing in real time and in order. They cannot simultaneously be making changes on the same old state.
Durability - information must be stored in a manner that successfully persists the data.

2.

T1	T2
read(A)	
	write(A)
	Com.
read(A)	
Com.	

3.
Undo log records represent operations that happened in the past. When recorded they are stored as a stack, adding the newest operation to the end. To traverse back in time we must follow a last in first out manner, to preserve the order of events in time. Redo logs represents transactions that will take the database to state it was once in. The instructions must be followed in order from beginning to end to be consistent with the order they were committed in.

4.1



4.2.a
When cascading aborts are avoided the persisted transaction maintains recoverability in that it read the original data value, which therefore couldn't have been previously written by any other transaction.

The tables below represent the next steps in schedule S:
4.2.b.

	T1	T2	T3
Step 1	commit		
Step 2			commit
Step 3		commit	

4.2.c.

	T1	T2	T3
Step 1	commit		
Step 2		abort	
Step 3			commit

5.1

```
T1:
shared-lock(A);
read(A);
exclusive-lock(B);
unlock(A);
read(B);
if A = 0 then B := B +1;
write(B);
unlock(B);
```

```
T2:
shared-lock(B);
read(B);
exclusive-lock(A);
read(A);
if B = 0 then A := A +1;
write(A);
unlock(A)
unlock(B);
```

5.2

T1	T2
lock-s(A)	
read(A)	
lock-x(B)	
read(B)	
If A = 0 then B:= B + 1	
write(B)	
unlock(A)	
unlock(B)	
	lock-s(B)
	read(B)
	lock-x(A)
	read(A)
	If B = 0 then A:= A + 1
	write(A)
	unlock(A)
	unlock(B)

6.1
Undo-list: T1,T0,T4
Redo-list: T2,T3

6.2
Undo:
Set E to 23
Set A to 41
Set B to 102
Set B to 66
Redo:
Set C to 99
Set D to -40

6.3
A: 41 B: 66 C: 99 D: -40 E: 23

7.1.a
HyPer is a database system that runs in main memory.

7.1.b
HyPer runs OLTP transactions sequentially. Since it is all in memory it can avoid costly lock structures.

7.1.c
OLTP transactions are lagged while OLAP queries are completed, since all operations lock

7.1.d
HyPer aims to process OLTP transactions at state of the art speeds while also processing OLAP queries that operate on up to date data.

7.1.e
HyPer allows hybrid workloads by operating on a consistent virtual memory snapshot. It is created by forking the OLTP transaction and maintained consistent by using the operating systems copy-on-write mechanism.

7.1.f
OLTP transactions operate on little data; usually they update just a single or a few values in a server. OLAP transactions are the opposite, generally querying large amounts of data, to make generalizing analyses. HyPer is trying to create a DBMS framework that simultaneously supports both, with high-performance.

7.2.a
HyPer translates high level scripting code into the respective low level OLTP request code. HyPer then executes concurrent OLTP transactions serially on a single thread. This way there is no need for locking structures since other transactions don't happen at the same time.

7.2.b
HyPer's success with serial single thread operations relies on being in-memory. It works because OLTP transactions take just 10 microseconds in memory!

7.3.a
HyPer efficiently creates a snapshot that represents the current state of the database. It does this often and on the fly at the beginning of OLAP sessions to stay up to date. Concurrent OLTP changes are stored as deltas, to maintain the old state, dubbed current for the OLAP session and when the OLAP session ends, the fork resolves these deltas with the old snapshot to create an up to date session for the next fork.

7.3.b
Fork functionally makes a copy of the memory. It does this by telling the OS to retain the state of the virtual memory and proceed by making changes in deltas which can be added to get the new state. Calling fork at the beginning of a session ensures that the OLAP transaction is operating on the freshest data when it starts.

7.3.c
Locks are avoided because OLAP transactions are read-only so their snapshots are immutable and so there is no need to avoid read-write or write-write conflicts.

7.3.d
HyPer can run different OLAP queries on different threads, each with a snapshot representing the state of the DB when the respective OLAP session started.

7.4.a
HyPer keeps track of the changes made since the snapshot. These changes are resolved when the current state is queried by OLTP transactions. Otherwise, the OLAP session operates on the unadulterated snapshot, consistent with its execution time.

7.4.b
OLTP transactions can include write operations. To avoid write-write and read-write conflicts while maintaining a lockless architecture, HyPer halts all other thread processes while doing write operations. Otherwise, read-only OLTP operations can be executed concurrently!

(a) What isolation level (e.g. Serializable, Repeatable Read, Snapshot Isolation, etc..) and what concurrency control protocol does HyPer use to achieve this (e.g. 2PL, Strict 2PL, OCC, etc..)?
(b) How does HyPer make a transaction archive? This is analogous to what concept in recovery that we discussed in class?
(c) How does HyPer handle partition-crossing transactions? Why is this extremely expensive?
(d) How does HyPer use the transaction archive and a redo-log to ensure durability?
(e) Why does Hyper not need to ”undo” failed transactions after a system crash (so-called R3 recov- ery) and when is the undo-log necessary in HyPer?
(f) How does HyPer perform recovery after a system failure?

7.5.a
HyPer achieves virtual memory snapshot isolation using process forks!

7.5.b
HyPer can leverage VM snapshots as records of changes. Since they are always stored after the fact, they are analogous to a deferred database log!

7.5.c
HyPer engages exclusive mode, stopping all other processes to ensure no conflicts. This is expensive because it simultaneously halts all other processes!

7.5.d
The transaction archive is used to restore the DB to a transaction consistent state. The redo log is then used to restore the DB to an action consistent state. This way, all effects of committed transactions can be restored after a failure and thus durability is ensured.

7.5.e
Hyper stores transaction consistent snapshots in the database archive. This makes undo logging unnecessary unless attempting to turn a snapshot which might be action consistent to transaction consistent.

7.5.f
The recovery process starts with the youngest transaction consistent process in the database archive, then the redo log is executed in chronological order.