Data-driven systems

Transactions in relational databases

Automatizálási és
Alkalmazott
Informatikai Tanszék

Transactions

Problems

Problems 2 ☺

Problems

- Concurrent transactions
- A unit of data (here: a record of a database table) is used by **multiple actors** at the same time, and **at least one of the actors edits** the data.

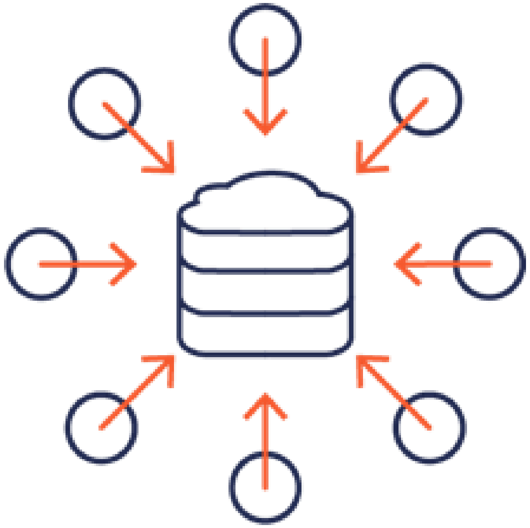


Image source: https://blog.yugabyte.com/a-primer-on-acid-transactions/

Transaction

- The logical unit of a process; a series of operations that only make sense together.
- Basic properties:
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Atomicity and Consistency

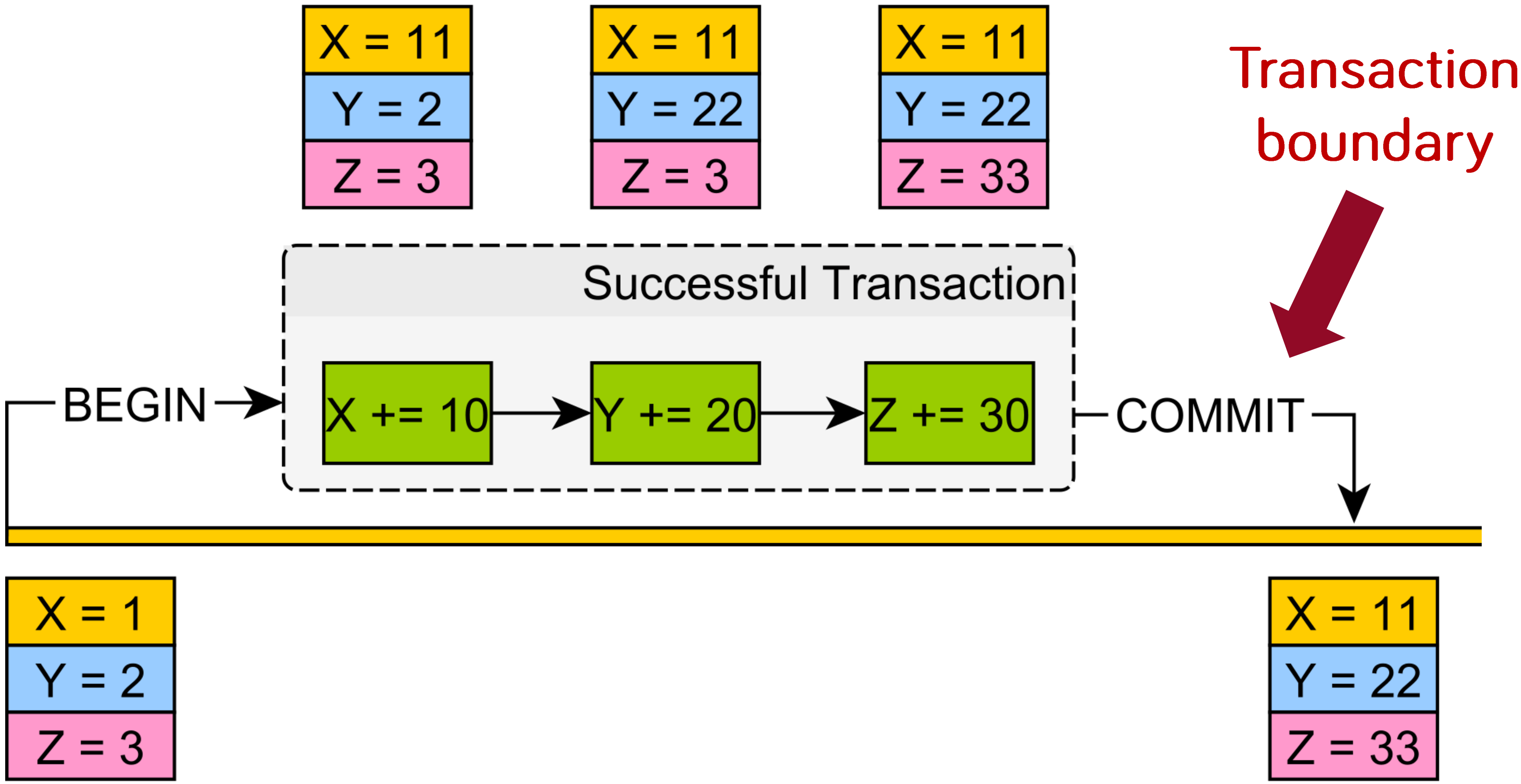
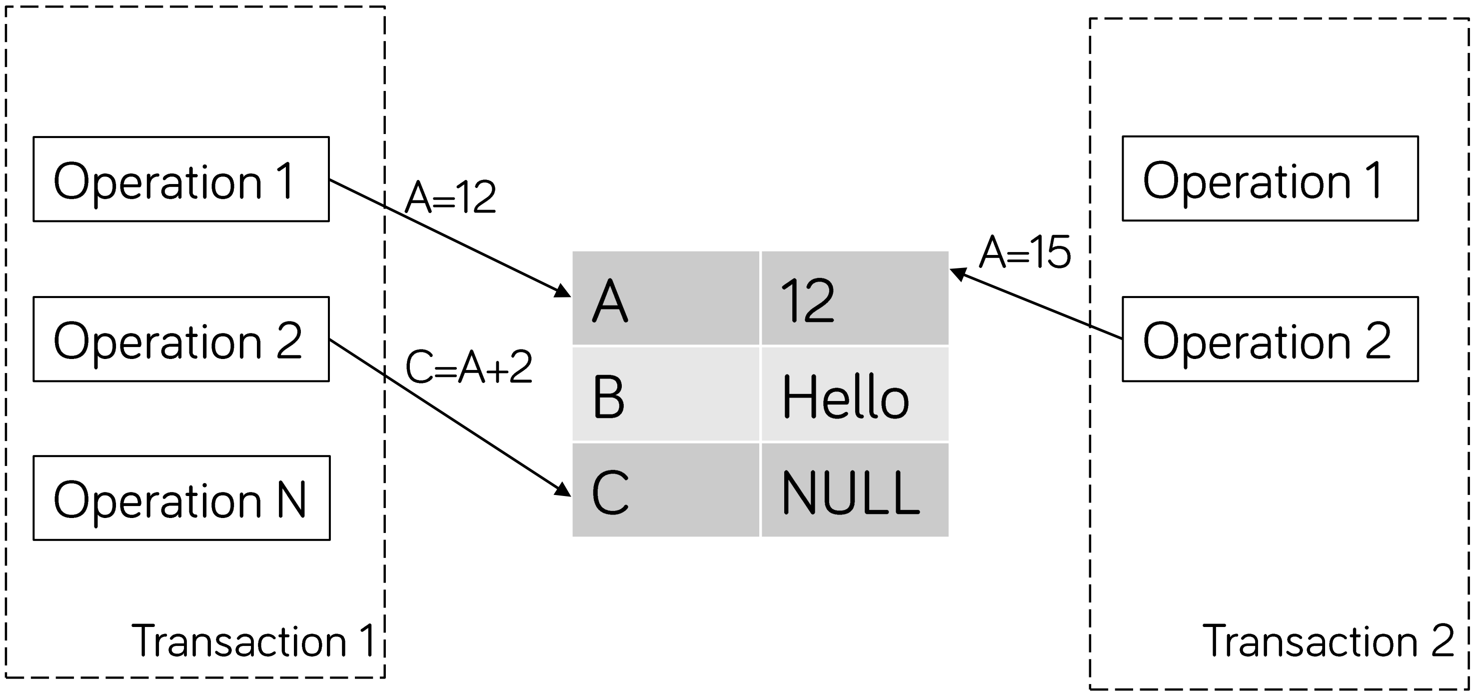


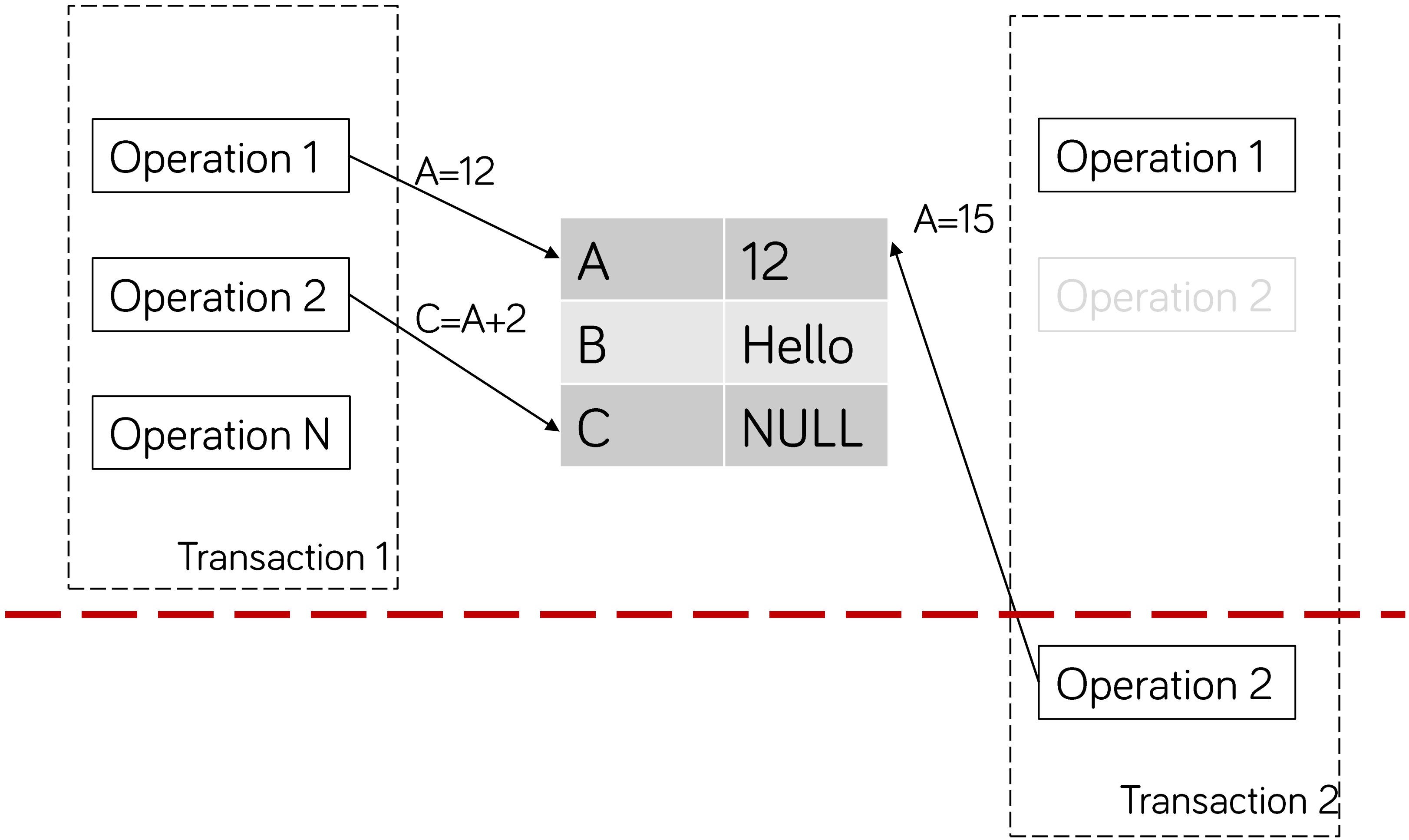
Image source: https://vladmihalcea.com/current-database-transactio

Isolation



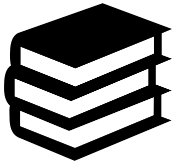
Operation 1
Operation 2
Operation N
Transaction 1
A=12
C=A+2
A
12
B
Hello
C
NULL
A=15
Operation 1
Operation 2
Transaction 2

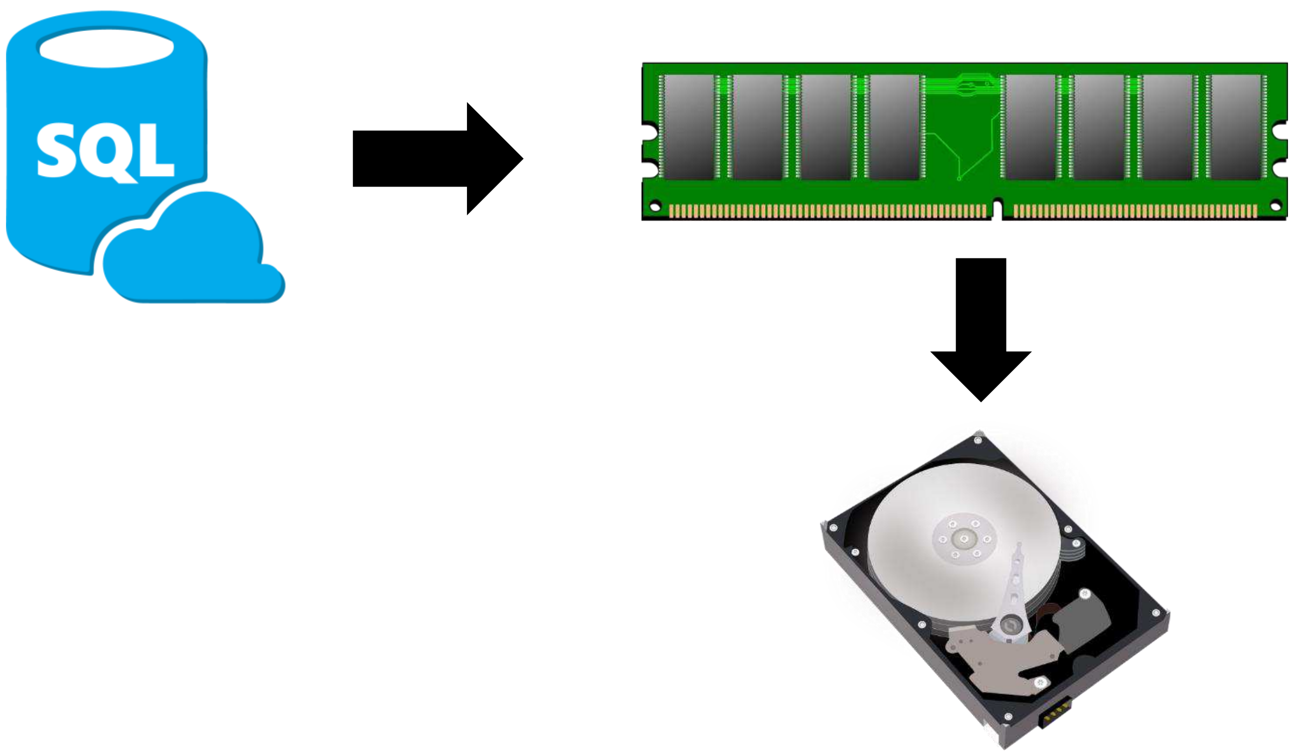
Isolation


Operation 1
Operation 2
Operation N
Transaction 1
A=12
C=A+2
A
12
B
Hello
C
NULL
A=15
Operation 1
Operation 2
Operation 2
Transaction 2

Durability

- Transactional logging





Transaction boundaries

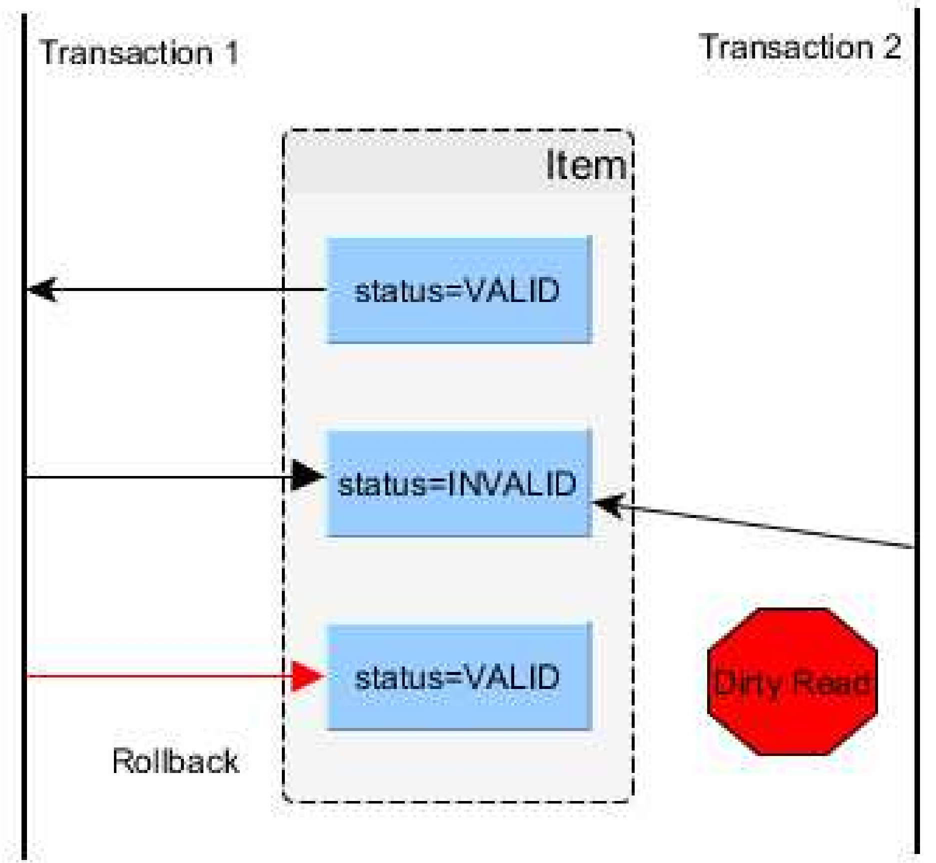
- We can start and close a transaction explicitly
- begin transaction -> commit / rollback
- We can set the transaction level
- Nested transactions
- If not in a transaction, each statement runs as a transaction by itself
- A delete statement that deletes multiple rows cannot delete only half of the records

Transaction isolation

Isolation problems

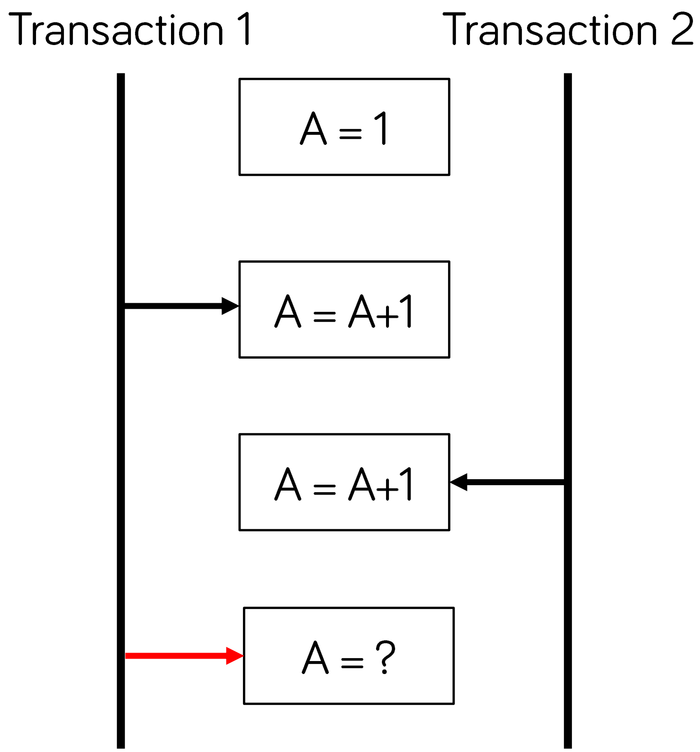
- Multiple concurrent transactions
- Must be executed ***as if*** they were executed after each other and not concurrently
- Problems
  - Dirty read
  - Lost update
  - Non-repeatable read
  - Phantom records
- **Problem is not specific to relational data bases!**
  - Same issue is present is some form in all distributed systems.

Dirty read

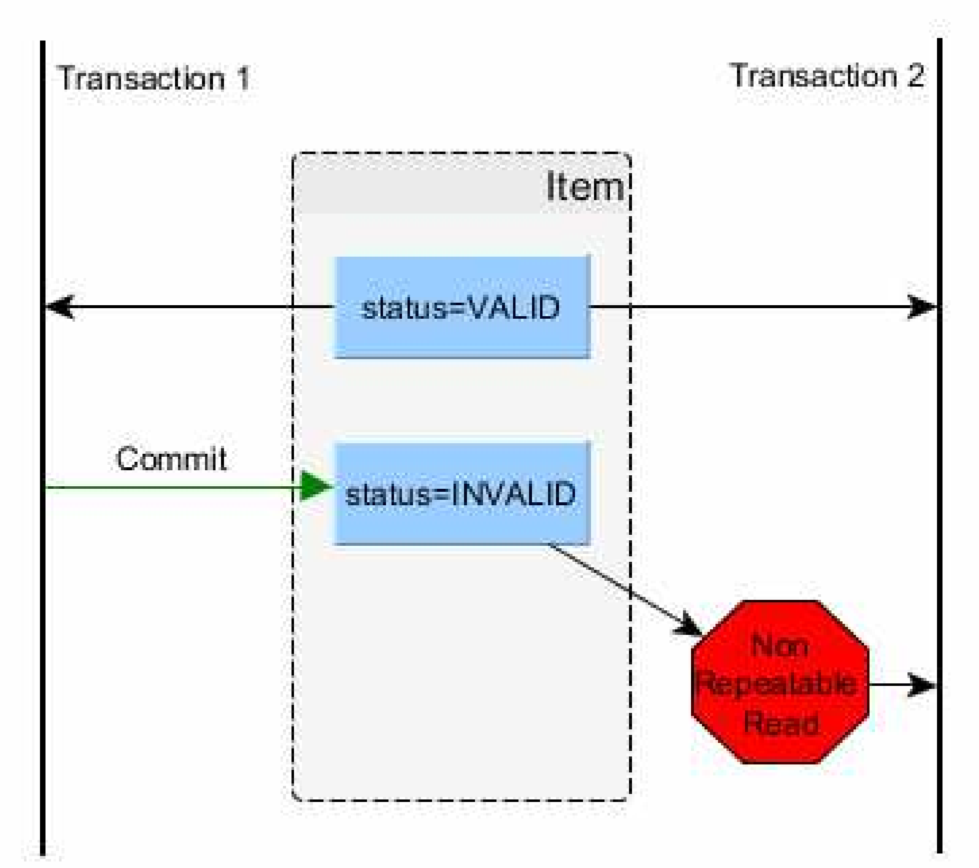


https://vladmihalcea.com/2014/01/05/a-beginners-guide-to-acid-and-database-transactions/

Lost update

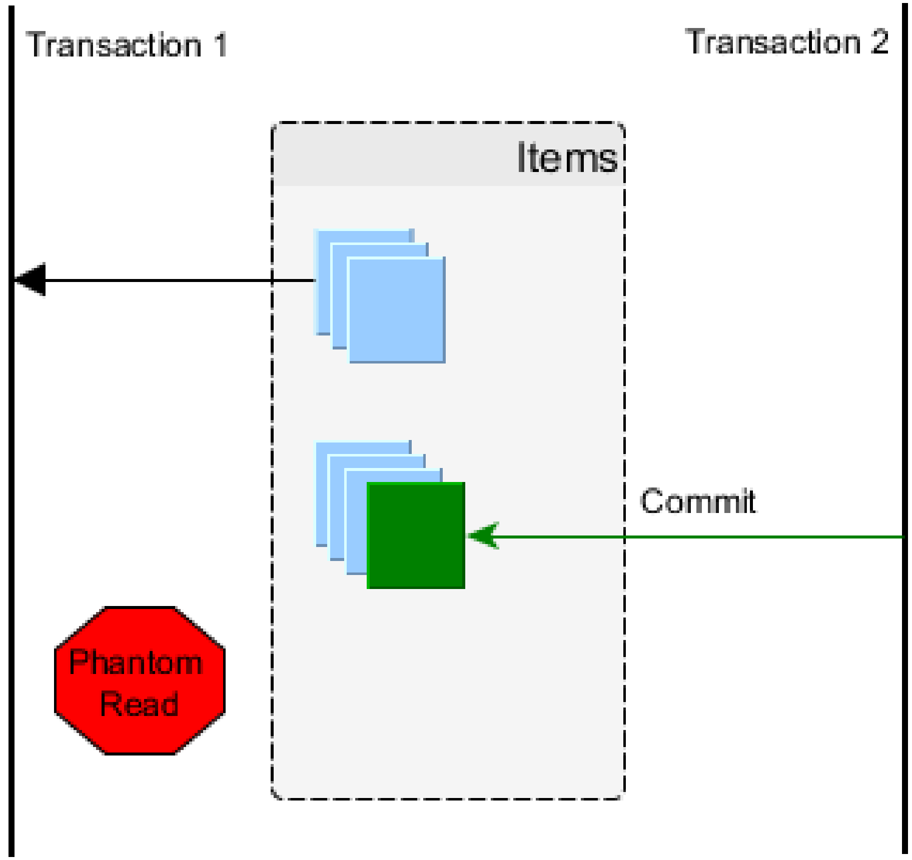


Non-repeatable read



https://vladmihalcea.com/2014/01/05/a-beginners-guide-to-acid-and-database-transactions/

Phantom records



https://vladmihalcea.com/2014/01/05/a-beginners-guide-to-acid-and-database-transactions/

Solution

- Transaction scheduling
- Only operations that do not violate right scheduling are allowed
- If the scheduling is violated, the transaction must wait
- A scheduling is allowed if there exists a conflict-equivalent serial scheduling
  - Can be transformed to a serial scheduling that avoids conflicts

Isolation levels

- Isolation levels in the SQL standard
  - Read uncommitted → all 4 problems
  - Read committed → no dirty read
  - Repeatable read →no dirty read, no non-repeatable read
  - Serializable → no problems
- Non-standard isolation levels (MS SQL)
  - Read committed snapshot isolation
  - Snapshot isolation

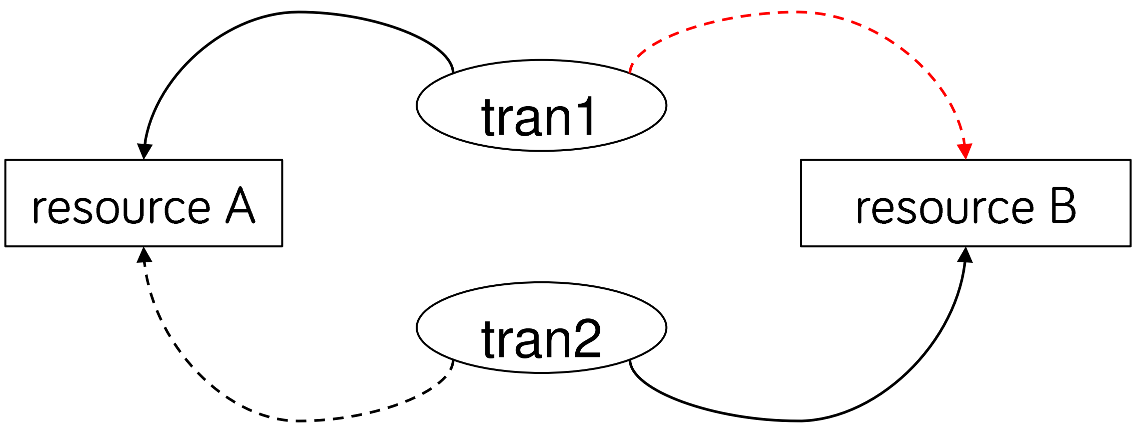
Isolation with locks

- Locks: the data has only one storage location, it is protected by the lock
- One transaction modifies it -> puts a lock on it
- The others cannot read it, because they would read the new, dirty data

- Depending on the implementation, there are several types of locks
  - Read / write
  - Row / page / table

Provides scheduling

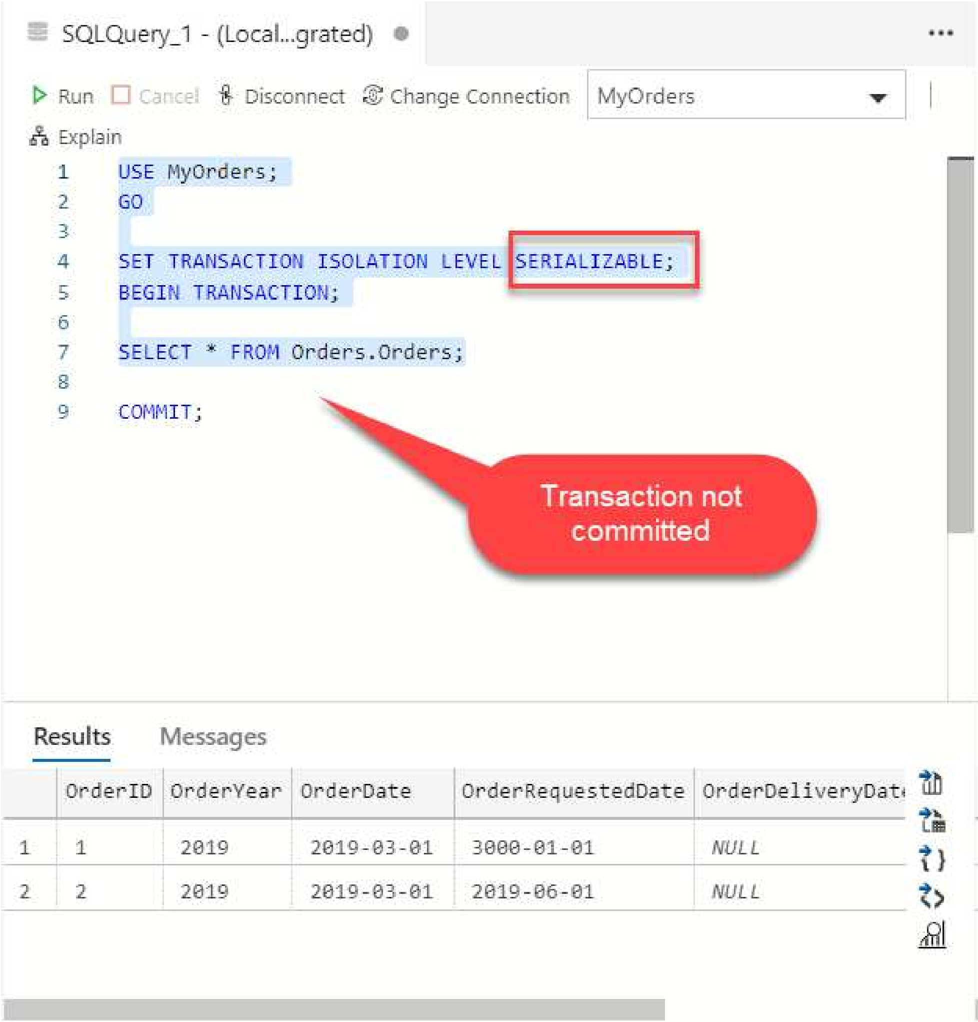
- Two-phase locking

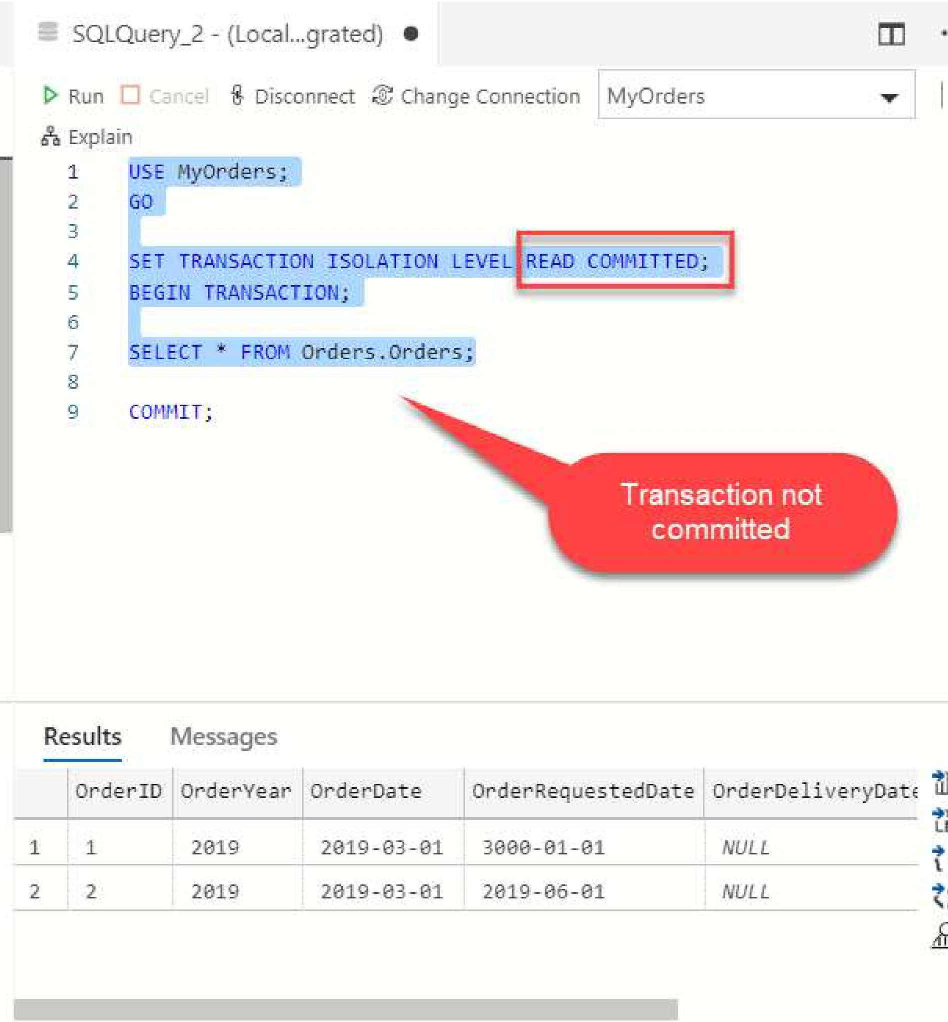


Snapshot isolation

- Instead of locks, we store previous versions of the data
- Before each write, we copy the record to tempdb (copy-on-write)
- The other transactions read the previous data version

Snapshot isolation



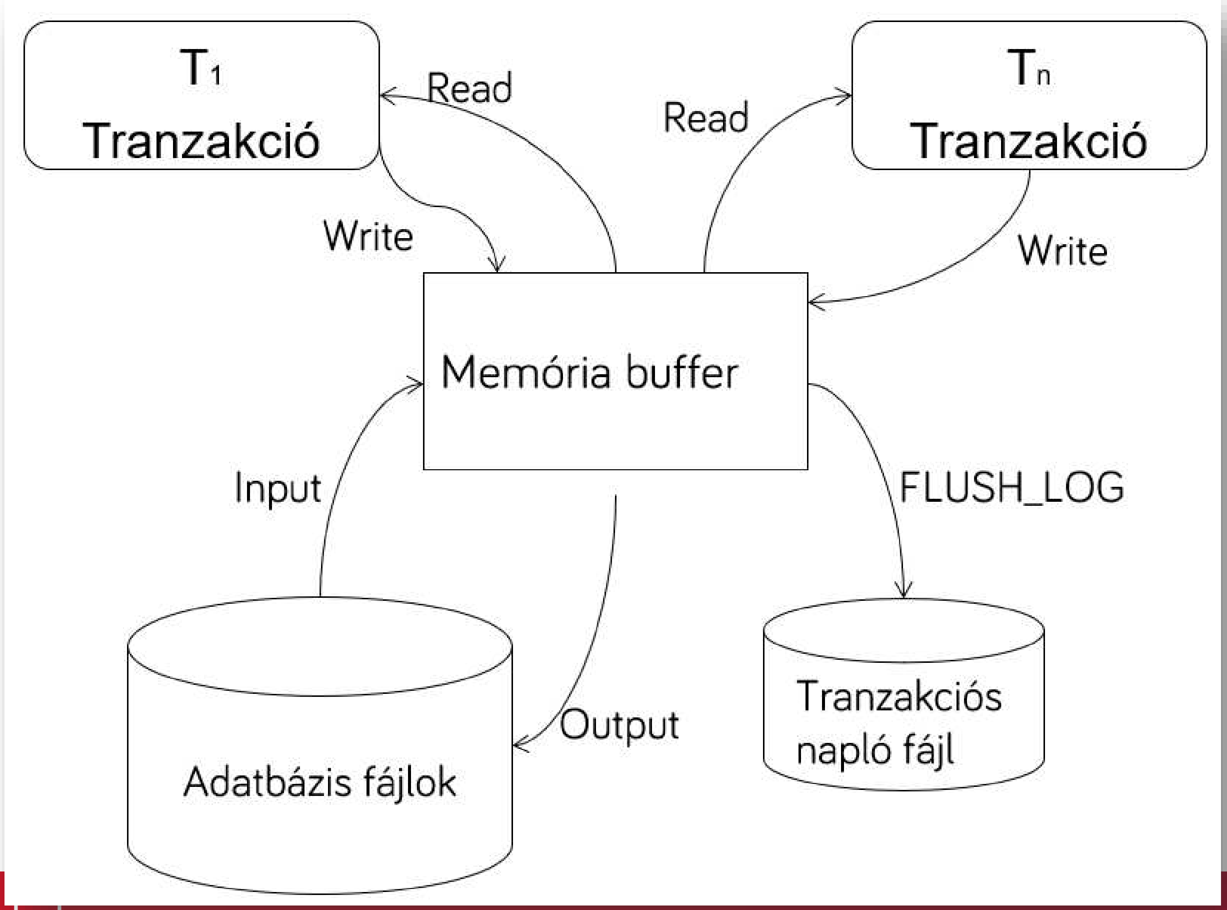


Snapshot vs standard

- The previous four levels work with locks, pessimistic
  - Stops the transaction
- Snapshot creates versions, optimistic
  - Later, during the commit, it detects the collision and kills one of the transactions
- Fewer locks, waits, deadlocks - more tempdb usage
- Useful for fewer writing and more reading

Transaction log

- Tracking transactions
  - Rollback / soft crash



T:

$A = A - 2$

$B = B + 2$

Undo logging 1

$A = A - 2$
$B = B + 2$

- Read data from disk into memory

Command	A (database)	B (database)	A (buffer)	B (buffer)	Transaction log
Begin(T1)	10	20	-	-	**Begin T1**
Input(A)	10	20	**10**	-	
Input(B)	10	20	10	**20**	

Undo logging 2

$A = A - 2$
$B = B + 2$

- Actions are performed in memory
- The value *before* the modification is entered in the log

Command	A (database)	B (database)	A (buffer)	B (buffer)	Transaction log
Read(A)	10	20	10	20	
Write(A)	10	20	8	20	T1, A, 10
Read(B)	10	20	8	20	
Write(B)	10	20	8	22	T1, B, 20

Undo logging 3

A = A − 2
B = B + 2

- Commit
  - Write transaction log to the disk
  - Modifies the database file

Command	A (database)	B (database)	A (buffer)	B (buffer)	Transaction log
Flush_LOG	10	20	8	22	
Output(A)	8	20	8	22	
Output(B)	8	22	8	22	
					Commit T1

Undo logging

- The transaction log must be written first, then the database file
- A commit entry can only be made after the database has been written
- Having to write a transaction log twice - expensive

- Recovery
  - Before commit: there is no task, it is lost (in memory)
  - Reading the transaction log from the back, for uncommitted transactions, the original values must be written back to the database from the log

Redo logging 1

$A = A - 2$
$B = B + 2$

- Read data from disk into memory
- Actions are performed and the value after modification is written into the log

Command	A (database)	B (database)	A (buffer)	B (buffer)	Transaction log
Begin(T1)	10	20	-	-	**Begin T1**
Input(A)	10	20	**10**	-	
Input(B)	10	20	10	**20**	
Read(A)	10	20	10	20	
Write(A)	10	20	**8**	20	T1, A, 8
Read(B)	10	20	8	20	
Write(B)	10	20	8	**22**	T1, B, 22

Redo logging 2

$A = A - 2$
$B = B + 2$

- Commit
  - Write the transaction log, commit entry
  - Write data to the database file

Command	A (database)	B (database)	A (buffer)	B (buffer)	Transaction log
					Commit T1
Flush_LOG	10	20	8	22	
Output(A)	**8**	20	8	22	
Output(B)	8	**22**	8	22	

Redo logging

- The transaction log must be written first, then the database file
- A commit entry must be made before writing to the database
- Less cost, longer recovery process

- Recovery
  - Before commit: there is no task, it is lost
  - The log must be processed from the beginning and all committed transactions must be redone

Undo/redo logging 1

$A = A - 2$
$B = B + 2$

- Read data from disk into memory
- Actions are performed and the value before AND after the modification is logged

Command	A (database)	B (database)	A (buffer)	B (buffer)	Transaction log
Begin(T1)	10	20	-	-	Begin T1
Input(A)	10	20	**10**	-	
Input(B)	10	20	10	**20**	
Read(A)	10	20	10	20	
Write(A)	10	20	**8**	20	T1, A, 10, 8
Read(B)	10	20	8	20	
Write(B)	10	20	8	**22**	T1, B, 20, 22

Undo/redo logging 2

$A = A - 2$
$B = B + 2$

- Commit
  - Write a transaction log
  - Write the commit signal and data in parallel

Command	A (database)	B (database)	A (buffer)	B (buffer)	Transaction log
Flush_LOG	10	20	8	22	
Output(A)	8	20	8	22	
					Commit T1
Output(B)	8	22	8	22	

Undo/redo logging

- The transaction log must be written first, then the database
- The commit entry can be made in parallel with the database write
- Less internal synchronization

- Recovery
  - Before commit: there is no task, it is lost
  - Replay a committed transaction from the beginning (redo), restore interrupted transactions (undo)

Size of transaction log

- It is reduced from time to time
- All transaction data that has already been entered into the database files can be deleted

- Long-running transactions can increase the size of the log!

Distributed transactions

- Several database servers participate in a transaction
  - There is a dedicated transaction manager that controls the execution, two phase commits
- It has ACID properties
  - Short transaction and commit time
- A tightly coupled system
  - Database level integration

Long running transactions

- No database-like commit/rollback
- Custom manual mechanism ensures the restoration of data consistency
- Special data repositories
- Distributed heterogeneous solutions
- "Saga" transactions
  - User input is also part of the transaction!

Globally distributed transactions

- Huge amount of data, big load, large distances
  - AWS, Azure, etc.
  - Multiple copies of data, far apart
- CAP theorem: any distributed data store can provide only two of the following three guarantees
  1. Consistency: Every read receives the most recent write or an error
  2. Availability: every request receives a (non-error) response, without the guarantee that it contains the most recent write
  3. Partition tolerance: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

Alternatives

- Single server -> no partitioning
  - It is consistent AND available
  - One server -> doesn't scale, SPoF
- More servers -> there will be network failure -> there will be more partitions
  - Consistency OR availability
- Alt1: stop the service until network is problems are solved
- Alt2: consistency in a given time window
  - Like DNS registration or an FB comment etc.

Strong vs eventual consistency

- Strong consistency: the system synchronizes internally and serves external requests based on this
- Eventual consistency: prioritizes availability over consistency
  - It lets you read data that is not the most recent
  - Inconsistency window: depends on load
  - Different nodes have different data versions
  - DNS, Facebook message board, etc.
  - Financial transactions often allow 24 hours