

# DATA BASES 2

Federico Mainetti Gambera

29 settembre 2020

## Indice

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Prerequisites: References to Relational Algebra and SQL . . . . .	2
1.1.1	Relational Algebra . . . . .	2
1.1.2	SQL . . . . .	2
1.2	Books . . . . .	2
<b>2</b>	<b>Transactional Systems</b>	<b>3</b>
<b>3</b>	<b>Concurrency Control</b>	<b>4</b>
3.1	Anomalies . . . . .	4
3.1.1	Lost Update . . . . .	4
3.1.2	Dirty Read . . . . .	4
3.1.3	Nonrepeatable Read . . . . .	4
3.1.4	Phantom Update . . . . .	4
3.1.5	Phantom Insert . . . . .	5
3.2	Principles of Concurrency Control . . . . .	5
3.2.1	Serializable schedule . . . . .	5
3.2.2	View-serializability . . . . .	5
3.2.3	Conflict-serializability . . . . .	6
3.2.4	Testing conflict-serializability . . . . .	6
3.3	Concurrency control approaches: locking . . . . .	7
3.3.1	Locking . . . . .	7

# 1 Introduction

Introduction slides:

`../other/professor'sslides/0-Intro.pdf`

## 1.1 Prerequisites: References to Relational Algebra and SQL

### 1.1.1 Relational Algebra

`../other/RelationalAlgebraAndSQL/04-RelationalAlgebra.pdf`

`../other/RelationalAlgebraAndSQL/05-RelationalAlgebra2.pdf`

### 1.1.2 SQL

`../other/RelationalAlgebraAndSQL/08-SQL1.pdf`

`../other/RelationalAlgebraAndSQL/09-SQL2.pdf`

`../other/RelationalAlgebraAndSQL/10-SQL3.pdf`

## 1.2 Books

All books can be found at:

`../other/books`

## 2 Transactional Systems

../other/professor'sslides/1\_Transactions\_NEW.pdf

A **transaction** is an elementary, atomic, unit of work performed by an application. Each transaction is conceptually encapsulated within two commands:

- **begin of transaction** (bot);
- **end of transaction** (eot).

Within a transaction, **one** of the commands below is executed (**exactly once**) to signal the end of the transaction:

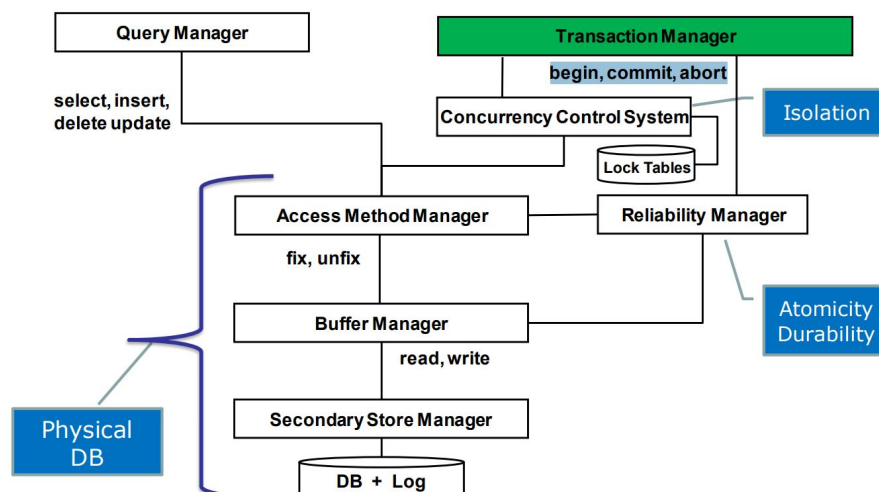
- **commit**;
- **rollback**.

**Transactional System** (OLTP) is a system that supports the execution of transactions on behalf of concurrent applications.

A transaction is a unit of work enjoying the **ACID** properties:

- **Atomicity** (abort-rollback-restart, commit protocols): either all the operations in the transaction are executed or none is executed;
- **Consistency** (integrity checking of the DBMS): a transaction must satisfy the DB integrity constraints, if the initial state is consistent then the final state is also consistent, but this is not necessarily true for the intermediate states of the transaction;
- **Isolation** (concurrency control): the concurrent execution of a number of transaction must produce the same result as the execution of the same transactions in a sequence;
- **Durability** (recovery management): the effect of a transaction that has successfully committed will last "forever".

Lets take a look at where are the ACID properties managed in the DBMS architecture:



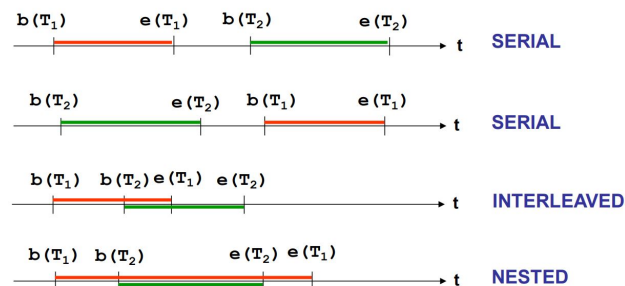
## 3 Concurrency Control

../other/professor'sslides/2-ConcurrencyControl.pdf

The **concurrency control system**:

- manage the simultaneous execution of transactions;
- avoids the insurgence of anomalies;
- ensure performance: exploit parallelism to maximise transactions per second.

Two transaction can have a **serial** execution, an **interleaved** execution or a **nested** execution.



### 3.1 Anomalies

An anomaly happens when two transactions are executed interleaved or nested and the result is different if they were executed serially.

#### 3.1.1 Lost Update

$$r_1(x) \rightarrow r_2(x) \rightarrow w_1(x) \rightarrow w_2(x)$$

An update is applied from a state that ignores a preceding update, which is lost.

#### 3.1.2 Dirty Read

$$r_1(x) \rightarrow w_1(x) \rightarrow r_2(x) \rightarrow abort_1 \rightarrow w_2(x)$$

An uncommitted value is used to update the data.

#### 3.1.3 Nonrepeatable Read

$$r_1(x) \rightarrow r_2(x) \rightarrow w_2(x) \rightarrow r_1(x)$$

Someone else updates a previously read value.

#### 3.1.4 Phantom Update

$$r_1(x) \rightarrow r_2(x) \rightarrow w_2(x) \rightarrow r_1(x)$$

Someone else updates data that contributes to a previously read datum.

Lets see an example: given the constraint  $A+B+C = 100$  and the initial values  $A = 50, B = 30, C = 20$

$T_1 : r(A), r(B)$

$T_2 : r(B), r(C)$

$T_2$  : subtract 10 from  $C$  and add 10 to  $B$ .

$T_2 : w(B), w(C)$

$T_1 : r(C)$  ( $T_1$  reads that the sum is 90).

So for  $T_1$  it is as if "somebody else" had updated the value of the sum, but for  $T_2$  the update is perfectly legal (does not change the value of the sum).

### 3.1.5 Phantom Insert

$$r_1(x) \rightarrow w_2(\text{new tuple}) \rightarrow r_1(x)$$

Someone else inserts data that contributes to a previously read datum.

This anomaly does not depend on data already present in the DB when  $T_1$  executes, but on a “phantom” tuple that is inserted by “someone else” and satisfies the conditions of a previous query of  $T_1$ .

## 3.2 Principles of Concurrency Control

**Operations:** a read or write of a specific datum by a specific transaction.

**Schedule:** a sequence of operations performed by concurrent transactions that respects the order of operations of each transaction.

How many distinct schedules exist for  $n$  transaction each with  $k_i$  operations?

$$N_D = \frac{(\sum_{i=1}^n k_i)!}{\prod_{i=1}^n (k_i!)}$$

How many of them are serial?

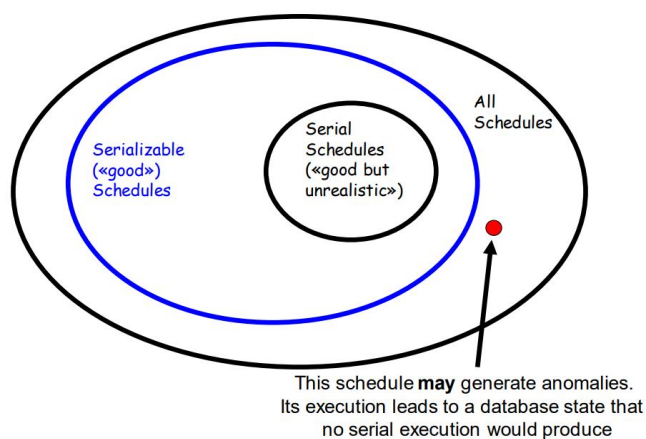
$$N_S = n!$$

**Scheduler:** a component that accepts or rejects the operations requested by the transactions.

**Serial schedule:** a schedule in which the actions of each transaction occur in a contiguous sequence.

### 3.2.1 Serializable schedule

A **serializable schedule** is a schedule that leaves the database in the **same state** as **some** serial schedule of the same transactions.



### 3.2.2 View-serializability

$r_i(x)$  **reads-from**  $w_j(x)$  in a schedule  $S$  when  $w_j(x)$  precedes  $r_i(x)$  and there is no  $w_k(x)$  in  $S$  between  $r_i(x)$  and  $w_j(x)$ .

$w_i(x)$  in a schedule  $S$  is a **final write** if it is the last write on  $x$  that occurs in  $S$ .

Two schedules are **view-equivalent** if they have the same operations, the same reads-from relation, and the same final writes.

A schedule is **view-serializable** if it is view-equivalent to a serial schedule of the same transactions. The class of view-serializable schedules is named **VSR**.

$S$  is **view-serializable** if:

1. every read operation sees the same values;

2. the final value of each object is written by the same transaction as if the transactions were executed serially in some order.

Deciding if a generic schedule is in VSR is an **NP-complete problem** but is a heavy task, so we look for a stricter definition that is easier to check, even if we reject some schedule that would be acceptable: VSR schedules are "too many".

### 3.2.3 Conflict-serializability

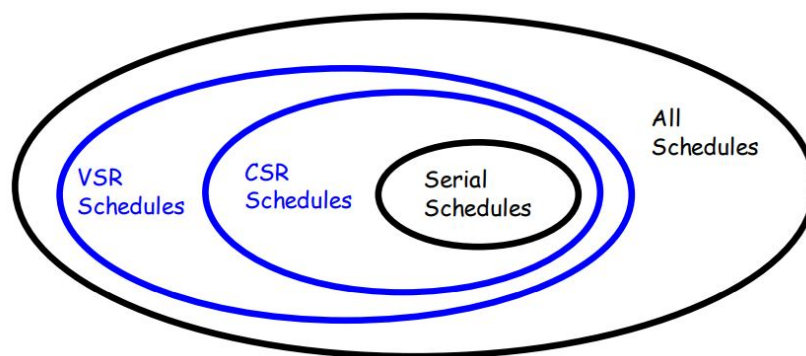
Two operations  $o_i$  and  $o_j$  are in **conflict** if they address the same resource and at least one of them is a write: **read-write** conflicts (r-w or w-r) and **write-write** conflicts (w-w).

Two schedules are **conflict-equivalent** if contain the same operations and in all conflicting pairs transactions occur in the same order.

A schedule is **conflict-serializable** if it is conflict-equivalent to a serial schedule of the same transactions.

The class of conflict-serializable schedules is named **CSR**.

All conflict-serializable schedules are also view-serializable, but the inverse is not necessarily true:  $CSR \rightarrow VSR$ .



### 3.2.4 Testing conflict-serializability

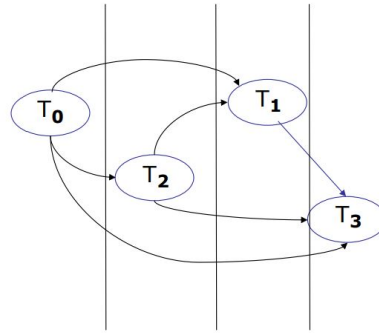
To check if a schedule is conflict-serializable we use a **conflict graph**:

- one node for each transaction  $T_i$ ;
- one arc from  $T_i$  to  $T_j$  if there exists at least one conflict between an operation  $o_i$  of  $T_i$  and an operation  $o_j$  of  $T_j$  such that  $o_i$  precedes  $o_j$ ;
- a schedule is in CSR if and only if its conflict graph is **acyclic**.

To check if a graph is acyclic we use this algorithm:

- if a graph is acyclic, then it must have at least one node with no targets, called leaf (with no arrows going away from the node).
- if we "peel off" a leaf in an acyclic graph, then we are always left with an acyclic graph, and if we keep peeling off leaf nodes, one of two things will happen: we will eventually peel off all nodes and so the graph is acyclic, or we will get to a point where there is no leaf, yet the graph is not empty and so the graph is cyclic.

If  $S$ 's graph is acyclic then it induces a **topological (partial) ordering** on its nodes, that is easier to understand graphically with an example:



In general, there can be many compatible serial schedules.

### 3.3 Concurrency control approaches: locking

Everything we have seen until now is based on study of schedules after they have been executed. In real life we need a system that can assure concurrency control in "live".

There are two main families of techniques:

- Pessimistic: based on locks, resource access control.
- Optimistic: based on timestamps and versions, serve as many requests as possible, possibly using out-of-date versions of the data.

#### 3.3.1 Locking

A transaction is **well-formed w.r.t. locking** if

- read operations are preceded by **r\_lock** (shared lock) and followed by **unlock**;
- write operations are preceded by **w\_lock** (exclusive lock) and followed by **unlock**.

Transactions that first read and then write an object may:

- acquire a w\_lock already when reading or
- acquire a r\_lock first and then upgrade it into a w\_lock (lock escalation).

Possible states of an object:

- free;
- r-locked (locked by one or more readers);
- w-locked (locked by a writer);

When a lock request is granted, the resource is acquired; when an unlock is executed, the resource becomes available.

The lock manager grants access to resources according to the **conflict table**:

	FREE	R-LOCKED	W-LOCKED
r_lock	OK (R-LOCKED)	OK (R-LOCKED $n++$ )	NO (W-LOCKED)
w_lock	OK (W-LOCKED)	NO (R-LOCKED)	NO (W-LOCKED)
unlock	ERROR	OK (DEPENDS $n--$ )	OK (FREE)

where  $n$  is the counter of the current readers