Transactions

Chapter 17 of Sail (minus 17.3 and 17.8-10)

What is a Transaction? (17.1)

Transaction - A unit of program execution that accesses and possibly updates various data items.

But what makes transactions so special?

Problems we Have to Deal with

- Failures and crashes of various kinds hardware, software, etc
- Concurrent execution of multiple transactions on the same data

```
A problem has been detected and system has been shut down to prevent damage
to your computer.

IRQL_NOT_LESS_OR_EQUAL

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, folow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any system updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup options, and then
select Safe Mode.

Technical information:

*** STOP: 0x0000000A (0x000000000000004A, 0x0000000000000002, 0x0000000000000001,
0xFFFFF80002B37ABF)


Collecting data for crash dump ...
Initializing disk for crash dump ...
Beginning dump of physical memory.
Dumping physical memory to disk: 95
```

Transaction Required Properties (ACID)

Atomicity - Transactions are indivisible and the whole transaction is required to occur or not occur. I.e. you can't have only part of the transaction occur

Consistency - Execution of a transaction in isolation (i.e. with no other transaction executing concurrently) preserves the consistency of the database.

Isolation - Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T1 and T2 it appears to T1 that either T2 finished execution before T1 started or T2 started execution after T1 finished

Durability - After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Venmo Example

Is there a place where if this failed the Database would become inconsistent?

Venmo Example: $T_{D \to E}$:= Transaction from Drew to Evan

```
T_{D→E}: read(D);          //get Drew's Account Balance
         D:= D-100;
         write(D);         //update Balance
         read(E);
         E := E+100;
         write(E);
```

Venmo Example Failure

Venmo Example: $T_{D \to E}$:= Transaction from Drew to Evan

```
T_{D->E}: read(D);      // get Drew's Account Balance
          D := D-100;
          write(D);     // update Balance
          Fail
          read(E);
          E := E+100;
          write(E);
```

How To Prevent This?

We need to include measures to keep the ACID Properties

Consistency: It is required that the sum of D + E is the same before and after the Transaction or else there was an error.

Atomicity: Need to keep track of the Transaction steps before we run them so in the case of a failure like this we know what we have or haven't done. I.e insure updates of partial transactions are not reflected in the Database

Durability: The updates made by the Transaction are stored on the disk or information on how to do the updates is. So even if there are failures after the Database remains consistent.

Isolation: Need to create ability for correct concurrent transactions.(slides 14-23)

Durability

- After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

- We talked about this before but having a log file that keeps track of the events that a transaction did and what it did them on

- This makes it so those transactions can be replicated in the case of a hardware or software failure

Transaction States

Active - the initial state and the during state.

Partially Committed - After the final statement has been executed .

Failed - When the transaction fails and cannot finish.

Aborted - After the transaction has been rolled back and restored to its state before the beginning of the transaction.

Committed - After successful completion(data is stored on disk) and a new consistent database is created.

Graph of Transaction Lifecycle

- From the aborted state there are 2 options:
  - Restart the Transaction in the case of a hardware error or other non logic error
  - Kill the transaction if there was an underlying error in the transaction

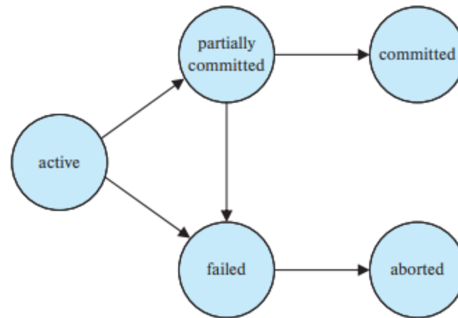


Figure 17.1 State diagram of a transaction.

Ensuring Atomicity in Transaction Errors

Abort - Transaction failures and errors will occur meaning there will be times when the transaction must be aborted

Roll back - Once the changes caused by an error full transaction have to be undone

Log - A log is usually maintained in order to keep track of transaction history and changes. Making for easy roll backs.

Observable External Writes

- Ex) Writes to a user’s screen, or sending an email
- These writes are tricky as once they are seen external to the database system they cannot be erased
- Therefore it gets tricky to deal with and most systems decide to only allow these types of writes after the transaction has already been committed

Solution to Observable External Writes

- Create systems that only do external writes after the transaction has been committed and deal with these problems case by case
- Example) If an ATM fails after the transaction is committed in the database but before the ATM prints the cash(external action).
- Solution: When the ATM starts back up we don't want it to dispense the cash because the person could have left. Instead we have it do a compensating transaction to make up for the error.

Isolation

- This is a problem because what is one way that you think we could ensure that the transactions are totally isolated? i.e. how could we make it so only one transaction happens at a time
- We could make them serial but as we will discuss later concurrency is too important for performance and therefore serial transactions is not an option

What Concurrency Allows

Overall: Greatly Improved performance

Specifically:
-Improved **throughput** and **resource utilization**: Running transactions in parallel allow CPU and disk resources to be used to their full potential and not wait around for work. This allows more transactions to be executed quicker

Reduced waiting time: Running serially could create blocking scenarios where short transactions are bottle necked by Longer ones(Similar to how synchronous javascript code can slow everything down)

Atomicity With Concurrency

- With this constraint if a transaction fails the effect of the transaction must be undone
- In concurrent transactions this makes it so that any other transaction dependent on the one that failed must also be rolled back
- Dependency = T2 reads data written by T1 (T2 is dependent on T1)

Schedules

- Transaction execution sequences that detail the order in which instructions are executed
- These can be used to our advantage to keep the benefits of concurrency while insuring that our database is isolated and uses the correct data
- The rules are:
  - A schedule must contain all rules from the transactions
  - keep the order of the steps in individual transactions the same

Serial Schedule

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

Figure 17.2 Schedule 1—a serial schedule in which T_1 is followed by T_2.

Concurrent Transactions

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

Figure 17.4 Schedule 3—a concurrent schedule equivalent to schedule 1.

Nonrecoverable Schedules

T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

Figure 17.14 Schedule 9, a nonrecoverable schedule.

Cascading Schedules

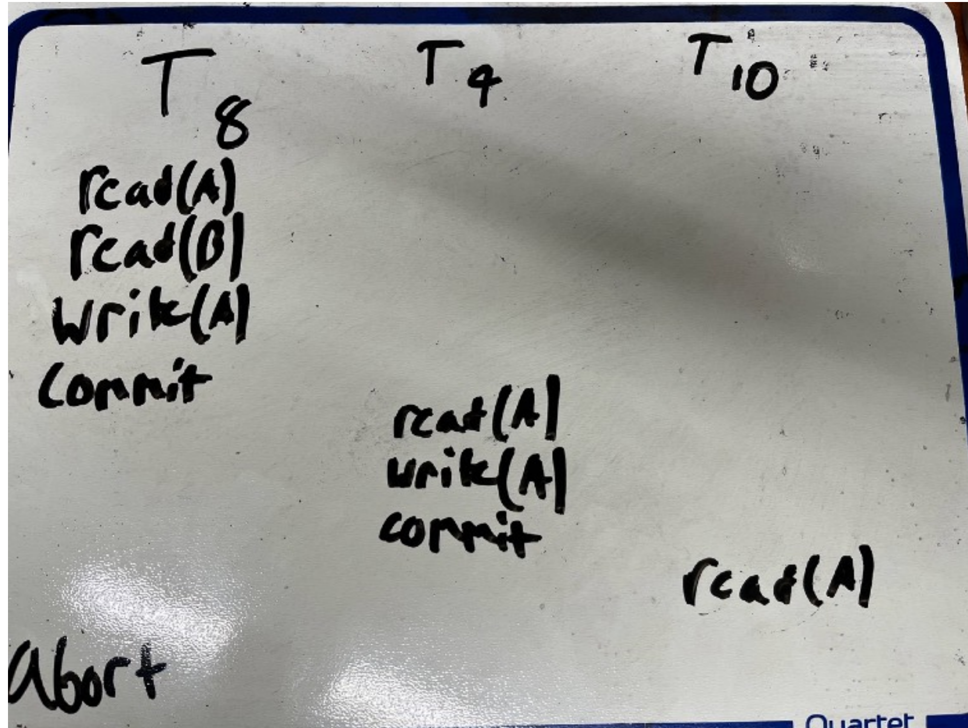
T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

Figure 17.15 Schedule 10.

Cascadeless Schedules

- Cascading rollbacks is terrible and in a complex series of transactions adds a ton of extra work
- In order to create cascadeless schedules we just need to ensure that for any pair of Transactions T1 and T2 where T2 is dependent on T1, the commit operation for T1 happens before the read operation for T2

Cascadeless Schedule



Serializability

- A schedule is said to be serializable if it can be turned into its serial schedule and they both return the same results(Figure 17.4 from the last slide)
- In these slides we will be focusing on **Conflict serializability**
- We will also only be focusing on **Read** and **Write** operations to make it simpler

Conflict

- We say that Transactions I and J conflict if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Figure 17.6 Schedule 3—showing only the read and write instructions.

Using Conflicts To Find New Schedules

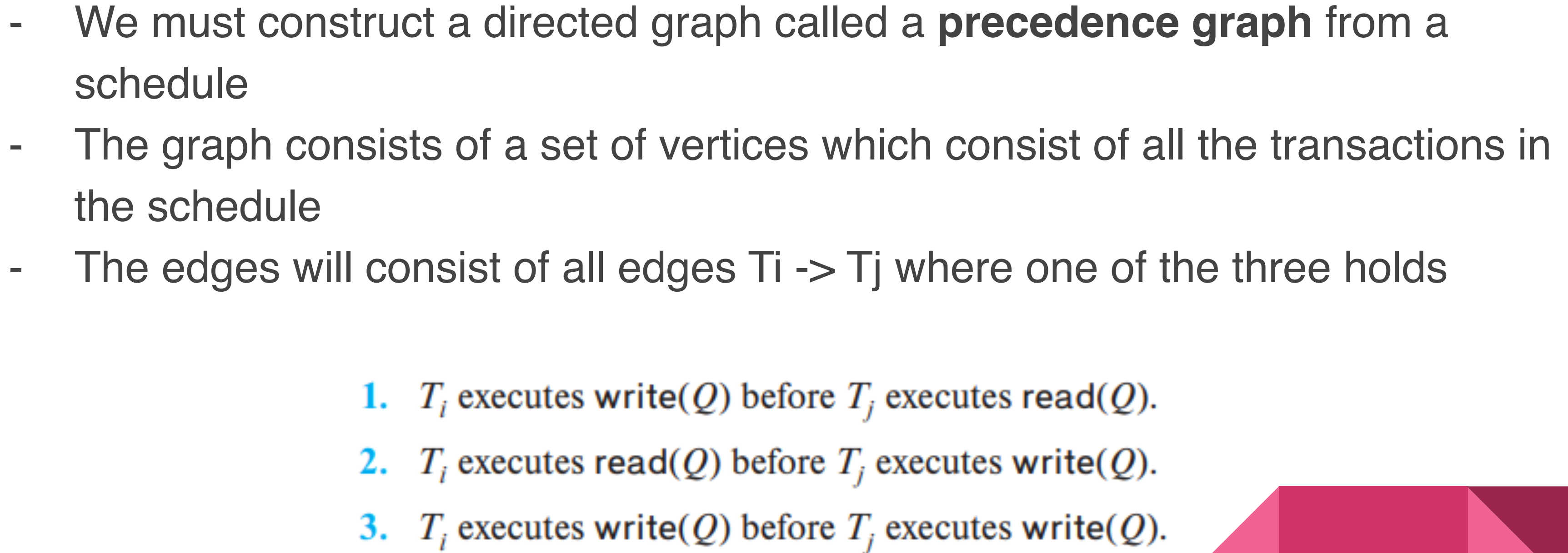
- Now we know that if we let I and J be non conflicting consecutive instructions from different transactions we can flip their order
- Therefore if we can transform a schedule S to a schedule S' through a series of swaps of non conflicting instructions we say S and S' are **Conflict Equivalent**
- If a schedule is conflict equivalent with a serial schedule then we say that the schedule is **Conflict Serializable**

Example Conflict Serializable

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

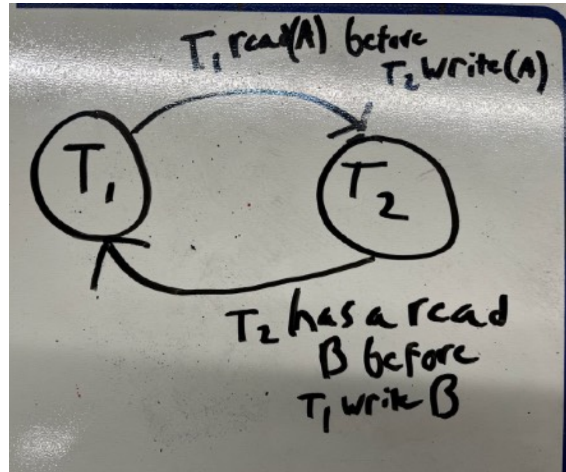
How to Determine Conflict Serializability



Precedence Graph Continued

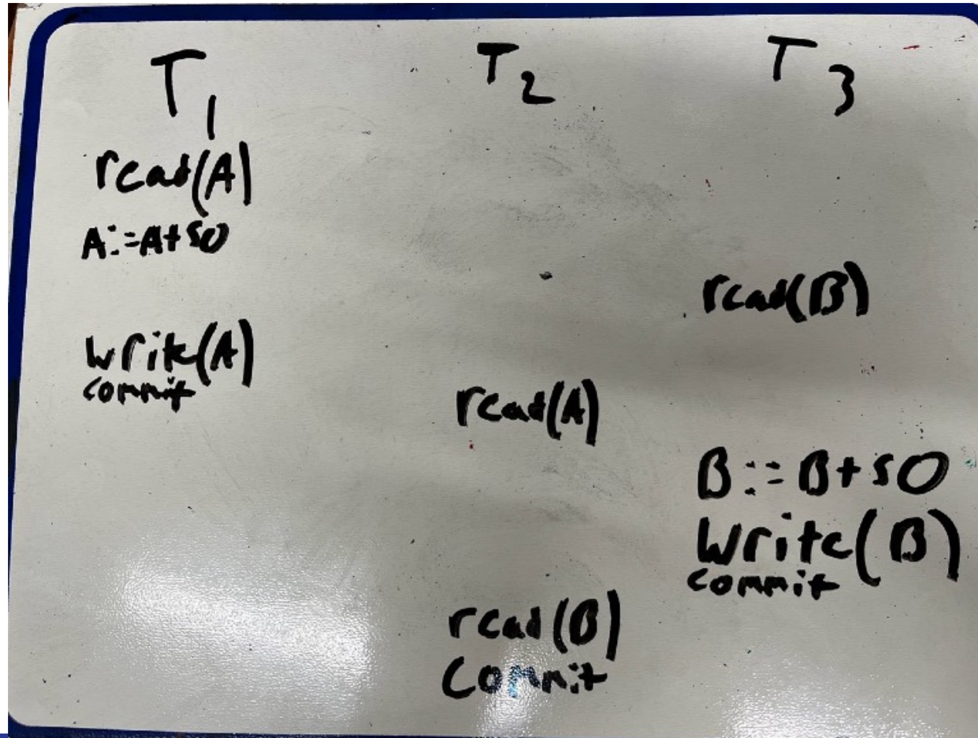
- After building this graph if there are no cycles than the schedule S is **conflict serializable**
- Though if there is a cycle than the schedule S is not conflict serializable

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit



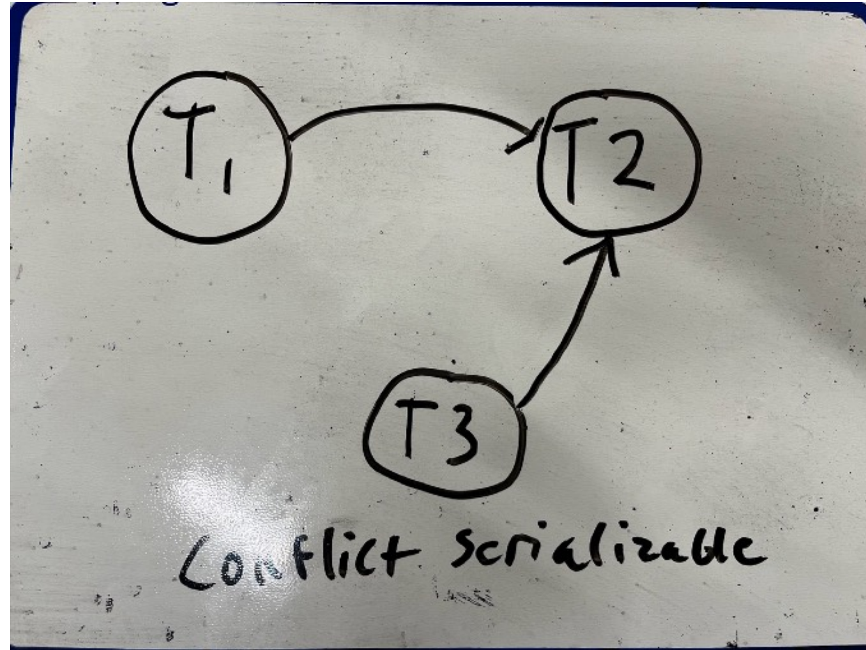
Practice

Make a directed graph from these 3 Transactions



Solution:

From this if we wanted to we could perform topological sort to get a serial Transaction.



Sources

https://www.geeksforgeeks.org/precedence-graph-for-testing-conflict-serializability-in-dbms/

https://db-book.com/slides-dir/index.html