Acid - Atomicity, Consistency, Isolation & Durability

From: http://en.wikipedia.org/wiki/ACID

Atomicity

Atomicity requires that each transaction is "all or nothing".

If one part of the transaction fails, the entire transaction fails, and the database state is left unchanged.

An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes

A committed transaction appears (by its effects on the database) to be indivisible ("atomic"),

An aborted transaction does not leave effects on the database at all, as if it never existed.

Consistency

Consistency ensures that any transaction will bring the database from one valid state to another.

Any data written to the database must be valid according to all defined rules.

This does not guarantee correctness of the transaction in all ways the application programmer intended but indicates programming did not violate defined rules.

Isolation

Isolation ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially

Providing isolation is the main goal of concurrency control.

Durability

Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors.

As an example, once a group of SQL statements execute, the results need to be stored permanently (even if the system crashes)

Atomicity/Isolation Example w/Traditional File Approach

Let's consider a typical transaction without a DBMS that simply relies on an OS with no extra transaction management.

A husband and wife share a bank account with a $100 balance

They are about to each perform a transaction at the same moment.

The husband walks up to an ATM in NYC and is about to withdraw $1 => balance = balance -1

The wife walks up to an ATM in San Fran and is about to deposit $1 => balance = balance +1

Now each of these High Level Language statements will get converted to machine language however
I will represent them in assembly language for readability

Assembly Language Syntax Convention => OPERATION Destination, Source

Husband in NYC withdraws $1		Wife in San Fran deposits $1
balance = balance -1 converted to:	Line Number	balance = balance + 1 converted to:
MOV R1,Bal	1	MOV R1,Bal
SUB R1, 1	2	ADD R1,1
MOV Bal,ACC	3	Mov Bal, ACC

Ok, so let's execute this code serially drawing from our OS knowledge
Husband and Wife walk up to ATMs in NYC & San Fran at exactly the same time.
The balance is $100 before the transaction – Balance == 100

Ok, so let's execute this code serially drawing from our OS knowledge
For no particular reason the Husband/NYC event occurs first
NYC process executes Line #1 and moves Balance to R1 (NYC R1 = 100)
NYC process executes Line #2 and subtracts 1 from R1 leaving result in Accumulator (NYC ACC = 99)
At this point this process has exceeded its time slice so the state is saved, a context switch occurs and the CPU is given to another process
San Fran process executes Line #1 and moves Balance to R1 (San Fran R1 = 100)
San Fran process executes Line #2 and adds 1 to R1 leaving result in Accumulator (San Fran ACC = 101)
San Fran process executes Line #3 and moves/stores the ACC in **Balance => 101**
San Fran process ends and at some point NYC process resumes where it left off
NYC process executes Line #3 and moves/stores the ACC in **Balance => 99**
Result => starting balance of $100, add a dollar & subtract a dollar can possibly result in $99 (or $101) without Atomicity/Isolation provision