Transactions

Credit: Dr. Bruns

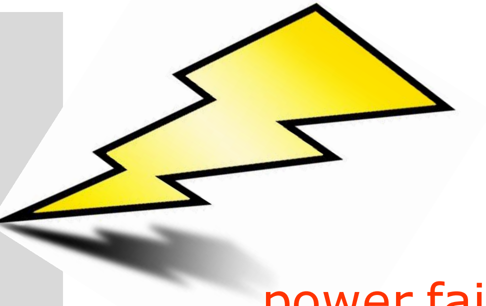
Motivation

Scenario 1: bank funds transfer

account_id	name	balance
101	Rene Wells	4250.55
102	Janie Abbott	34.88
103	Andrea Owen	955.20
104	Javier Benson	772.59

To transfer $60.00 from Rene to Andrea:

1. Decrease balance of Rene by $60.00
2. Increase balance of Andrea by $60.00



power failure after step 1, before step 2!

Issue 1: Atomicity

In the bank example, it's bad if only the account 101 operations happen.

We often want a group of DB operations to either:

- execute completely, or
- not execute at all

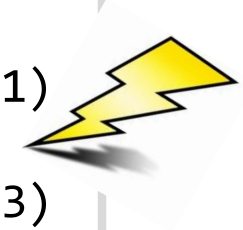
In other words, the group of operations work like one "atomic" unit.

This is **atomicity**.

“all or nothing”

```
amt = 60
x = read(“balance”, 101)
x = x - amt
write(“balance”, x, 101)

y = read(“balance”, 103)
y = y + amt
write(“balance”, 103)
```



Issue 2: Isolation

"no interference"

Don't want other DB operations happening here

```
amt = 60
x = read("balance", 101)
x = x - amt
write("balance", x, 101)

y = read("balance", 103)
y = y + amt
write("balance", 103)
```

In the bank example, we don't want other DB operations to read the balances of accounts 101 and 103 in the middle of this group of operations.

We want the group of operations to act as if it were running by itself.

This is **isolation**.

Issue 3: Consistency

In the bank example, we want the value of x + y to be the same before and after the accounts are updated.

This is a kind of consistency constraint, but it can't be written in SQL.

The constraint may not hold in the middle of the operations.

It is the job of the programmer to ensure the constraint.

This is **consistency**.

{ x + y = c }

```
amt = 60
x = read("balance", 101)
x = x - amt
write("balance", x, 101)

y = read("balance", 103)
y = y + amt
write("balance", 103)
```

{ x + y = c }

Issue 4: Durability

In the bank example, after the accounts are changed, we want the changes to be permanent.

For example, a failure after the operations are finished shouldn't cause the account updates to be lost.

This is **durability**.

```
amt = 60
x = read("balance", 101)
x = x - amt
write("balance", x, 101)

y = read("balance", 103)
y = y + amt
write("balance", 103)
```

ACID Properties

We often want the execution of a related group of DB operations to have these properties:

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

Transaction concept

```
begin;
  update account
     set balance = balance - 60
   where account_id = 101;
  update account
     set balance = balance + 60
   where account_id = 103;
commit;
```

A **transaction** is a group of DB operations that form a unit.

Transactions are initiated by programs written in SQL, C++, etc.

A simple transaction model

To make things easy, we allow only two DB operations:

- read a simple variable from DB
- write a simple variable to DB

We assume 'write' writes to disk immediately.

A simple transaction T

```
T:
read(A)
A = A - 50
write(A)
read(B)
B = B + 50
write(B)
```

How to make transactions ACID?

How to achieve Atomicity?

keep a log of changes

How to achieve Isolation?

run transactions one after the other

How to achieve Durability?

make sure updates written to disk before transaction completes

A simple transaction T

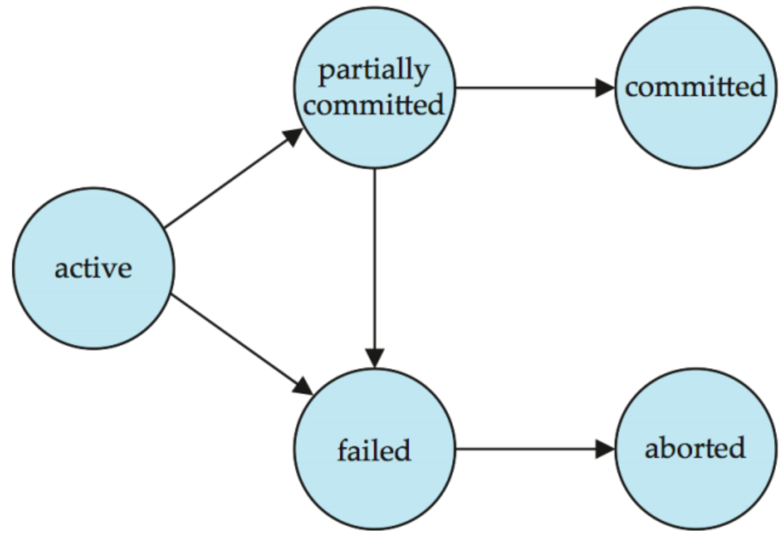
```
T:
read(A)
A = A – 50
write(A)
read(B)
B = B + 50
write(B)
```

Atomicity and Durability

An aborted transaction must have no effect on the database.

Changes caused by an aborted transaction must be rolled back.

A transaction that completes successfully is said to be committed.



Rolling back a transaction: logs

```
T:
read(A)
A = A – 50
write(A)
read(B)
B = B + 50
write(B)
```

Log entry

old value

new value

T: A, 950 -> 900

transaction name

data item being modified

With a log, operations performed in a transaction can be:

- undone
- re-done

Isolation

Isolation can be achieved by running transactions one after another.

But we want to run them in parallel:

- higher throughput and resource utilization
- reduced waiting time

When is it safe to run transactions in parallel?

An interleaved schedule

A = 1000
B = 2000

T1	**T2**

`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`

Schedule 3

What is the result of this interleaving?

This schedule has the same effect as schedule 1, where we ran T1 then T2.

If a schedule is equivalent to a serial schedule, then it is **serializable**.

Lock modes

S – shared mode

With shared mode access, a transaction can read but not write a data item

X – exclusive mode

A transaction can read and write a data item

lock mode compatibility

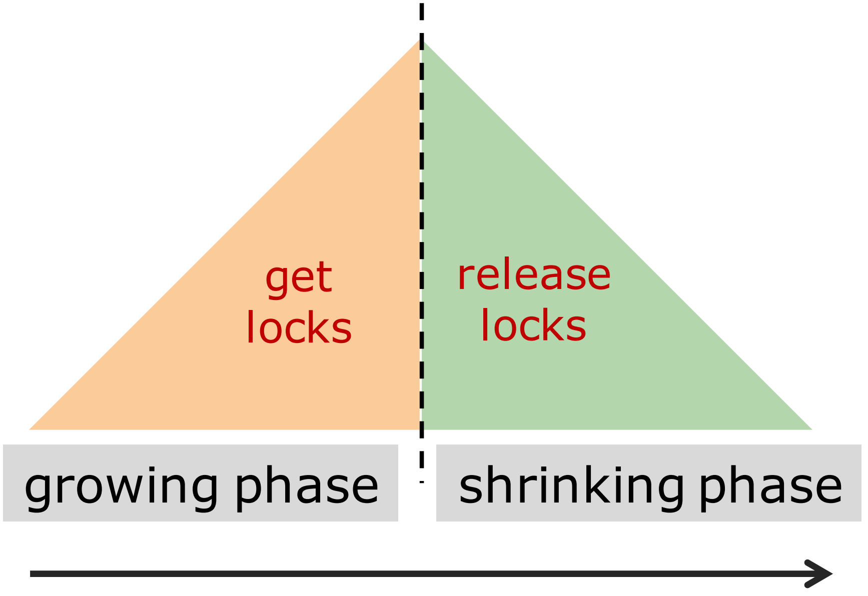
	S	**X**
S	true	false
X	false	false

Basically, a transaction can be granted a shared mode lock on an item immediately even if another transaction holds a shared mode lock on the item.

The two-phase locking protocol

Rule: transaction must be split into two phases

- in the growing phase, a transaction may obtain locks, but not release them
- in the shrinking phase, a transaction may release locks, but not obtain them



SQL - Isolation levels

Weak

Strong

	dirty Reads	Non-repeatable reads	phantoms
Read Uncommitted	yes	yes	yes
Read Committed	no	yes	yes
Repeatable Read	no	no	yes
Serializable	no	no	no

SQL statement:
Set Transaction Isolation Level Serializable;