Transaction Management

Managing Concurrent Access to Data Objects

CS3524 Distributed Systems

Lecture 07

Transaction

A transaction is a controlled sequence of actions for the manipulation of a database

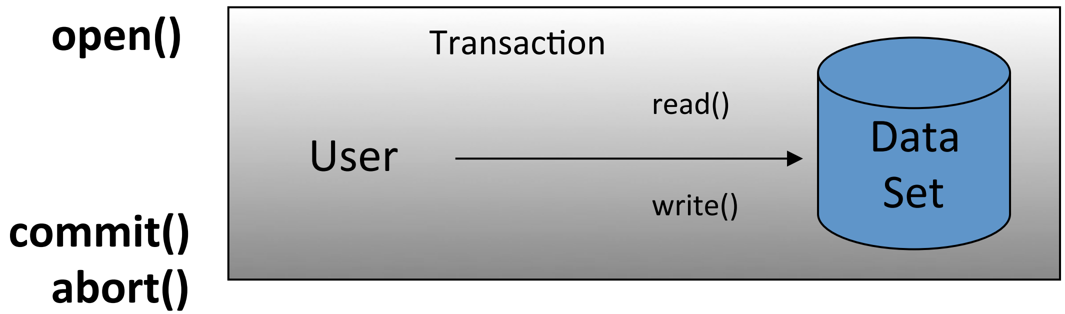
- Objective:
  - Guarantee that data always remains in a consistent state
- Avoid:
  - Loss of data due to system failure (Database Integrity, Durability of data)
  - Problems resulting from concurrent access of multiple users (consistent manipulation)

Database Integrity

- There are two categories of problems that may compromise the integrity of databases:

System failure	Concurrent access
- Malfunction of hardware and software - crash during data manipulation - leaves data in a corrupted / inconsistent state	- Users interfere with each others’ data manipulation operations - overwrite / delete each others data - may leave data in a corrupted / inconsistent state

Transaction



- Transactions are a controlled sequence of read and write operations:
  - Have a clear “start”: the first operation of a transaction
  - Have a clear “end”:
    - Commit: transaction ends successfully
  - Transactions can be aborted any time
    - Abort: transaction is “undone”

Transaction Atomicity

- Transactions are regarded as ***atomic***:
  - "All-or-nothing": the complete sequence of operations is regarded as an indivisible unit - either all of the operations occur or none of them
- A transaction transforms databases from one **consistent state** to another **consistent state**
- Transactions are either committed at the end or aborted (anytime):

Commit: transaction ends successfully, all manipulations are stored in database, database is in **new consistent state**

Abort: transaction is "undone" – manipulations performed during transaction have no effect on the database, database remains in **old consistent state**

Programming Transactions

Transaction Commit	Account A	Account B

`account_a = getAccount(db,"account_1")`
`account_b = getAccount(db,"account_2")`
 	
`open_transaction()`
`    balance1 = `**`read( account_a )`**
`    balance1 = balance1 - 200`
`    `**`write( account_a, balance1 )`**
`    balance2 = `**`read( account_b )`**
`    balance2 = balance2 + 200`
`    `**`write( account_b, balance2 )`**
`commit()`
	1000.00
	
	800.00
		1000.00
		
		1200.00

Both accounts have changed	800.00	1200.00

Programming Transactions

- Transaction Abort:
  - A user can decide to abort a transaction (e.g. Pressing a button in a GUI to abort a payment transaction):

Transaction Abort	Account A	Account B

`open_transaction()`
`    balance1 = `**`read( account_a )`**
`    balance1 = balance1 - 200`
`    `**`write( account_a, balance1 )`**
`    balance2 = `**`read( account_b )`**
`    balance2 = balance2 + 200`
`    `**`write( account_b, balance2 )`**
`    user_decision = getUserDecision()`
 	
`if( user_decision == “abort” ) `**`abort()`**
	1000.00
	
	800.00
		1000.00
		
		1200.00

Both accounts have to be returned to their original balance	**1000.00**	**1000.00**

How can the Database system do that?

Handling Transaction Abort

- Transaction Abort
  - Intentionally by client / user programs interacting with database server
  - Aborted by database system
    - Client session timeout
    - For resolving conflicts between concurrent transactions (e.g. deadlocks)
- Action: Transaction "Rollback"
  - Database is "rolled back" to its previous consistent state
  - All manipulative operations are undone (or not applied, depending on the implementation of the transaction management)

System Failure

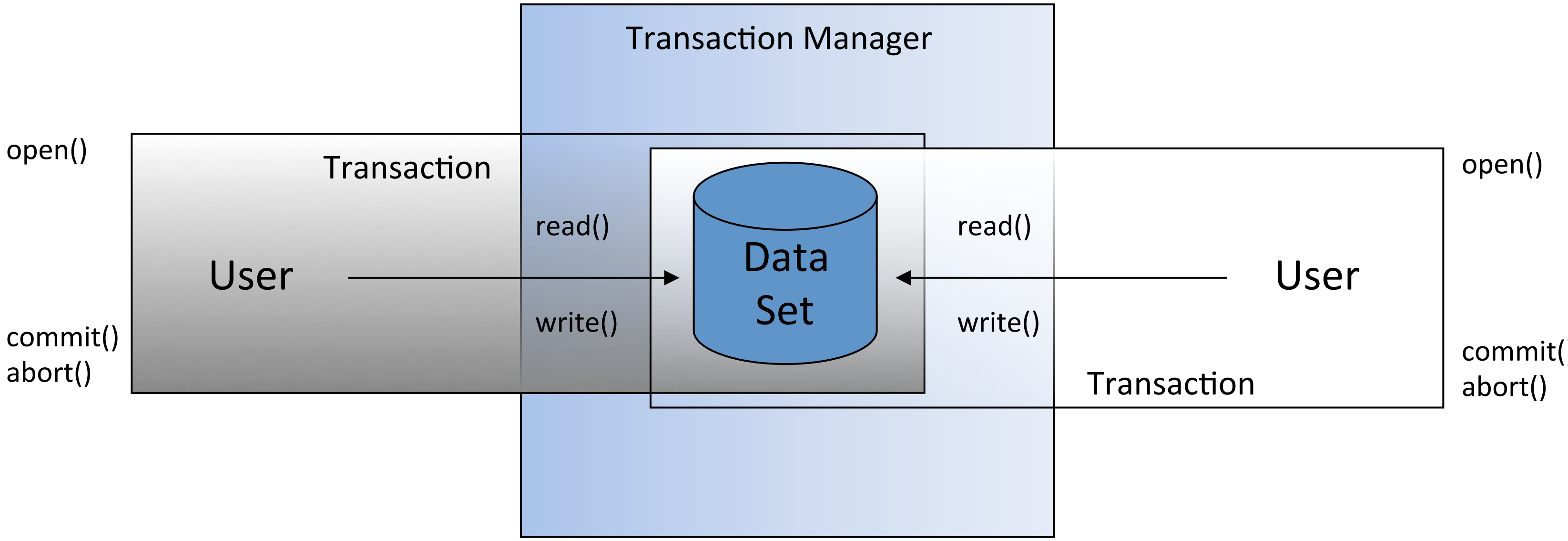
Transaction Commit	Account A	Account B
`open_transaction()`		
`balance1 =` **`read( account_a )`**	**1000.00**	
`balance1 = balance1 - 200`		
`write( account_a, balance1 )`	**800.00**	
`balance2 =` **`read( account b )`**		**1000.00**
`balance2 = balance2 + 200`		
`write( account_b, balance2 )`		
`commit()`		

- A system failure occurs, after change of account A, but before account B can be changed – the database is "inconsistent"!

Recovery from System Failure

- System failures may corrupt / destroy data
- Database management systems employ a "Recovery Manager" to re-instate a database to its last consistent state before system failure
  - Backup: record last consistent state
  - Log files / Transaction Journals:
    - A Recovery manager records each data manipulation actions since last backup
  - "Roll Forward": recovery manager uses log / journal records to redo all committed transactions

Concurrent Transactions



- Usually many transaction may run concurrently
- They may manipulate the same data concurrently
- Isolation:
  - The Transaction management has to employ concurrency control mechanisms and protocols to properly “isolate” transactions from each other so that they do not interfere with each other

Database Integrity

Transaction

Consistency of Data
Saveguard against undesired/wrong
Manipulation of Data

Durability of Data
Saveguard against System Failure
Support Recovery

- Consistency:
  - Saveguard against “undesired” / “wrong” manipulation of data (user concern)
- Durability:
  - Saveguard against loss of data (system concern)
- Transaction management systems have to guarantee:
  - In case of system failure, data can be recovered
  - Isolate concurrently executing transactions from each other
    - Avoid race conditions (overwrite each others’ results)
    - Protect transactions from reading partial / uncommitted results of other transactions

The ACID Properties

- **Atomicity:** Either *all operations* of a transaction are performed, or *none* of them
- **Consistency:** A transaction transforms the system from one *consistent state* into another
- **Isolation:** An incomplete transaction cannot reveal its *intermediate state* to other transactions until committed
- **Durability:** Once a transaction is committed, the system must guarantee that the results of the transaction will *persist*, even if the management system subsequently fails.

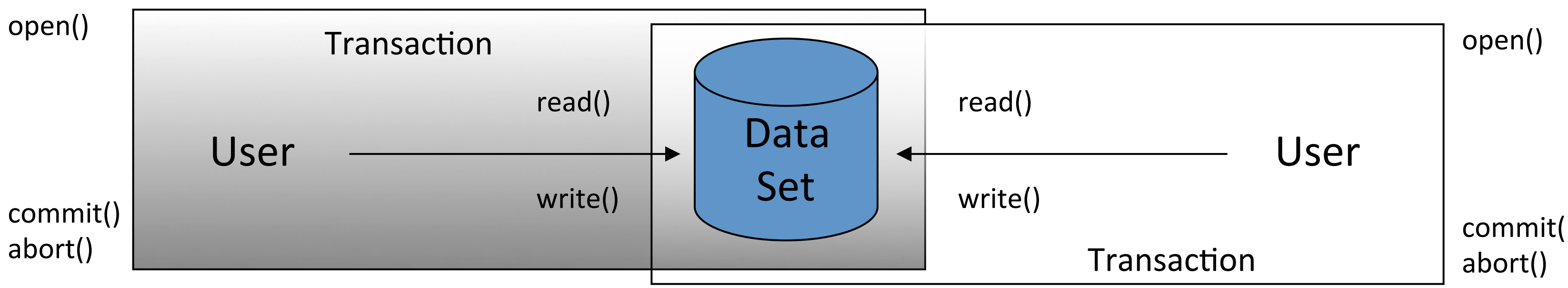
Atomicity

- Principle of "All-or-nothing":
  - A transaction either completes successfully, and the effects of all of its operations are recorded or (due to system failure or deliberate abort) it has no effect at all
- Failure atomicity ("*nothing*"):
  - the effects are atomic even when the server crashes – intermediate results are "rolled back"
- Durability ("*all*"):
  - after a transaction has completed successfully, all its effects are saved in persistent storage

Durability

- **Durability:**
  - Once a transaction is committed, the system must guarantee that the results of the transaction will persist, even if the management system subsequently fails.
- In order to support failure atomicity and durability, data objects must be *recoverable*
  - regular backups
  - transaction logs that record manipulation actions

Concurrency Problems



- The interleaving of actions of concurrently executing transactions may lead to severe problems
- Lack of Isolation:
  - Lost Update: Transactions may overwrite each others’ updates
  - Inconsistent Retrieval: Transactions base their calculations on retrieved data that is not yet committed by other transactions (“dirty read”)
- It depends on the sequence of actions, scheduled from different transactions, whether we create inconsistencies in our databases

The Lost Update Problem

Time	Transaction T1	Account A	Transaction T2
t1	`open_transaction()`	100.00	
t2	**`read( balance_a )`** ←	100.00	`open_transaction()`
t3	`balance_a = balance_a + 100`	100.00 →	**`read( balance_a )`**
t4	**`write( balance_a )`** →	200.00	`balance_a = balance_a - 10`
t5	`commit()`	90.00 ←	**`write( balance_a )`**
t6		90.00	`commit()`

- Schedule:

  S = { T1.read(), T2.read(), T1.write(), T2.write() }

- Problem:
  - Last write of transaction T2 overwrites previous write of transaction T1

- **Race condition** !

Inconsistent Retrieval Problem

- Also called the “inconsistent analysis” problem
- Occurs when
  - One transaction first retrieves data via a sequence of read operation
  - A second transaction performs write operations concurrently (after read operations) and overwrites some of this data in the database
- Problem:
  - The reading transaction bases its calculations on the original retrieved data (not yet committed) that have been changed in the meantime in the database itself – calculations are out-of-date!

Inconsistent Retrieval Problem

(due to“Dirty Read”)

Time	Transaction T1	A	B	Transaction T2
t1	`open_transaction()`	100	50	
t2	`sum = 0`	100	50	`open_transaction()`
t3	**`read( balance_a )`**	100	50	**`read( balance_a )`**
t4	`sum = sum + balance_a`	100	50	`balance_a = balance_a - 10`
t5		90	50	**`write( balance_a )`**
t6	**`read( balance_b )`**	90	50	`commit()`
t7	**`sum = sum + balance_b`**	90	50	
t8	`commit()`			

- Transaction T1 calculates:
  - sum = 100 + 50 = 150
- Problem: transaction T2 changed the database
  - Balance of account A + balance of account B = 140 !

Problems with Aborted Transactions

- Effects of transactions are recorded in persistent memory only if they are committed
- If a transaction is aborted, then they should not have any effect on subsequent transactions
- Problems
  - Cascading Aborts
  - Premature Writes

Uncommitted Dependency Problem

(due to "Dirty Read")

Time	Transaction T1	Account A	Transaction T2
t1	`open_transaction()`	100.00	
t2	**`read( balance_a )`**	100.00	
t3	`balance_a = balance_a + 100`	100.00	
t4	**`write( balance_a )`**	200.00	`open_transaction()`
t5		200.00	**`read( balance_a )`**
t6	`rollback()` →	100.00	`balance_a = balance_a - 10`
t7		190.00	**`write( balance_a )`**
t8		190.00	`commit()`

- Problem:
  - Transaction T2 reads an intermediate, uncommitted result of T1 and bases its own calculations on that "dirty read"
  - T1 aborts transaction and performs a "rollback"

Cascading Aborts

- Cascading aborts
  - If a transaction has seen the effects of an aborted transaction, it has to abort as well
  - This can cause for other transactions to abort as well – a cascade of aborts can occur
- Solution
  - Read is only allowed as long as there are no concurrent uncommitted write operations during transaction:
    - A transaction is only allowed to read objects that are committed by other transactions
    - Any read operation must be delayed until other transactions have committed or aborted write operations on shared data objects

Concurrency Control

- How can we avoid these problems?
- We need a concurrency control mechanism

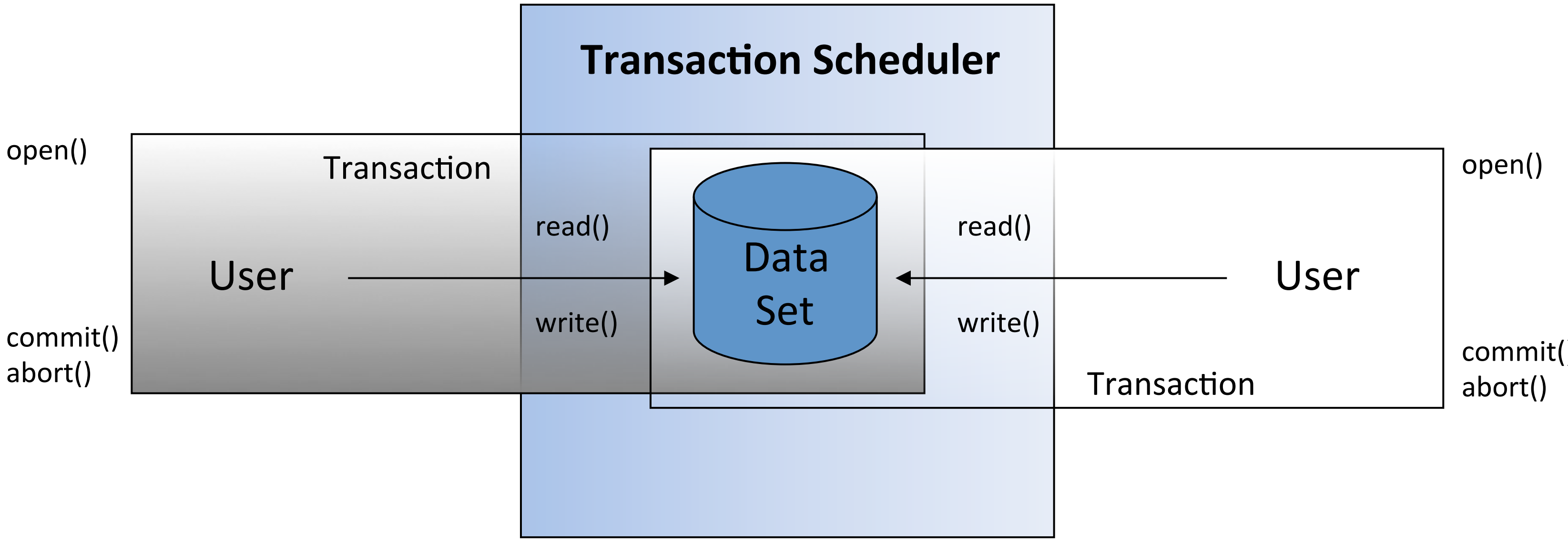
Concurrency Control

Saveguard Isolation between Transactions

Concurrency Control

- Concurrency control enforces a particular order of operations of concurrent transactions
  - It creates a ***Transaction schedule***
- The mechanisms commonly employed for concurrency control are
  - Locking:
    - Reserve a data object for exclusive access by a single transaction
    - Most practical systems use locking
  - Optimistic concurrency control
  - Timestamp ordering
- How do we know that such a concurrency control mechanism is "correct" ?
- How do we know that the enforced order of operations represents a "correct" transaction schedule?

What is a correct Transaction Schedule?



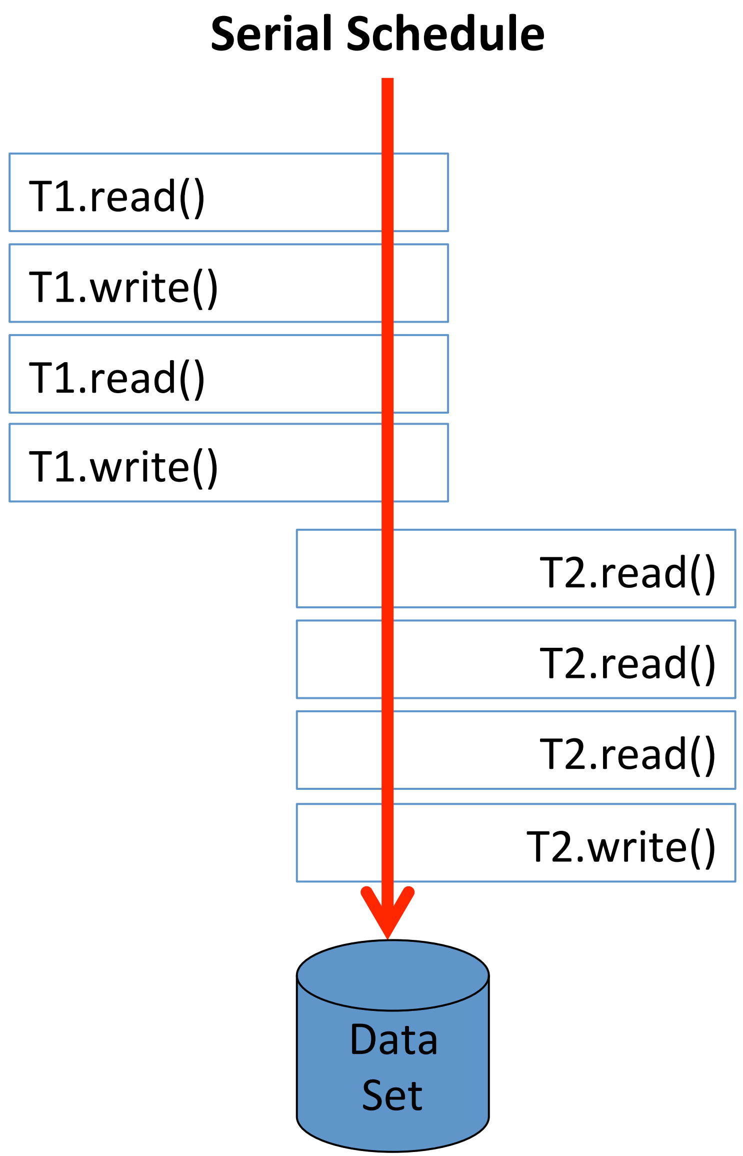
- A "schedule" is a sequence of operations from different transactions – transactions operate in an interleaved fashion
  - Such a schedule may compromise the integrity / consistency of a database

Transaction Schedule

- We specify
  - A schedule is "correct", if it preserves the consistency of a database
    - No lost update
    - No inconsistent retrieval
- Radical solution
  - Completely serialized execution of transactions

Serial Schedule

- Serial Schedule:
  - A schedule where the operations of each transaction are scheduled for execution consecutively, without any interleaving of operations from different transactions
- There is no interference or concurrency problem
- Therefore, outcome of the execution of such a schedule preserves consistency of database
- It is a “correct” schedule!



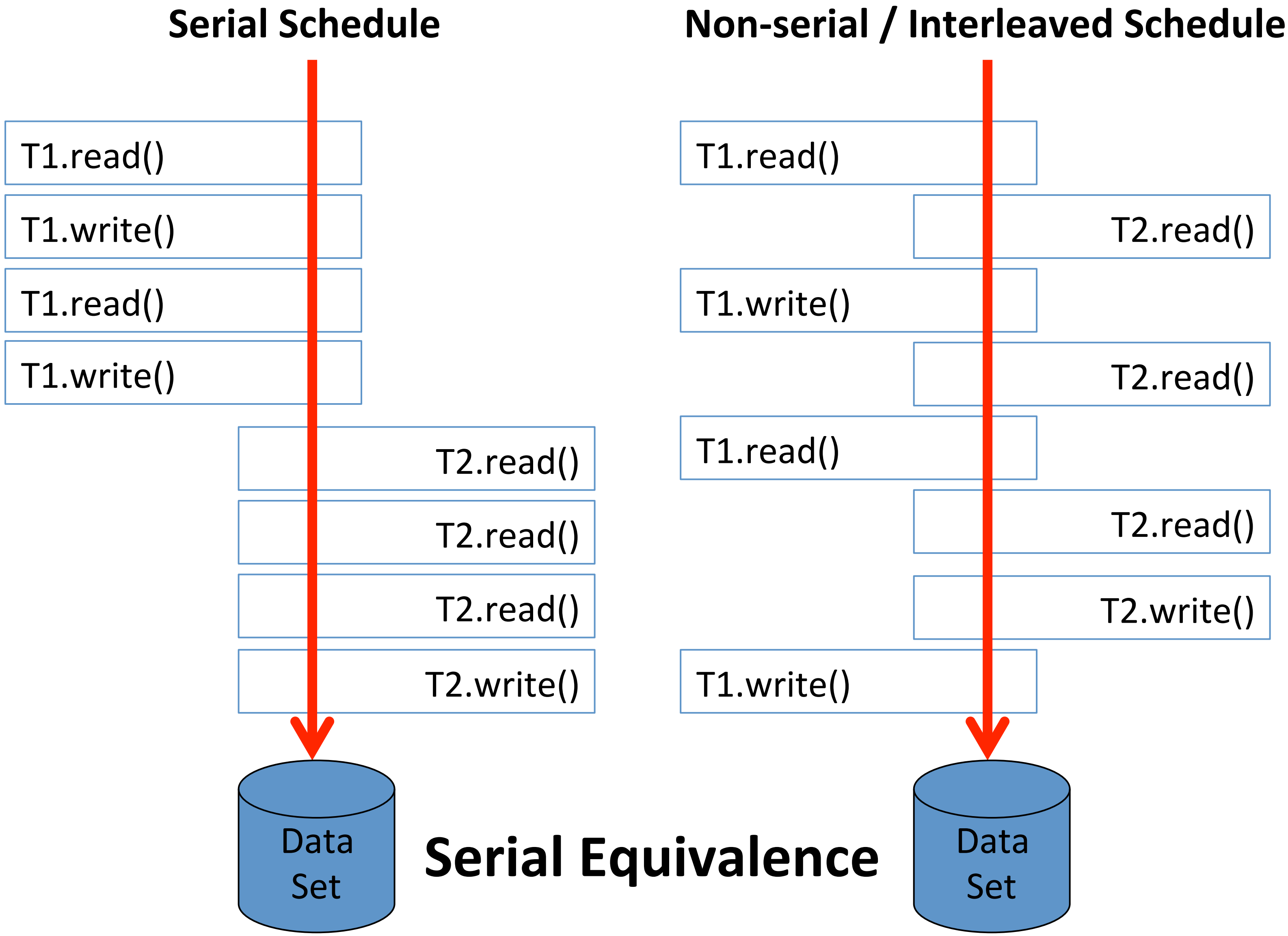
Serialised Transaction Schedules

- Radical solution
  - Completely serialised execution of transactions
  - In order to avoid the concurrency problems described, one obvious solution would be to schedule only one transaction at a time for execution
- Such a completely “serialised” schedule can be regarded as “correct”:
  - Enforces Isolation: transactions are completely isolated and cannot interfere with each other
  - Enforces consistency: Serial execution of transactions never leaves a database in an inconsistent state
- However: we want a concurrent execution of transactions, that preserves the ACID principles

Serialised and Concurrent Execution

- Objective:
  - The aim of modern database management systems is to allow multi-user access and to maximise the degree of concurrency between transactions
- How can we guarantee that a particular transaction schedule does not violate the consistency of a database – that it is “correct”?
- We use the observation that completely serialised schedules are always correct

Transaction Schedules



Serial Equivalence, Serialisability

- We try to find a non-serial schedule that is ***equivalent*** to a corresponding serial schedule:
  - A non-serial schedule over a set of transactions is correct if it produces the same results as a completely serial execution of the same set of transactions
- If we can show or guarantee serialisability of a transaction schedule, then we can guarantee consistency of data manipulation

Serial Equivalence

- Definition: Serial Equivalence

Two or more transactions are serial equivalent, if they produce the same result operating in an interleaved fashion, as if they would operate in a completely serialized fashion.

- Serial Equivalence is used as a design criterion for concurrency control protocols!

Serialisability

- What makes a non-serial schedule equivalent to a serial schedule?
- Observation:
  - It is the sequence and order of **read / write** operations (of different transactions) in a schedule that determines whether the schedule is serialisable
- A concurrency control protocol must enforce such a correct sequencing!

Concurrency Control Protocols

- Two kinds of concurrency control behaviour
  - Transactions wait to avoid conflicts (pessimistic concurrency control)
  - Transactions are restarted after conflicts have been detected (optimistic concurrency control)
- Methods
  - Locking
  - Optimistic concurrency control
  - Timestamp ordering