# Data Management Systems

- Transaction Processing
    - Concurrency control and recovery
    - Transactions
    - Recovery

ACID
Transaction model
Concurency Control
Recovery

**Gustavo Alonso**
**Institute of Computing Platforms**
**Department of Computer Science**
**ETH Zürich**

# A bit of theory

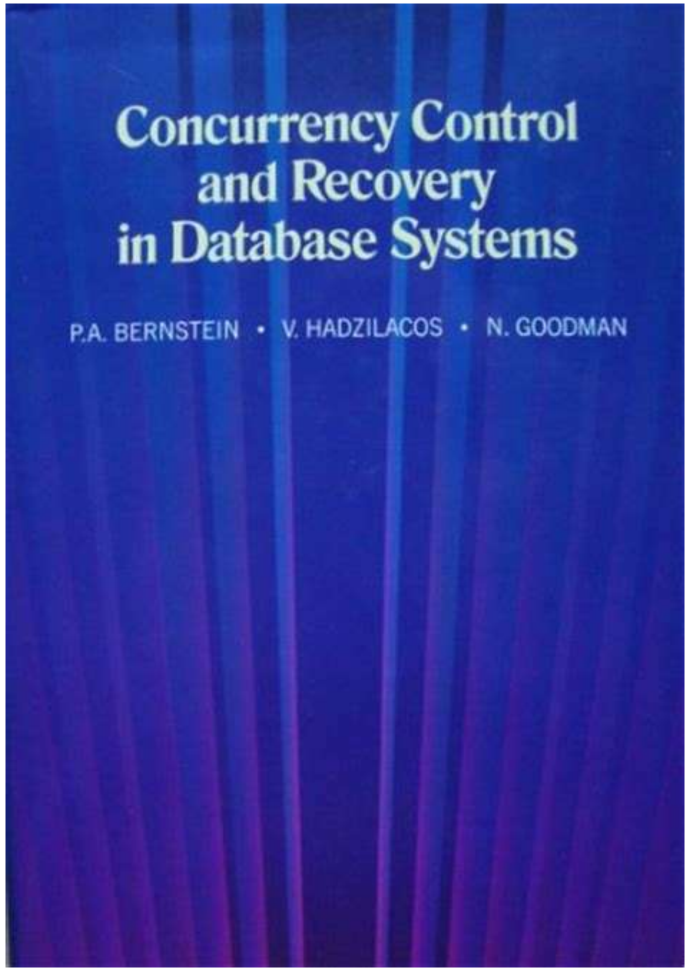
- Before discussing implementations, we will cover the theoretical underpinning behind concurrency control and recovery
    - Discussion at an abstract level, without relation to implementations
    - No consideration of how the concepts map to real elements (tuples, pages, blocks, buffers, etc.)
- Theoretical background important to understand variations in implementations and what is considered to be correct
- Theoretical background also key to understand how system have evolved over the years

## Reference

Concurrency Control and Recovery in Database Systems

Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman

- https://www.microsoft.com/en-us/research/people/philbe/book/


Concurrency Control and Recovery in Database Systems
P.A. BERNSTEIN • V. HADZILACOS • N. GOODMAN

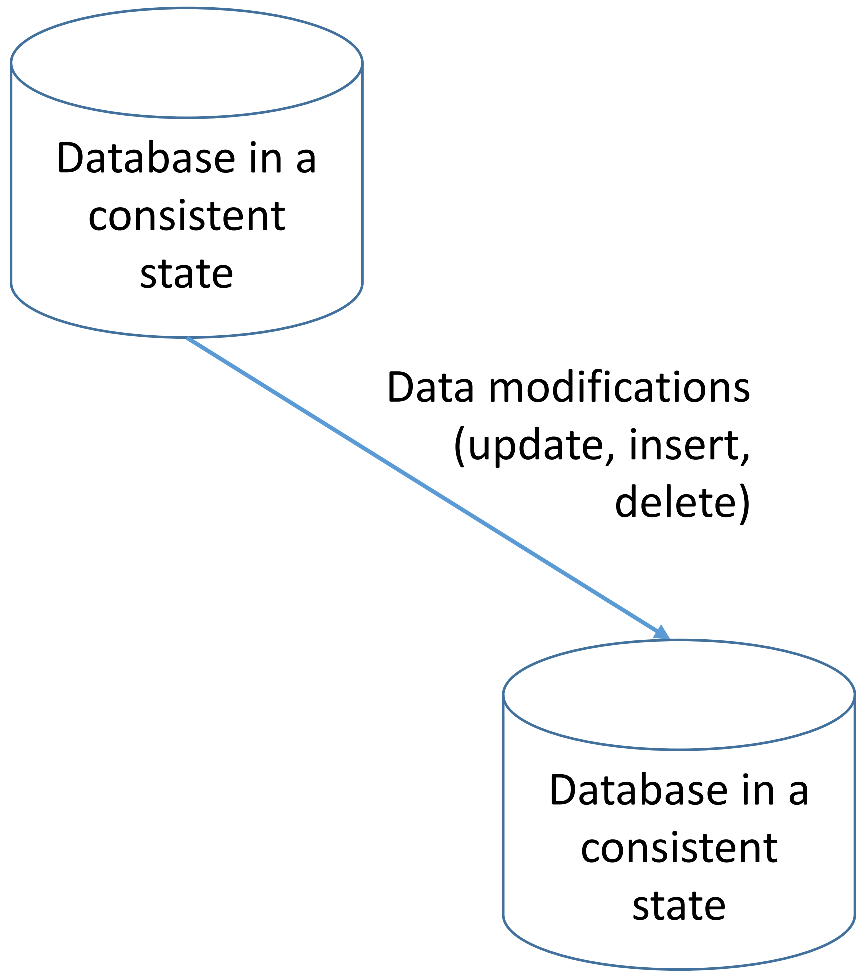
# ACID

# Conventional notion of database correctness

- ACID:
  - Atomicity: the notion that an operation or a group of operations must take place in their entirety or not at all
  - Consistency: operations should take the database from a correct state to another correct state
  - Isolation: concurrent execution of operations should yield results that are predictable and correct
  - Durability: the database needs to remember the state it is in at all moments, even when failures occur
- Like all acronyms, more effort in making it sound cute than in saying something formal

## The formalism behind ACID: consistent state

- To understand ACID, we need to formalize a few concepts:
- Database state: the actual values stored in a database at a given point in time (both in memory as well as in storage)
- Consistent state: A database state that is the result of *correctly* applying *operations* to the database


Database in a consistent state
Data modifications (update, insert, delete)
Database in a consistent state

# More formalism

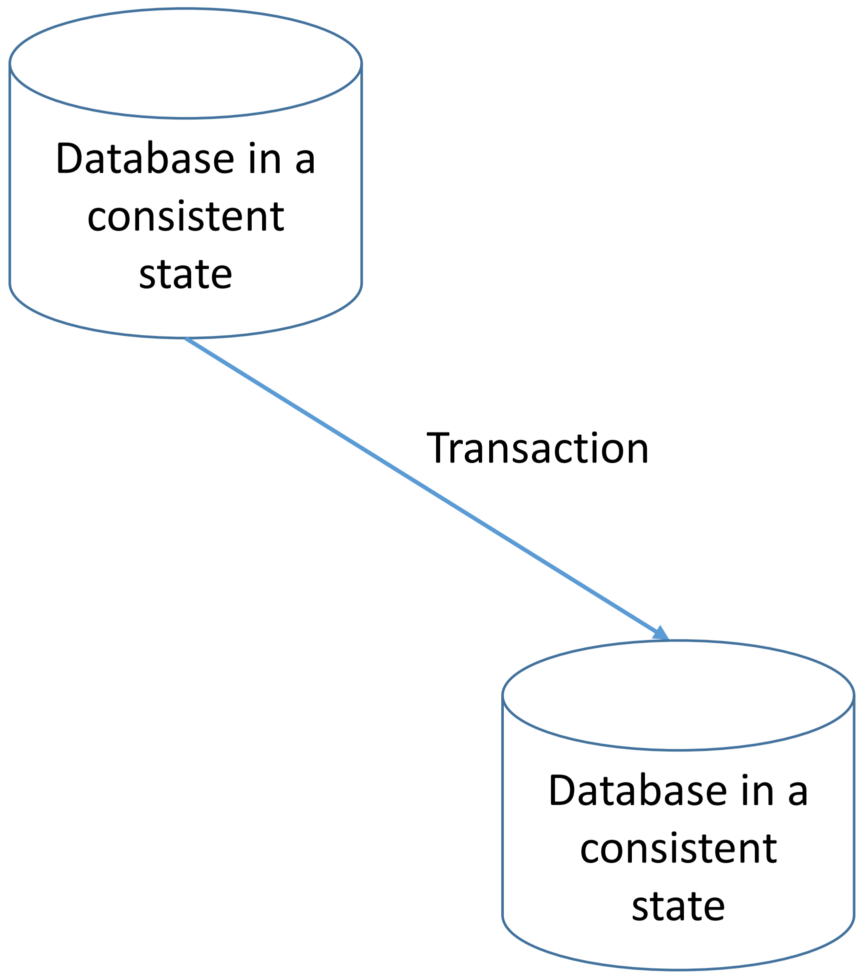
- … correctly …
    - In databases, the underlying model of correct execution of operations is sequential execution: all operations are executed one after each other with no concurrency
- … operations …
    - The operations relevant to transactions are inserts, updates, and deletes of tuples
    - But dealing with these operations one by one is not enough, instead of single operations, data modifications are grouped into a transaction (a series of data modification operations)
- Transactions
    - We assume that transactions are correct. Given a consistent database state as input, they produce a consistent database state as output

## Consistency

- The assumption that transactions are correct is based on two things:
  - The user will not apply wrong transactions (the database cannot control this, it can only assume the user knows what is correct; many database errors are actually data entry errors)
  - The database has mechanisms to prevent incorrect data modifications:
    - Consistency constraints on tables
    - Constraints on values
    - Referential integrity (foreign key constraints)
    - Triggers applying consistency checks
- The database engine will reject transactions that violate constraints specified in the schema
- Database developers can use triggers to perform more checks on data modifications before accepting a transaction

## Atomicity

- Atomicity dictates that a transaction has to be executed in its entirety or not at all
- Only a transaction executed in its entirety brings the database from a consistent state to another consistent state
- Intermediate results cannot be assumed to be correct


Database in a consistent state
Transaction
Database in a consistent state

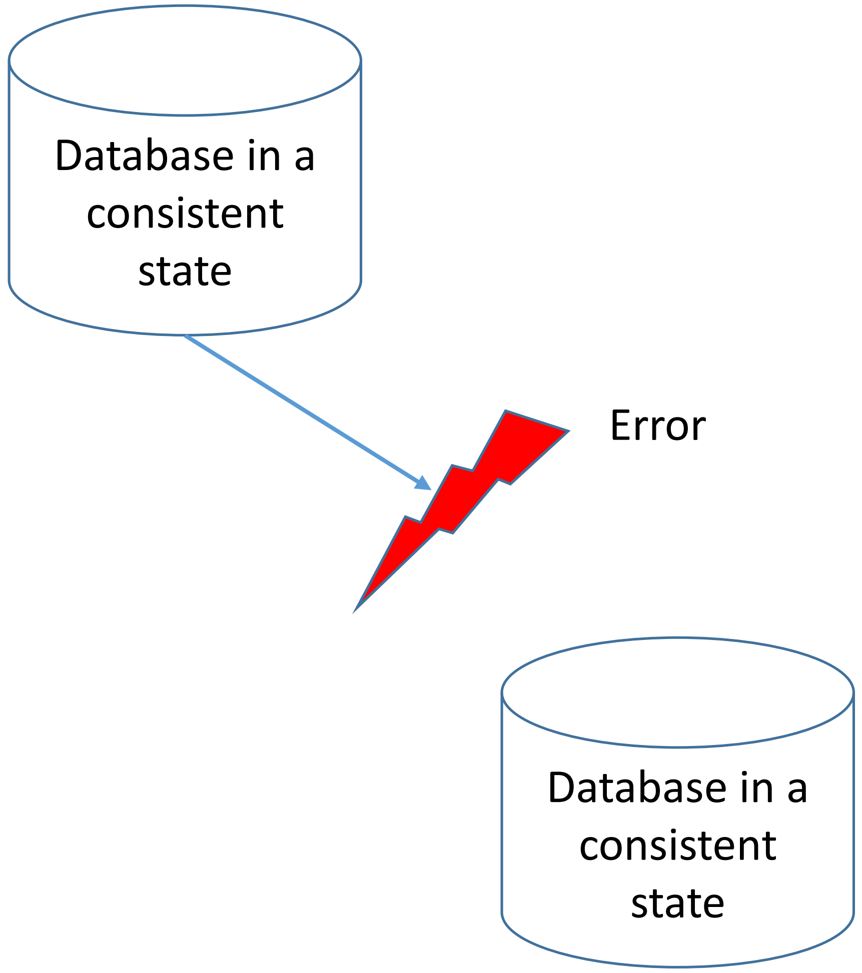
## Why atomicity?

- Logically, many data modification operations go through several states
  - x = x+1
    - read x
    - increment x by 1
    - write x
  - Even more states when more variables are involved or more complex operations (z = x + y)
- In general, the transaction is correct only if executed completely. Intermediate states are not guaranteed to be correct
- Atomicity violations are costly in terms of the effort needed to correct them!!

```
UPDATE employees
SET salary = salary * 1.02
WHERE dept = D11
```

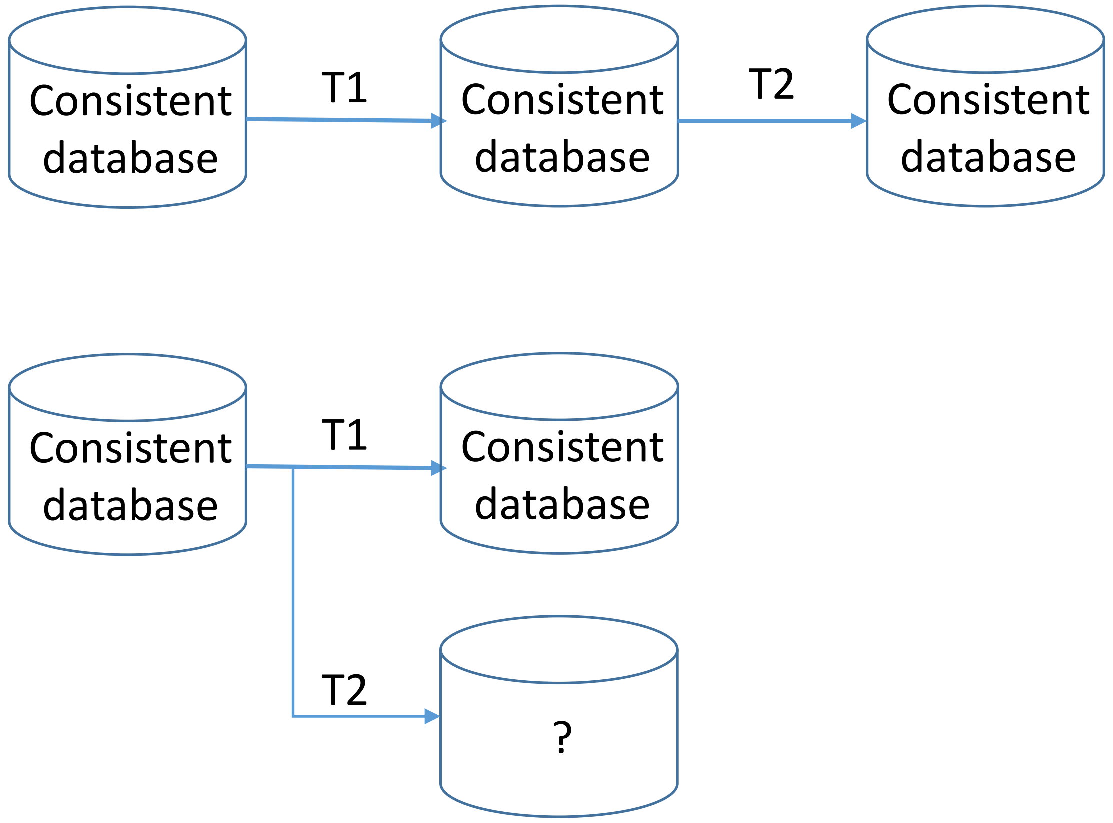
## Implications of atomicity

- If a transaction does not complete, we need to make sure that either:
  - The initial state is restored and any effects of the transaction are removed (undo)
  - The intended final state is reached (redo)
- Anything in between is assumed to be incorrect
- Has to happen even when the system crashes!!


Database in a consistent state
Error
Database in a consistent state

# Atomicity requires isolation

- Since a transaction takes the database from a consistent state to another consistent state, it follows that a sequential execution of transactions leaves the database in a consistent state
- If a transaction can see the intermediate state created by another transaction, its input is not guaranteed to be consistent and, thus the output is not guaranteed to be consistent



## Simple example

T1

```
UPDATE employees
SET salary = salary * 1.02
WHERE dept = D11
```

Q1

```
SELECT sum(salary)
FROM employees
WHERE dept = D11
```

T1 -> Q1 : we read the total value of the salaries after the salaries are raised
Q1 -> T1 : we read the total value of the salaries before the salaries are raised

If the executions are intertwined, Q1 will read a value that is incorrect: it will correspond neither to the state before salaries were raised nor to the state after they were raised.

## Isolation

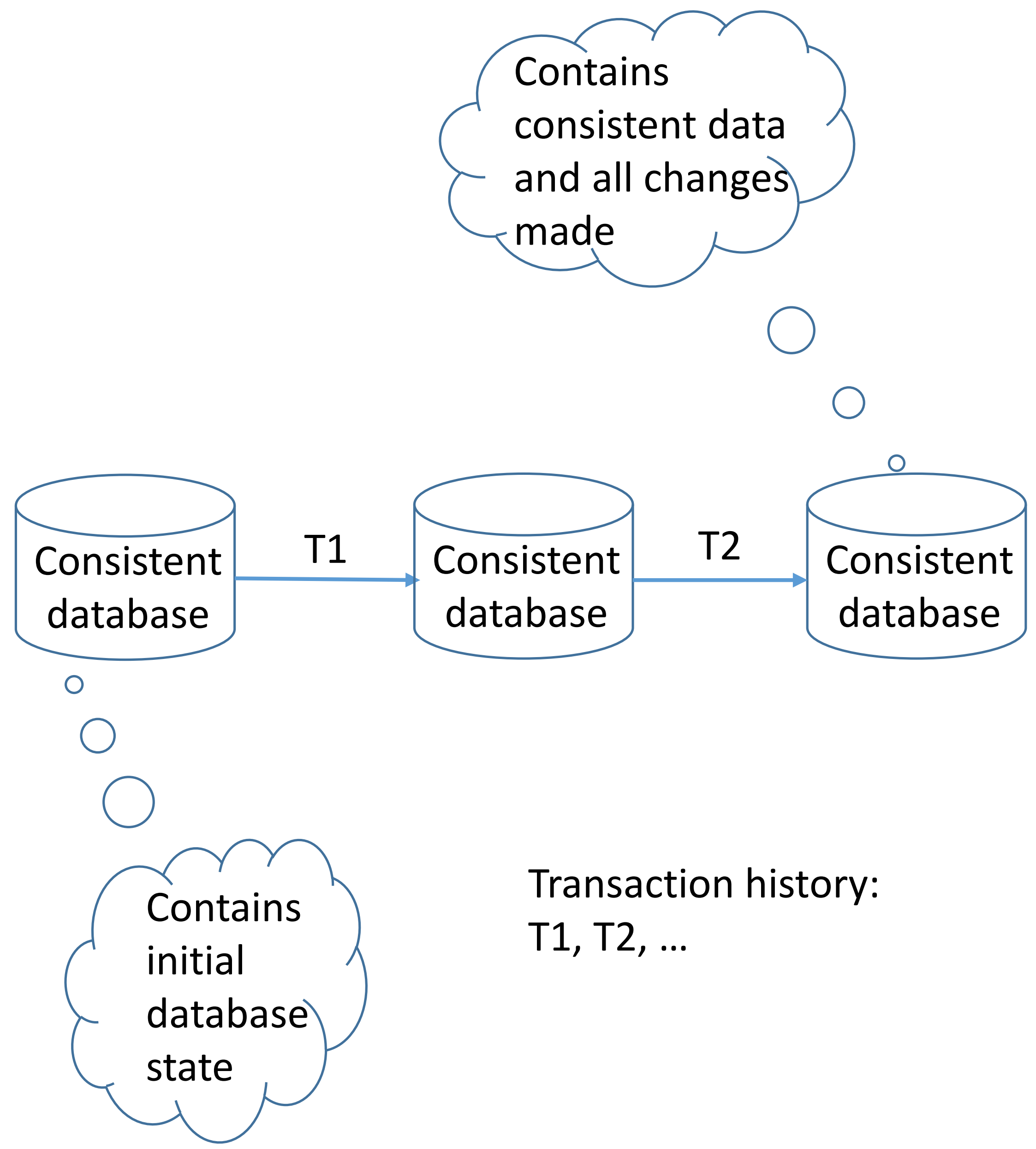
- Transactions are executed in a database as if they were alone in the database and nothing else would be running at the same time
  - Transactions should not see the intermediate state of other transactions
  - Queries should not see the intermediate state of transactions
- The consequences of enforcing isolation:
  - We need to be able to determine when a transaction is done with its changes so that they become visible
  - We need to detect conflicts (operations one the same data that will result in intermediate results being visible)
  - In case of conflicts, we need to have mechanisms to resolve them or prevent them

## Concurrency control and locking

- Isolation in databases is enforced through concurrency control mechanisms
  - Prevent conflicts
  - Ensure a level of consistency by controlling what can be seen of a transaction
- The canonical concurrency control mechanism in databases is locking
  - Use locks to prevent conflicting accesses
  - Use locking policies to ensure transactions are isolated

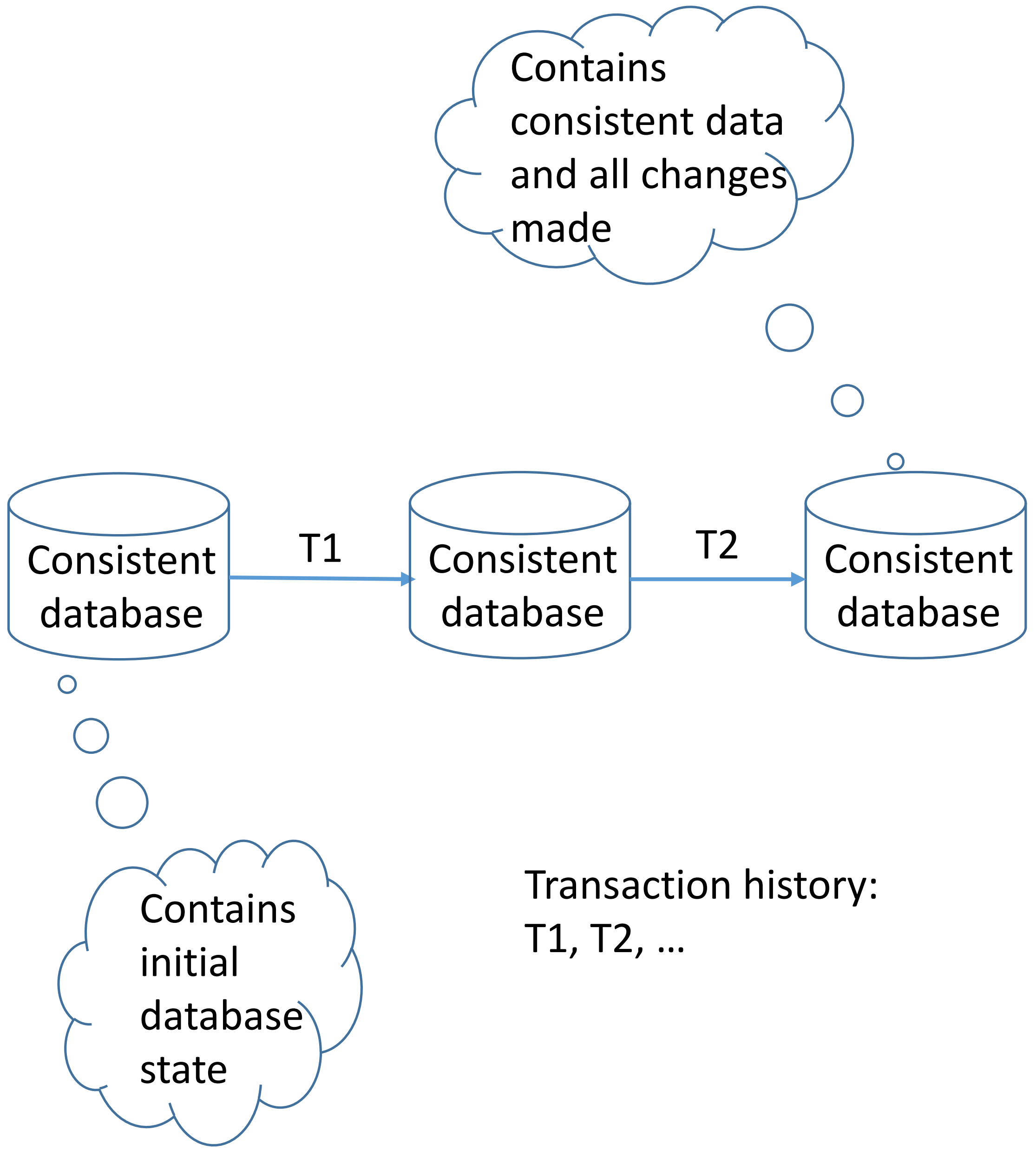
## Durability

- Databases are all about keeping the data persistent and consistent
- Durability ensures that the changes of a completed transaction are remembered by the system and can be recovered
- Often more strict:
  - Databases will keep the database state
  - Databases will keep a history of all transactions


Contains consistent data and all changes made
Consistent database
T1
Consistent database
T2
Consistent database
Contains initial database state
Transaction history: T1, T2, …

# Recovery and the log

- The transaction history is captured in the database log
- The consistent database state at a given point in time is captured in snapshots
- The log and the snapshots allow to go back and forward in the history of the database by undoing or redoing transactions from a given snapshot


Contains consistent data and all changes made
Consistent database
T1
Consistent database
T2
Consistent database
Contains initial database state
Transaction history: T1, T2, …

## Small warning

- Often, concurrency control and recovery are treated as separate problems
- In reality, they are deeply interrelated both at the theoretical as well as at the practical, implementation level
  - How concurrency control is implemented will affect how recovery is implemented
  - How recovery is implemented will restrict hat can be done in terms of concurrency control
- Will become clear when we discuss implementations

# Transaction model

# Defining a transaction

- Begin of transaction (BOT) : often implicit
- Commit: transaction has finished, database confirms to client when all changes of the transaction have been made persistent
- Abort/rollback: transaction is cancelled, database rollbacks all changes done by the transaction
- Read: read a data item
- Write: write a data item
- $a <_T b$ : a happens before b, partial order; implies a will be done before b

## Transactions in real systems

- Often associated with sessions:
  - The firs update statement in a session indicates the beginning of a transaction
  - Subsequent SQL statements are considered to be part of the transaction
  - Finalized with either “COMMIT” or “ROLLBACK”
- Quite a few different possibilities and approaches:
  - Autocommit = will treat every SQL statement as a transaction and commit it when it finishes executing
  - Explicit “BEGIN TRANSACTION” or “BEGIN” statement

## Transactions in real systems

- COMMIT = indicates the transaction will not do any more changes and all the changes made must be made persistent and durable
  - The database engine will confirm the commit only after it is sure the changes are persistent and/or have been recorded
- ROLLBACK = indicates that the transaction is being cancelled and all of its changes must be removed from the system
- SAVEPOINTS = allow to temporarily commit the changes made by a transaction up to the savepoint
- ROLLBACK TO SAVEPOINT = allow to cancel all the changes made by a transaction after the indicated savepoint, bringing the state seen by the transaction to that of the savepoint

## Transaction operations

- Read operations: represent access to a tuple  without modifying it:
  - $r_1[x]$
- Write operations: represent access to a tuple that change the value of the tuple:
  - $w_1[y]$
- Conflicting operations:
  - two operations over the same item with one of them being a write
    - $r_1[x]$ $w_2[x]$
    - $w_1[y]$ $w_2[y]$

## Conflicts: Aborts and Commits

**Abort**

- $r_i(x)$ and $a_j$: Conflict if $T_j$ updated x.
- $w_i(x)$ and $a_j$: Conflict if $T_j$ updated x.
  - N.B. Reads of $T_j$ are irrelevant.

**Commit**

- $r_i(x)$ and $c_j$:  no conflict
- $w_i(x)$ und $c_j$: no conflict

## Histories

- A history H is a partial ordered ($<_H$) sequence of operations from a set of transactions where
  - If two operations are ordered within a transaction, they are equally ordered in the history
  - If two operations, p and q, conflict then they are ordered with respect to each other
    - $p <_H q$ or $q <_H p$

# History of three TAs

$$H = \begin{tikzcd}
 & r_2(A) \arrow[r] & w_2(B) \arrow[r] & w_2(C) \arrow[r] & c_2 \\
r_3(B) \arrow[r] & w_3(A) \arrow[r] \arrow[u] & w_3(B) \arrow[r] \arrow[u] & w_3(C) \arrow[r] \arrow[u] & c_3 \\
r_1(A) \arrow[r] & w_1(A) \arrow[r] \arrow[u] & c_1 & &
\end{tikzcd}$$

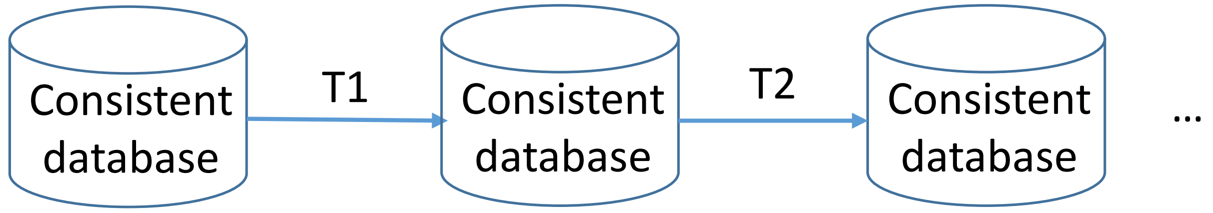
## Concurrency Control

## Goals

- Concurrency control has the goal of ensuring correct executions of concurrent transactions
  - Define a baseline for correctness (serial execution)
  - Define equivalence between histories
  - Define correct histories as those equivalent to serial histories
- Serializability
- Serializability theorem

# Serial history

- A history H is serial if, for every two transactions $T_i$ and $T_j$ that appear in H, either all operations from $T_i$ appear before all operations of $T_j$ or viceversa.
- A serial history with only committed transactions is correct by definition
  - Transactions are isolated
  - Each transaction starts in a consistent state and leaves the database in a consistent state



## Equivalent histories

Two histories are equivalent iff

1. They are over the same transactions and contain the same operations
2. Conflicting operations of non aborted transactions are ordered in the same way in both histories

- History equivalence captures the fact that, in the two histories, committed transactions see the same state (read the same values) and leave the database in the same state.

# Equivalent Histories

$$\begin{tikzcd}
 & & w_1[y] \arrow[dr] \arrow[dddrr, bend left] & & & \\
H_2 = r_1[x] \arrow[r] \arrow[dddr] & r_1[y] \arrow[ur] \arrow[dr] & & c_1 & & \\
 & & w_1[x] \arrow[ur] & & & \\
 & & & r_2[z] \arrow[r] & w_2[y] \arrow[dr] & \\
 & w_2[x] \arrow[uur] \arrow[rrrr] & & & & c_2
\end{tikzcd}$$

$$\begin{tikzcd}
H_3 = r_1[x] \arrow[r] \arrow[ddr] & r_1[y] \arrow[r] & w_1[x] \arrow[r] & w_1[x] \arrow[r] & c_1 \arrow[d] & \\
 & & & r_2[z] \arrow[r] & w_2[y] \arrow[dr] & \\
 & w_2[x] \arrow[uur] \arrow[rrrr] & & & & c_2
\end{tikzcd}$$

$$\begin{tikzcd}
 & & & w_1[x] \arrow[dr] & \\
H_4 = & r_1[x] \arrow[r] & r_1[y] \arrow[ur] \arrow[dr] & & c_1 \\
 & & & w_1[y] \arrow[ur] & \\
 & & r_2[z] \arrow[r] & w_2[y] \arrow[u] \arrow[dr] & \\
 & w_2[x] \arrow[uuu] \arrow[rrr] & & & c_2
\end{tikzcd}$$

# Serializable History

A history is serializable iff it is equivalent to a serial history.

Is the following history serializable? If yes, what is the serial history?

$$\begin{tikzcd}
 & r_1(A) \arrow[rr] & & w_1(A) \arrow[rr] \arrow[ddr, bend left] & & w_1(B) \arrow[r] & c_1 & \\
H = & & r_2(A) \arrow[ur] \arrow[rrr] \arrow[drrrr] & & & w_2(B) \arrow[u] \arrow[r] & c_2 & \\
 & & & & r_3(A) \arrow[rr] & & w_3(A) \arrow[r] & c_3
\end{tikzcd}$$

## Serializability Graph

$$SG(H)= \begin{tikzcd} & T_3 \\ T_2 \arrow[ur] \arrow[dr] & \\ & T_1 \arrow[uu] \end{tikzcd}$$

- $w_1(A) \rightarrow r_3(A)$ in $H$ implies $T_1 \rightarrow T_3$ in SG(H)
- Compact representation of the dependencies in a history.
- If the graph is acyclic, a topological sort gives an equivalent serial history

# Serializability Theorem

A history is serializable iff its serializability graph is acyclic.

**History** H = $w_1(A) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow r_3(B) \rightarrow w_2(A) \rightarrow c_2 \rightarrow w_3(B) \rightarrow c_3$

**Serializability Graph**

$SG(H)=$

$$\begin{tikzcd}
 & T_2 \\
T_1 \arrow[ur] \arrow[dr] & \\
 & T_3
\end{tikzcd}$$

**Topological Sorting**

$$H_s^1 = T_1 \mid T_2 \mid T_3$$

$$H_s^2 = T_1 \mid T_3 \mid T_2$$

$$H \equiv H_s^1 \equiv H_s^2$$

## Summary

- A history captures the execution of transactions over the database
- A history is correct if it is serializable (equivalent to a serial history)

- Serializability is the canonical correctness criterion for databases
- In practice, databases implement different levels of isolation
- Some engines use related but slightly different criteria

- Serializability is not complete, it does not cover a number of important cases that appear when the level of abstraction is lowered (operations, items being updated, tables, inserts and deletes, etc.)
- It also does not cover the case when transactions are still active

## Recovery

## Goals

- Recovery has the aim to make sure that, even in the case of failures, the changes from transactions that have not completed are not visible and those of committed transactions are recorded and visible
  - Define undesirable situations
  - Classify histories according these situations
  - Define how histories can avoid such situations
- Covers conflict cases when transactions are still active
  - Recoverable
  - Avoids Cascading Aborts
  - Strict

# R1-R4: Recovery procedures

- Abort/Rollback of a single transaction (regardless of why)
  - *R1*: Undo all changes from the transaction

- System crash: lose main memory, keep disk
  - *R2*: Redo the changes from committed transactions
  - *R3*: Undo the changes that remain in the system from active transactions

- System crash with loss of disks
  - *R4*: Read consistent snapshot form backup, if available, apply log

# What do these procedures imply

- These procedures imply:
  - We can undo the changes of an active transaction
    - We need to keep a copy of the value before it was modified
    - Shadow pages is a mechanism that would support this
  - We can redo the changes made by committed transactions
    - We need to keep a persistent record of all the changes made by committed transactions
    - This is why there is a redo log
  - We have a consistent snapshot to start with
    - Either move forward by taking a snapshot and apply committed transactions
    - Or move backward by taking the current state and go back to a consistent state by undoing changes from transactions that should not be there
- Focus now is on interaction between concurrent transactions, we will explain logging later on

## Undo - redo

- The changes of an aborted transaction are undone:
  - Typically by restoring the before image of the value modified
- The changes of a committed transaction can be redone:
  - Typically by restoring the after image of the value modified
- Databases log transactions by keeping before and after images of their changes
- Review shadow paging when we studied the storage system

## Recovery on histories

- A transaction $T_1$ reads from another transaction $T_2$ if $T_1$ reads a value written by $T_2$ at a time when $T_2$ was not aborted
- Recoverable (RC) history
  - If $T_i$ reads from $T_j$ and commits, then $c_j < c_i$
- Avoids cascading aborts (ACA) history
  - If $T_i$ reads x from $T_j$, then $c_j < r_i[x]$
- Strict (ST) history
  - If $T_i$ reads from or overwrites a value written by $T_j$ , then $c_j < r_i[x]/ w_i[x]$ or $a_j < r_i[x]/ w_i[x]$

## What do they mean?

- Recoverable:
  - No need to undo a committed transaction because it read the wrong data
  - Transactions commit in their serialization order
- ACA:
  - Aborting a transaction does not cause aborting others
  - Transactions only read from committed transactions
- Strict:
  - Undoing a transaction does not undo the changes of other transactions
  - Transactions do not read or overwrite updates of uncommitted transactions

## Why Recoverable?

- Recoverable
  - … w2[x] r1[x] c1 …
  - if T2 aborts, T1 has read invalid data
  - Since T1 committed the data has been returned to the user
  - Recovery now involves correcting the application that has processed the data and might have acted upon it => typically a very expensive procedure
  - It is avoided by making sure that T1 does not commit until T2 commits, if T2 aborts, then we can safely abort T1 since the results have not been returned to the user

## Why ACA?

- ACA
  - … w2[x] r1[x] a2 …
  - When T2 aborts, we have to abort T1 because it has read invalid data (the change made by T2 which will be undone as part of the rollback of T2)
  - If cascading of aborts happen very often, performance will suffer since aborting one transaction results in other transactions or queries being aborted
  - It is avoided by making sure that uncommitted data is never read (later on, this will be called "read committed")

## Why Strict?

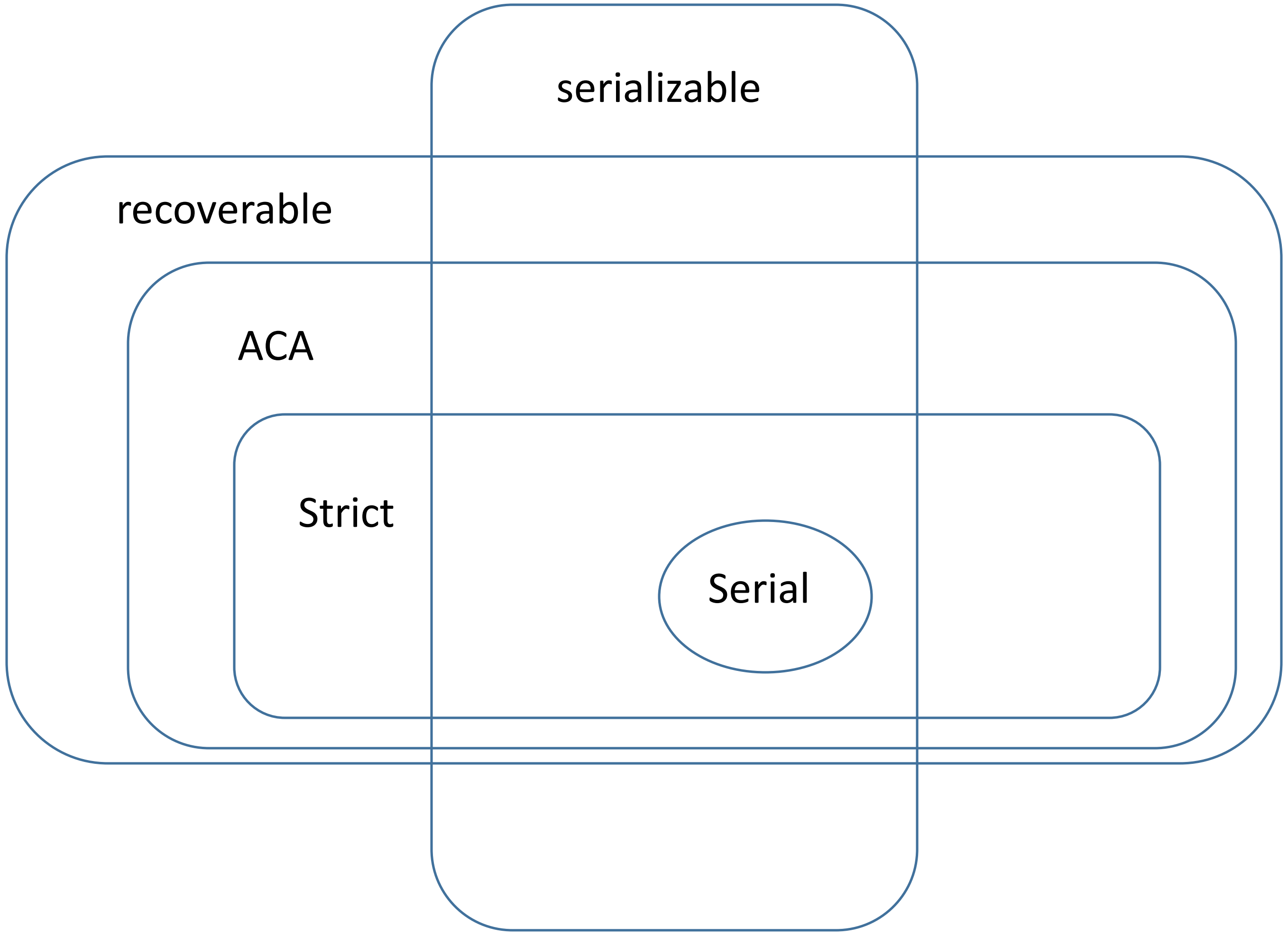
- Strict
  - … w2[x] w1[x] a2 …
  - If we undo w2[x], then we also remove the changes made by T1
  - This is because undoing changes is done by restoring the before image and the before image for w2 does not include the change made by w1
  - Undoing T2 implies aborting T1
  - Like in ACA, non-strict execution leads to cascading aborts of transactions that update the same item which affects performance
  - It is avoided by not letting a value to be read or updated unless it is committed

## Recoverability matters

- Often, most of the emphasis is in concurrency control
- Recovery is equally important because it ensures the state in the database is correct
- Problems with recovery are very expensive:
  - Not RC: I run my program, get some data, finish the program and issue a report. The data read was later removed because another program aborted
  - Not ACA: thrashing behavior when transactions keep aborting each other (my program makes no progress because of other programs)
  - Not Strict: recovery after a failure becomes very complex (or impossible)

## Putting it all together

## Histories



## ANSI isolation levels

- SQL standardized the isolation levels but it did it in a different way than we have just explained.
- The way SQL defines these levels is problematic (and has generated decades of controversy)
  - Consistency levels provided by each system varies slightly
  - Some corner cases become difficult
- SQL defines 3 phenomena to be avoided
- Then it defines 4 isolation levels on the basis of how they avoid these phenomena

# SQL “phenomena”

- Dirty read
  - Occurs when uncommitted data is read
  - Dirty reads result in non-ACA histories
- Non-repeatable read
  - Occurs when a transaction reads the same item at different times and sees different values each time
  - It is the result of somebody else updating the item concurrently
  - Non-repeatable reads correspond to a non-serializable execution
- Phantom reads
  - See next page

## Phantom reads

- Occur when a tuple is inserted or deleted on a table:
  - An aggregate over the table will be different before and after the insert/delete
  - Not a conflict in the formal sense
- It is not necessarily a repeatable read problem because I can read all the values I read before and they have not changed. However, I am missing the new one.
- This is a conflict at the table level (a change to the table rather than a tuple)

| $T_1$ | $T_2$ |
|---|---|
| | **select** sum(balance) |
| | **from** Account |
| **insert into** Account | |
| **values** (*C*,1000,...) | |
| | **select** sum(balance) |
| | **from** Account |

## SQL isolation levels

| ISOLATION LEVEL | Dirty read | Non-repeatable read | Phantom read |
|---|---|---|---|
| Read uncommitted | Allowed | Allowed | Allowed |
| Read committed | Not Allowed | Allowed | Allowed |
| Repeatable read | Not Allowed | Not Allowed | Allowed |
| Serializable | Not Allowed | Not Allowed | Not Allowed |

## These criteria are not comparable

- Read uncommitted:
  - It is not ACA and it allows non-recoverable executions
- Read committed:
  - It is Recoverable but it might not be serializable and not ACA
- Repeatable read
  - It is ACA but might not be serializable and not strict
- Serializable
  - Does not match the canonical definition of serializability (equivalence to a serial history)

# What about updates?

- Note that the phenomena considered do not include write-write conflicts
- The following histories are serializable according to the SQL criteria.
  - … $w_1[x]w_2\,[x]w_2[y]w_1[y]$ …
  - … $r_1[x]w_2[x]r_2[y]w_1[y]$ …
- This issue appears when snapshot isolation is used to implement concurrency control (see later)
- Locking protocols as implemented by existing engines are stronger than the SQL isolation levels because they consider writes (and recovery)

# Isolation and Recovery (Atomicity)

- The following history:

  … $w_1[x]w_2\ [x]w_2[y]w_1[y]$ …
- has several problems:
  - It is not serializable (has a cycle in the conflict graph)
  - It is not strict (values are overwritten before being committed)
- Why is it an issue?
  - Recovery becomes complex if something like that happens
    - Undoing $T_1$ would undo $w_2[x]$
    - Undoing $T_2$ would undo $w_1[y]$
    - If one aborts, one would have to undo the aborted transaction and redo the other one
    - Using before and after images, not enough

## Why the confusion?

- These notions have been developed over several decades
- Different systems used different approaches and different interpretations of the same concepts
- The SQL standard is too vague and focus solely on isolation not recovery
- Commercial systems need to solve the recovery problem so they have implementations that are close to the canonical definitions, which are cleaner, than to the SQL levels
- These days, no longer controversial, only two approaches: locking and snapshot isolation with the differences well understood