EPL646 – Advanced Topics in Databases

Lecture 8

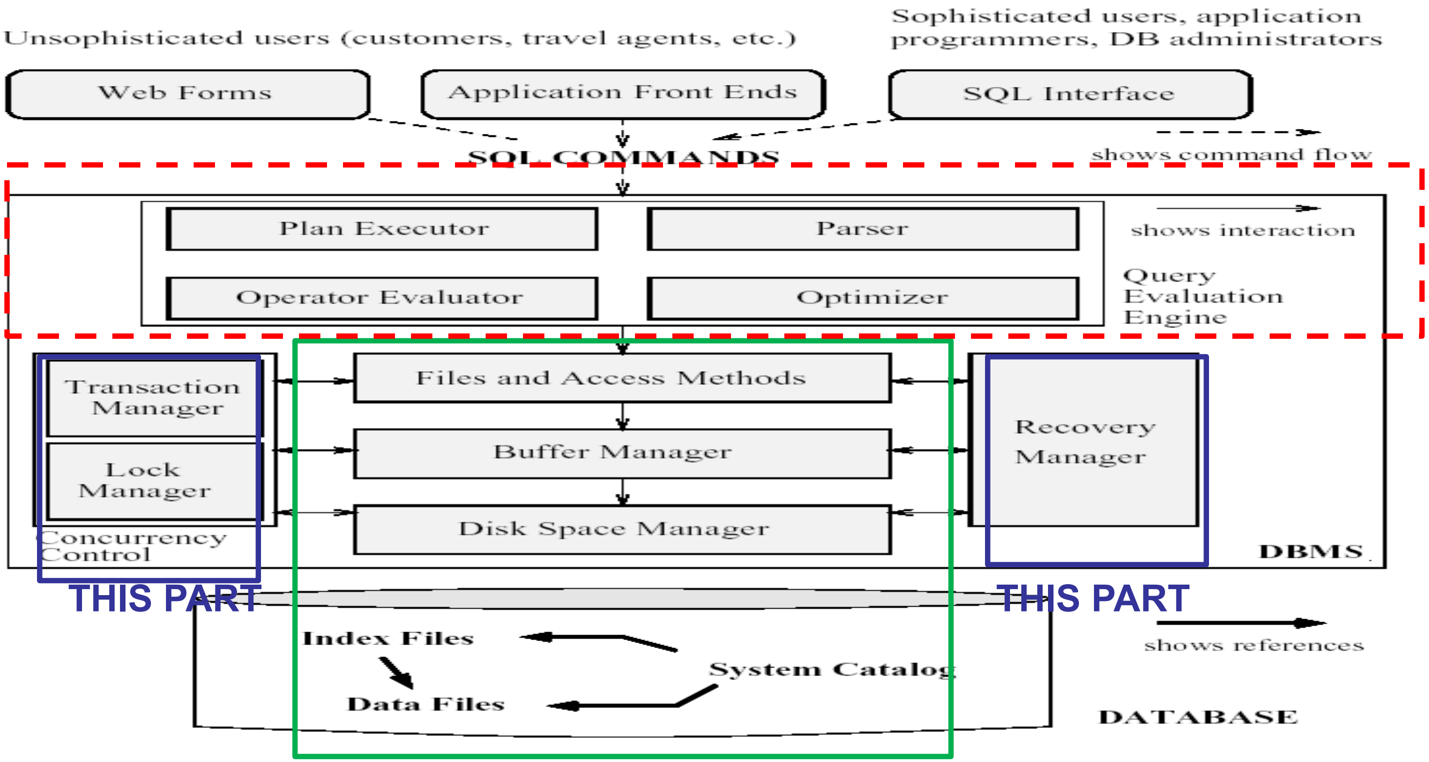
Transaction Management Overview

Chapter 17.1-17.6: Elmasri & Navathe, 5ED
Chapter 16.1-16.3 and 16.6: Ramakrishnan & Gehrke, 3ED

Demetris Zeinalipour
http://www.cs.ucy.ac.cy/~dzeina/courses/epl646

Overview of Transaction Processing (Επισκόπηση Επεξεργασίας Δοσοληψιών)

- We will now focus on **Concurrency Control (Έλεγχο Ταυτοχρονίας)** and **Recovery Management (Τεχνικές Ανάκαμψης)** in cases of failures



Lecture Outline
Transaction Management Overview

- 16.0) Introduction to **Transactions** (**Δοσοληψίες** ή **Συναλλαγές**)
- 16.1) The **ACID** (Atomicity-Consistency-Isolation-Durability) Properties
- 16.2) **Transactions** and **Schedules** (Χρονοπρόγραμμα)
- 16.3) **Concurrent Executions** of Transactions (**Ταυτόχρονες Εκτελέσεις** Δοσοληψιών) and Problems
- 16.6) **Transaction Support in SQL**

Below topics will be covered as part of subsequent lectures

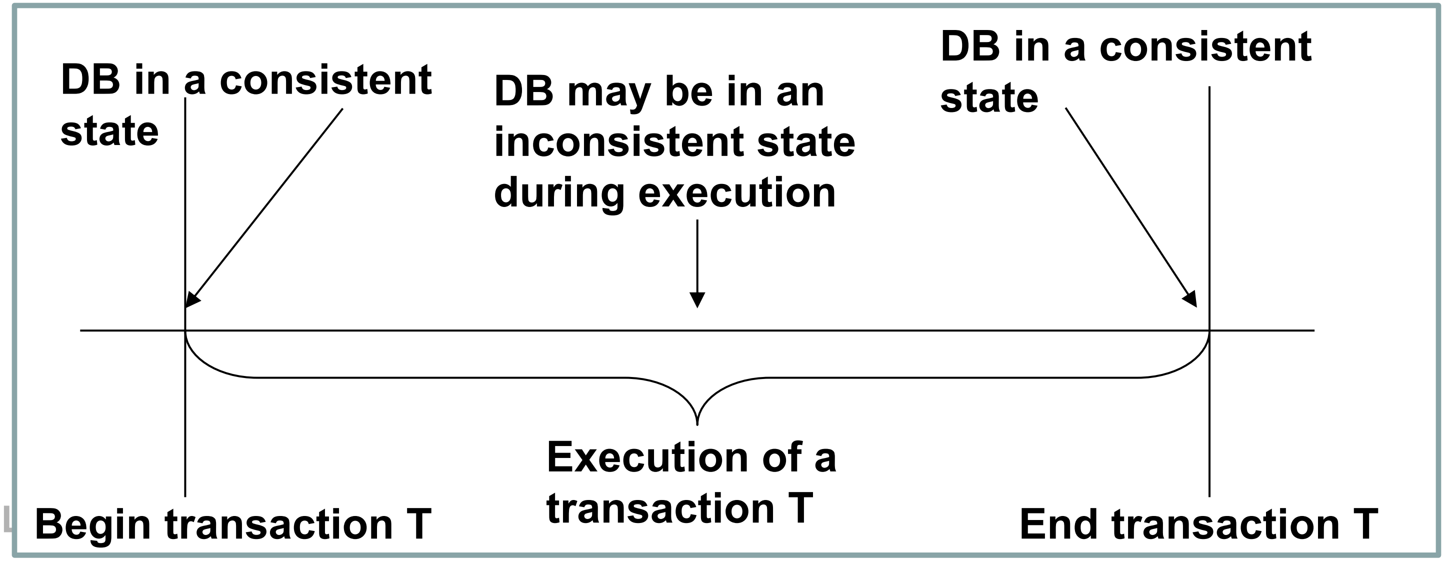
- 16.4) Concurrency with Locks (Κλειδαριές)
- 16.5) Concurrency with Timestamps (Χρονόσημα)
- 16.7) Introduction to Crash Recovery (Ανάκαμψη Σφαλμάτων)

Introduction to Transactions (Εισαγωγή σε Δοσοληψίες)

- The concept of **transaction (δοσοληψία)** provides a mechanism for describing logical units of database processing.
  - **Analogy:** A **transaction** is to a **DBMS** as a **process** is to an **Operating System**.
- **Transaction Processing Systems (Συστήματα Επεξεργασίας Δοσοληψιών)** are systems with large databases and hundreds of concurrent users executing database transactions
  - **Examples:** *Airline Reservations* (Αεροπορικές Κρατήσεις), *Banking* (Εφαρμογές Τραπεζικού Τομέα), *Stock Markets* (Χρηματιστήρια), *Supermarkets* (Υπεραγορές),

Introduction to Transactions (Εισαγωγή σε Δοσοληψίες)

- **Transaction (Δοσοληψία) (Xact),** is an **atomic** (i.e., all-or-nothing) sequence of **database operations** (i.e., read-write operations).
- It is the **DBMS's abstract view** of a **user program**!
- A **transaction** (collection of actions) makes **transformations** of system states while preserving the **database consistency (συνέπεια βάσης)** … next slide.



DB in a consistent state

DB may be in an inconsistent state during execution

DB in a consistent state

Begin transaction T

Execution of a transaction T

End transaction T

DB Consistency vs. **Trans.** Consistency (Συνέπεια ΒΔ vs. Συνέπεια Δοσοληψίας)

- **Database Consistency (Συνέπεια Βάσης)**
  - A **database** is in a **consistent state** if it **obeys all of** the **Integrity Constraints (Κανόνες Ακεραιότητας)** defined over it.
  - **Examples of Integrity Constraints:**
    - **Domain Constraints** (Πεδίου Ορισμού): e.g., SID must be integer
    - **Key Constraints** (Κλειδιού): e.g., no 2 students have the same SID
    - **Foreign Key Constraints** (Ξένου Κλειδιού): e.g., DepartmentID in Student must match the DepartmentID in Department's table.
    - **Single Table Constraints:** e.g., CHECK (age>18 AND age<25)
    - **Multiple Table Constraints:**
      - e.g., Create ASSERTION C CHECK ((SELECT A) + (SELECT B) < 10)
- **Transaction consistency (Συνέπεια Δοσοληψίας)**
  - Complements Database consistency by incorporating user semantics (e.g., σημασιολογία όπως ορίζεται από τον χρήστη)
  - T.C is the user's responsibility (e.g., previous example)

Introduction to Transactions (Εισαγωγή σε Δοσοληψίες)

- One way of specifying the **transaction boundaries** is by specifying explicit **BEGIN TRANSACTION** and **END TRANSACTION** statements in an application program
  - **Transaction Example in MySQL**

    START TRANSACTION;

    SELECT @A:=SUM(salary) **FROM** table1 **WHERE** type=1;

    UPDATE table2 **SET** summary=@A **WHERE** type=1;

    COMMIT;
  - **Transaction Example in Oracle (similar with SQL Server)**
    - When you connect to the database with **sqlplus** (Oracle command-line utility that runs SQL and PL/SQL commands interactively or from a script) a transaction begins.
    - BEGIN | SET AUTOCOMMIT OFF | insert... ; insert... ; update... ; commit; exit; END;
  - **Transaction Example in C: See Next Slide**

    Note that the given example has no explicit START/END statements as the whole program is essentially 1 transaction (as the previous example with Oracle's sqlplus utility.

Transaction Consistency
Example with Embedded SQL

Consider an airline reservation example with the relations*:
FLIGHT(**FNO**, **DATE**, SRC, DEST, **STSOLD**)
CUST(**CID**, ADDR)
FC(**FNO**, **DATE**, **CID**, SPECIAL)

- **The below example shows a Transaction constraint (not captured by ICs)**

```
main {
…
    EXEC SQL BEGIN DECLARE SECTION; // define C host program variables (accessible in SQL environment)
    char flight_no[6], customer_id[20];    // these host-language variable are prefixed with ":" in SQL statements
    char day;
    EXEC SQL END DECLARE SECTION;

    scanf("%s %d %s", flight_no, day, customer_id);

    EXEC SQL UPDATE FLIGHT
    SET STSOLD = STSOLD + 1
    WHERE FNO = :flight_no AND DATE = :day;
```

DB in inconsistent state

```
    EXEC SQL INSERT
    INTO FC(FNO, DATE, CID, SPECIAL);
    VALUES(:flight_no, :day, :customer_id, null);

    printf("Reservation completed");
    return(0);
}
```

Sell a seat on a given flight and date by increasing the **SeaTSOLD** attribute

Store the sale in the Flight-Customer table

If only the first action is executed then relations FLIGHT and FC will be inconsistent
*➔ Although **not a concurrent program**, we need to ensure **transaction consistency** (all-or-nothing)!*

---

* *We make some **simplifying assumptions** regarding the schema and constraints*

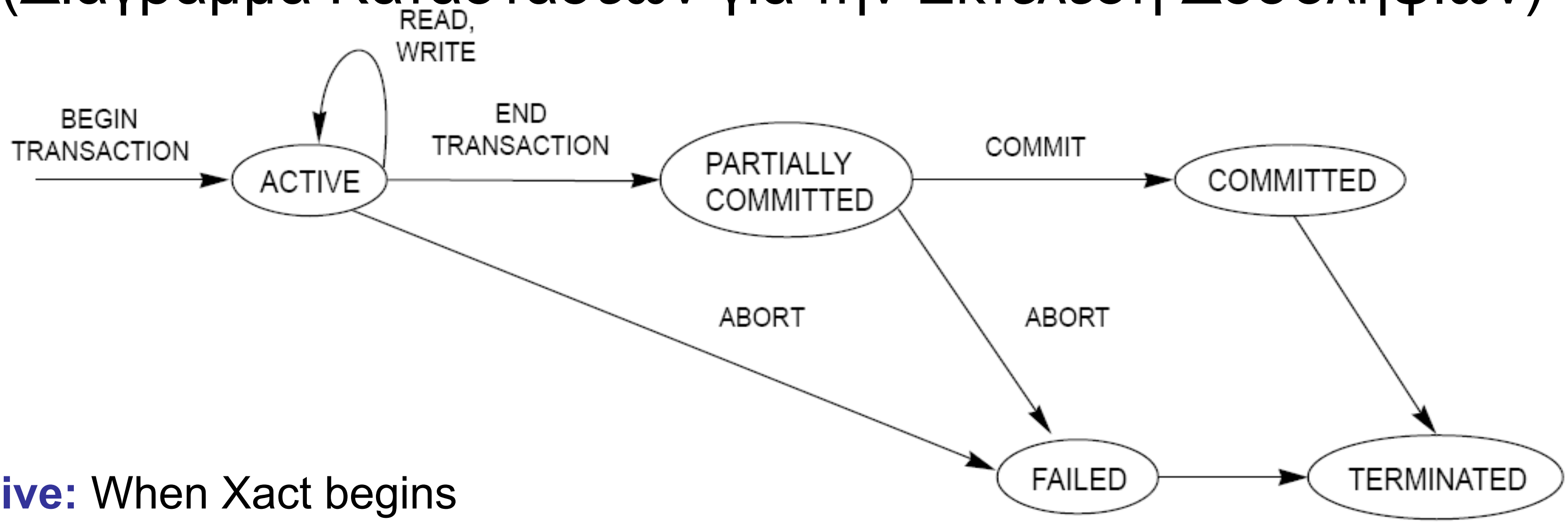
Introduction to Transactions (Εισαγωγή σε Δοσοληψίες)

- Things get even **more complicated** if we have several DBMS programs (transactions) **executed concurrently**.
- **Why do we need concurrent executions?**
  - **It** is **essential** for good DBMS performance!
    - **Disk accesses are frequent**, and relatively **slow**
    - Overlapping I/O with CPU activity increases **throughput** and **response time.**
- **What is the problem with concurrent transactions?**
  - **Interleaving (Παρεμβάλλοντας)** transactions might lead the system to an inconsist state (like previous example):
    - **Scenario:** A Xact prints the monthly bank **account statement** for a user U (one bank transaction at-a-time).**Before** finalizing the report another Xact **withdraws $X** from user **U**.
    - **Result:** Although the report contains an updated **final balance,** it shows nowhere the bank transaction that caused the decrease (unrepeatable read problem, explained next)
- A DBMS **guarantees** that these **problems** will **not arise**.
  - *Users are given the impression that the transactions are executed* ***sequentially (σειριακά),*** *the one after the other.*

Introduction to Transactions (Εισαγωγή σε Δοσοληψίες)

State Diagram for Transaction Execution

(Διάγραμμα Καταστάσεων για την Εκτέλεση Δοσοληψιών)



Active: When Xact begins

Partially Committed: When Xact ends, several recovery checks take place making sure that the DB can always recover to a consistent state

Failed: If Xact **aborts** for any reason (**rollback** might be necessary to return the system to a consistent state)

Committed: After partial committed checks are successful. Once committed we never return (roll-back) to a previous state

The ACID properties
(Οι ιδιότητες ACID)

- What are the **fundamental (θεμελιώδεις) properties** that a DBMS must **enforce** so that **data remains consistent** (in the face of **concurrent access** & **failures**)?
- A DBMS needs to enforce **four (4) properties:**

Atomicity – Consistency – Isolation - Durability

Ατομικότητα – Συνέπεια - Απομόνωση – Μονιμότητα

- Jim Gray defined the key transaction properties of a reliable system in the late 1970.
- Acronym **ACID** was coined by Reuter and Haerder in 1983
  - Reuter, Andreas; Haerder, Theo "Principles of Transaction-Oriented Database Recovery". ACM Computing Surveys (ACSUR) 15 (4): pp. 287-317, 1983

The ACID properties
(Οι ιδιότητες ACID)

1. Atomicity (Ατομικότητα): All or nothing!

- *An executing transaction completes in its **entirety** (i.e., ALL) or it is **aborted altogether** (i.e., NOTHING).*
- e.g., **Transfer_Money(Amount, X, Y)** means i) **DEBIT(Amount, X)**; ii) **CREDIT(Amount, Y)**. **Either both** take place **or none**.
- Reasons for Incomplete Transactions
  - **Anomaly Detection (**e.g., Constraint violation) or **System Crash (e.g., power)**
- **Responsibility:** Recovery Manager (use log file to record all writes)

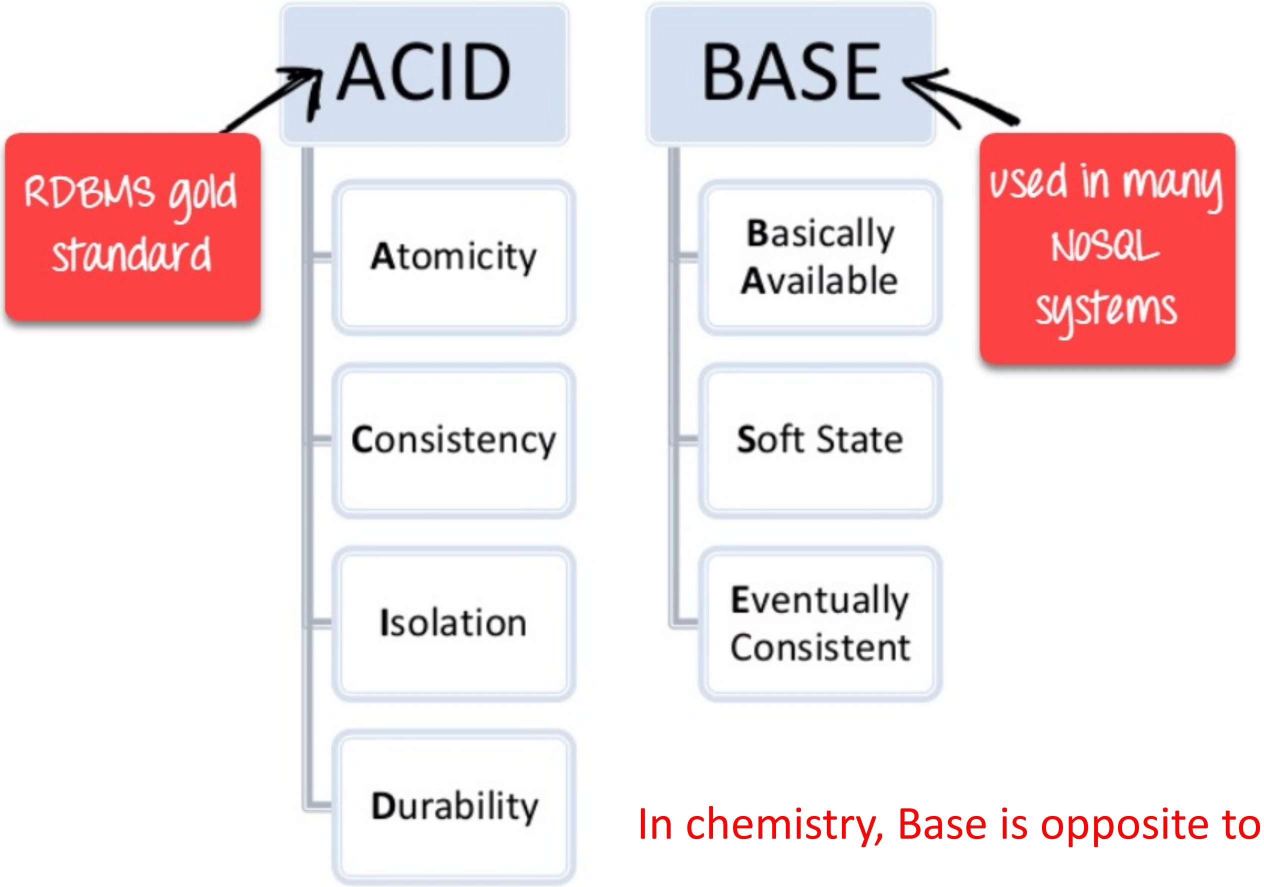
2. Consistency (Συνέπεια): Start & End Consistent!

- *If each **Transaction is consistent**, and the **DB starts consistent**, then the **Database ends up consistent**.*
- If a **transaction violates** the database's **consistency rules**, the entire transaction will be **rolled back** and the database will be **restored to a state consistent** with **those rules**
- **Responsibility: User** (DB only enforcing IC rules)

The ACID properties
(Οι ιδιότητες ACID)

3. **Isolation (Απομόνωση):** See your own data only!
   - *An executing transaction cannot* ***reveal its*** *(incomplete) results before* ***it commits****.*
   - Consequently, the **net effect** is **identical** to **executing** all transactions, the **one after the other** in some **serial order**.
     - e.g., if two transactions T1 and T2 exists, then the output is guaranteed to be either T1, T2 or T2, T1 (The DBMS cannot guarantee the order of execution, that is the user's job!) … see example next page
   - **Responsibility:** Lock Manager (i.e., Concurrency Control Manager)
4. **Durability (Μονιμότητα):** DBMS Cannot Regret!
   - *Once a* ***transaction commits****, the system must* ***guarantee*** *that the results of its operations will* **never be lost**, in spite of subsequent failures.
   - **Responsibility:** Recovery Manager (use log file to record all writes)

ACID vs. BASE



RDBMS gold standard

ACID

- Atomicity
- Consistency
- Isolation
- Durability

BASE

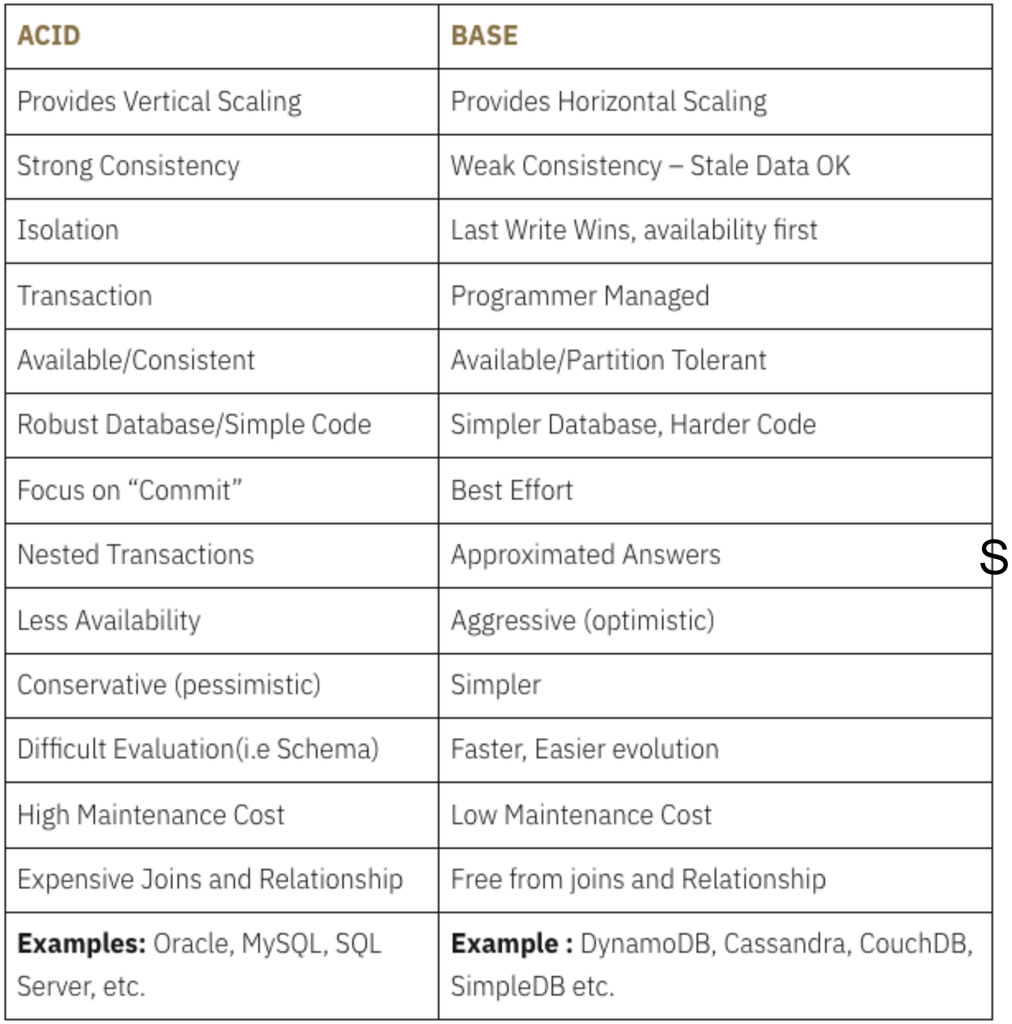
- Basically Available
- Soft State
- Eventually Consistent

used in many NoSQL systems

In chemistry, Base is opposite to ACID

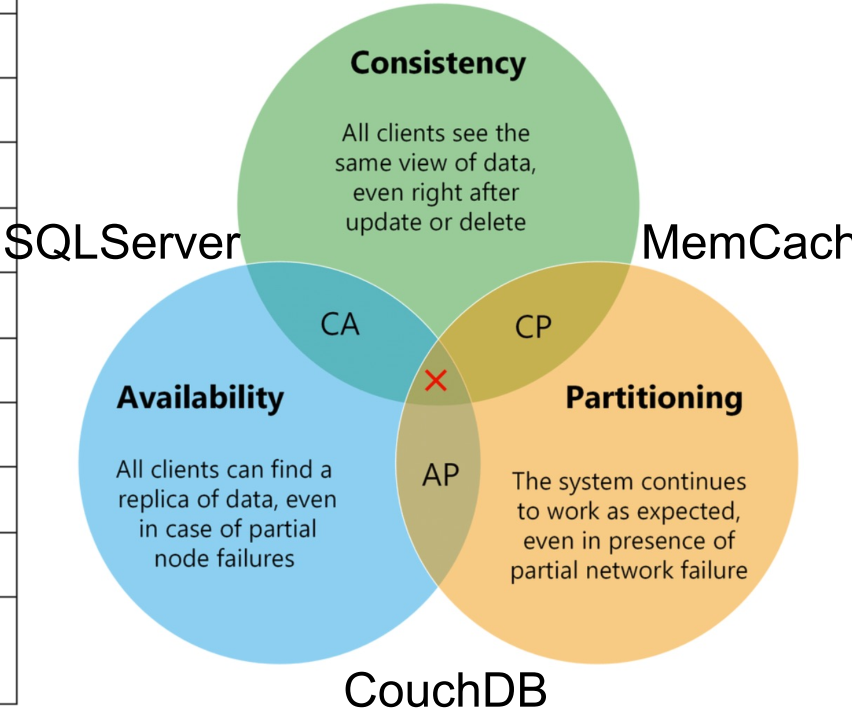
https://medium.com/analytics-vidhya/significance-of-acid-vs-base-vs-cap-philosophy-in-data-science-2cd1f78200ce

ACID vs. BASE (and CAP)



ACID	BASE
Provides Vertical Scaling	Provides Horizontal Scaling
Strong Consistency	Weak Consistency – Stale Data OK
Isolation	Last Write Wins, availability first
Transaction	Programmer Managed
Available/Consistent	Available/Partition Tolerant
Robust Database/Simple Code	Simpler Database, Harder Code
Focus on “Commit”	Best Effort
Nested Transactions	Approximated Answers
Less Availability	Aggressive (optimistic)
Conservative (pessimistic)	Simpler
Difficult Evaluation(i.e Schema)	Faster, Easier evolution
High Maintenance Cost	Low Maintenance Cost
Expensive Joins and Relationship	Free from joins and Relationship
Examples: Oracle, MySQL, SQL Server, etc.	**Example :** DynamoDB, Cassandra, CouchDB, SimpleDB etc.

There is also the **CAP Theorem** in the NoSQL world: you can only choose 2 out of 3 in a **distributed system**:



Consistency

All clients see the same view of data, even right after update or delete

Availability

All clients can find a replica of data, even in case of partial node failures

Partitioning

The system continues to work as expected, even in presence of partial network failure

Notation for Transactions

(Σημειογραφία για Δοσοληψίες)

- **Actions** executed by a transaction include **reads** and **writes** of **database objects**
- **Notation**
  - **R_T(O):** The Transaction **T Reads** an Object **O**.
  - **W_T(O):** The Transaction **T Writes** an Object **O**.
    - When Transaction is clear in context we shall omit the T
    - Although written, the data is in really **pending** until **committed**.
  - **$Commit_T$**: Complete successfully **writing** data to disk
  - **$Abort_T$**: Terminate and **undo** all carried out actions
- **Assumptions**
  - Transaction **Communication** only **through the DBMS**
  - Database Objects: **Static Collection (i.e., tables, etc. not added/removed** ... dynamic case more complex)

Transactions and Schedules
(Δοσοληψίες και Χρονοπρόγραμμα)

- **Schedule (Χρονοπρόγραμμα)**
  - List of **actions** (read, write, abort, or commit) from a **set (ομάδας)** of transactions (T1, T2, …) where the **order of actions** inside each transaction **does not change**.
    - e.g., if T1=R(A), W(A) then W(A), R(A) is not the same schedule (as it is in opposite order)

Schedule

T1	T2
R(A)	
W(A)	
	R(B)
	W(B)
R(C)	
W(C)	

- Note that the DBMS might carry out **other actions** as **well** (e.g., evaluate arithmetic expressions) .
- Yet these do not affect the other transactions, thus will be omitted from our presentation
- We shall introduce **Commits/Aborts** subsequently.

Transactions and Schedules

(Δοσοληψίες και Χρονοπρόγραμμα)

- **Serial Schedule (Σειριακό Χρονοπρόγραμμα)**
  - A schedule in which the different **transactions** are **NOT interleaved** (i.e., transactions are executed from start to finish one-by-one)

Serial Schedule

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

Serial Schedule

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

N! Possible Serial Schedules, where N the number of Xacts

Problems due to Interleaved Xact
(Προβλήματα από την Παρεμβολή Δοσοληψιών)

Problems that arise when interleaving Transactions.

- **Problem 1: Reading Uncommitted Data (WR Conflicts)**
    - *Reading the value of an uncommitted object might yield an inconsistency*
  - **Dirty Reads** or **Write-then-Read (WR)** Conflicts.
  - **In Greek:** Ασυνεπείς αναγνώσεις

W
R

- **Problem 2: Unrepeatable Reads (RW Conflicts)**
    - *Reading the same object twice might yield an inconsistency*
  - **Read-then-Write (RW) Conflicts** (ή Write-After-Read)
  - **In Greek:** Μη-επαναλήψιμες αναγνώσεις

R
W
R

- **Problem 3: Overwriting Uncommitted Data (WW Conflicts)**
    - *Overwriting an uncommitted object might yield an inconsistency*
  - **Lost Update** or **Write-After-Write (WW)** Conflicts.
  - **In Greek:** Απώλειες ενημερώσεων

W
W

- **Remark:** There is no notion of **RR-Conflict** as no object is changed

Reading Uncommitted Data (WR Conflicts) (Ασυνεπείς αναγνώσεις)

"Reading the value of an uncommitted object yields an inconsistency"

- To illustrate the **WR-conflict** consider the following problem:

  T1: Transfer **$100** from Account **A** to Account **B**

  T2: Add the annual interest of **6%** to both **A** and **B.**

(Correct)Serial Schedule

Trace	T1	T2
	R(A)	
A=A-100	W(A)	
	R(B)	
B=B+100	W(B)	
		R(A)
*A=A*1.06*		W(A)
		R(B)
*B=B*1.06*		W(B)

(Correct)Serial Schedule

Trace	T1	T2
		R(A)
*A=A*1.06*		W(A)
		R(B)
*B=B*1.06*		W(B)
	R(A)	
A=A-100	W(A)	
	R(B)	
B=B+100	W(B)	

WR-Conflict (Wrong)

Trace	T1	T2
	R(A)	
A=A-100	**W(A)**	Dirty Read
		R(A)
*A=A*1.06*		W(A)
		R(B)
B=B*1.06		W(B)
	R(B)	
B=B+100	W(B)	

Problem caused by the WR-Conflict? Account B was credited with the interest on a smaller amount (i.e., 100$ less), thus the result is not equivalent to the serial schedule

Unrepeatable Reads (RW Conflicts)

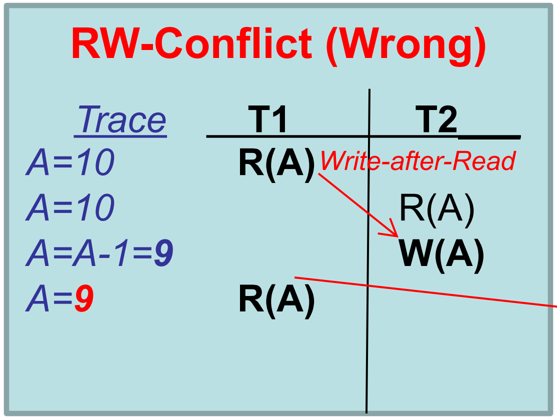
(Μη-επαναλήψιμες αναγνώσεις)

“Reading the same object twice yields an inconsistency”

- To illustrate the **RW-conflict** consider the following problem:

  T1: Print Value of **A**

  T2: Decrease Global counter **A by 1.**



RW-Conflict (Wrong)

Trace	**T1**	**T2**
A=10	**R(A)**	
A=10		R(A)
A=A-1=9		**W(A)**
*A=**9***	**R(A)**	

Write-after-Read

Note that if I read at this point we would see 9 rather than 10 (i.e., read is unrepeatable)

Problem caused by the RW-Conflict?

*Although the **“A” counter is read twice within T1 (without any intermediate change) it has two different values (unrepeatable read)!** … what happens if T2 aborts?*

➔ T1 has shown an incorrect result.

Overwriting Uncommitted Data (WW Conflicts) (Απώλειες ενημερώσεων)

“Overwriting an uncommitted object yields an inconsistency”

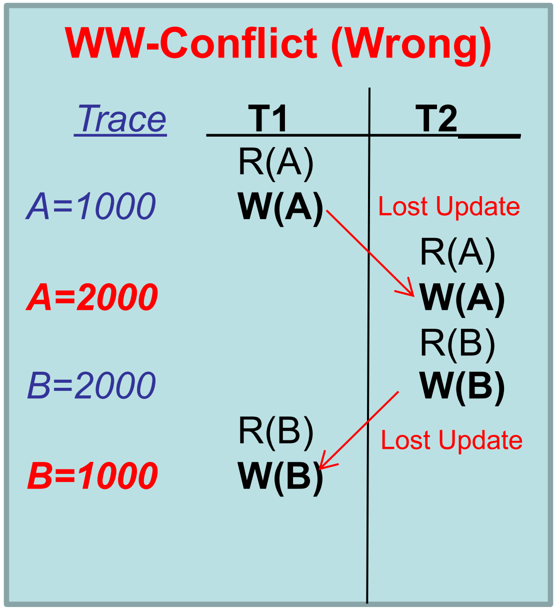
- To illustrate the **WW-conflict** consider the following problem:
  Constr: Salary of employees **A** and **B** must be **kept equal**
  T1: Set Salary to 1000; **T2:** Set Salary equal to 2000

(Correct)Serial Schedule

Trace	T1	T2
	R(A)	
A=1000	W(A)	
	R(B)	
B=1000	W(B)	
		R(A)
A=2000		W(A)
		R(B)
B=2000		W(B)

(Correct)Serial Schedule

Trace	T1	T2
		R(A)
A=2000		W(A)
		R(B)
B=2000		W(B)
	R(A)	
A=1000	W(A)	
	R(B)	
B=1000	W(B)	



WW-Conflict (Wrong)

Trace	T1	T2
	R(A)	
A=1000	**W(A)**	
		R(A)
A=2000		**W(A)**
		R(B)
B=2000		**W(B)**
	R(B)	
B=1000	**W(B)**	

Problem caused by the WW-Conflict?

*Employee “**A**” gets a salary of 2000 while employee **“B” gets a salary of 1000**, thus result is not equivalent to the **serial schedule!***

Lecture Roadmap
Transactions and Schedules

- 16.2) **Transactions** and **Schedules** (Χρονοπρόγραμμα)
  - **Serial Schedule (Σειριακό Χρονοπρόγραμμα) … one after the other…**
  - **Complete Schedule (Πλήρες Χρονοπρόγραμμα) … with Commit, Abort**
- 17.1) **Serializability** (Σειριοποιησιμότητα)
  - **"Correctness Measure"** of some Schedule
  - **Why is it useful?** It answers the question: "Will an interleaved schedule execute correctly"
  - i.e., a Serializable schedule will execute as **correctly as a serial** schedule … but in an **interleaved manner**!
- 17.1) **Recoverability** (Επαναφερσιμότητα)
  - **"Recoverability Measure"** of some Schedule.
  - **Why is it useful?** It answers the question: "Do we need to rollback a some (or all) transactions in an interleaved schedule after some Failure (e.g., ABORT)"
  - i.e., in a Recoverable schedule no transaction needs to be **rolled back (διαδικασία επιστροφής)** once committed!

Transactions and Schedules
(Δοσοληψίες και Χρονοπρόγραμμα)

- **Serial Schedule (Σειριακό Χρονοπρόγραμμα)**
  - A schedule in which the different **transactions** are **NOT interleaved** (i.e., transactions are executed from start to finish one-by-one)

Serial Schedule

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

Serial Schedule

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

N! Possible Serial Schedules, where N the number of Transactions

Transactions and Schedules
(Δοσοληψίες και Χρονοπρόγραμμα)

- **Complete Schedule (Πλήρες Χρονοπρόγραμμα)**
  - A schedule that contains either a **commit (ολοκλήρωση δοσοληψίας)** or an **abort (ματαίωση δοσοληψίας)** action for **EACH** transaction.*

Complete Schedule

T1	T2
R(A)	
	R(B)
W(A)	
	W(B)
Commit	
	Abort

Complete Schedule

T1	T2
R(A)	
W(A)	
	R(B)
Commit	
	W(B)
	Abort

Complete (Serial) Schedule

T1	T2
R(A)	
W(A)	
Commit	
	R(B)
	W(B)
	Abort

* ***Note:*** *consequently, a complete schedule* ***will not*** *contain any* ***active transactions*** *at the* ***end of the schedule***

Transactions and Schedules (Δοσοληψίες και Χρονοπρόγραμμα)

- **Interleaved Schedules** of transactions **improve performance**
  - **Throughput (*ρυθμαπόδοση*):** *More Xacts per seconds; and*
  - **Response Time (χρόνος απόκρισης):** A **short** transaction will not get stuck behind a **long-running** transaction
- Yet it might lead the DB to an **inconsistent** **state as we have shown**
- **Serial schedule** (σειριακό χρονοπρόγραμμα) **is slower but** guarantees consistency (correctness)
- We seek to identify schedules that are:
  - **As fast** as **interleaved** schedules.
  - **As consistent** as **serial** schedules

Transactions and Schedules (Δοσοληψίες και Χρονοπρόγραμμα)

- We shall now **characterize** different schedules based on the following two **properties**:

A. Based on Serializability (Σειριοποιησιμότητα)

- We shall ignore **Commits** and **Aborts** for this section

Characterize which schedules are **correct** when **concurrent transactions** are **executing.**

- **Conflict Serializable Schedule** (Σειριοποιησιμότητα Συγκρούσεων)
- **View Serializable Schedule** (Σειριοποιησιμότητα Όψεων)

B. Based on Recoverability (Επαναφερσιμότητα)

- **Commits** and **Aborts** become important for this section!

Characterize which schedules **can be recovered and how easily.**

- **Recoverable Schedule** (Επαναφέρσιμο Χρονοπρόγραμμα).
- **Cascadeless schedule** (Χρονοπρ. χωρίς διαδιδόμενη ανάκληση).
- **Strict Schedules** (Αυστηρό Χρονοπρόγραμμα).

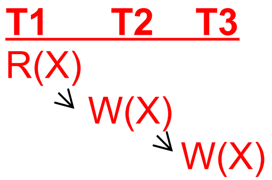
Conflicting Actions
(Συγκρουόμενες Πράξεις)

- **Conflicting Actions (Συγκρουόμενες Πράξεις)**

  Two or more **actions** are said to be **in conflict** if:
  - The actions belong to **different transactions**.
  - **At least** one of the actions is a **write operation**.
  - The actions **access** the **same object** (read or write).

- **The following set of actions is conflicting:**

  T1:R(X), T2:W(X), T3:W(X)



- **While the following sets of actions are not:**

  T1:R(X), T2:R(X), T3:R(X) // No Write on same object

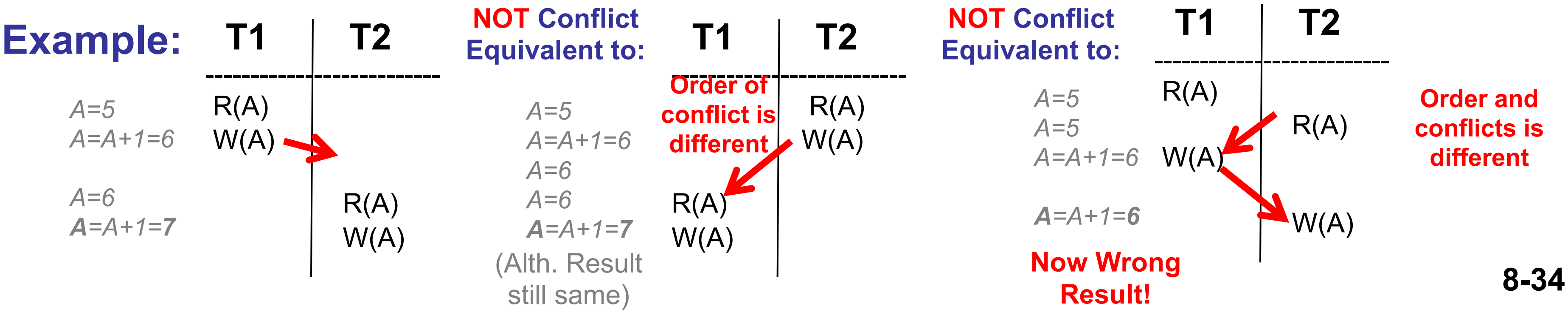
  T1:R(X), T2:W(Y), T3:R(X) // No Write on same object

Characterizing Schedules based on:
Serializability
Recoverability

Conflict Equivalence
(Ισοδυναμία Συγκρούσεων)

- **Conflict Equivalence (Ισοδυναμία Συγκρούσεων)**

  The schedules **S1** and **S2** are said to be **conflict-equivalent** if the following conditions are satisfied:
  - Both schedules **S1** and **S2 involve** the **same set** of **transactions** (including ordering of actions within each transaction).
  - **The order (διάταξη) of each pair of conflicting actions in S1 and S2 are the same.**
- **Why is the order of Conflicts important?** If two **conflicting operations** are **applied in different orders**, the **net effect** can be **different** on the database or on other transactions in the schedule. See example below:



Example:

	T1	T2
A=5	R(A)	
A=A+1=6	W(A)	
A=6		R(A)
A=A+1=7		W(A)

NOT Conflict Equivalent to:

	T1	T2
A=5		R(A)
A=A+1=6		W(A)
A=6		
A=6	R(A)	
A=A+1=7	W(A)	

Order of conflict is different

(Alth. Result still same)

NOT Conflict Equivalent to:

	T1	T2
A=5	R(A)	
A=5		R(A)
A=A+1=6	W(A)	
A=A+1=6		W(A)

Order and conflicts is different

Now Wrong Result!

* Note that non-conflicting operations can arbitrary be **swapped around without compromising the order**.

Conflict Serializability
(Σειριοποιησιμότητα Συγκρούσεων)

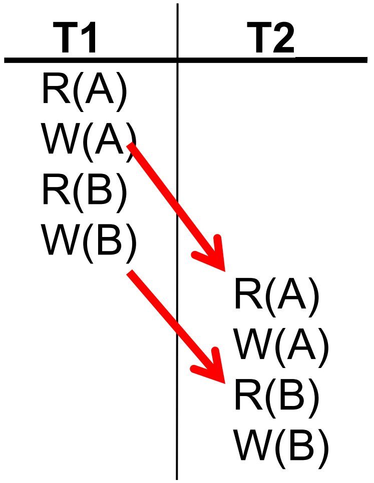
- **Conflict Serializability (Σειριοποιήσιμο Συγκρούσεων)**

  When the schedule is **conflict-equivalent (ισοδύναμο συγκρούσεων)** to **some (any!) serial schedule**.

Serializable == Conflict Serializable

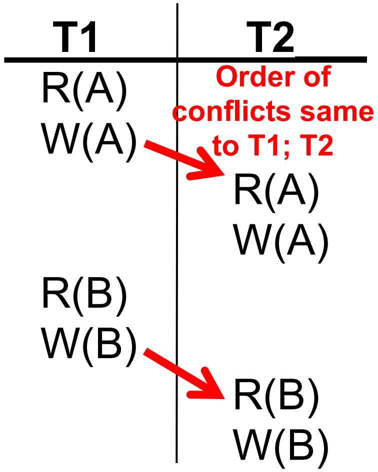
(that definition is in some textbooks different)

Serial Schedule



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

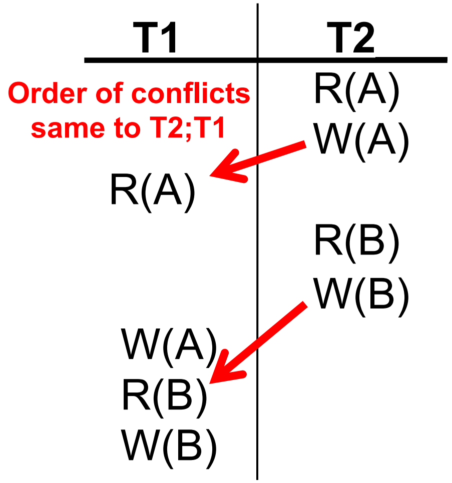
Serializable Schedule A



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

Order of conflicts same to T1; T2

Serializable Schedule B



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

Order of conflicts same to T2;T1

Conflict Serializability
(Σειριοποιησιμότητα Συγκρούσεων)

- **Why is Conflict Serializability important?**
- We have already said that any **serial schedule** leaves the DB in a **consistent (correct)** state, but is **inefficient**
  - i.e., T1; T2 is as correct as T2; T1 (although they might have a different outcome).

Serializable != Serial: NOT the same thing

- Being **Serializable** implies:

A.That the schedule is a correct schedule.
- It will leave the database in a **consistent state**.
- The **interleaving** is **appropriate** and will result in a state as if the transactions were **serially executed**.

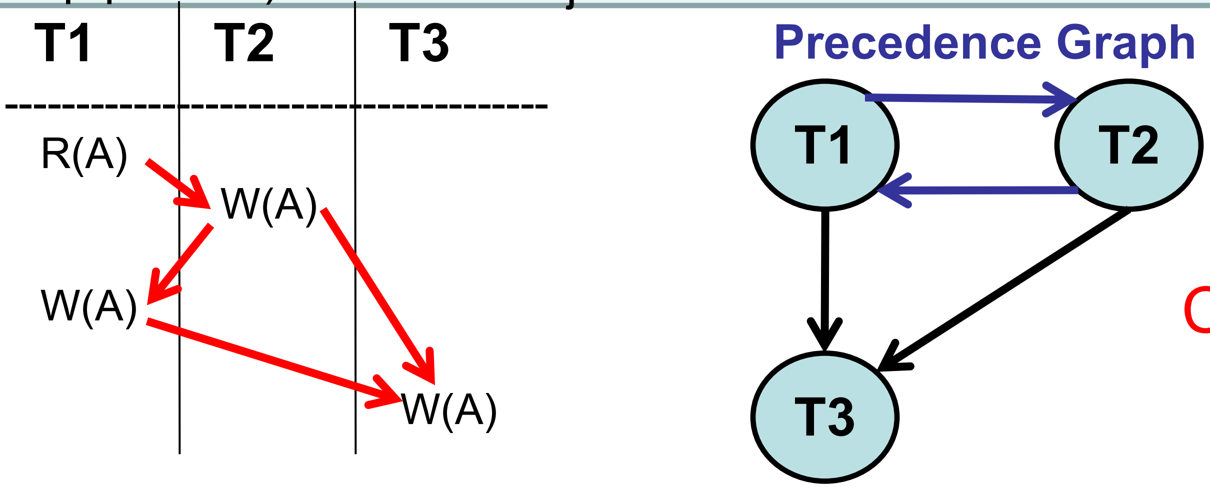
B.That a schedule is a efficient (interleaved) schedule
- That parameter makes it better than Serial ☺!

Testing for Serializability (Έλεγχος **Σειριοποιησιμότητας**)

- **How can we test if a schedule is Conflict Serializable?**
  - There is a simple algorithm **detailed next** (that is founded on a **Precedence Graph**)
- **Does the DBMS utilizes this algorithm? NO**
  - We detail it only to gain a better understanding of the definitions.
- **Why is the DBMS not using it?**
  - Serializability is hard to check at runtime
    - Difficult to **determine beforehand** how the **operations** in a schedule will be **interleaved** (as it depends on the OS)
- Subsequently, we will see that a DBMS utilizes a set of **protocols** (e.g., **2PL or other Concurrency Control techniques w/out locking**), which **guarantee** that a schedule is always **serializable**.

Precedence Graph (Γράφος Προτεραιότητας)

- **Why is it useful?** To find if a schedule is Conflict Serializable
- A **Precedence Graph (Γράφος Προτεραιότητας)** for a schedule S contains:
  - A **node** for each **transaction** in S
  - An arch from **T_i** to **T_j**, if an action of **T_i** precedes (προηγείται) and conflicts (συγκρούεται) with one of **T_j's** actions.



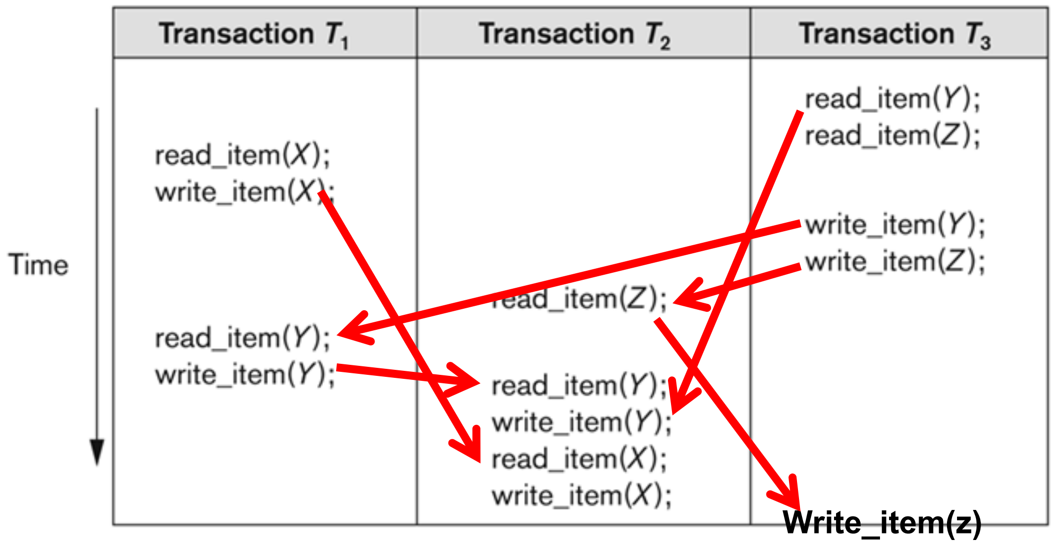
T1	T2	T3
R(A)		
	W(A)	
W(A)		
		W(A)

Precedence Graph

Cycle exists!

- A schedule S is **conflict serializable** if and only if its **precedence graph** is **acyclic**.
  - The above schedule is **not** Conflict Serializable!

Conflict Serializability Testing
(Έλεγχος Σειριοποιησιμότητας Συγκρούσεων)



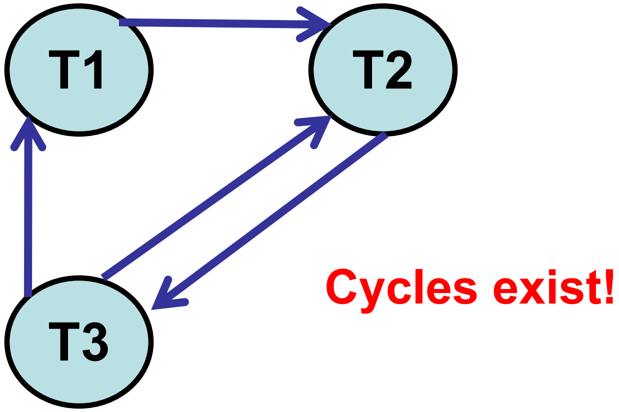
Transaction T_1	Transaction T_2	Transaction T_3
		read_item(Y); read_item(Z);
read_item(X); write_item(X);		
		write_item(Y); write_item(Z);
	read_item(Z);	
read_item(Y); write_item(Y);		

	read_item(Y);
	write_item(Y);
	read_item(X);
	write_item(X);	

		Write_item(z)

Schedule F

Precedence Graph



Above schedule is NOT Conflict Serializable!

Although efficient (interleaved) the above might NOT produce a correct result!

Introduction to Recoverability
(Εισαγωγή στην Επαναφερσιμότητα)

- So far we have characterized schedules based on **serializability (σειριοποιησιμότητα),** i.e., correctness.
- Now it is **time** to characterize schedules based on **recoverability (επαναφερσιμότητα)**
- **Why is this important?**
  - For some schedules it is easier to recover from transaction failures than others.
- In summary, a **Recoverable Schedule (Επαναφέρσιμο Χρονοπρόγραμμα)** is a schedule where no transaction needs to be **rolled back (διαδικασία επιστροφής)** once committed.
- Commit/Abort points now become quite important!

Recoverable Schedule (Επαναφέρσιμο Χρονοπρόγραμμα)

- Recoverable Schedule (Επαναφέρσιμο Χρονοπρόγραμμα)
  - A **schedule** S is **recoverable** if no transaction T in S commits until all transactions T', that have written an item that T **reads, have committed.**
    - **Rule:** In other words, the parents of **dirty reads** need to commit before their children can commit

Consider the Following schedule:

T1	T2
R(X)	
W(X)	
	R(X)
R(Y)	
	W(X)
	Commit
Abort	

dirty read

T1 should have committed first!

Is this schedule recoverable?

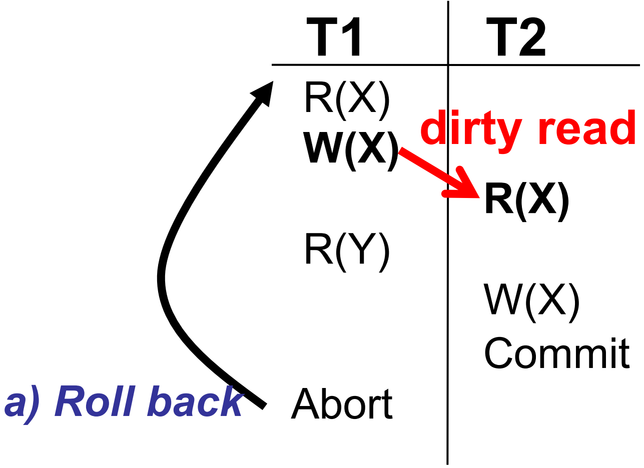
Answer: NO

Why NOT recoverable?

- Because **T2** made a **dirty read** and committed before **T1 … next slide** explains why this is a problem …

Recoverable Schedule (Επαναφέρσιμο Χρονοπρόγραμμα)

- **But why is the schedule Nonrecoverable (Μη-επαναφέρσιμο)?**
  - Because when the **recovery manager** rolls back (step a) **T1** then A gets its initial value.
  - But T2 has already utilized this wrong value and committed something to the DB
  - The DB is consequently in an inconsistent state!



T1	T2
R(X)	
W(X)	
	R(X)
R(Y)	
	W(X)
	Commit
Abort	

dirty read

a) Roll back

B) Now DB is inconsistent! (because X was committed based on T1's aborted transaction)

Nonrecoverable

Recoverable Schedule (Επαναφέρσιμο Χρονοπρόγραμμα)

- **How can we make the Schedule Recoverable?**

Initial Unrecoverable Schedule

dirty read

T1	T2
R(X)	
W(X)	
	R(X)
R(Y)	
	W(X)
	Commit
Abort	

Nonrecoverable

Recoverable Schedule A

dirty read

T1	T2
R(X)	
W(X)	
	R(X)
R(Y)	
	W(X)
Abort	
	Commit

Commit after parent of dirty read

Recoverable

Recoverable Schedule B

T1	T2
R(X)	
	R(X)
W(X)	
R(Y)	
	W(X)
	Commit
Abort	

Not Serializable
(order of conflicting actions has changed)

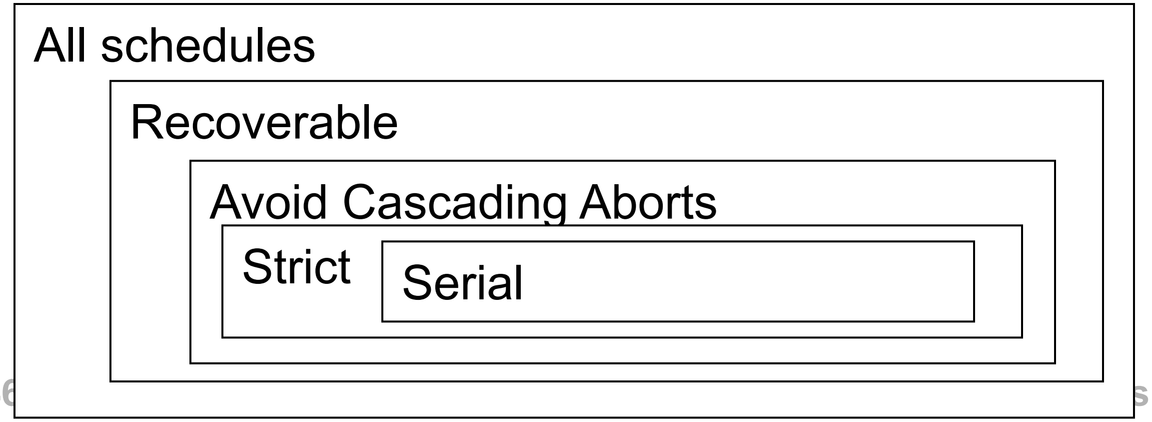
Remove Dirty Read

Recoverable

Other Schedules based on Recoverability (Άλλα χρονοπρογράμματα βάση Επαναφερσιμότητας)

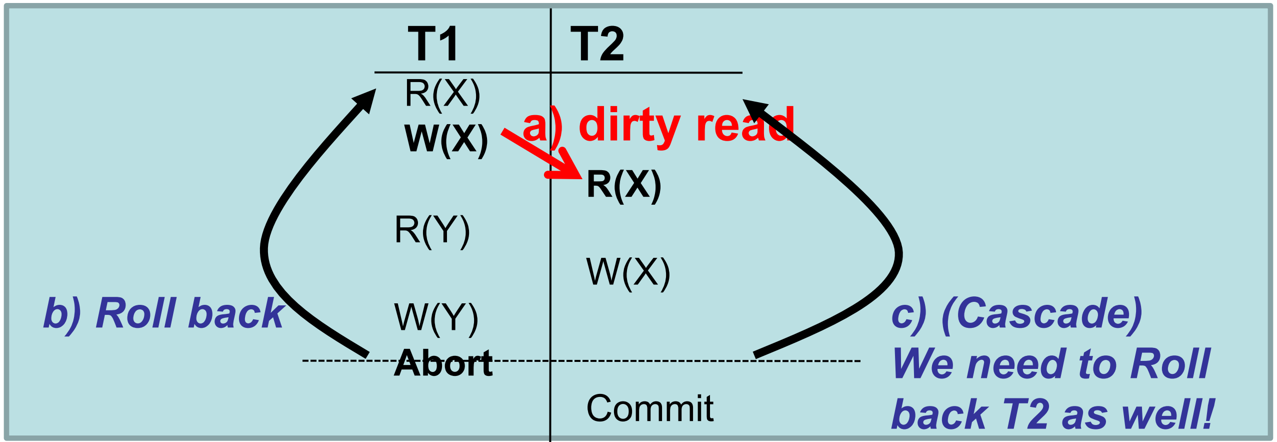
- There are **more strict types** of Schedules (based on the **Recoverability properties**)
  - **Cascadeless schedule (Χρονοπρόγραμμα χωρίς διαδιδόμενη ανάκληση):**

    (or Schedule that **Avoids Cascading Rollbacks**)
    - Refers to cases where we have aborts.
  - **Strict Schedules (Αυστηρό Χρονοπρόγραμμα)**
    - These schedules are very simple to be recovered!
    - Thus, the DBMS prefers this class of Schedules.



Cascadeless Schedule (Χρονοπρόγραμμα χωρίς διαδιδόμενη ανάκληση)

- **Cascadeless schedule (Χρονοπρόγραμμα χωρίς διαδιδόμενη ανάκληση):** a Schedule that **Avoids Cascading Rollbacks**)
  - **One where a rollback does not cascade to other Xacts**
  - **Why is this necessary? Rollbacks are Costly!**
  - **How can we achieve it?** Every transaction **reads** only the items that are **written** by **committed** transactions.



T1	T2
R(X)	
W(X)	
	R(X)
R(Y)	
	W(X)
W(Y)	
Abort	
	Commit

a) dirty read

b) Roll back

c) (Cascade) We need to Roll back T2 as well!

NOT Cascadeless (but Recoverable)

Cascadeless Schedule (Χρονοπρόγραμμα χωρίς διαδιδόμενη ανάκληση)

- **Let us turn the previous example into a Cascadeless Schedule**
  - Recall, in order to get a Cascadeless Schedule, every transaction **must read** only **committed data**

Cascadeless Schedule

T1	T2
R(X)	
W(X)	
R(Y)	
W(Y)	
Abort	
	R(X)
	W(X)
	Commit

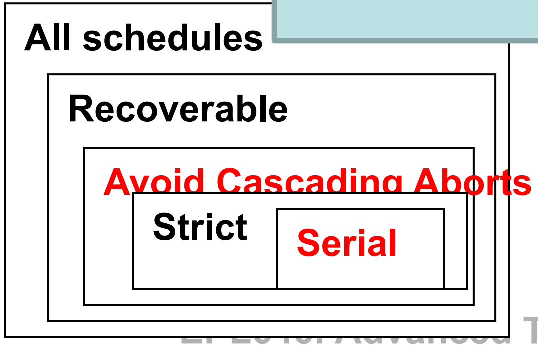
a) Roll back

b) Now T2 reads the X value that existed before T1 started.

Another Cascadeless Schedule

T1	T2
R(X)	
W(X)	
R(Y)	
	W(X)
W(Y)	
Abort	
	Commit

… but not strict



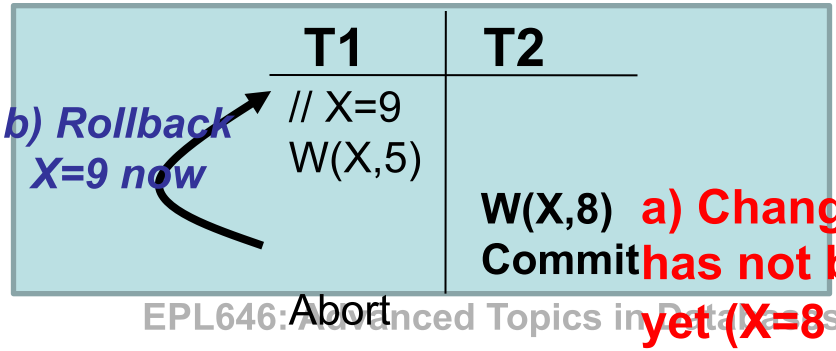
Strict Schedule
(Αυστηρό Χρονοπρόγραμμα)

- **Strict Schedule (Αυστηρό Χρονοπρόγραμμα):**
- **A schedule is strict if overriding of uncommitted data is not allowed.**
- Formally, if it satisfies the following conditions:
  - Tj **reads** a data item X ***after*** Ti has terminated (aborted or committed)
  - Tj **writes** a data item X ***after*** Ti has terminated (aborted or committed)

$T_i \searrow T_j$

- Why is this necessary? **Eliminates Rollbacks!**
  - If a schedule is strict, a rollback can be achieved simply by resetting the Xact variables to the value before its start value, e.g.,

Cascadeless but NOT Strict



T1	T2
// X=9	
W(X,5)	
	W(X,8)
	Commit
Abort	

b) Rollback X=9 now

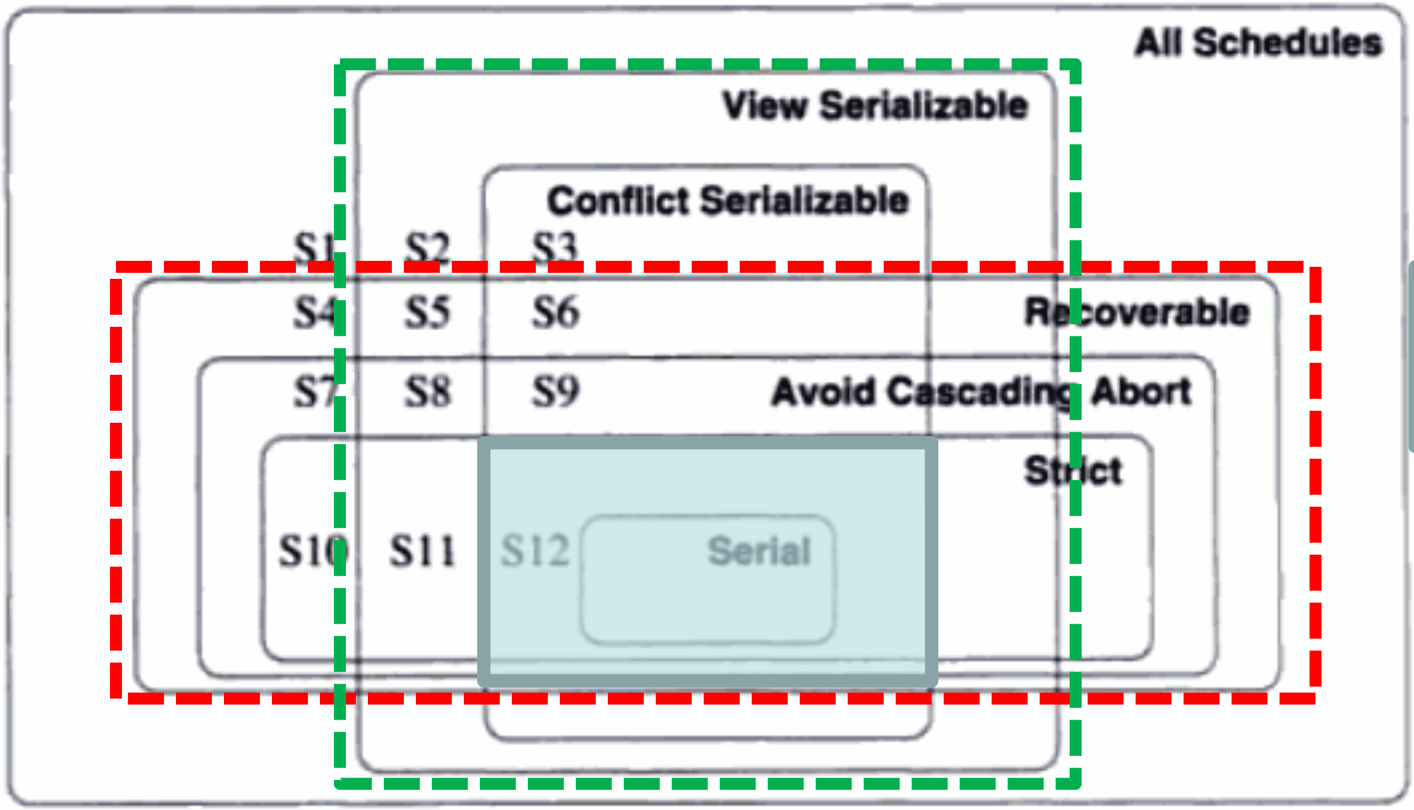
a) Changing an item that has not been committed yet (X=8 now)

c) Why is this a problem?

Because X now became again 9 rather than X=8 (committed)!

Characterizing Schedules
(Χαρακτηρίζοντας Χρονοπρογράμματα)

- **Venn Diagram** Illustrating the different ways to characterize a Schedule based on Serializability and Recoverability



Focus of DB Schedules