Transactional systems

COURSE 4: Databases

Transactional systems

Transaction

- Set of operations on the database, set of statements:
  - insert, update, delete
- Delimited by statements or function calls of type:
  - begin transaction
  - end transaction
- All operations are finalized with success or none is saved in the db.
- A transactional system must
  - manage concurrent transactions.
  - ensure consistent data in case of failure.

Transaction

Statement 1

Statement 2

commit -- end transaction 1

Statement 3

Statement 4

Statement 5

commit -- end transaction 2

Transaction properties

ACID

ATOMICITY | CONSISTENCY | ISOLATION | DURABILITY

- all changes or none
  - collection of steps → single indivisible unit.

- If one operation fails all changes to the database must be undone
  - Failures in transaction, example: statement error, violating unique constraint.
  - System failures, OS crashed.

ATOMICITY | **CONSISTENCY** | ISOLATION | DURABILITY

- If a transaction is run starting from a database in a consistent state, the database must be consistent at the end of the transaction.
  - Database constraints
    - PRIMARY KEY key constraint, UNIQUE, NOT NULL, FOREIGN KEY referential integrity, CHECK
  - Business constrains

ATOMICITY | **CONSISTENCY** | ISOLATION | DURABILITY

- The database may at some point be in an *inconsistent state*.
- Inconsistencies are not visible in a database system (ensured by *atomicity*).
- The old values of any data on which a transaction performs is written to a log file used by a

  → *recovery system*

ATOMICITY | CONSISTENCY | **ISOLATION** | DURABILITY

- The database system must ensure that transactions run without interference.
  - For any pair of transactions T_i, T_j,

  first statement of transaction T_i is executed after T_j finished or

  first statement of transaction T_j is executed after T_i finished.

ATOMICITY

CONSISTENCY

ISOLATION

DURABILITY

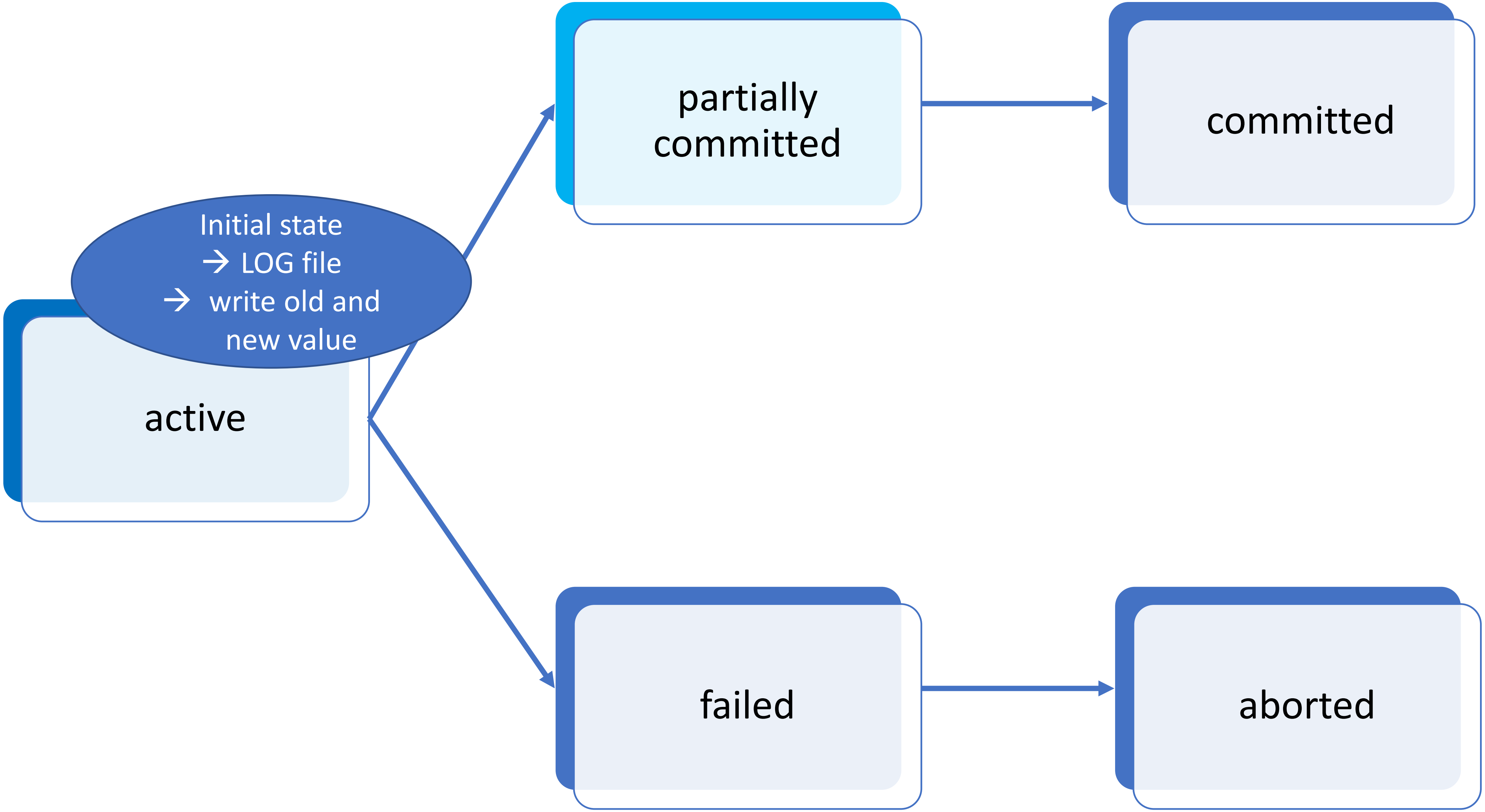
- After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

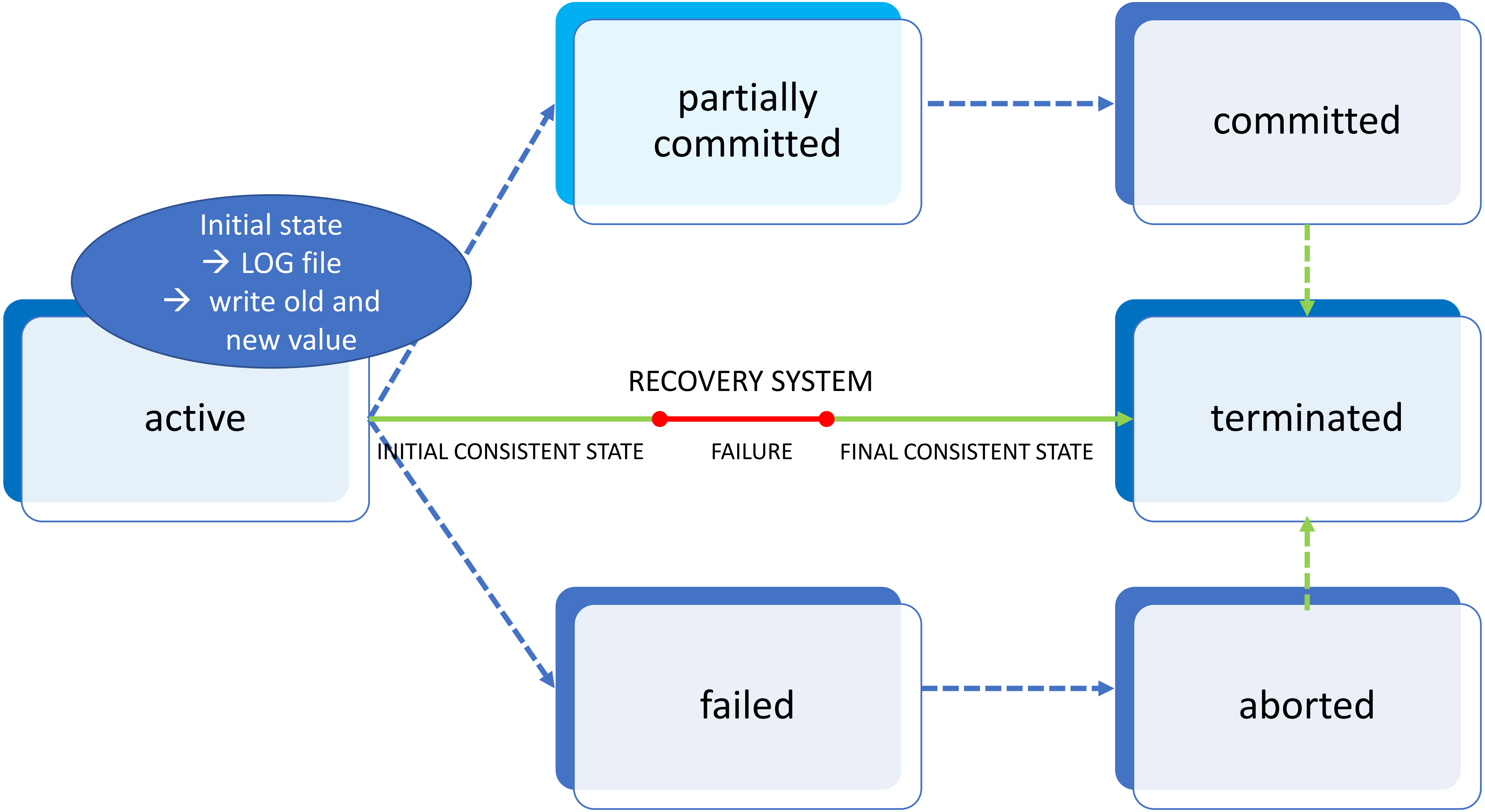
- Information about the updates performed by the transaction is written to disk and used to reconstruct the database after failure.

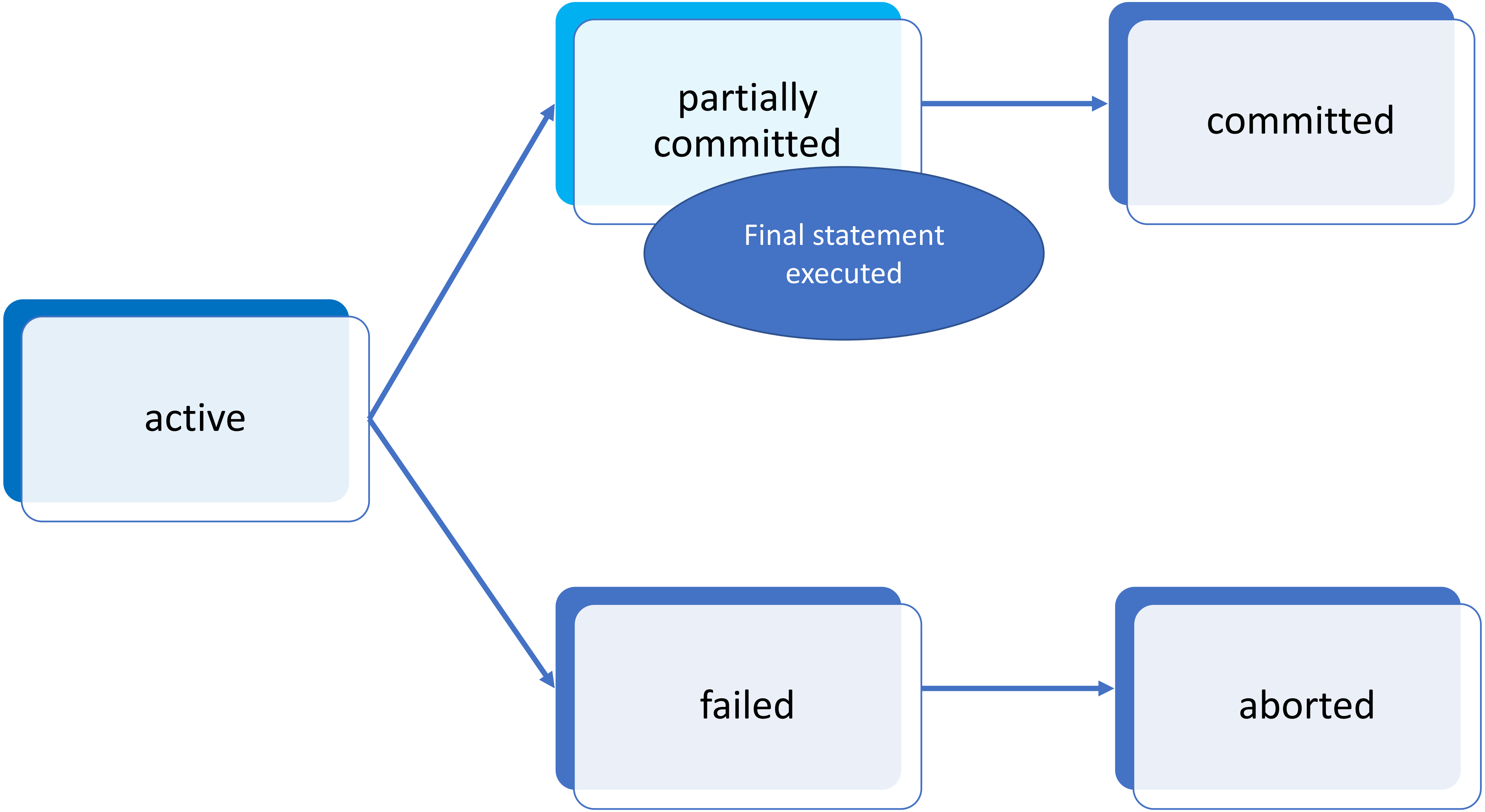
  → *recovery system*

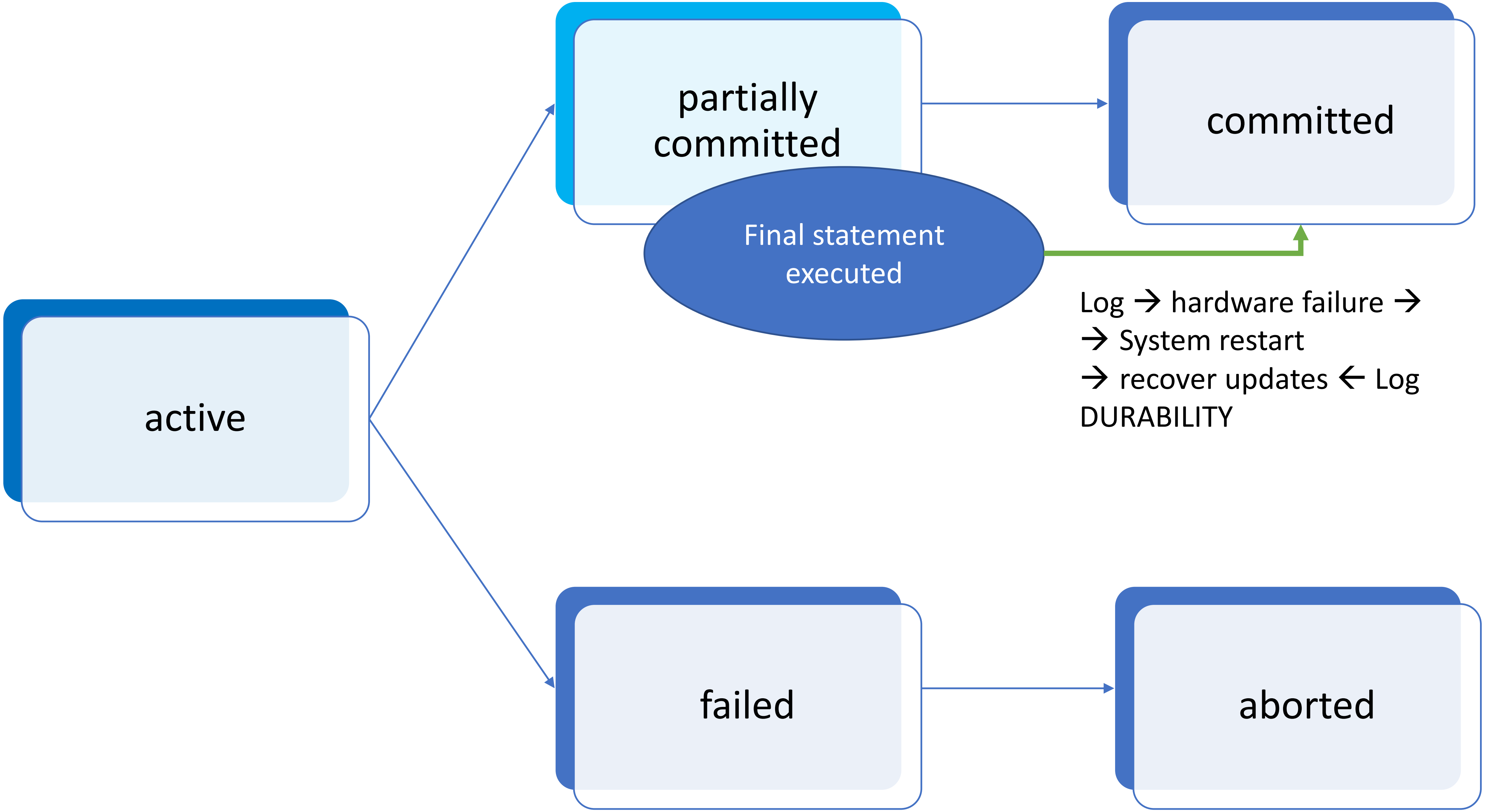
- Please answer www.menti.com 13 52 85   Q1, Q2, Q3, Q4

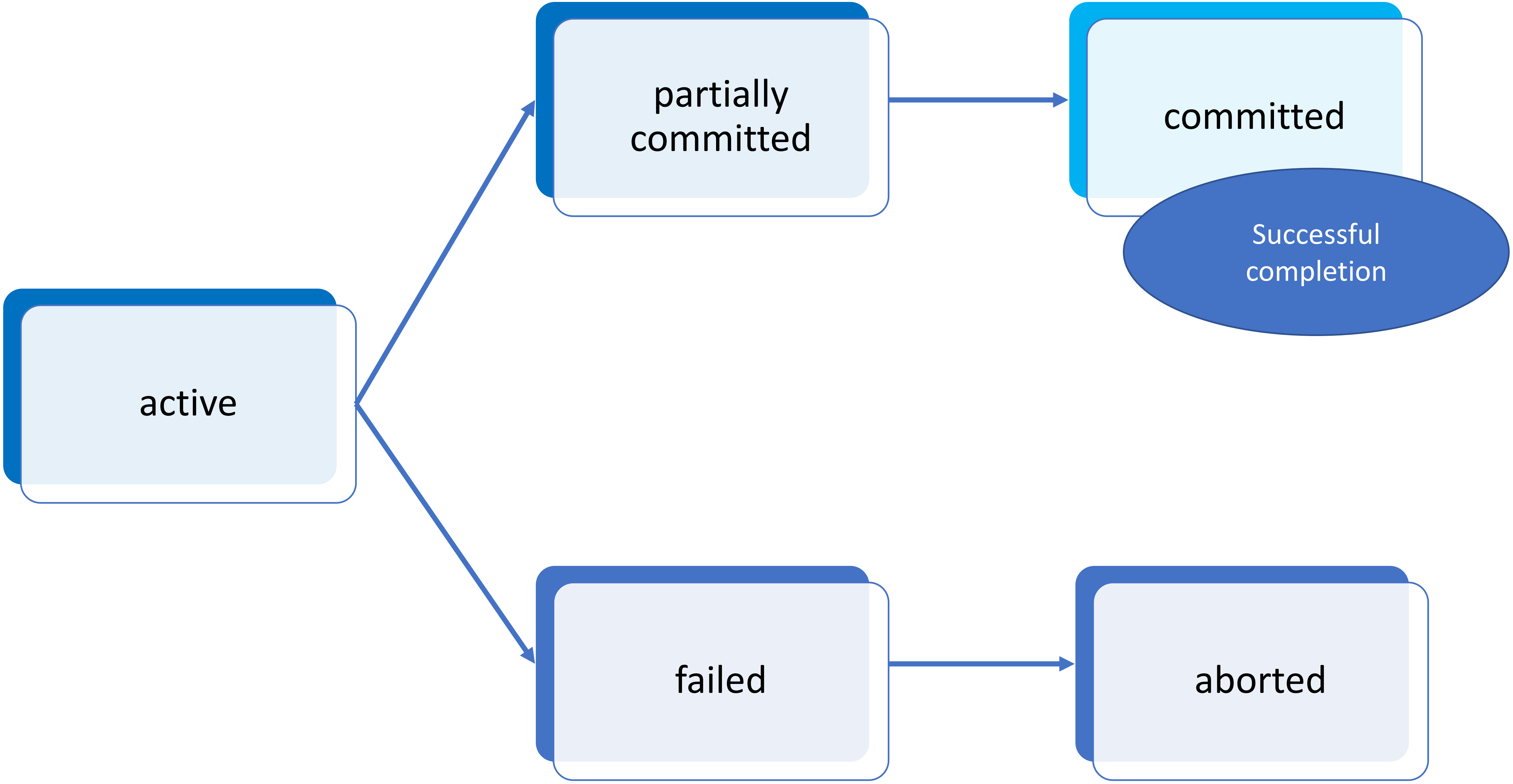
Transaction states

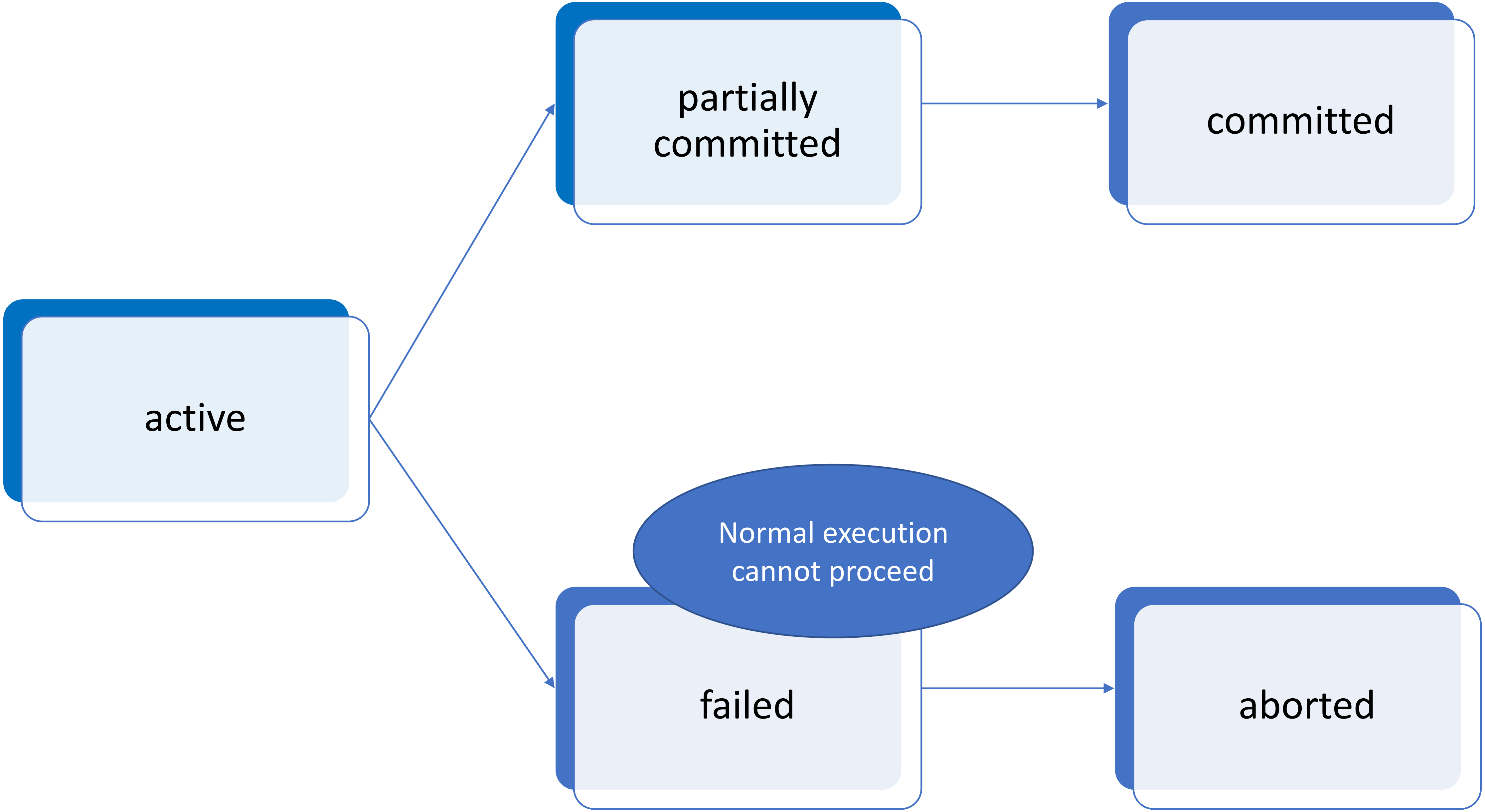

Initial state
→ LOG file
→ write old and new value
active
partially committed
committed
failed
aborted

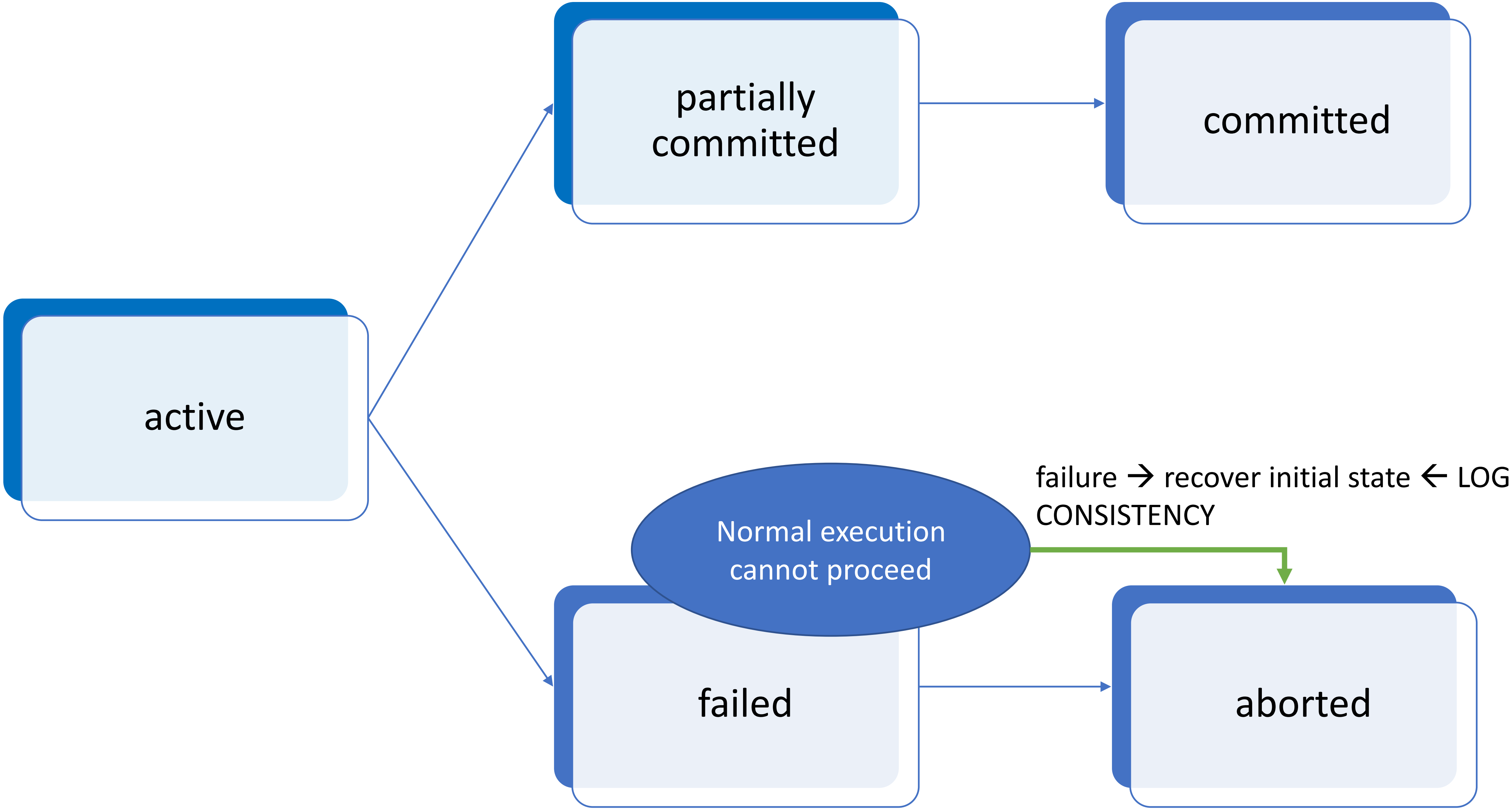

partially committed
committed
Initial state
→ LOG file
→ write old and new value
active
RECOVERY SYSTEM
INITIAL CONSISTENT STATE
FAILURE
FINAL CONSISTENT STATE
terminated
failed
aborted

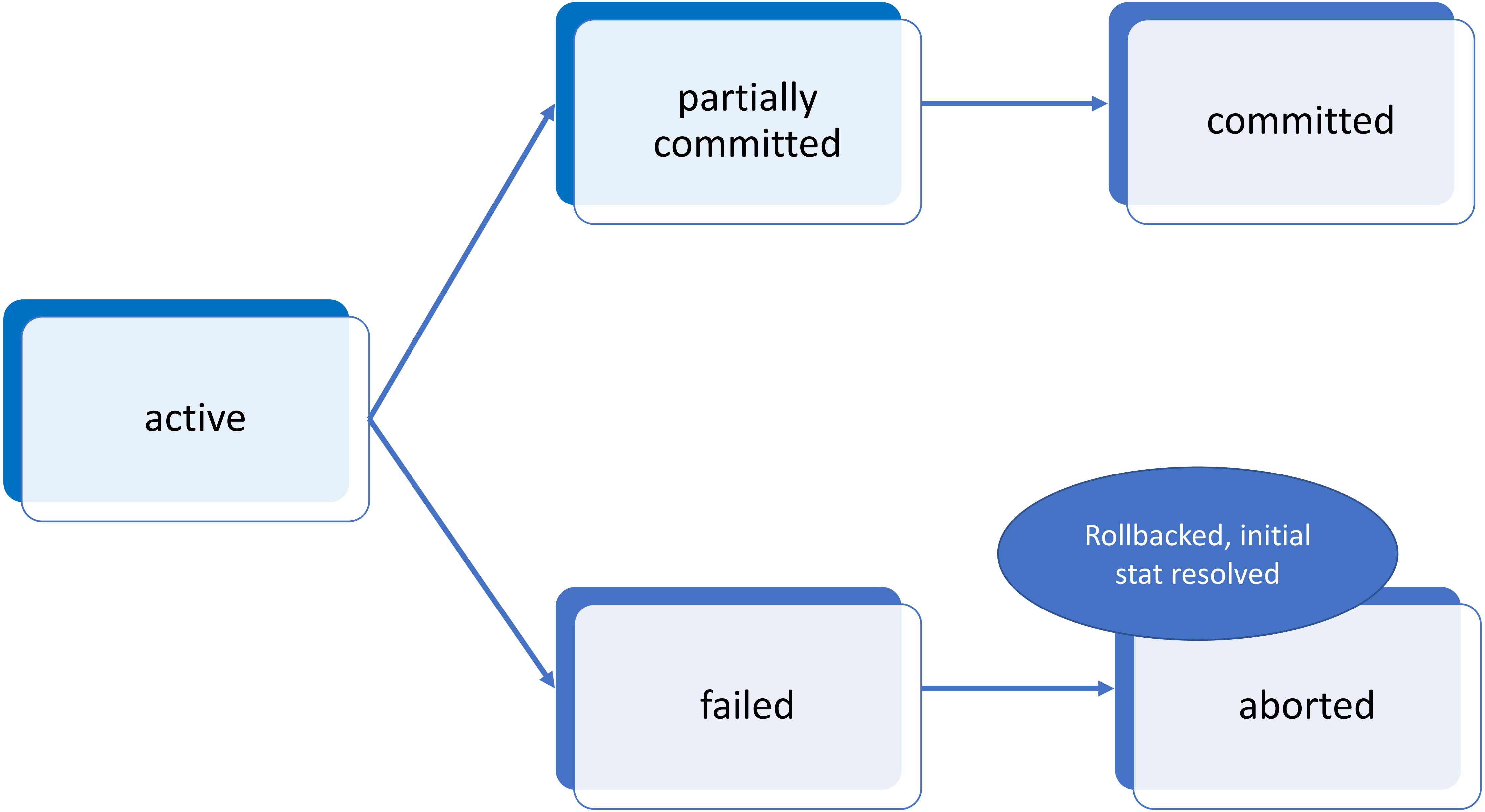

partially committed
committed
Final statement executed
active
failed
aborted


partially committed
committed
Final statement executed
Log → hardware failure →
→ System restart
→ recover updates ← Log
DURABILITY
active
failed
aborted


partially committed
committed
Successful completion
active
failed
aborted


partially committed
committed
active
Normal execution cannot proceed
failed
aborted


partially committed
committed
active
Normal execution cannot proceed
failure → recover initial state ← LOG
CONSISTENCY
failed
aborted


partially committed
committed
active
Rollbacked, initial stat resolved
failed
aborted

Concurrent transactions

Concurrent transactions

- Reduce response time: time for a transaction to be completed.
- Improved workload/resource utilization.
- ISOLATION may be violated → as a result database may be found in an inconsistent state
  - → *Concurrency control*

Concurrent transactions - *conflicts*

- Serial execution preserves consistency, assuming that transactions preserve consistency.

  first statement of transaction T_i is executed after T_j finished or
  first statement of transaction T_j is executed after T_i finished
  single threaded transactions

- Instructions I of T_i and J of T_j conflict ⇔ there exists a *data* accessed by both I and J, and at least one of I an J write *data*.

1. I = read(data)	J = read(data)	I and J don't conflict.
2. I = read(data)	J = write(data)	conflict
3. I = write(data)	J = read(data)	conflict
4. I = write(data)	J = write(data)	conflict

Concurrent transactions -- Schedules

- Schedules: sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions.
  - A schedule must preserve the order in which the instructions appear in each individual transaction.

  - A transaction that successfully completes its execution will have a commit instructions as the last statement
    - By default transaction assumed to execute commit instruction as its last step.
  - A transaction that fails to successfully complete its execution will have an abort instruction as the last statement.

Schedules example S1

- Serial execution.
- No conflicts.
- DB in consistent state
  - A.new + B.new = A.old + B.old

T1	T2

`read (A)`
`A := A - 50`
`write (A)`
`read (B)`
`B := B + 50`
`write (B)`
`commit`	

	`read (A)`
	`temp := A * 0.1`
	`A := A - temp`
	`write (A)`
	`read (B)`
	`B := B + temp`
	`write (B)`
	`commit`

Schedules example S2

- Not a serial execution.
- Equivalent to Schedule S1.
- DB in consistent state
  - A.new + B.new = A.old + B.old

T1	T2

`read (A)`
`A := A - 50`
`write (A)`	

	`read (A)`
	`temp := A * 0.1`
	`A := A - temp`
	`write (A)`

`read (B)`
`B := B + 50`
`write (B)`
`commit`	

	`read (B)`
	`B := B + temp`
	`write (B)`
	`commit`

Concurrent transactions

- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence:
  1. Conflict serializability

  2. View serializability

Concurrent transactions

- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence:

1. Conflict serializability

   If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are conflict equivalent.

2. View serializability

Schedules example S2

- Not a serial execution.
- Equivalent to Schedule S1.
- DB in consistent state
  - A.new + B.new = A.old + B.old

no conflict, no data item is updated by both blocks, by swapping the two blocks we obtain S1

T1	T2

`read (A)`
`A := A - 50`
`write (A)`	

	`read (A)`
	`temp := A * 0.1`
	`A := A - temp`
	`write (A)`

`read (B)`
`B := B + 50`
`write (B)`
`commit`	

	`read (B)`
	`B := B + temp`
	`write (B)`
	`commit`

Schedules example S3

- Not a serial execution.
- Not equivalent to Schedule S1.
- DB in inconsistent state
  - A.new + B.new != A.old + B.old

T1	T2
read (A) A := A - 50	
	read (A) temp := A * 0.1 A := A - temp write (A)
write (A) read (B) B := B + 50 write (B) commit	
	read (B) B := B + temp write (B) commit

conflict, A is updated by both blocks

Concurrent transactions

1. Conflict serializability
2. View serializability

Let S and S’ be 2 schedules with the same set of transactions. S and S’ are view equivalent if the following 3 conditions are met, for each data item Q:

- If in schedule S, transaction Ti reads the initial value of Q, then in schedule S’ also transaction Ti must read the initial value of Q.
- If in schedule S transaction Ti executes read(Q), and that value was produced by transaction Tj (if any), then in schedule S’ also transaction Ti must read the value of Q that was produced by the same write(Q) operation of transaction Tj .
- The transaction (if any) that performs the final write(Q) operation in schedule S must also perform the final write(Q) operation in schedule S’.

View equivalence is also based purely on reads and writes alone.

Concurrent transactions

- Test serializability:
  1. Conflict serializability

     - Consider some schedule of a set of transactions T1, T2, ..., Tn
     - Precedence graph — a direct graph where the vertices are the transactions (names).
     - We draw an arc from Ti to Tj if the 2 transaction conflict, and Ti accessed the data item on which the conflict arose earlier.
     - We may label the arc by the item that was accessed.
     - A schedule is CS if and only if its precedence graph is acyclic.
     - If precedence graph is acyclic, the serializability order can be obtained by a **topological sorting** of the graph.

Concurrent transactions

- Test serializability :

   2. View serializability

      - The problem of checking if a schedule is view serializable falls in the class of NP-complete problems. Thus, existence of an efficient algorithm is extremely unlikely.

      - Practical algorithms that just check some sufficient conditions for view serializability can still be used.

Please answer www.menti.com 13 52 85 Q5

Isolation levels

Isolation levels

- **Isolation:** execute a transaction ***as if*** there are no other concurrent transactions running simultaneously.
  - Prevent read or write of incorrect, temporary, aborted data processed by concurrent transactions
- **Isolation levels:** trade off between *perfect* isolation and performance
  - response time: time before a transaction completes
  - throughput: number of transactions per second

Level **Serializability**, perfect isolation

- The final state of the database is equivalent to a state of the database if the transactions were run sequentially.
  - serializable schedule

- Way of obtaining serializability:
  - locking
  - timestamp validation
  - multi-versioning

Transactions errors

T1	T2
select qte into **:nS** from stock where n_prod = 100 --nS = 13	
	select qte into **:nS** from stock where n_prod = 100
update stock set qte = **:nS - 1** where n_prod = 100	
	update stock set qte = **:nS - 1** where n_prod = 100
insert into orders(n_prod, qte) values(100, 1) commit	
	insert into orders(n_prod, qte) values(100, 1)

lost-update anomaly

final stock 12!

Transactions errors

T1	T2
select qte into **:nS** from stock where n_prod = 100 --nS = **13**	
update stock set qte = **:nS - 1** where n_prod = 100	
	select sum(qte) into **:nO** from orders where n_prod = 100 select qte into **:nS** from stock where n_prod = 100 --nO+nS!=**init_stock**
insert into orders(n_prod, qte) values(100, 1) commit	

dirty-read anomaly
number of products ordered + qte_stock != initial stock
1 product missing!
Read uncommitted data

Transactions errors

T1	T2
…	select qte into **:nS** from stock where n_prod = 100 --nS = **10**
	if nS < 15 and nS >= 10 insert into restock(n_prod, qte) values(100, 5)
update stock set qte = **:nS - 1** where n_prod = 100 insert into orders (n_prod, qte) values(100, 1) commit	
	select qte into **:nS** from stock where n_prod = 100 if nS < 10 insert into restock(n_prod, qte) values(100, 15)

non-repeatable read anomaly

only one insert into restock is needed!

read twice, different values

Transactions errors

phantom-read anomaly
AVG > MAX!

new lignes inserted

T1	T2
	select MAX(qte) into **:max** from orders where n_prod = 100
insert into orders(n_prod,qte) values(100, **789455**) commit	
	select AVG(qte) into **:avarage** from orders where n_prod = 100

Transactions errors

T1	T2

`select qte into :nS`
`from stock`
`where n_prod = 100`
`--nS = 13`	

`update stock`
`set qte = :nS - 1`
`where n_prod = 100`	

	`select qte into :nS`
	`from stock`
	`where n_prod = 100`
	`--nS = 12`

	`update stock`
	`set qte = :nS - 1`
	`where n_prod = 100`
	`--nS = 11`

`abort`	
	`insert …` `commit`

dirty-write anomaly
final stock 11! In the first transaction, the stock returns to 13. Only one update should decrease the number of products.

Isolation levels

- weaker the isolation level → more anomalies may occur

ERROR / LEVEL	lost-update	dirty-reads	non-repeatable reads	phantom
READ UNCOMMITTED	✗	✓	✓	✓
READ COMMITTED	✗	✗	✓	✓
REPEATABLE READ	✗	✗	✗	✓
SERIALIZABLE	✗	✗	✗	✗

Isolation levels

ERROR / LEVEL	lost-update	dirty-reads	non-repeatable reads	phantom
REPEATABLE READ	✗	✗	✗	✓

- read only committed
- between two reads of an item by a transaction, no other transaction is allowed to update it.
- a transaction may find other data inserted by a committed transaction

Isolation levels

ERROR / LEVEL	lost-update	dirty-reads	non-repeatable reads	phantom
READ COMMITTED	✗	✗	✓	✓

- read only committed
- does not require repeatable reads. Between two reads of a data item by the transaction, another transaction may have updated the data item and committed.

Isolation levels

ERROR / LEVEL	lost-update	dirty-reads	non-repeatable reads	phantom
READ UNCOMMITTED	✗	✓	✓	✓

- allows uncommitted data to be read
- all the isolation levels prevent writes to a data item that has already been written by another transaction not yet committed or aborted (rollbacked).

- Please answer www.menti.com 13 52 85 Q6, Q7

Achieving isolation

- Versioning
  - Transactions read from a “snapshot” of the database.
- Locking
- Timestamp

Locking

- Locks prevent destructive interactions between transactions accessing the same resource.
  - Shared access to read
  - Exclusive access to read and write

  - Locks (Shared, Shared) compatible.
  - Locks (Shared, Exclusive) not compatible.

- A transaction waits until all incompatible locks held by other transactions are released.

- https://oracle-base.com/articles/misc/deadlocks
- https://docs.oracle.com/cd/B19306_01/server.102/b14220/consist.htm

Snapshot isolation

- Snapshot of the database at the beginning of each transaction.
- The transaction operates only on that snapshot.
- The snapshot consists only of committed values.
- Updates are kept in transaction workspace until commit.
- Implemented with timestamp-versioning

Consistency levels

To be added, more info in the following video.

BASE

NoSql consistency model

To be added, more info in the following video.