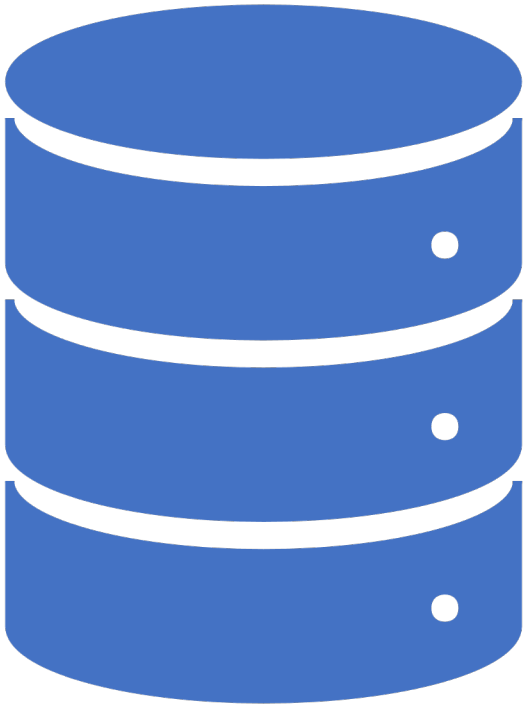


Lecture 14

DATABASES
Transactions. Concurrency Control

Concurrency in a DBMS

Scenarios

1. Train reservation system
   - Travel agent 1 (TA1) processes the request of customer 1 (C1) related to a seat on the train 10 → system retrieves from the database the tuple <train: 10, available seats: 1> and displays it to TA1
   - C1 takes time to decide whether or not to make the reservation

   - Travel agent 2 (TA2) processes the request of customer 2 (C2) related to a seat on the train 10 → system retrieves from the database the tuple <train: 10, available seats: 1> and displays it to TA2
   - C2 immediately reserves the seat

   - system:
     - update the tuple retrieved by TA2: <train: 10, available seats: 0>
     - updates the database, i.e. replaces flight 10's tuple in the database with the new tuple *<flight: 10, available seats: 0>*

   - C1 also decides to reserve the seat
   - however, TA1 is still working on the previously retrieved tuple, not knowing another user has changed the data

Concurrency in a DBMS

Scenarios

- system:
  - update the tuple retrieved by TA1: <train: 10, available seats: 0>
  - updates the database, i.e. replaces flight 10's tuple in the database with the new tuple *<flight: 10, available seats: 0>*

- The last remaining seat on train 10 has been sold 2 times!

2. Bank system
- 500 accounts: A_1, A_2, ..., A_{500}
- program P1 is computing the total balance: $A_1+A_2+ ... +A_{500}$
- program P2 transfers 1000 lei from A_1 to A_{500}
- if P2's transfer occurs after P1 has "seen" A_1, but before it has "seen" A_{500} => the total balance will be incorrect: $A_1+A_2+ ... +A_{500} + 500$
- The objective is to prevent such anomalies without disallowing concurrent access!

Concurrency in a DBMS

Transactions

- The database is partitioned into *items* (e.g. tuple, attribute)

Transaction

- any *one execution* of a program in a DBMS
- sequence of one or more operations on a database: *read* / *write* / *commit* / *abort*
- *commit* occurs if and only if the transaction is not aborted
- final operation: commit or abort

Let *I* be an item in a DB (database)

- fundamental operations in a transaction T (T can be omitted when clear from context):
  - read(T, I)
  - write(T, I)
  - commit(T) - successful completion: T's changes are to become permanent
  - abort(T) - T is *rollbacked*: its changes are undone

Concurrency in a DBMS

Transaction properties - ACID

- properties that a DBMS must ensure for a transaction in order to maintain data in the presence of concurrent execution / system failures

Atomicity

- a transaction is atomic
- either all the operations in the transaction are executed, or none are (*all-or-nothing*)

Consistency

- a transaction must preserve the consistency of the database after execution, i.e. a transaction is a correct program

Isolation

- a transaction is protected from the effects of concurrently scheduling other transactions

Durability

- committed changes are persisted to the database
- the effects of a committed transaction should persist even if a system crash occurs before all the changes have been written to disk

Concurrency in a DBMS

Atomicity

- a transaction can commit / abort:
  - *commit* – after finalizing all its operations (successful completion)
  - *abort* – after executing some of its operations; the DBMS can itself abort a transaction
- user's perspective:
  - either all operations in a transaction are executed, or none are
  - the transaction is treated as an indivisible unit of execution
- the DBMS *logs* all transactions' actions, so it can *undo* them if necessary (i.e. undo the actions of incomplete transactions)

e.g. consider the following transaction transferring money from Account_A to Account_B:

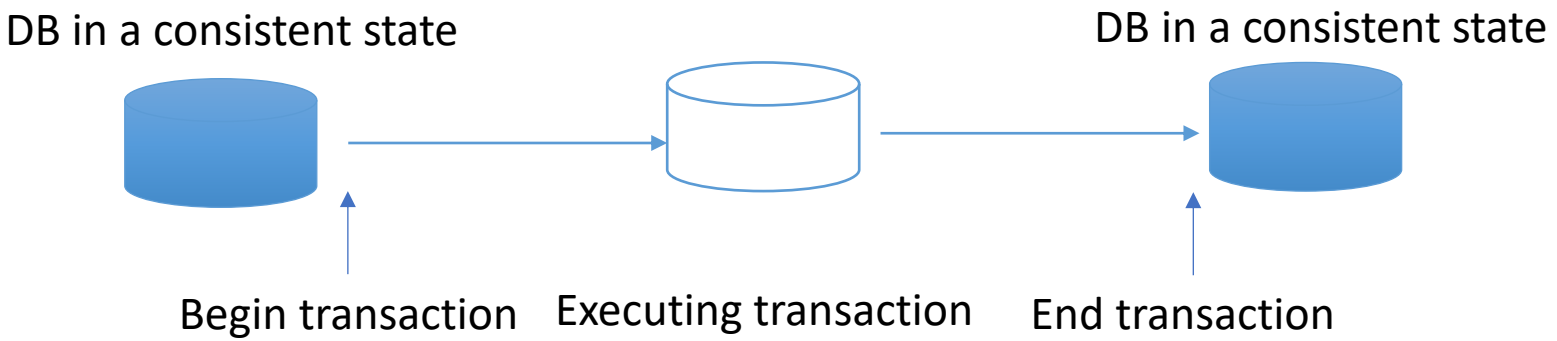
Account_A ← Account_A - 100
Account_B ← Account_B + 100

- if the transaction fails in the middle, 100 lei must be put back into Account_A, i.e. partial effects are undone

Concurrency in a DBMS

Consistency

- in the absence of other transactions, a transaction that executes on a consistent database state leaves the database in a consistent state



- transactions do not violate the integrity constraints (ICs) specified on the database (e.g. restrictions declared via CREATE TABLE statements) and enforced by the DBMS
- however, users can have consistency criteria that go beyond the expressible integrity constraints

Concurrency in a DBMS

Consistency

e.g. consider again the money transfer transaction:

Account_A ← Account_A - 100

Account_B ← Account_B + 100

- assume the transaction starts execution at time t_i and completes at t_f
- consistency constraint: Total(Account_A, Account_B, t_i) = Total(Account_A, Account_B, t_f)
- it is the responsibility of the user to write a correct transaction that meets the above constraint
- the DBMS does not understand the semantics of the data, of user actions (e.g. how is the interest computed, the fact that Account_B must be credited with the amount of money debited from Account_A)

Concurrency in a DBMS

Isolation

- transactions are protected from the effects of concurrently scheduling other transactions
- users concurrently submit transactions to the DBMS, but can think of their transactions as if they were executing in isolation, independently of other users' transactions
- i.e. users must not deal with arbitrary effects produced by other concurrently running transactions; a transaction is seen as if it were in single-user mode

Durability

- once a transaction commits, the system must guarantee the effects of its operations won't be lost, even if failures subsequently occur
- i.e. once the DBMS informs the user a transaction has committed, its effects should persist, even if a system crash occurs before all changes have been saved on the disk

Concurrency in a DBMS

Lock-Based Concurrency Control

- *lock*
  - a tool used by the transaction manager to control concurrent access to data
  - prevents a transaction from accessing a data object while another transaction is accessing the object
- *transaction protocol*
  - a set of rules enforced by the transaction manager and obeyed by all transactions
  - e.g. – simple protocol: before a transaction can read / write an object, it must acquire an appropriate lock on the object
  - locks in conjunction with transaction protocols allow interleaved executions
- *SLock* (*shared* or *read lock*)
  - if a transaction holds an SLock on an object, it can read the object, but it cannot modify it
- *XLock* (*exclusive* or *write lock*)
  - if a transaction holds an XLock on an object, it can both read and write the object

Concurrency in a DBMS

Lock-Based Concurrency Control

- if a transaction holds an SLock on an object, other transactions can be granted SLocks on the object, but they cannot acquire XLocks on it
- if a transaction holds an XLock on an object, other transactions can be granted either SLocks or Xlock on the object

	Shared	Exclusive
Shared	Yes	No
Exclusive	No	No

Concurrency in a DBMS

Transaction Support in SQL - Isolation Levels

- SQL provides support for users to specify various aspects related to transactions

e.g. isolation levels

- *isolation level*
  - a transaction's characteristic
  - determines the degree to which a transaction is isolated from the changes made by other concurrently running transactions
- greater concurrency -> concurrency anomalies

- 4 isolation levels
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE

Concurrency in a DBMS

Transaction Support in SQL - Isolation Levels

- isolation levels can be set with the following command:
  SET TRANSACTION ISOLATION LEVEL *isolevel*

e.g. SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

Examples on Article(Aid, Title, PrincipalAuthor, Year, Type)

- Article contains (at least) the following tuples:

(3, ‘Artificial Intelligence for DNA’, ’Popescu Dan’, 2020, 5)
(4, ‘General AI characteristics', ‘Kelly James’, 2010, 4)

READ UNCOMMITTED

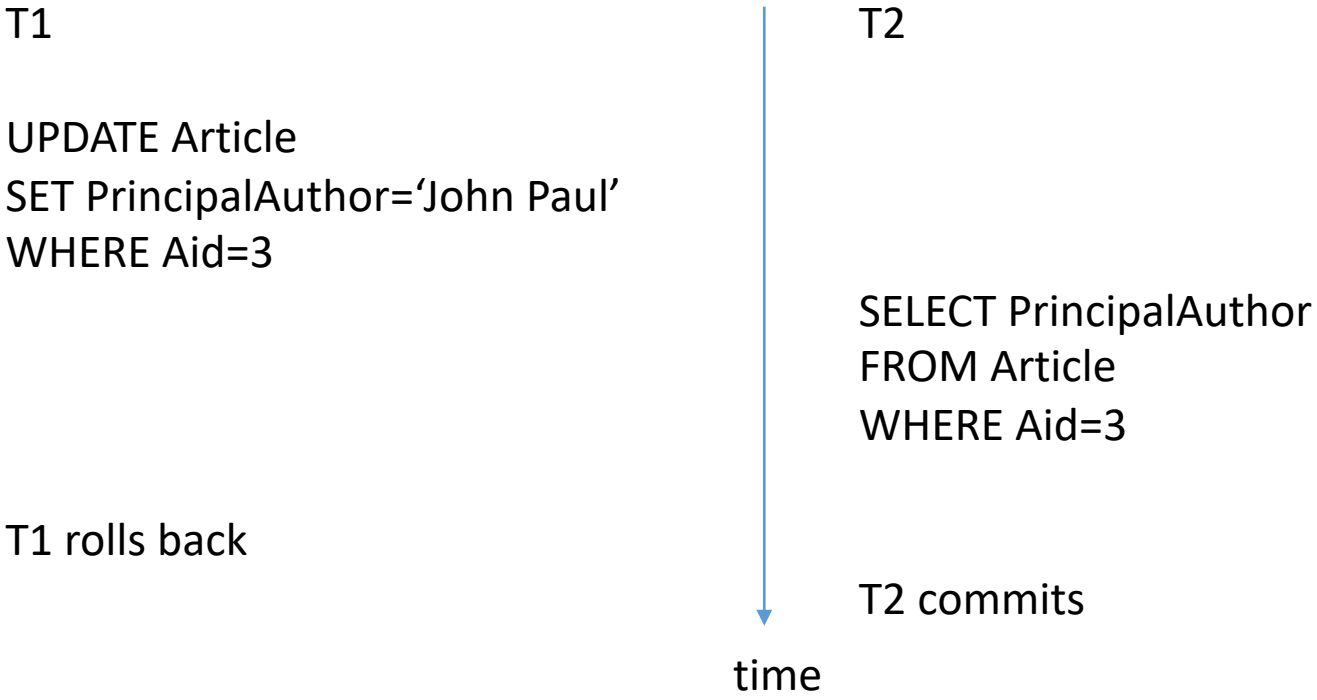
- no S locks when reading data
- a transaction can see uncommitted changes carried out by another transaction

Concurrency in a DBMS

Dirty reads

- a transaction reads data that has been modified by another transaction, but not committed
- T2 reads *John Paul* – under READ UNCOMMITTED

(3, 'Artificial Intelligence for DNA', 'Popescu Dan', 2020, 5)
(4, 'General AI characteristics', 'Kelly James', 2010, 4)



Concurrency in a DBMS

Isolation Levels

READ COMMITTED

- a transaction must acquire an exclusive lock prior to writing an object
- a transaction must acquire a shared lock prior to reading an object (i.e. the last transaction that modified the object is complete)
- exclusive locks are released at the end of the transaction
- shared locks are immediately released

→ a transaction cannot see uncommitted changes carried out by another transaction

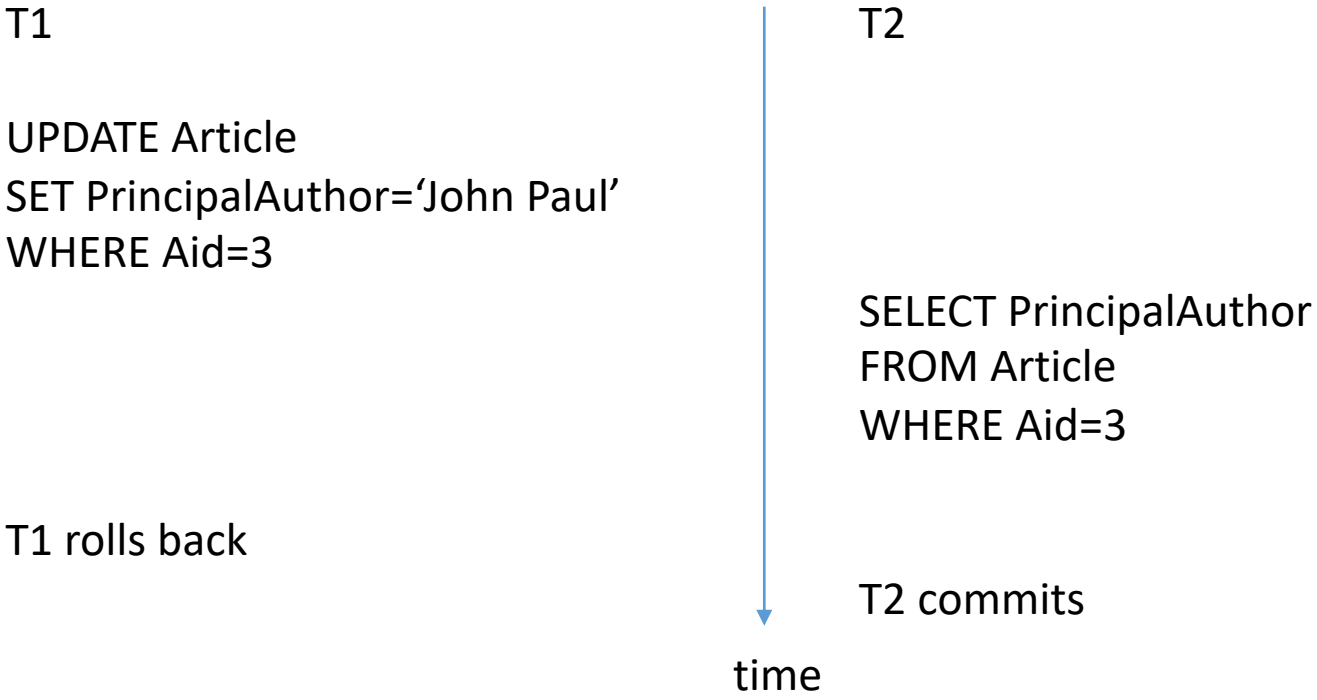
- a transaction can change data that has been read by the current transaction, prior to its completion
- it is the default isolation level in SQL SERVER

Concurrency in a DBMS

~~Dirty reads~~

- T2 reads *Popescu Dan*– under READ COMMITTED

(3, ‘Artificial Intelligence for DNA’, ’Popescu Dan’, 2020, 5)
(4, ‘General AI characteristics', ‘Kelly James’, 2010, 4)

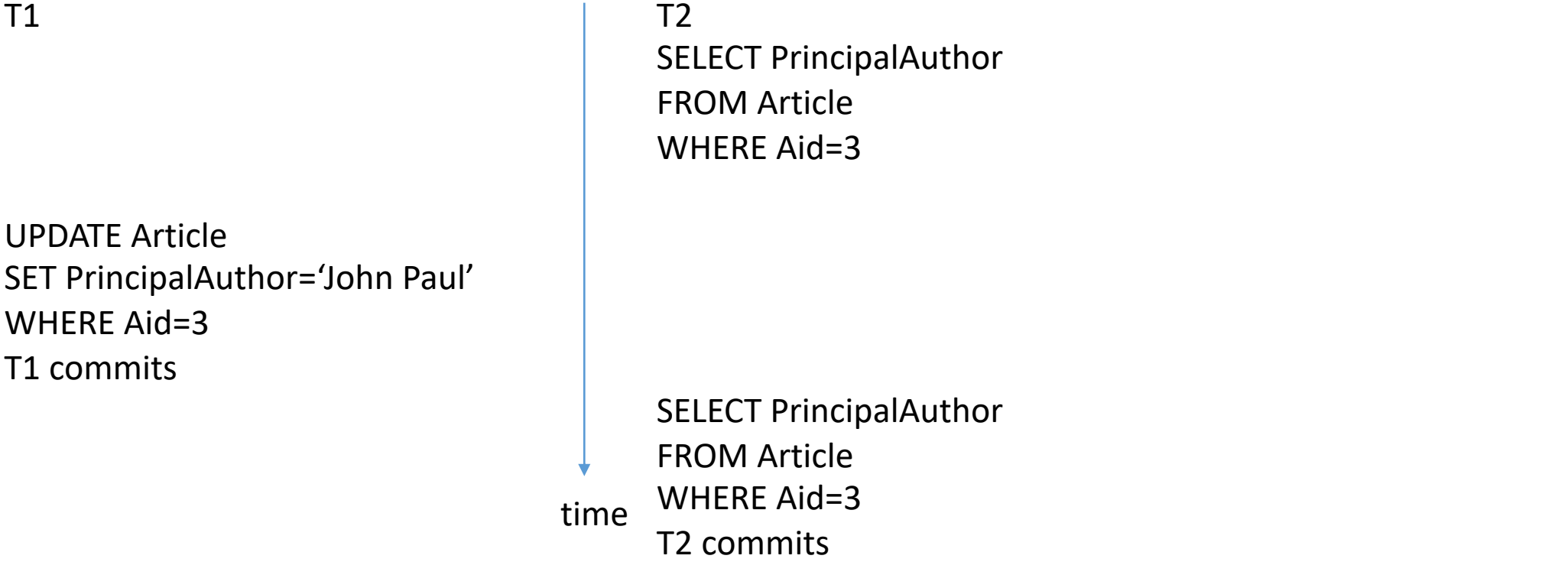


Concurrency in a DBMS

Non-repeatable reads

- T2 under READ COMMITTED; T1 can change data read by T2 while T2 is still in progress
- T2 reads 2 different values: *Popescu Dan* and *John Paul* – under READ COMMITTED

(3, 'Artificial Intelligence for DNA', 'Popescu Dan', 2020, 5)
(4, 'General AI characteristics', 'Kelly James', 2010, 4)



Concurrency in a DBMS

Isolation Levels

REPEATABLE READ

- a transaction must acquire an exclusive lock prior to writing an object
- a transaction must acquire a shared lock prior to reading an object
- exclusive locks are released at the end of the transaction
- shared locks are released at the end of the transaction

→ a transaction cannot change data that has been read by the current transaction, prior to its completion

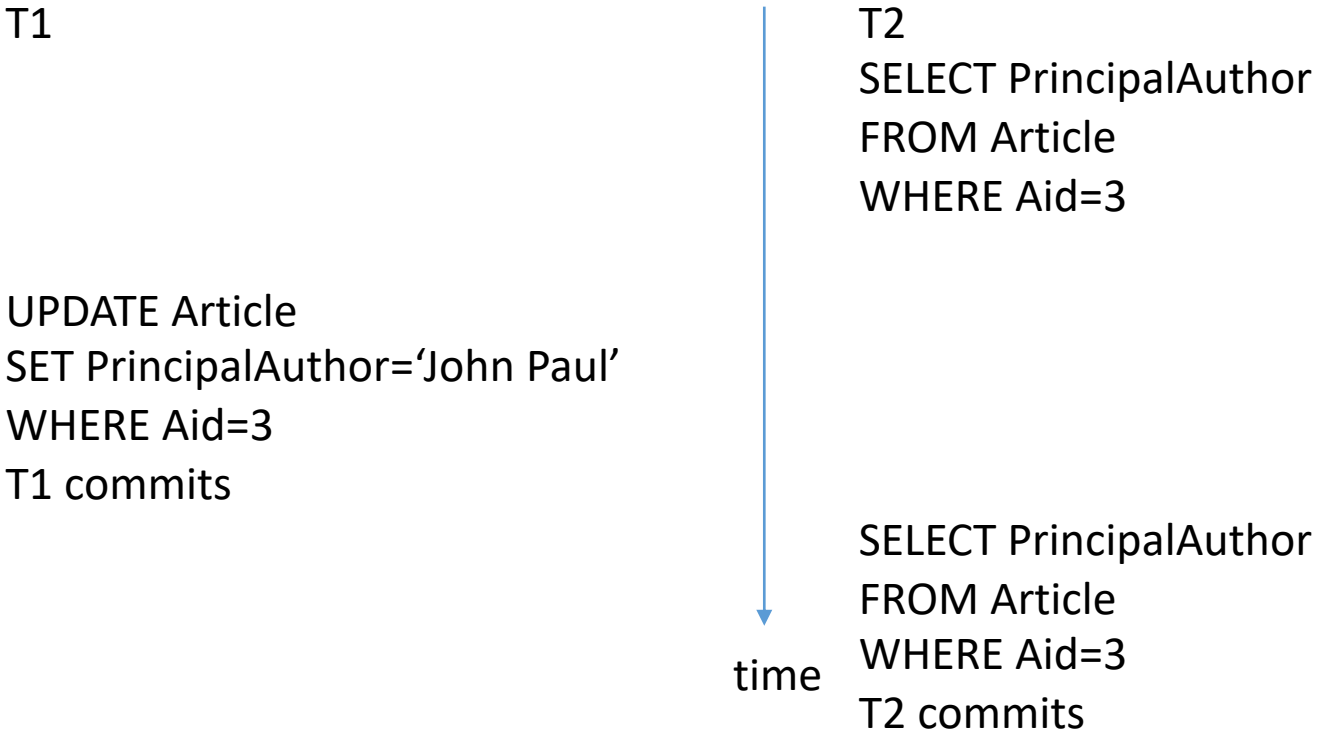
- transactions exposed to the phantom problem

Concurrency in a DBMS

~~Non-repeatable reads~~

- T2 reads the values: *Popescu Dan* and *John Paul* – under REPEATABLE READ

(3, ‘Artificial Intelligence for DNA’, ’Popescu Dan’, 2020, 5)
(4, ‘General AI characteristics', ‘Kelly James’, 2010, 4)

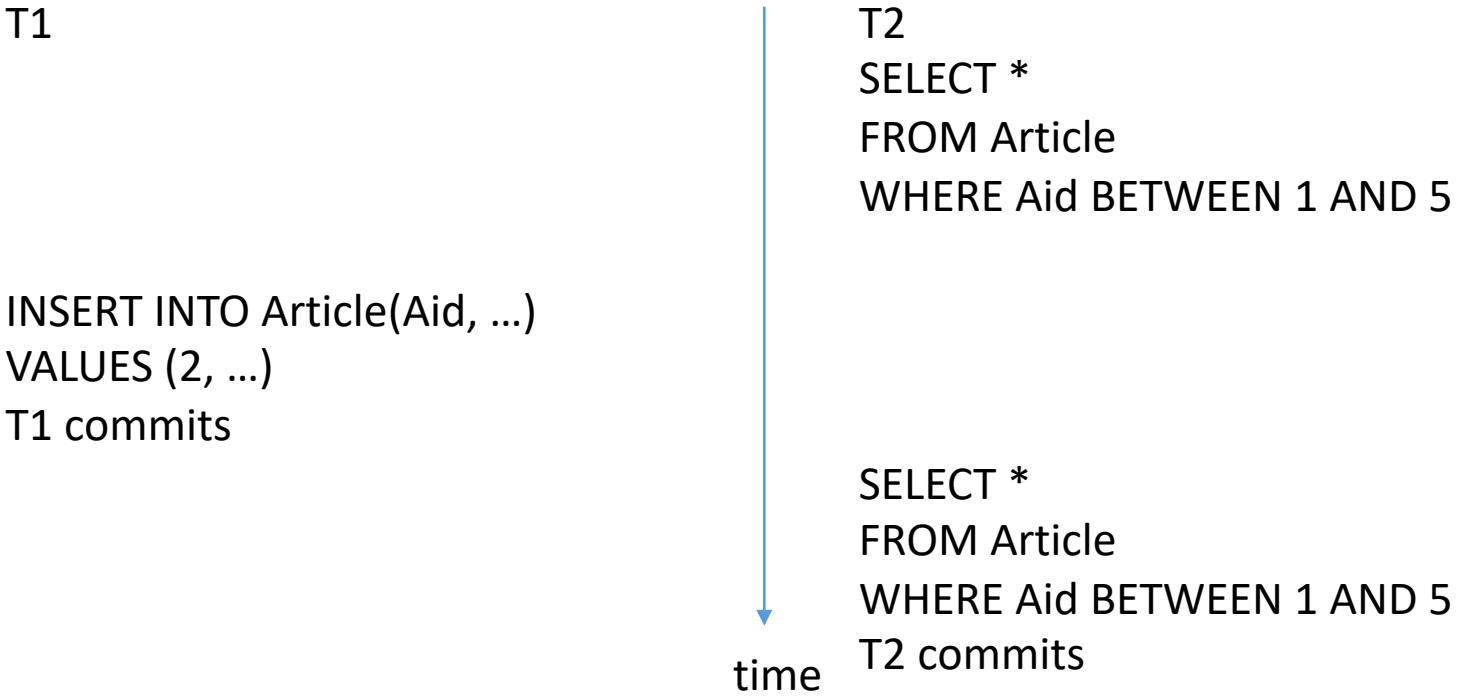


Concurrency in a DBMS

(3, ‘Artificial Intelligence for DNA’, ’Popescu Dan’, 2020, 5)
(4, ‘General AI characteristics', ‘Kelly James’, 2010, 4)

Phantoms

- T2 under REPEATABLE READ; T2 reads a set of tuples based on a search condition; T1 can add a row that alters the set of tuples, while T2 is still in progress
- T2 reads articles with Aids: (3, 4), (2, 3, 4) – under REPEATABLE READ



Concurrency in a DBMS

Isolation Levels

SERIALIZABLE

- strongest isolation level
- a transaction must acquire locks on objects before reading / writing them
- a transaction also acquires locks on sets of objects that must remain unmodified
  - if a transaction T reads a set of objects based on a search predicate, this set cannot be changed by other transactions while T is in progress (if query *SELECT * FROM Students WHERE Grade >= 8* is executed twice within a transaction, it must return the same answer set)
- locks are held until the end of the transaction

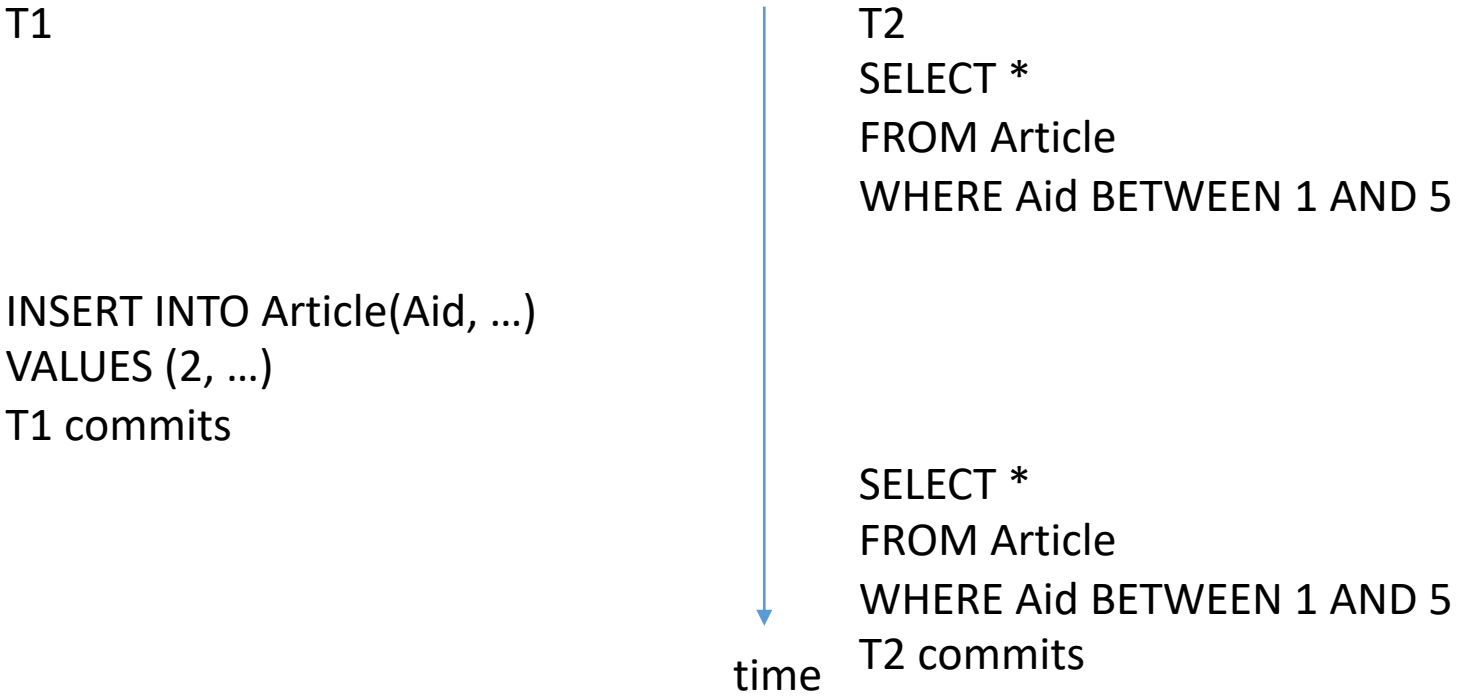
→ phantoms not possible

Concurrency in a DBMS

~~Phantoms~~

- T2 reads articles with Aids: (3, 4), (3, 4) – under SERIALIZABLE

(3, ‘Artificial Intelligence for DNA’, ’Popescu Dan’, 2020, 5)
(4, ‘General AI characteristics', ‘Kelly James’, 2010, 4)

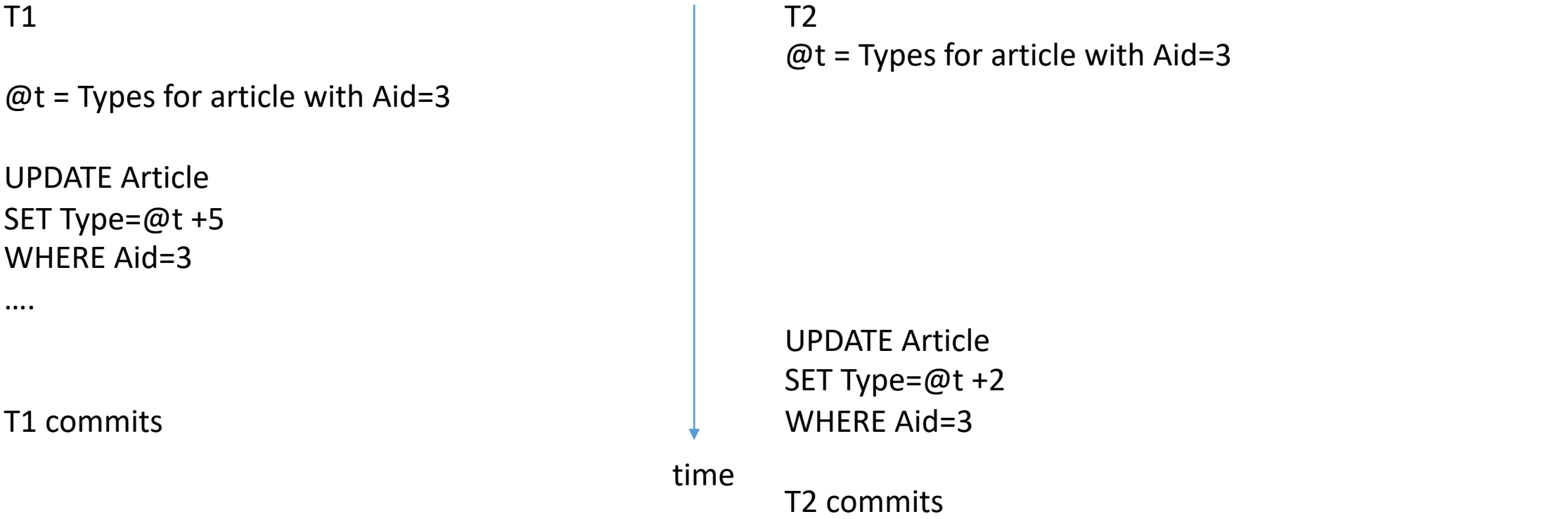


Concurrency in a DBMS

Deadlocks

- e.g. T1, T2 – under REPEATABLE READ → deadlock (cycle of transactions waiting for one another to release a locked resource)

(3, 'Artificial Intelligence for DNA', 'Popescu Dan', 2020, 5)
(4, 'General AI characteristics', 'Kelly James', 2010, 4)



Concurrency in a DBMS

Isolation Levels \ Concurrency Problems	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommittted	Yes	Yes	Yes
Read Committed	No	Yes	Yes
Repeatable Read	No	No	Yes
Serializable	No	No	No

References:

- C.J. Date, *An Introduction to Database Systems (8th Edition)*, Addison-Wesley, 2003.
- H. Garcia-Molina, J. Ullman, J. Widom, *Database Systems: The Complete Book, Prentice Hall Press*, 2008.
- G. Hansen, J. Hansen, *Database Management And Design (2nd Edition),* Prentice Hall, 1996.
- R. Ramakrishnan, J. Gehrke, *Database Management Systems*, McGraw- Hill, 2007. http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- R. Ramakrishnan, J. Gehrke, *Database Management Systems (2nd Edition),* McGraw-Hill, 2000.
- A. Silberschatz, H. Korth, S. Sudarshan, *Database System Concepts*, McGraw-Hill, 2010. http://codex.cs.yale.edu/avi/db-book/
- L. Țâmbulea, *Curs Baze de date*, Facultatea de Matematică și Informatică, UBB, 2013-2014.
- J. Ullman, J. Widom, *A First Course in Database Systems*, http://infolab.stanford.edu/~ullman/fcdb.html