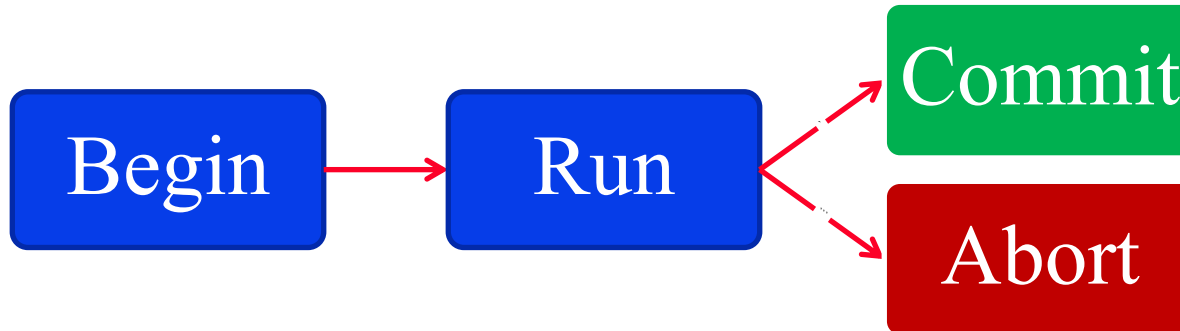


Δοσοληψίες

Γ. Κωτίδης

Transaction

- ◆ Programming abstraction
- ◆ Implement real-world transactions
 - » Banking transaction
 - » Airline reservation



Transactions

- ◆ Transactions are concepts that allow a system to guarantee certain semantic properties.
- ◆ These guarantees must be rigorously defined so that people can build correct systems above them.
- ◆ Theory meets practice here in a nice way.

History

- ◆ System R team, led by Jim Gray, codified the formal notion of transactions and serializability.
 - » Led to 1998 Turing Award for Gray.
- ◆ Christos Papadimitriou did early work formalizing theoretical results on transactions and has a book on the topic:
 - » C. Papadimitriou: The Theory of Database Concurrency Control, CS Press, 1988
- ◆ Notion of transaction processing exists outside of DBs
 - » Distributed applications and services. (Jim Gray, Andreas Reuter: Transaction Processing: Concepts and Techniques. M. Kaufmann, 1993)
 - » Web sessions and services.
- ◆ When the strict versions of these concepts are absolutely needed, and when in practice they can be (and are) relaxed?
 - » Business Intelligence: small “errors” are statistically insignificant
 - » NOSQL/Big data systems (**BASE**: Basically Available, Soft state, Eventual consistency vs **ACID**: discussed in what follows)

DBMS guarantees **ACID** Properties

◆ Atomicity

- » All changes of the transaction recorded or none at all

◆ Consistency

- » The database should start out "consistent" (legal state), and at the end of transaction remain "consistent"

◆ Isolation

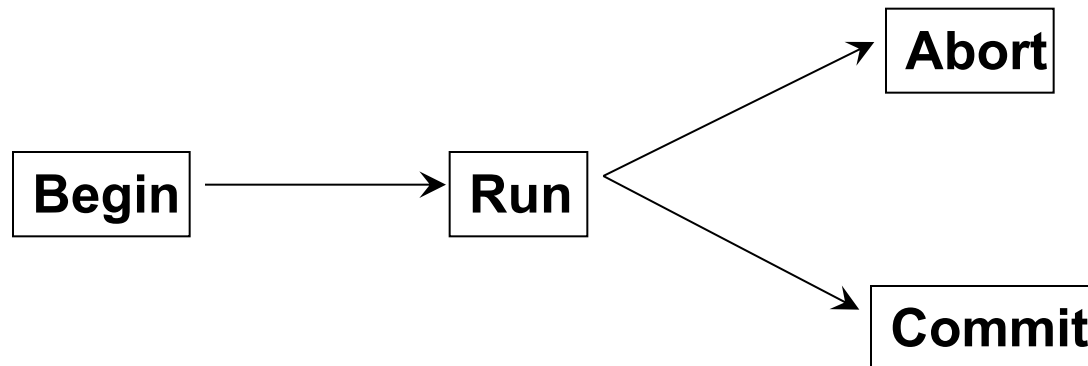
- » Net effect as if the transaction executed in isolation

◆ Durability

- » All future transactions see the changes made by this transaction if it completes

Atomicity

- ◆ The "**all or nothing**" property.
 - » Programmer needn't worry about partial states persisting.
 - » Two possible outcomes: transaction commits or rollbacks (aborts)



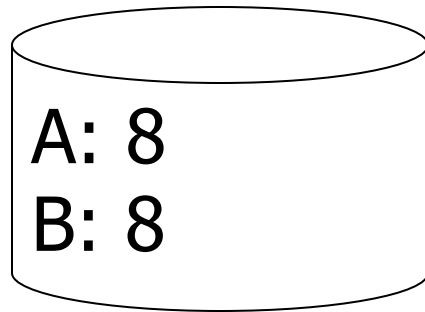
- ◆ Examples:
 - » T1: Delete person from consultants table, insert person into employees table
 - » T2: Transfer funds from account A to account B

Preliminaries

- ◆ **Read(A,t)**: Αντίγραψε το περιεχόμενο της εγγραφής A στην μεταβλητή t . Αν η A δε βρίσκεται στη μνήμη, πρώτα διάβασε την από το δίσκο στη μνήμη.
- ◆ **Write(A,t)**: Αντίγραψε το περιεχόμενο της μεταβλητής t στο αντίγραφο της εγγραφής A που έχεις στη μνήμη.
- ◆ **Output(A)**: Εφόσον η εγγραφή A έχει τροποποιηθεί στη μνήμη, αντίγραψε το περιεχόμενο της μνήμης πίσω στο δίσκο.

Example

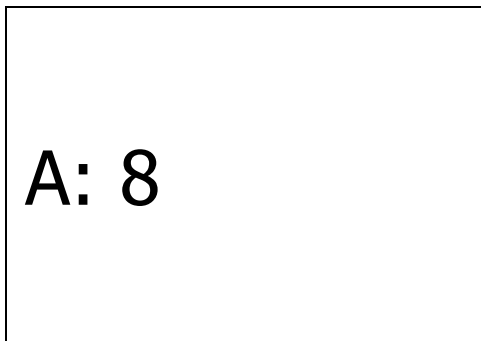
- ◆ Transaction T1: double the amounts of user accounts A, B



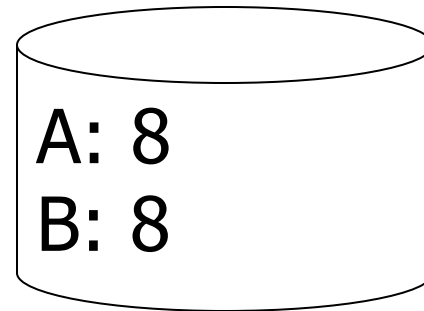
disk

Atomicity: execute all actions of a transaction or none at all

→ T₁: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



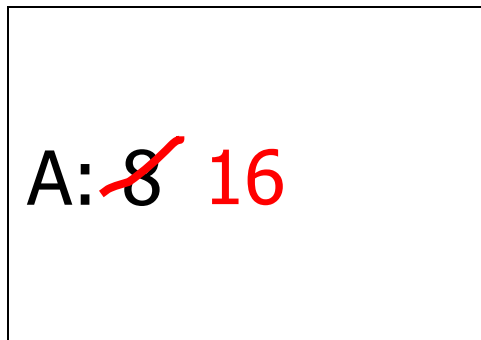
memory



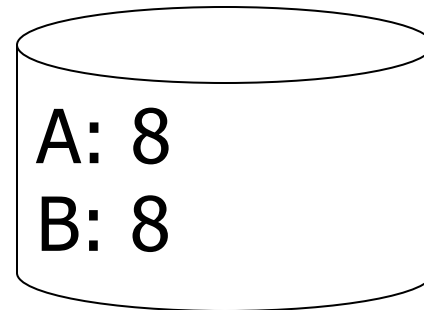
disk

Atomicity: execute all actions of a transaction or none at all

T₁: Read (A,t); $t \leftarrow t \times 2$
→ Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



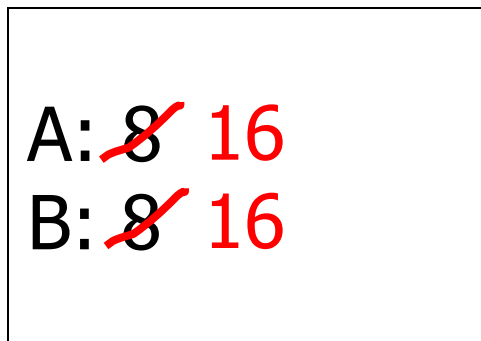
memory



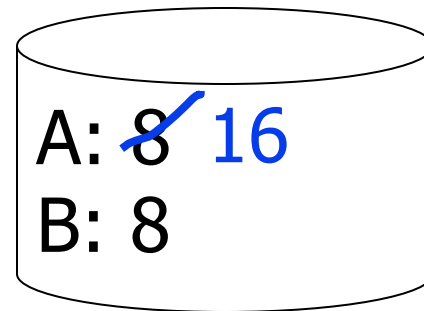
disk

Atomicity: execute all actions of a transaction or none at all

T₁:
Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
→ Output (A);
~~Output (B),~~ failure!



memory



disk

Types of failures

- ◆ **Transaction failure:** Transactions may fail because of incorrect input, deadlock, violation of constraints, etc.
- ◆ **System failure:** System may fail because of operating system fault, RAM failure, etc.
- ◆ **Media failure:** Disk head crash, power disruption kills disk, power surge fries SSD, etc.
 - » For this we need redundant copies of the database

One solution: Undo Logging

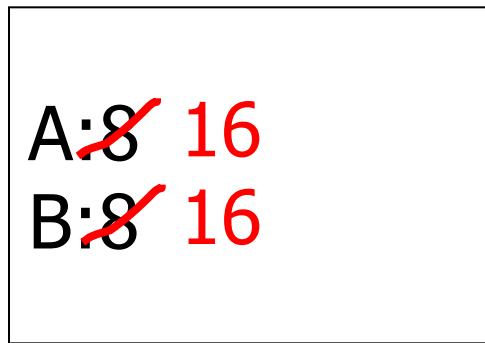
- ◆ If we are not sure that work for some transactions has been completed and stored on disk
 - » **Undo** all actions of the transaction and return DB in a prior state
- ◆ If work has been completed do nothing

Undo-Logging Rules

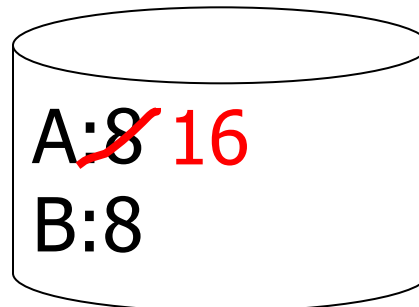
- ◆ Log records with **old** values must be written before **new** values appear on disk
 - » So that we can recover old value when something goes wrong
- ◆ COMMIT log record must be written only after all changes are flushed to disk
 - » But as soon thereafter as possible
 - » **Acts as a proof that work has been completed and flushed to disk**
- ◆ Thus, if I see a COMMIT in the log I am sure that all changes were flushed to disk

Undo logging

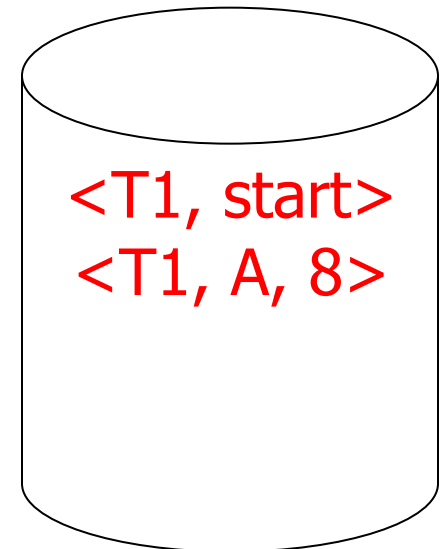
T1: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
→ Output (A);
Output (B);



memory



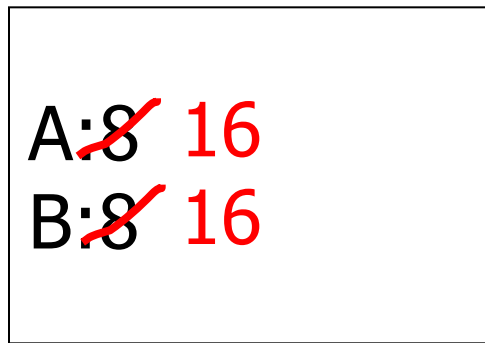
disk



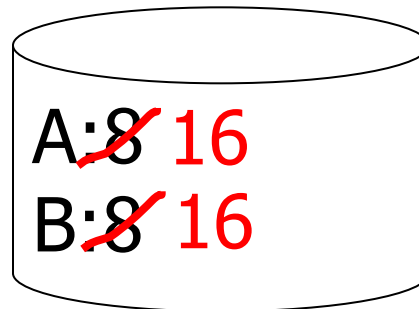
log

Undo logging

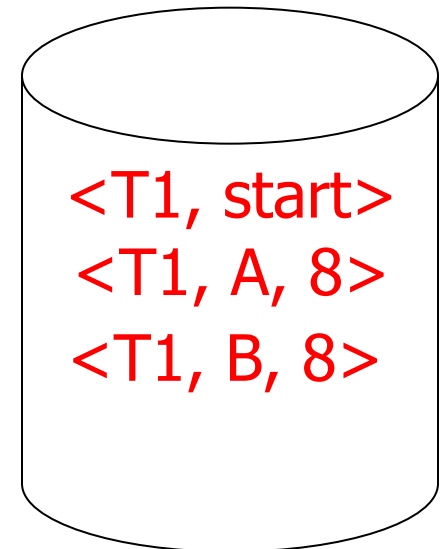
T1: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
→ Output (B);



memory



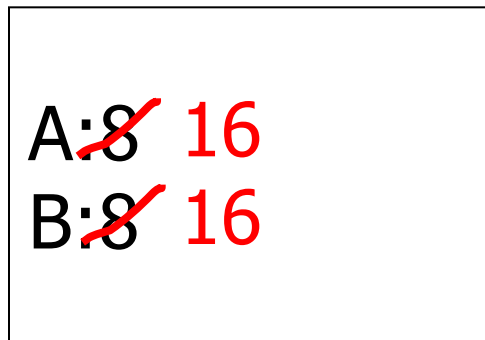
disk



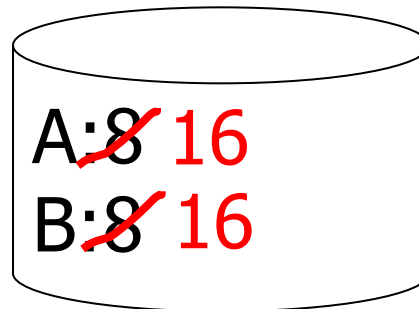
log

Undo logging

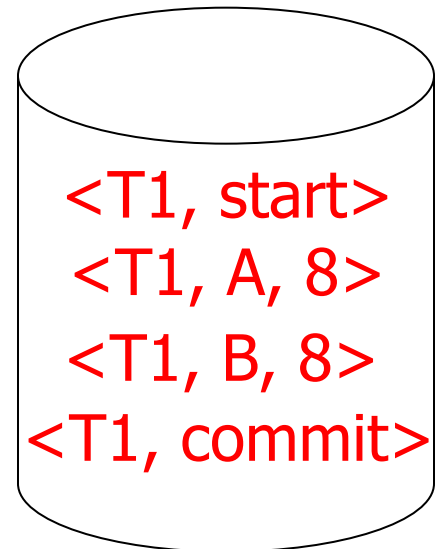
T1: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



memory



disk



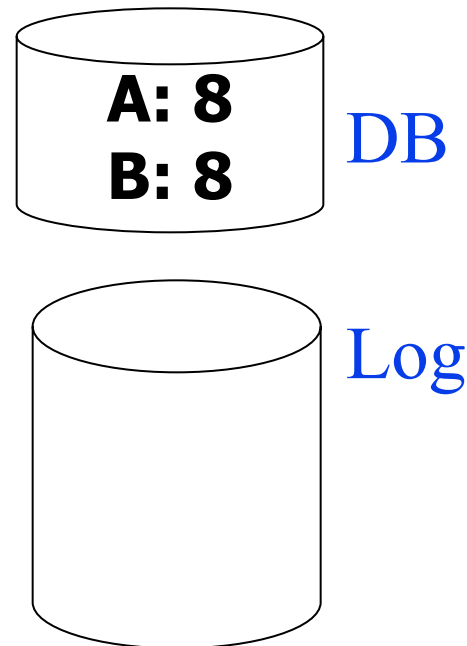
log

One “complication”

- ◆ Log is first written in memory
- ◆ Not written to disk on every action (why?)

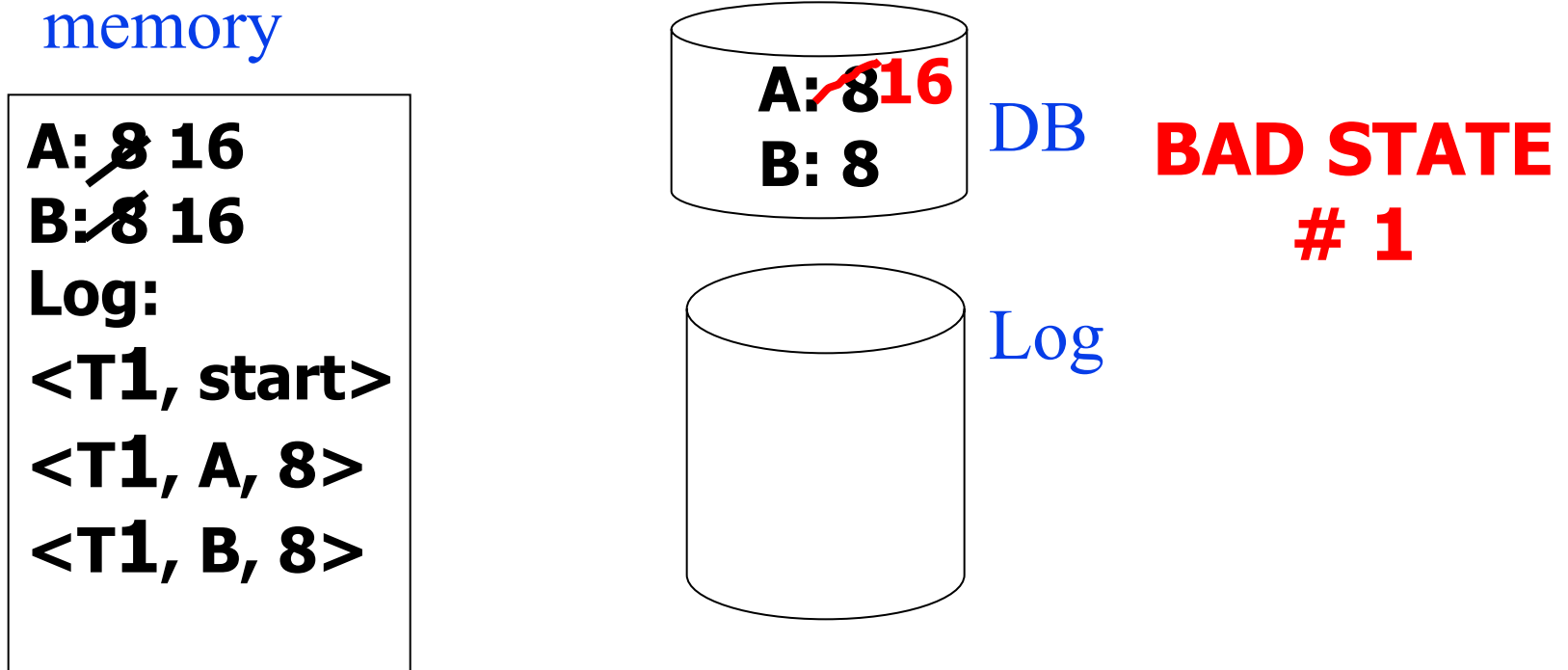
memory

```
A: 8 16  
B: 8 16  
Log:  
<T1, start>  
<T1, A, 8>  
<T1, B, 8>
```



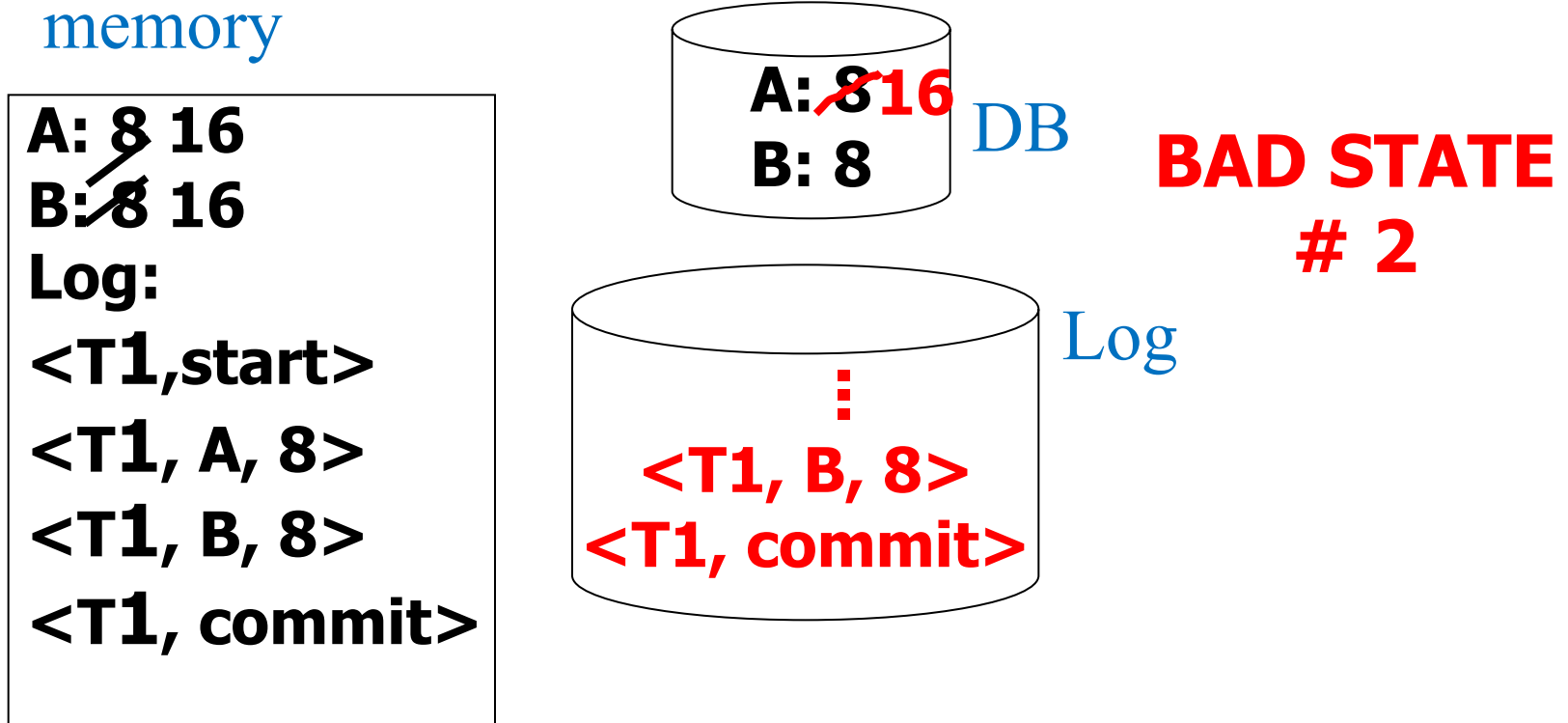
Data page flushed before relevant log entry

- ◆ What happens if there is a crash?



Commit flushed while not all changes have been written back to disk

- ◆ What happens if there is a crash?



Undo logging rules

- (1) For every action generate undo log record (containing old value)
- (2) Before x is modified on disk, log records pertaining to x must be on disk (**Write Ahead Logging/WAL**)
- (3) Before commit is flushed to log, all writes of transaction must be reflected on disk

Now suppose we have a failure

- ◆ Need to find transactions whose actions have not been committed to disk and undo them (due to atomicity property)
- ◆ Committed transactions (in log) are OK (why?)

Recovery rules: Undo logging

- (1) Let S = set of transactions with $\langle T_i, \text{start} \rangle$ in log, but no $\langle T_i, \text{commit} \rangle$ (or $\langle T_i, \text{abort} \rangle$) record in log
- (2) For each $\langle T_i, X, v \rangle$ in log,
in *reverse* order (latest \rightarrow earliest) do:
 - if $T_i \in S$ then $\left\{ \begin{array}{l} \text{write } (X, v) \\ \text{output } (X) \end{array} \right.$
- (3) For each $T_i \in S$ do
 - write $\langle T_i, \text{abort} \rangle$ to log

Example

◆ Undo Log:

<T,start>

<T,A,10>

<W,start>

<T,B,20>

<U,start>

<U,C,30>

<U,commit>

FAILURE

Identify transactions with no commit or abort record

◆ Undo Log:

<T,start>

<T,A,10>

<W,start>

<T,B,20>

<U,start>

<U,C,30>

<U,commit>

Set S={T,W}

Revert Modified Values ($S=\{T,W\}$)

◆ Undo Log:

<T,start>

<T,A,10>

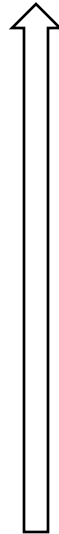
<W,start>

<T,B,20>

<U,start>

<U,C,30>

<U,commit>



Undo Actions:

write(B,20)

output(B)

write(A,10)

output(A)

Append log with abort records

◆ Undo Log:

<T,start>

<T,A,10>

<W,start>

<T,B,20>

<U,start>

<U,C,30>

<U,commit>

<T,abort>

<W,abort>

Complications

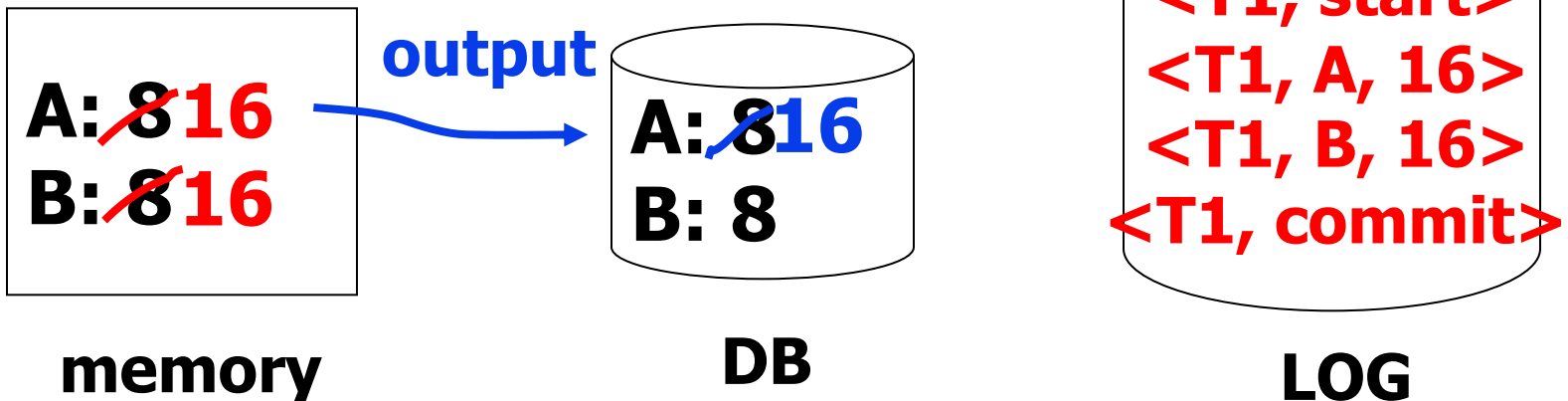
- ◆ Undo logging
 - » cannot bring backup DB copies up to date
 - » **need to output all changes before commit (may induce unnecessary I/Os)!!!**
- ◆ Redo Logging
 - » for every action, generate **redo log** record (containing **new** value)
 - » before X is modified on disk (DB), **all log records for transaction** that modified X (**including commit**) must be on disk
 - » + can play back log to update an old DB copy
 - » - deferred updates: need to keep all modified blocks in memory until commit

Rules of Redo Logging

- ◆ For every action, generate redo log record (containing **new** value)
- ◆ Before X is modified on disk (DB), **all** log records for transaction that modified X (including commit) must be on disk

Redo logging (deferred modification)

T1: Read(A,t); $t \leftarrow t \times 2$; write (A,t);
Read(B,t); $t \leftarrow t \times 2$; write (B,t);
Output(A); Output(B)

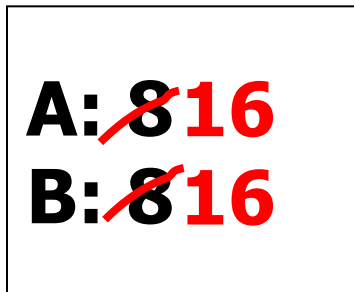


Now suppose we have a failure

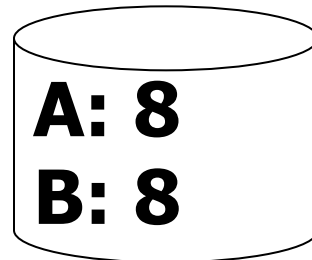
- ◆ If I see a commit log entry I am not sure whether changes are written to disk
 - » Need to redo all such transactions!
 - » But at least I know what to do (all details are in the log)
- ◆ Do not worry about **uncommitted transactions** (**why?**)

Uncommitted transaction

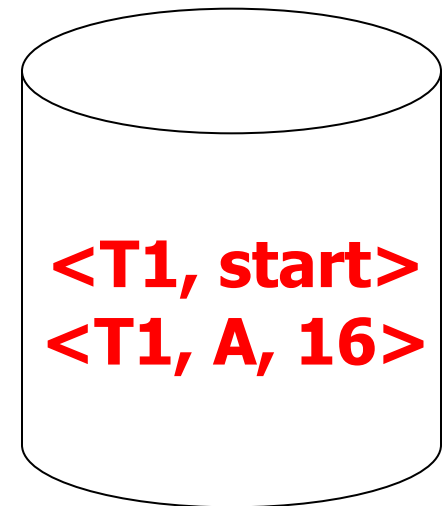
T1: Read(A,t); $t \leftarrow t \times 2$; write (A,t);
Read(B,t); $t \leftarrow t \times 2$; write (B,t);
Output(A); Output(B)



memory



DB



LOG

Recovery (redo logging)

- (1) Let S = set of transactions with
 $\langle T_i, \text{commit} \rangle$ in log
- (2) For each $\langle T_i, X, v \rangle$ in log, in forward
order (earliest \rightarrow latest) do:
 - if $T_i \in S$ then $\left\{ \begin{array}{l} \text{Write}(X, v) \\ \text{Output}(X) \end{array} \right.$

Example

◆ Redo Log:

<T,start>

<T,A,10>

<W,start>

<T,B,20>

<U,start>

<U,C,30>

<U,commit>

FAILURE

Identify committed transactions

◆ Redo Log:

<T,start>

<T,A,10>

<W,start>

<T,B,20>

<U,start>

<U,C,30>

<U,commit>

Set S={U}

Playback Changes (S={U})

◆ Redo Log:

<T,start>

<T,A,10>

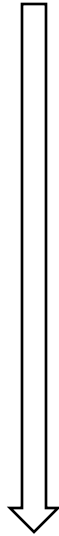
<W,start>

<T,B,20>

<U,start>

<U,C,30>

<U,commit>



Redo Actions:

write(C,30)

output(C)

Comparison

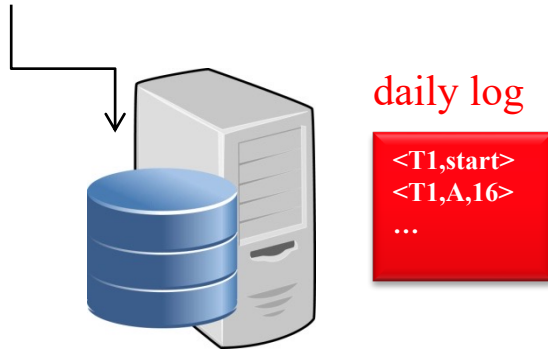
- ◆ Undo logging: need to output changes before commit (may induce unnecessary I/Os)
- ◆ Redo logging: need to keep all modified blocks in memory until commit

ALSO

- ◆ Redo logging: can play log on a DB copy to bring it up to date

Backup database server (+redo log)

Daily transactions



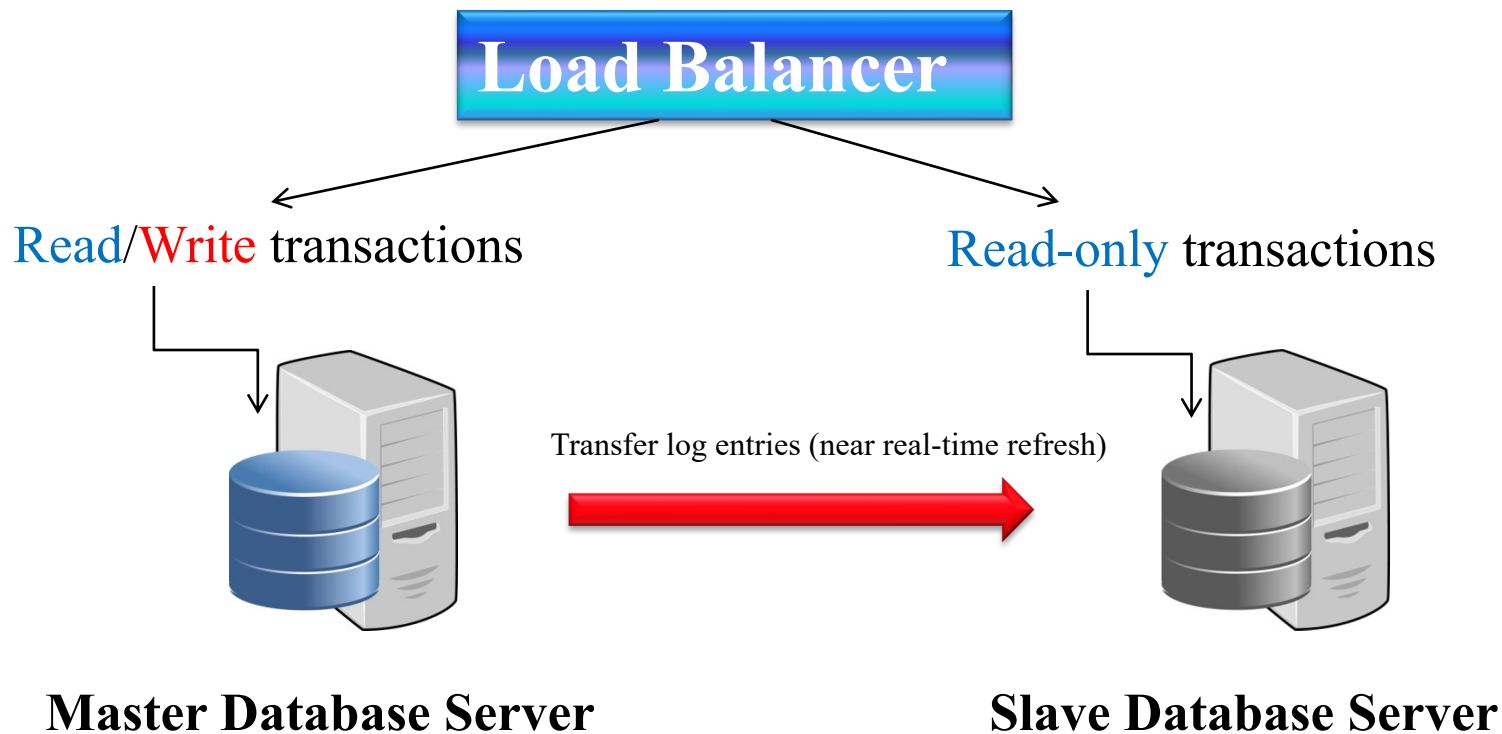
Active Database Server



Backup Database Server
(backup taken this morning)

**How to bring backup
server up to date?**

Similar Idea: Master-Slave Replication



Recall

- ◆ Undo logging: **need to output changes before commit** (may induce unnecessary I/Os)
- ◆ Redo logging: **need to keep all modified blocks in memory until commit** (memory utilization)

Solution: undo/redo logging!

- ◆ Update \Rightarrow $\langle T_i, X_{id}, \text{New } X \text{ val}, \text{Old } X \text{ val} \rangle$
- ◆ page X

Rules

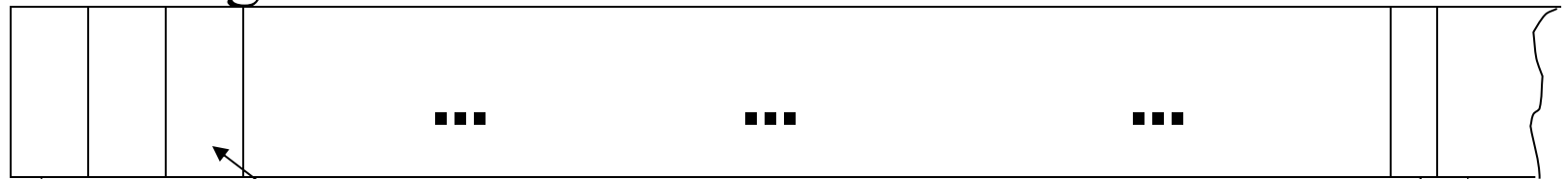
- ◆ Page X can be flushed before or after transaction commits (unlike redo log)
- ◆ Log record flushed before corresponding updated page (WAL)

Recovery Rules

- ◆ Identify transactions that committed
- ◆ Undo uncommitted transactions
- ◆ Redo committed transactions

Recovery is very, very **SLOW** !

Redo log:



First
Record
(1 year ago)

T1 wrote A,B
Committed a year ago
--> STILL, Need to redo after crash!!

Last
Record **Crash**

Checkpoints

- ◆ The entire log file needs to be processed in case of a failure.
 - » Need to redo all committed transactions from start of the log
 - » Need to undo uncommitted transactions, playing back the log in reverse
- ◆ To simplify recovery, we can *checkpoint* the log periodically.
 - » Then, old parts of the log (before the checkpoint) can be discarded

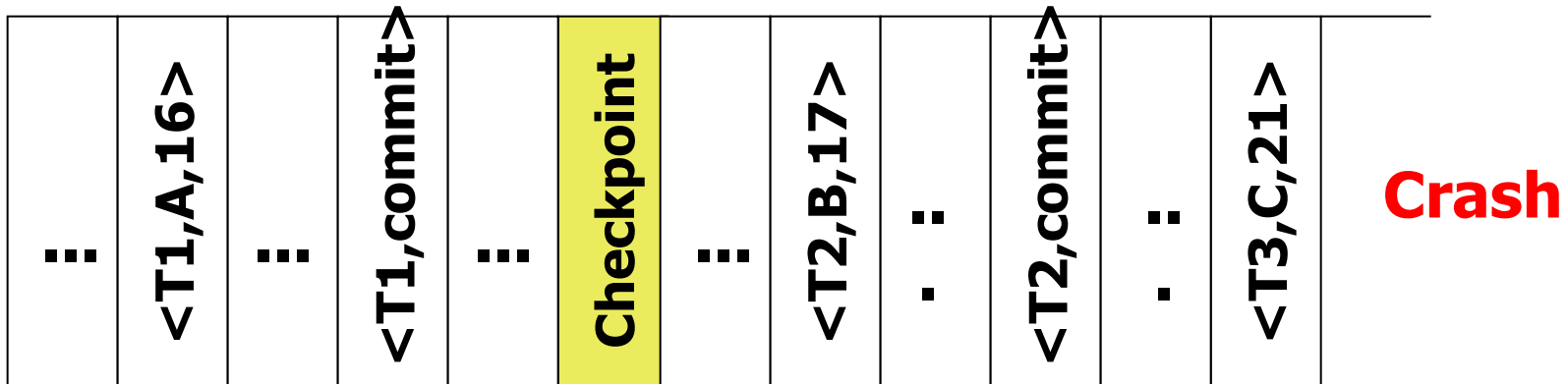
Checkpoint (simplified)

- ◆ Periodically:

- (1) Do not accept new transactions
- (2) Wait until all transactions finish
- (3) Flush all log records to disk (log)
- (4) Flush all buffers to disk (DB) (do not discard buffers)
- (5) Write “checkpoint” record on disk (log)
- (6) Resume transaction processing

Example: what to do at recovery?

Redo log (disk):

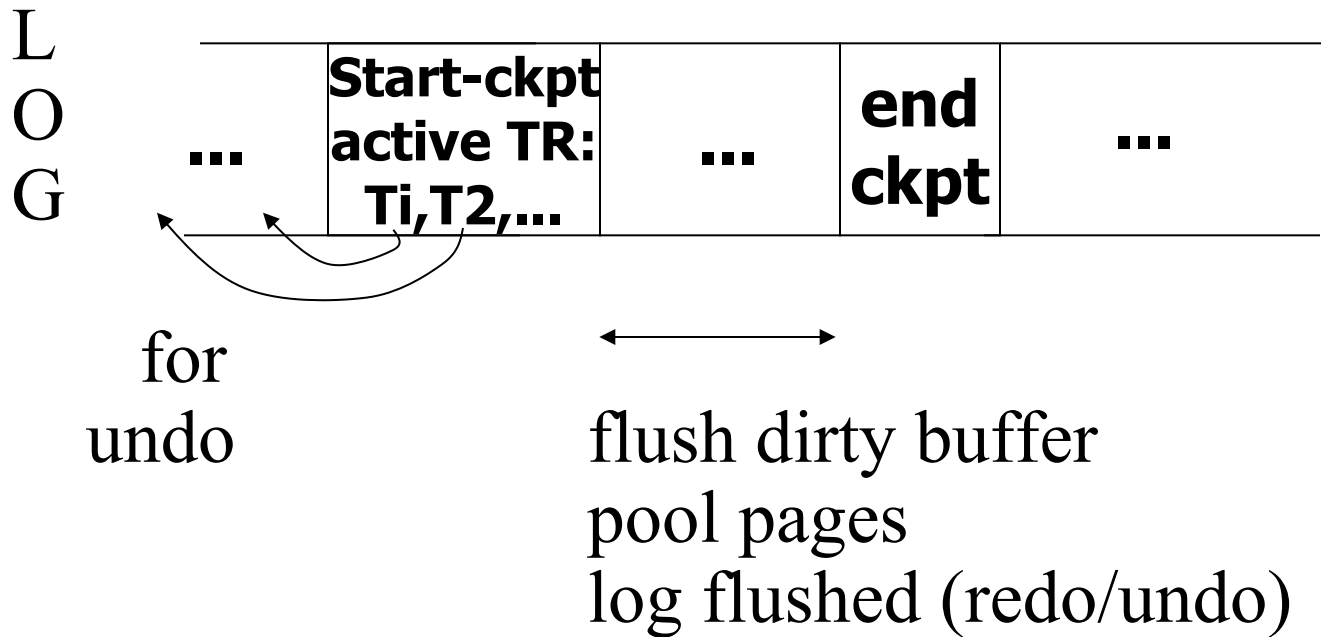


Redo <T2,B,17>

Issue with simple checkpoint

- ◆ The DBMS effectively **halts** until all current transactions have committed or aborted.
 - » This is not desirable in a production system
- ◆ *Nonquiescent checkpointing (μη αδρανή σημεία ελέγχου)* is a technique that avoids this bottleneck.
- ◆ *Nonquiescent checkpointing (undo/redo log)*:
 1. Write a record <START CKPT(T1...Tk)>. T1,...Tk are the active transactions. New transactions are allowed during checkpointing.
 2. Flush log and all pages modified by all active transactions.
 3. Write <END CKPT> to the log.

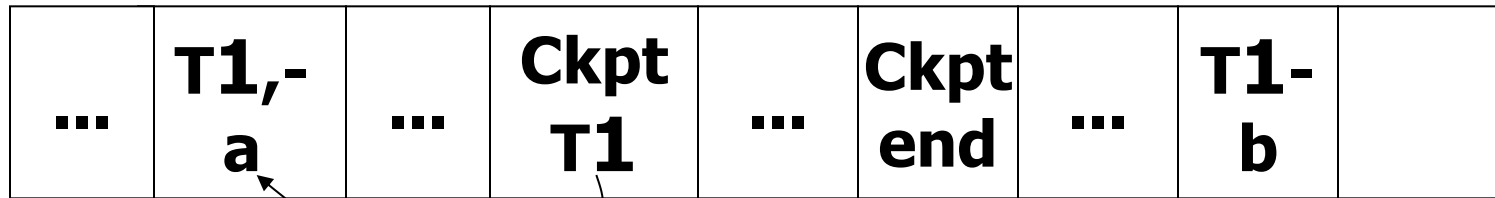
Non-quietescent checkpoint



Examples what to do at recovery time?

no T1 commit

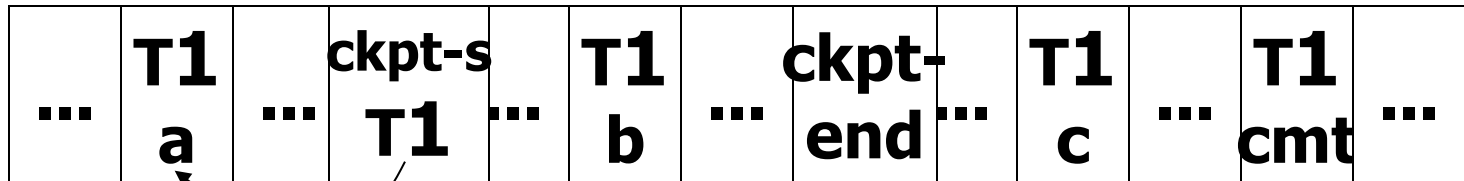
L
O
G



Undo T1 (undo b,a)

Example

L
O
G

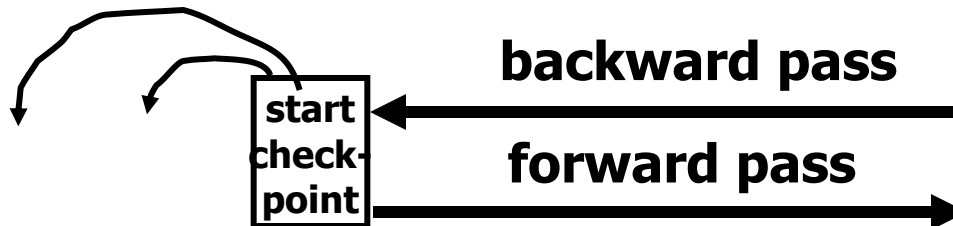


☒ **Redo T1: (redo b,c)**

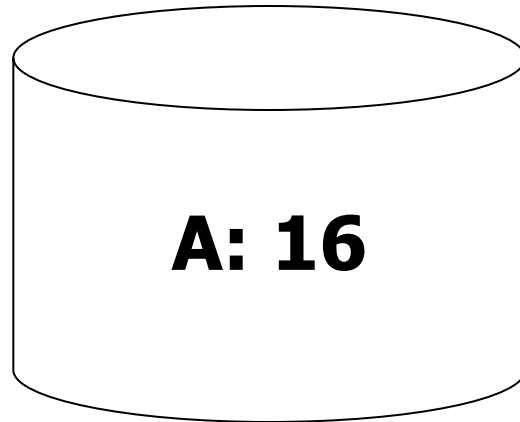
(note: this is for undo/redo case)

Recovery process:

- ◆ Backwards pass (end of log \Rightarrow latest checkpoint start)
 - » construct set S of committed transactions
 - » undo actions of transactions **not in S**
- ◆ Undo pending transactions
 - » follow undo chains for transactions in (checkpoint active list) - S
- ◆ Forward pass (latest checkpoint start \Rightarrow end of log)
 - » redo actions of S transactions

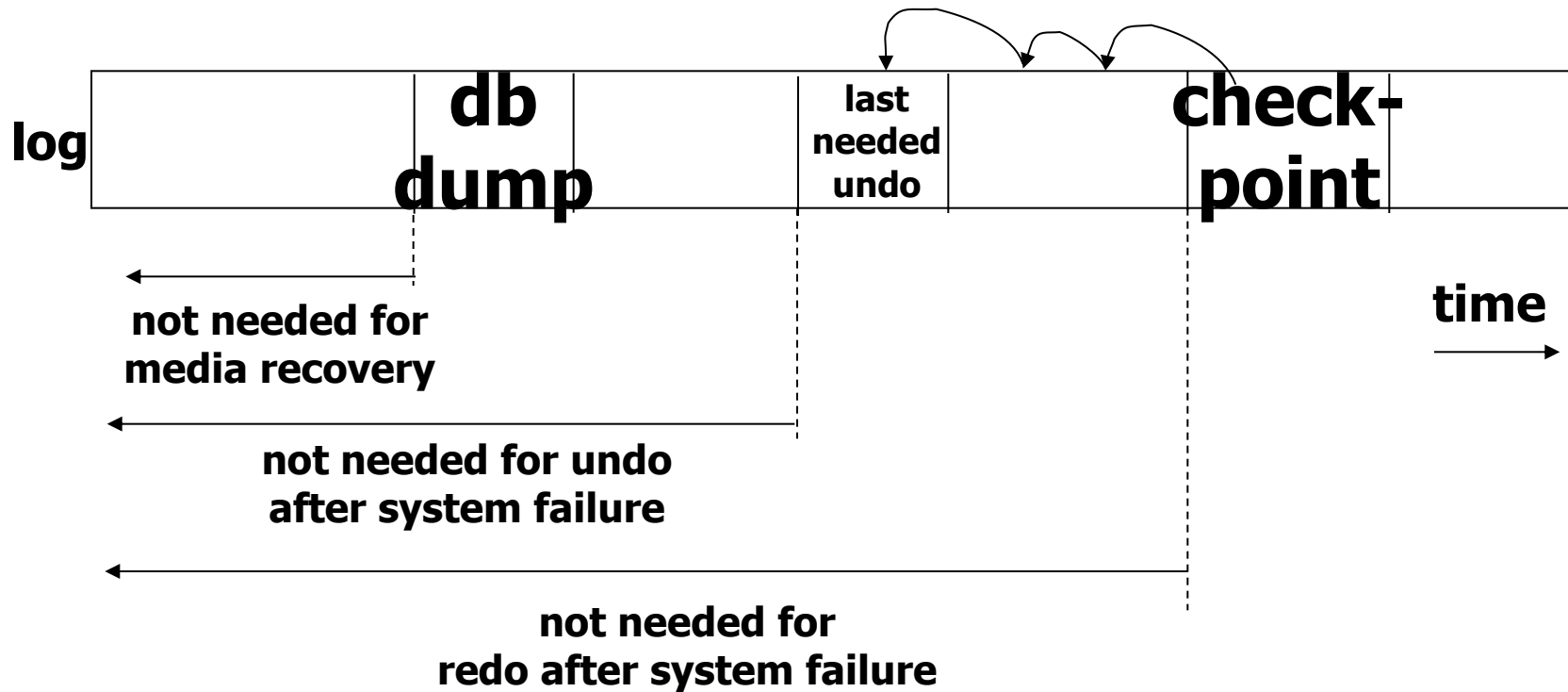


Media failure (loss of non-volatile storage)



Solution: Make copies of data!

Nonquiescent Checkpointing (with db dump)



Consistency (A.C.I.D)

- ◆ The database should start out "consistent" (legal state), and at the end of transaction remain "consistent".
- ◆ The definition of "consistent" is up to the database administrator to define to the system
 - » integrity constraints
 - » other notions of consistency must be handled by the application.

Integrity or correctness of data

- ◆ Would like data to be “accurate” or “correct” at all times

EMP:

Name	Age
John	52
Jim	24
Helen	1

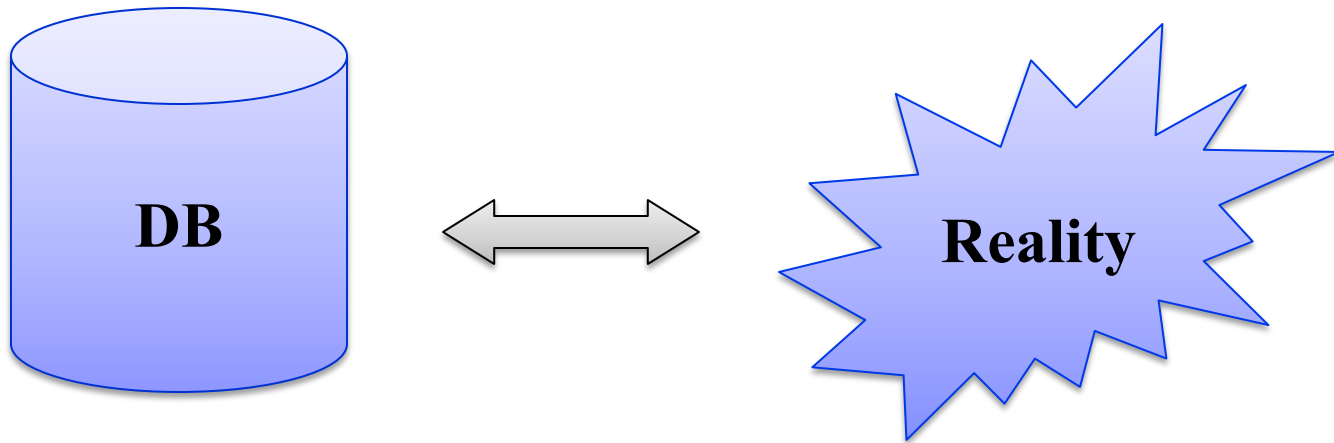
```
CREATE TABLE EMP (  
    Name varchar(255) NOT NULL,  
    Age int,  
    CHECK (Age>=18)  
);
```


Integrity/consistency constraints

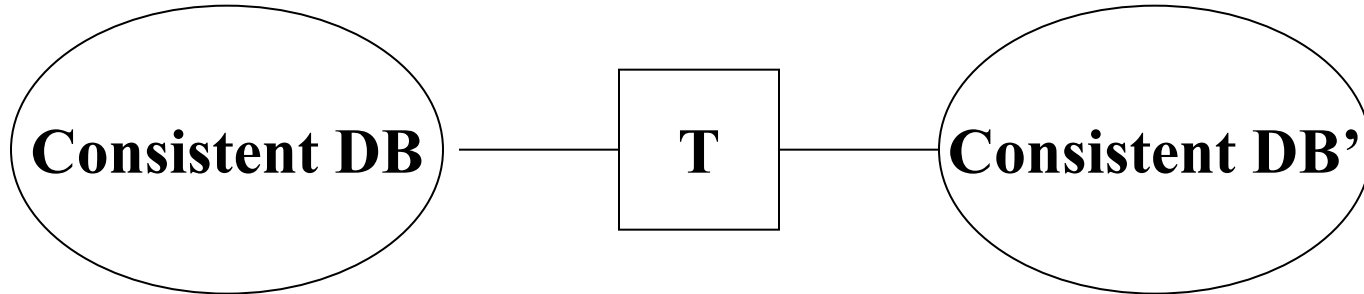
- ◆ Predicates data must satisfy
- ◆ Examples:
 - » $\text{age} \geq 18$ and $\text{age} < 65$
 - » x is key of relation R
 - » $x \rightarrow y$ holds in R
 - » $\text{Domain}(x) = \{\text{Red, Blue, Green}\}$
 - » no employee should make more than twice the average salary

Ultimate Goal

- ◆ Database should reflect real world



But will settle for this...



**Transaction: collection of actions
that preserve consistency**

Isolation (A.C.I.D)

- ◆ Each transaction must **appear** to be executed as if no other transaction is executing at the same time.
- ◆ Transfer funds from A to B (T1).
- ◆ Another teller makes a query on A and B (T2).
- ◆ T2 could see funds on A or B but not in both!
 - » Result may be independent of the time transactions were submitted

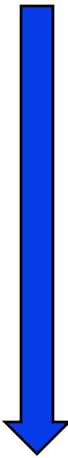
Isolation & Lost Update

- ◆ Consider two transactions
 - » T1: Deposit 200 euros in account A
 - » T2: Deposit 100 euros in the same account
- ◆ What could go wrong?

Assume depicted order of execution

Assume $A=1000$ (time=0)

Time	Transaction T1	Transaction T2
1	read(A,x)	
2	x:=x+200	
3		read(A,y)
4		y:=y+100
5	write(x,A);output(A)	
6		write(y,A);output(A)
7		commit
8	commit	



What is the final outcome?

Isolation Level

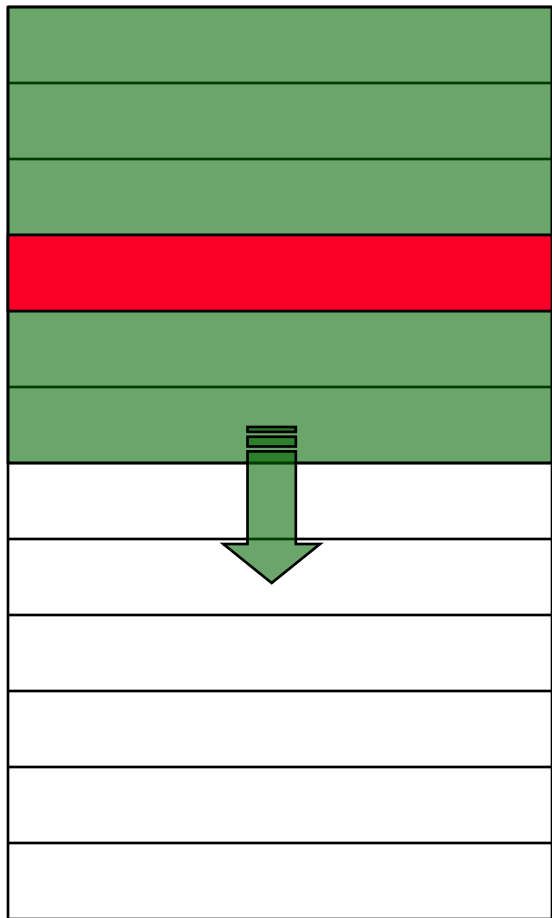
- ◆ Default is **Serializable**
 - » T1->T2 or T2->T1
- ◆ In reality we want to allow parallelism
 - » **Serializable refers to the net-effect**
- ◆ Can be implemented via **locks**
 - » Locks reduce concurrency
 - » Fine granularity of locks reduces risk of deadlocks
 - database, table, tuple
- ◆ Serializability can be guaranteed using locks in a certain fashion
 - » Tests for serializability are redundant!
 - » Rollover when required (e.g. deadlocks)

Is it always worth it?

- ◆ T1: compute average salary per dept of your 20000 employees
- ◆ T2: tries to move employee “Smith” to “Sales” department
 - » Eventually aborted, no office space can be allocated
- ◆ It is worth for T1 to be executed in isolation?
 - » What if we allow “dirty” (=uncommitted) reads?

Example 2

BankData(accountId,bankLocation,amount)



Read Locks

Write Lock

Compute total balance
of all accounts in Athens:

```
Select sum(amount)  
From BankData  
Where bankLocation="Athens"
```

Ενημέρωσε λογαριασμό
ενός πελάτη από στην Αθήνα

```
UPDATE BankData  
SET amount=amount+10  
WHERE accountId='12345'
```

Isolation Levels

- ◆ Read Uncommitted (dirty reads allowed)
- ◆ Read Committed
- ◆ Repeatable Reads
- ◆ Serializable (default)

Read Committed Isolation Level

- ◆ Forbids reading of dirty (uncommitted) data
- ◆ It is possible for a transaction to issue the same query several times and get different answers, as long as the answers reflect data has been written by transactions that already committed

Repeatable Read

- ◆ If a tuple is retrieved the first time, the system guarantees that the identical tuple will be retrieved again if the query is repeated
 - » Insertions that happen while the transaction is executed will be seen (phantom tuples)
- ◆ Assume relation with banking accounts
 - » The balances of accounts I read, will not change every time I make an inquiry within the same transaction
 - » But I will be able to see newly generated accounts

SQL-92 Isolation Levels

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantom Tuples
Read Uncommitted	Yes	Yes	Yes
Read Committed	No	Yes	Yes
Repeatable Read	No	No	Yes
Serializable	No	No	No

Durability (A.C.I.D.)

- ◆ Once committed, the transactions effects should not disappear.
 - » Of course, they may be overwritten by subsequent committed transactions.

Implementation

- ◆ A, C, and D are mostly guaranteed by recovery (usually implemented via logging).
- ◆ I is mostly guaranteed by concurrency control (usually implemented via locking).
- ◆ Of course, life is not so simple. For example, recovery typically requires concurrency control and depends on certain behavior by the buffer manager...

Quick Lesson on Recovery

- ◆ Every change is logged
- ◆ Log manager negotiates with buffer manager to flush changes to non-volatile storage
- ◆ In case of a “problem” (machine crash, violation of “A” and “C” properties) recovery manager examines logs and returns database to some consistent state

Quick Lesson on Concurrency Control

- ◆ Usually implemented via Locking
- ◆ Most OLTP systems measure success via #transactions/second
 - » Airline reservation, banking
- ◆ Required for Isolation

Bottom Line

- ◆ DBMS are monsters
 - » Complicated systems
 - » Guarantee integrity, longevity the data
 - » Goal is to execute as many transactions/secs as possible
- ◆ Is it always worth it?
 - » Want to analyze millions of facebook/twitter records
 - » Do we need strict ACID properties?
 - » **Do we need SQL to do it?**

Note

- ◆ Οι παρακάτω διαφάνειες είναι εκτός ύλης

Top-10 most frequent words with NoSQL (Apache Spark + Scala)

```
val file = sc.textFile("/home/yannisk/wordfile")
val words = file.flatMap(line => line.split(" "))
val pairs=words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

val invCounts = wordCounts.map(x => (x._2,x._1)).sortByKey(false)

invCounts.take(10).foreach(println)
```

Sample text file

Panathinaikos B.C. also known simply as Panathinaikos, or by its full name, Panathinaikos BSA Athens, is the professional basketball team of the major Athens-based multi-sport club Panathinaikos A.O. It is owned by the billionaire Giannakopoulos family.

The parent athletic club was founded in 1908, while the basketball team was created in 1919, being one of the oldest in Greece. Alongside Aris, they are the only un-relegated teams with participation in every Greek First Division Championship until today. Panathinaikos has developed into the most successful basketball club in Greek basketball's history, and among the best in Europe, creating its own dynasty. They have won thirty-seven Greek Basket League Championships, eighteen Greek Cups, ten Doubles (all records), six EuroLeague Championships, one Intercontinental Cup and two Triple Crowns. They also hold the record for most consecutive Greek League titles, as they are the only team to have won nine consecutive championships (2003 - 2011), as well as for the most consecutive Greek Basketball Cup titles (six in a row 2012 to 2017). The team plays in the Olympic Indoor Hall, which has a capacity of 18,989 for basketball games.

Output

```
val file = sc.textFile("/home/yannisk/wordfile")  
val words = file.flatMap(line => line.split(" "))  
val pairs = words.map(word => (word, 1))  
val wordCounts = pairs.reduceByKey(_ + _)
```

```
val invCounts = wordCounts.map(x => (x._2, x._1)).sortByKey(false)
```

```
invCounts.take(10).foreach(println)
```



```
(12,the)  
(8,in)  
(6,Greek)  
(4,Panathinaikos)  
(4,as)  
(4,team)  
(4,basketball)  
(3,consecutive)  
(3,club)  
(3,for)
```

Issues with NoSQL

- ◆ Efficient but ugly
- ◆ Long-term maintenance of codebase?
- ◆ Query processing is procedural unlike SQL
- ◆ Declarative queries (SQL) allow optimizations by well designed systems

Top-10 most freq. words (DataFrames+SQL)

```
import org.apache.spark.sql.Session
import spark.implicits._
val spark = SparkSession.builder.config(sc.getConf).getOrCreate()

val file = sc.textFile("/home/yannisk/wordfile")
val words = file.flatMap(line => line.split(" "))

val wordsDF = words.toDF(colNames="word")
wordsDF.createOrReplaceTempView("R")

val result = spark.sql("Select word,count(*) as counter from R
                        group by word order by counter DESC limit 10")
result.show()
```


Create a DataFrame containing all words in the document

```
import org.apache.spark.sql.SparkSession
import spark.implicits._
val spark = SparkSession.builder.config(sc.getConf).getOrCreate()

val file = sc.textFile("/home/yannisk/wordfile")
val words = file.flatMap(line => line.split(" "))

val wordsDF = words.toDF(colNames="word")
wordsDF.createOrReplaceTempView("R")

val result = spark.sql("Select word,count(*) as counter from R
                        group by word order by counter DESC limit 10")
result.show()
```

```
scala> wordsDF.show(10)
+-----+
|      word|
+-----+
|Panathinaikos|
|      B.C.|
|      also|
|      known|
|     simply|
|        as|
|Panathinaikos,|
|         or|
|        by|
|        its|
+-----+
```

Define a View

```
import org.apache.spark.sql.SparkSession
import spark.implicits._
val spark = SparkSession.builder.config(sc.getConf).getOrCreate()

val file = sc.textFile("/home/yannisk/wordfile")
val words = file.flatMap(line => line.split(" "))

val wordsDF = words.toDF(colNames="word")
wordsDF.createOrReplaceTempView("R")

val result = spark.sql("Select word,count(*) as counter from R
                        group by word order by counter DESC limit 10")
result.show()
```

Query View R using SQL

```
import org.apache.spark.sql.Session
import spark.implicits._
val spark = SparkSession.builder.config(sc.getConf).getOrCreate()

val file = sc.textFile("/home/yannisk/wordfile")
val words = file.flatMap(line => line.split(" "))

val wordsDF = words.toDF(colNames="word")
wordsDF.createOrReplaceTempView("R")

val result = spark.sql("Select word,count(*) as counter from R
                        group by word order by counter DESC limit 10")
result.show()
```

Show Results


```
import org.apache.spark.sql.SparkSession
import spark.implicits._
val spark = SparkSession.builder.config(sc.getConf).getOrCreate()
```

```
val file = sc.textFile("/home/yannisk/wordfile")
val words = file.flatMap(line => line.split(" "))
```

```
val wordsDF = words.toDF(colNames="word")
wordsDF.createOrReplaceTempView("R")
```

```
val result = spark.sql("Select word,count(*) as counter from R
                        group by word order by counter DESC limit 10")
```

```
result.show()
```



word	counter
the	12
in	8
Greek	6
Panathinaikos	4
team	4
basketball	4
as	4
for	3
of	3
club	3