TRANSACTIONs

MANAGEMENT

What’s a transaction?

## A transaction is a logical processing corresponding to

a series of elementary physical operations (reads/writes) on the DB

Examples:

Transfer of a sum between bank accounts

UPDATE CC UPDATE CC

SET balance=balance-50 SET balance=balance+50

WHERE account=123 WHERE account=235

Updating wages of employees in a branch

UPDATE Emp

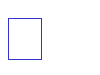
SET wage=1.1\*wage

WHERE branch=‘S01’

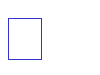
The ACID acronym denotes the 4 properties that the DBMS

should guarantee for every transaction:

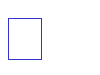
Atomicity: a transaction is a processing unit

The DBMS guarantees that the transaction is performed as a whole

Consistency: a transaction leaves the DB in a consistent state

The DBMS guarantees that no integrity constraint is violated

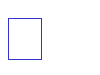
Isolation: a transaction is executed independently of the others

If more than transaction is executed concurrently, the DBMS guarantees

that the net effect is equivalent to one of the many possible sequential executions

of the same transactions

Durability: effects of a correctly terminated transaction should persist over time

The DBMS protects the DB against failures

Transaction Manager

Coordinates the execution of transactions, receiving relevant SQL commands

Logging & Recovery Manager

Is in charge of Atomicity and Durability

Concurrency Manager

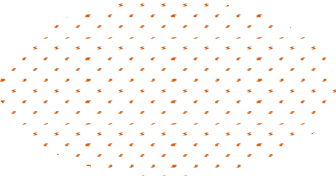
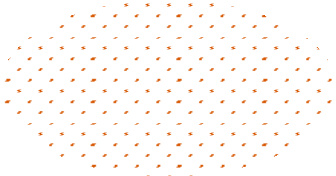
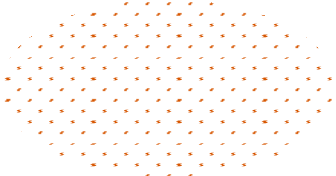
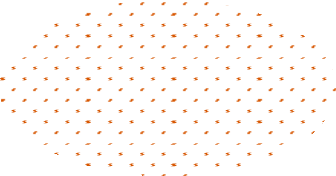
Guarantees Isolation

DDL Compiler

Generates code for controlling Consistency

## In the model we consider a transaction is viewed as a sequence of elementary read (R) and write (W) operations on objects (tuples-One Record) of the DB that, starting from an initial DB state,

brings the DB to a new consistent state



**Start**

**state**

**W(X)**

Intermediate

state

**W(Y)**

Intermediate

state

**R(Z)**

**End**

**state**

**W(Y)**

Intermediate

state

**W(Z)**

Intermediate

state

In general, it is not required that intermediate DB states are consistent - (intermediate state of a transaction is **invisible to other transactions)**

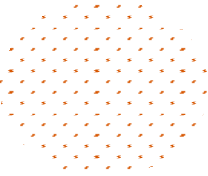
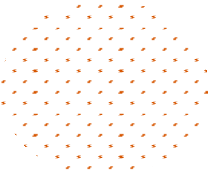
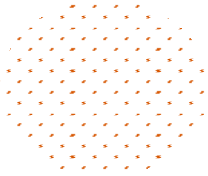
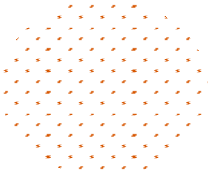
In the model we consider, a transaction (whose beginning is specified by the keyword BEGIN, although this is implicit in SQL) can only have two outcomes:

Complete successfully:

This happens only when the transaction, after having executed

all its operations, specifies a particular SQL statement, called COMMIT (or COMMIT WORK), that “formally”

communicates the successful completion to Transaction Manager



**Start**

**state**

Int. state

**BEGIN**

**W(X)**

Int. state

**R(Y)**

Int. state

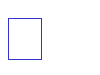
**W(Y)**

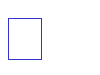
Int. state

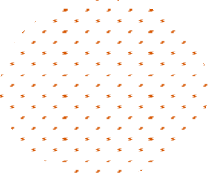
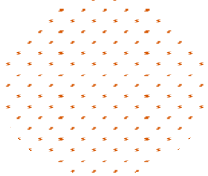
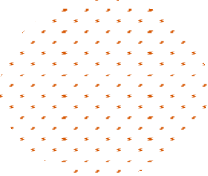
**COMMIT**

**End**

**state**

Complete unsuccessfully (beforehand); 2 cases are possible: The transaction itself, for some reason, decides that it makes no sense to continue and thus “aborts” executing the SQL statement ROLLBACK (or ROLLBACK WORK)

The system (e.g., due to a failure or to a constraint violation)cannot guarantee the successful execution of the transaction, which is thus aborted



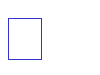
**Start**

**state**

**BEGIN W(X) R(Y)**

Int. state Int. state Int. state

**ROLLBACK**

If, for some reason, the transaction is unable to complete successfully, the DBMS should “undo” any change possibly made to the DB

## The transaction model used by DBMS is actually more complex;

in particular, it is possible to define some “savepoint”,

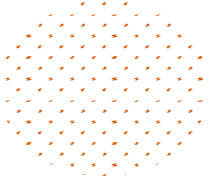
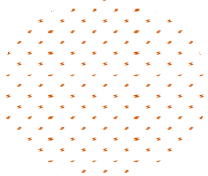
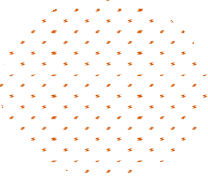
which can be used to undo the operations of a transaction only partially

**Start**

**state**

**BEGIN**

Int. state



**W(Y)**

**W(X)**

**SAVEPOINT**

Int. state

**Savepoint**

Int. state **ROLLBACK TO SAVEPOINT**

To define a savepoint in DB2 we use the command

SAVEPOINT <name> ON ROLLBACK RETAIN CURSORS

to execute a partial rollback

## Since a DBMS should be able to execute different transaction accessing to shared data, it could execute such transactions in sequence (serial execution)

E.g., two transactions T1 and T2 could be executed as follows, where the temporal succession of elementary operations

on the DB (schedule) is highlighted:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **T1** | R(X) | W(X) | Commit |  |  |  |
| **T2** |  |  |  | R(Y) | W(Y) | Commit |

Alternatively, the DBMS could execute multiple transactions concurrently, interleaving operations of one transaction with those of other transactions (enclosed execution)

Concurrent execution of multiple transactions is the key to guarantee performance :

We can exploit the fact that, when a transaction is waiting for an I/O operation to complete, another transaction can use the CPU, thus increasing the system “throughput” (no. of transactions processed in the time unit)

If we have one “short” and one “long” transactions, interleaved execution reduces the average response time of the system

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **T1** | R(X) |  |  |  | W(X) | Commit |
| **T2** |  | R(Y) | W(Y) | Commit |  |  |

## The Transaction Manager should guarantee that concurrently

executing transactions do not interfere with each other

If this is not the case, 4 basic types of problems could arise:

Lost Update: concurrent updates

Dirty Read: reading uncommitted data Unrepeatable Read: interleaving reads and writes

Phantom Row: new data not appearing in the result of a query

## Lost Update: two people, in different shops, buy the very last

ticket for the U2 concert in Rome (!?)

Dirty Read: the U2 tour schedule shows a date in Bologna on 15/07/17, but when you try to buy the ticket for that concert the system tells that no such date exists (!?)

Unrepeatable Read: for the U2 concert (finally, the date has been decided!) you see a price of 90 €, you think about it a little, but when you’re decided, the price is risen to 110 € (!?)

Phantom Row: you want to go see both U2 concerts in Italy, but when you try to buy tickets, you discover that there are now three dates (!?)

The following schedules show a typical lost update case, where we also highlight operations updating the value of X and show how the value of X in the DB varies over time

This update is lost

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **T1** | R(X) | X=X-1 |  |  | W(X) | Commit |  |  |
| **X** | ***1*** | 1 | 1 | 1 | ***0*** | 0 | ***0*** | 0 |
| **T2** |  |  | R(X) | X=X-1 |  |  | W(X) | Commit |

The problem arises because T2 reads the value of X before T1 (that already read it) updates it (“both transactions see the last ticket”)

## In this case, the problem arises because a transactions read a value that is not correct:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **T1** | R(X) | X=X+1 | W(X) |  | Rollback |  |  |
| **X** | ***0*** | 0 | ***1*** | 1 | ***0*** | 0 | ***0*** |
| **T2** |  |  |  | R(X) |  | … | Commit |

This read is “dirty”

What T2 does is based on an “intermediate”, non-stable value of X, (“the definitive date is not 15/07/17”)

Consequences are unpredictable(it depends on what T2 does)

and would be present even if T1 would not abort

## Now the problem is that a transaction reads a value twice,

with different outcomes (“meanwhile, the price has increased”):

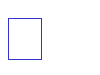
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **T1** |  | R(X) | X=X+1 | W(X) | Commit |  |  |
| **X** | ***0*** | 0 | 0 | 1 | ***1*** | 1 | 1 |
| **T2** | R(X) |  |  |  |  | R(X) | Commit |

The two reads

are inconsistent

Also in this case serious consequences could arise

The same problem can occur for “analysis” transactions

For example, T1 sums the balance of 2 accounts while T2 transfers money between the two (T1 could report an incorrect total value)

## This case could arise only when tuples (Record) are deleted or inserted that should be logically considered by another transaction

E.g.: record r4 is “phantom”, since T1 “dose not see it”

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **T1** | R(r2) | R(r3) | … | W(r2) | W(r3) |  | Commit |  |
| **T2** |  |  |  | R(X) |  | Insert(r4) |  | Commit |

T1:

*T1* cannot see this record

UPDATE Proj

SET location=‘Firenze’ WHERE location=‘Bologna’ T2:

INSERT INTO Proj

VALUES(‘P03’,‘Bologna’)

## Serial: a schedule with transactions executed sequentially Serializable: a schedule involving only committed transactions whose effect on any consistent DB instance is guaranteed

to be identical to that of some serial schedule

Recoverable: a schedule where, if transaction T1 reads a change made by transaction T2, T1 commits only after T2 commits

Cascadeless: a where every transaction can only read changes of committed transactions

Strict: a schedule where every transaction does not read or write values changed by any other active transaction

A technique commonly used by DBMSs to avoid previous

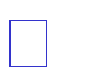
problems consists in locks

Locks are a mechanism normally used by operating systems to regulate access to shared resources

Before executing any operation, it is required to “acquire” a lock

on the requested resource (e.g., a record)

The lock request is implicit, thus invisible at SQL level

… but we will see that we can do something with SQL, anyway

## Locks come in different “flavors” (DB2 has 11 types!)

The basic ones are:

S (Shared): a shared lock is required for reading a value

X (eXclusive): an exclusive lock is required to write/update

a value

## The Lock Manager is a DBMS module in charge of keeping track which resources are currently used and which transactions are using them (and how)

When a transaction T wants to operate on a value Y, a lock request on Y is sent to the Lock Manager

Lock is granted to T according to the following compatibility

table

Another transaction has on Y a lock of type

T requests

|  |  |  |
| --- | --- | --- |
|  | **S** | **X** |
| **S** | **OK** | **NO** |
| **X** | **NO** | **NO** |

a lock of type

## When T finishes using Y, can release the lock (unlock(Y))

Strict 2-phase lock (Strict 2PL) protocol

The way transaction release acquired locks is the key to solve

concurrency problems

It can be proven that isolation is guaranteed if:

A transaction first acquires all necessary locks Locks are released only at the end of the execution (COMMIT or ABORT)

*no. of locks granted to T*

COMMIT/ABORT

*time*

## As a collateral effect, deadlocks (stalemate situation)

can happen, which are solved by aborting a transaction

Previous schedule is modified as follows:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **T1** | S-lock(X) | R(X) | X=X-1 |  |  |  | X-lock(X) | wait | wait |
| **X** | ***1*** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **T2** |  |  |  | S-lock(X) | R(X) | X=X-1 |  | X-lock(X) | wait |

Neither T1 nor T2 succeed in acquiring the lock needed to update X (they remain in “wait” state)

We thus have a deadlock

If the DBMS chooses to abort, say, T2, then T1 can proceed

## In this case, correct execution requires that T2 awaits T1

termination before reading the value of X

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **T1** | S-lock(X) | R(X) | X=X+1 | X-lock(X) | W(X) |  | rollback | unlock(X) |  |
| **X** | ***0*** | 0 | 0 | 0 | ***1*** | 1 | ***0*** | 0 | 0 |
| **T2** |  |  |  |  |  | S-lock(X) | wait | wait | R(X) |

Also in this case, T2 is put on hold, and T1 is therefore

Guaranteed to read always the correct value of X

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **T1** |  |  | S-lock(X) | R(X) | X=X+1 | X-lock(X) | wait | Wait | wait | W(X) |
| **X** | ***0*** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ***1*** |
| **T2** | S-lock(X) | R(X) |  |  |  |  | R(X) | Commit | unlock(X) |  |