Transactions

Why Transactions?

- Database systems are normally being accessed by many users or processes at the same time.
  - Both queries and modifications.
- Unlike operating systems, which support interaction of processes, a DMBS needs to keep processes from troublesome interactions.

Transactions

- A single "unit of work" in a DBMS.
- Can comprise more than one SQL command, but each individual command does not stand on its own.

Statement of Problem

- How do we allow concurrent running of independent transactions while preserving database integrity?
- Additionally, we want
  - good response time and minimal waiting.
  - correctness and fairness.

Another example: "lost update" problem

time

T1	T2
Read(N)	
	Read(N)
N=N-1	
	N= N-1
Write(N)	
	Write(N)

Concurrency

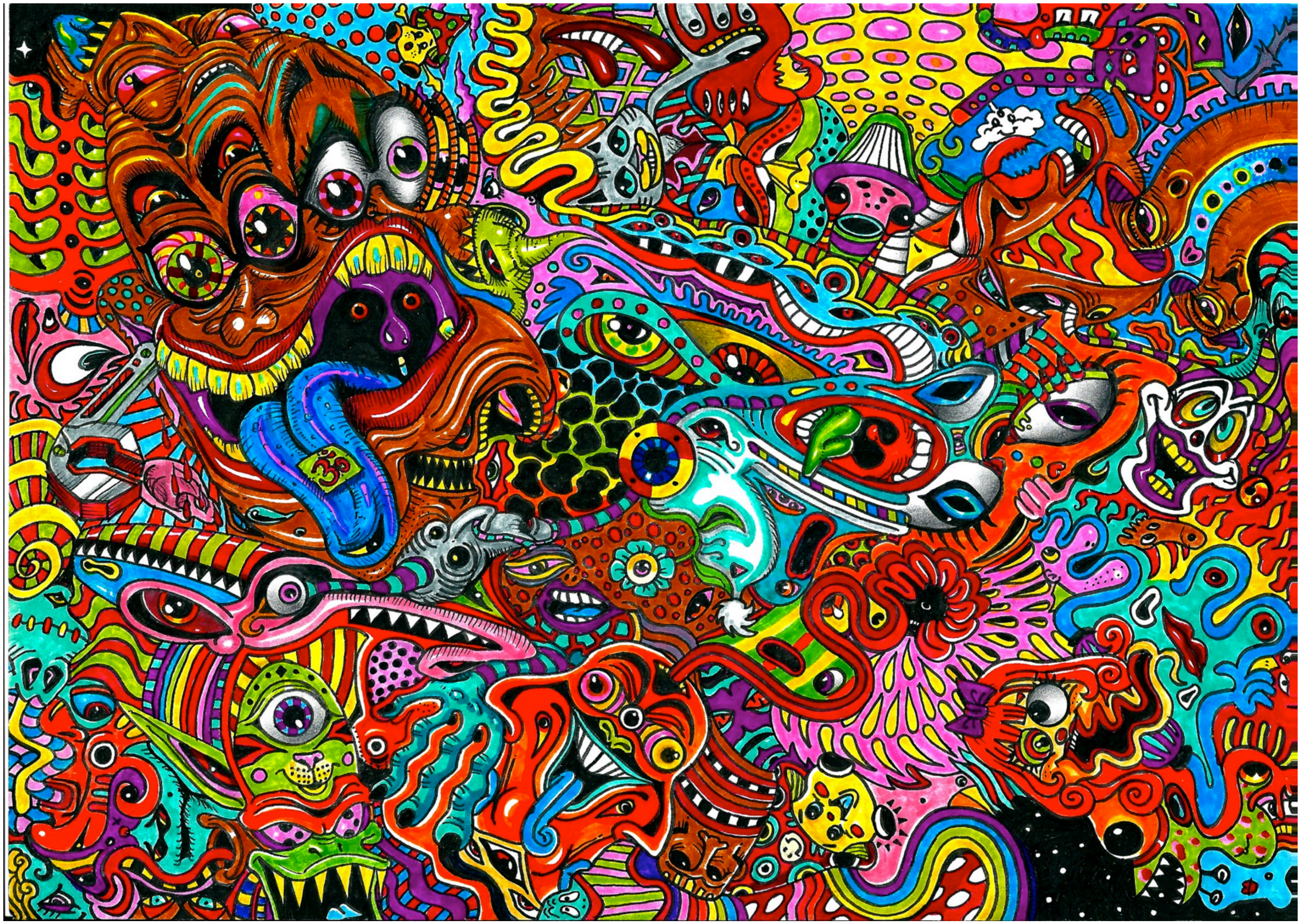
- Arbitrary interleaving can lead to
  - Temporary inconsistency (unavoidable)
  - "Permanent" inconsistency (bad!)

Example: Bad Interaction

- You and friend each take $100 from different ATMs at about the same time.
  - The DBMS had better make sure one account deduction doesn't get lost.
- Compare: An OS allows two people to edit a document at the same time. If both write, one's changes get lost.

Remember ACID?

Remember ACID?



ACID Transactions

- *We want transactions to be*:
  - ***Atomic***: Whole transaction or none is done.
  - ***Consistent***: Database constraints preserved.
  - ***Isolated***: It appears to the user as if only one transaction executes at a time.
  - ***Durable***: Effects of a transaction survive a crash.

SQL Transactions

- `BEGIN TRANSACTION`
- `// do SQL here`
- `either COMMIT or ROLLBACK`

COMMIT

- The SQL statement COMMIT causes a transaction to complete.
  - Any database modifications are now permanent in the database.

ROLLBACK

- The SQL statement ROLLBACK also causes the transaction to end, but by ***aborting***.
  - No effects on the database.
- Failures like division by 0 or a constraint violation can also cause rollback, even if the programmer does not request it.

Isolation Levels

- SQL defines four ***isolation levels***: choices about what interactions are allowed by transactions that execute at about the same time.
- Only one level (serializable) gives ACID transactions.
- Each DBMS implements transactions in its own way.
- Not all DBMS implement all four isolation levels.

Let's get abstract

- database - a fixed set of named data objects (A, B, C, …)
- transaction - a sequence of read and write operations (read(A), write(B), …)
  - DBMS's abstract view of a user program

ACID Transactions

- *ACID transactions* are:
  - *Atomic* : Whole transaction or none is done.
  - *Consistent* : Database constraints preserved.
  - *Isolated* : It appears to the user as if only one process executes at a time.
  - *Durable* : Effects of a process survive a crash.

A

Atomicity of Transactions

- Two possible outcomes of executing a transaction:
  - Xact might *commit* after completing all its actions
  - or it could *abort* (or be aborted by the DBMS) after executing some actions.
- DBMS guarantees that Xacts are *atomic.*
  - From user's point of view: Xact always either executes all its actions, or executes no actions at all.

A

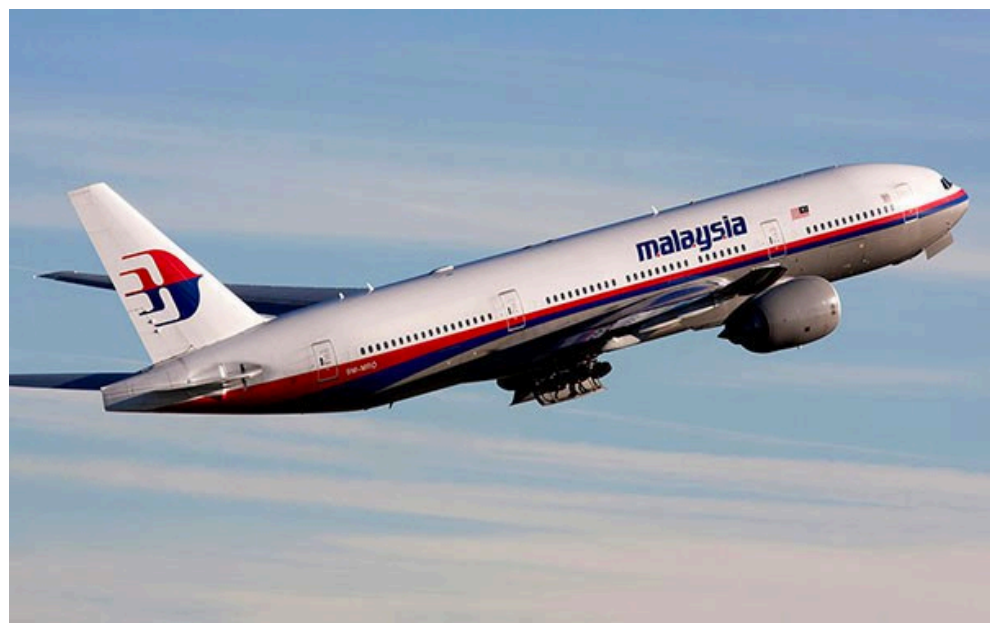
Mechanisms for Ensuring Atomicity

- What would you do?

A

Mechanisms for Ensuring Atomicity

- One approach: LOGGING
  - DBMS logs all actions so that it can undo the actions of aborted transactions.
- ~ like black box in airplanes …



A Mechanisms for Ensuring Atomicity

- Logging used by all modern systems.
- Q: why?

A

Mechanisms for Ensuring Atomicity

- Logging used by all modern systems.
- Q: why?
- A:
  - audit trail &
  - efficiency reasons

C

Transaction Consistency

- "Database consistency" - data in DBMS is accurate in modeling real world and follows integrity constraints

C

Transaction Consistency

- "Transaction Consistency": if DBMS consistent before Xact (running alone), it will be after also
- Transaction consistency: User's responsibility
  - DBMS just checks IC

C Transaction Consistency (cont.)

- Recall: Integrity constraints
  – must be true for DB to be considered consistent

  Examples:

  1. FOREIGN KEY R.sid REFERENCES S
  2. BALANCE >= 0

C Transaction Consistency (cont.)

- System checks ICs and if they fail, the transaction rolls back (i.e., is aborted).
  - Beyond this, DBMS does not understand the semantics of the data.
  - e.g., it does not understand how interest on a bank account is computed
- This is the user's responsibility; DB cannot do much other than enforce the rules and rollback if violated.

I

Isolation of Transactions

- Users submit transactions, and
- Each transaction executes as if it was running by itself.
  - Concurrency is achieved by DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
- Q: How would you achieve that?

I

Isolation of Transactions

- A: Many methods - two main categories:
- Pessimistic – don’t let problems arise in the first place
- Optimistic – assume conflicts are rare, deal with them after they happen.

I

Example

- Consider two transactions (Xacts):

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.01*A,	B=1.01*B	END

- 1st xact transfers $100 from B's account to A's
- 2nd credits both accounts with 1% interest.
- Assume at first A and B each have $1000. What are the **legal outcomes** of running T1 and T2?

I

Example

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.01*A,	B=1.01*B	END

- many - but A+B should be: $2000 * 1.01 = $2020
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. But, the net effect *must* be equivalent to these two transactions running serially in some order.
- **What are the legal ending values for the accounts?**

I

Example (Contd.)

- Legal outcomes: A=1111,B=909 or A=1110,B=910
- Consider a possible interleaved *schedule*:

```
T1:   A=A+100,                B=B-100
T2:               A=1.01*A,            B=1.01*B
```

- This is OK (same as T1;T2). But what about:

```
T1:   A=A+100,                          B=B-100
T2:               A=1.01*A, B=1.01*B
```

I

Example (Contd.)

- Legal outcomes: A=1111,B=909 or A=1110,B=910
- Consider a possible interleaved *schedule*:

T1:	A=A+100,		B=B-100	
T2:		A=1.01*A,		B=1.01*B

- This is OK (same as T1;T2). But what about:

T1:	A=A+100,		B=B-100
T2:		A=1.01*A, B=1.01*B	

- **Result: A=1111, B=910; A+B = 2021, bank loses $1**
- **The DBMS's view of the second schedule:**

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

I Anomalies with Interleaved Execution

- Reading uncommitted data (WR Conflicts, "dirty reads"):

T1:	R(A), W(A),		R(B), W(B), Abort
T2:		R(A), W(A), C	

I

Anomalies with Interleaved Execution

- Reading uncommitted data (WR Conflicts, "dirty reads"):

T1:	R(A), **W(A)**,		R(B), W(B), Abort
T2:		**R(A)**, W(A), C	

- Because T1 ends up aborting, the highlighted R(A) is reading an incorrect value for A.

I

Anomalies with Interleaved Execution

- Nonrepeatable reads (RW Conflicts):

```
T1:  R(A),                                R(A), W(A), C
T2:               R(A), W(A), C
```

I Anomalies with Interleaved Execution

- Nonrepeatable reads (RW Conflicts):

T1:	**R(A),** **R(A),** W(A), C
T2:	R(A), **W(A),** C

- Transactions always must appear to be isolated, so the two R(A) should return the same value.
- With a W(A) in between, the DB may or may not return the same R(A) both times.

I

Anomalies with Interleaved Execution

- ***Phantom read***: Special case of a non-repeatable read where the set of rows returned by the R(A) differs.

T1:	**R(A),**		**R(A),** W(A), C
T2:		R(A), **W(A),** C	

- Some people define a "non-repeatable read" to occur when A is a single value from a single row, and a "phantom read" when A is a set of rows.

I

Anomalies (Continued)

- Overwriting uncommitted data (WW conflicts):

T1:	W(A),			W(B), C
T2:		W(A),	W(B), C	

I

Anomalies (Continued)

- Overwriting uncommitted data (WW conflicts):

T1:	W(A),				W(B),	C
T2:		W(A),	W(B),	C		

- Two different WW conflicts here.

Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

- SET TRANSACTION
  ISOLATION LEVEL <level>

- (do after BEGIN TRANSACTION)

(Review) Goal: ACID Properties

- *ACID transactions* are:
  - *Atomic* : Whole transaction or none is done.
  - *Consistent* : Database constraints preserved.
  - *Isolated* : It appears to the user as if only one process executes at a time.
  - *Durable* : Effects of a process survive a crash.

What happens if system crashes between *commit* and *flushing modified data to disk* ?

D

Problem definition

- Records are on disk
- for updates, they are copied in memory
- and flushed back on disk, *at the discretion of the O.S.!*
  - *(although you can force it)*

D

Problem definition

- Records are on disk
- for updates, they are copied in memory
- and flushed back on disk, *at the discretion of the O.S.!*
  - *(although you can force it)*

- **Solution: Write-ahead log**
  - **All modifications are written to a log before they are applied to the DB.**

D Durability - Recovering From a Crash

- At the end – all committed updates and only those updates are reflected in the database.
  - All active Xacts at time of crash are aborted when system comes back up.
- Some care must be taken to handle the case of a **crash** occurring during the **recovery** process!