Databases 1

Daniel POP

Course Outline

1. ~~Introduction to database approach~~
2. ~~The database environment~~
3. ~~Introduction to The Relational Model~~
4. ~~Views~~
5. Transactions
6. ~~SQL Constraints~~
7. ~~Relational Database Design. Theory and practice~~
8. An Introduction to Database Performance. Indexing
9. JSON Support in Relational Database Management Systems
10. NoSQL Databases

Week 11

Transactions

Agenda

- Concurrency
- Transactions
- Locking, Blocking and Deadlocks
- Transaction Isolation Levels
- Demo session

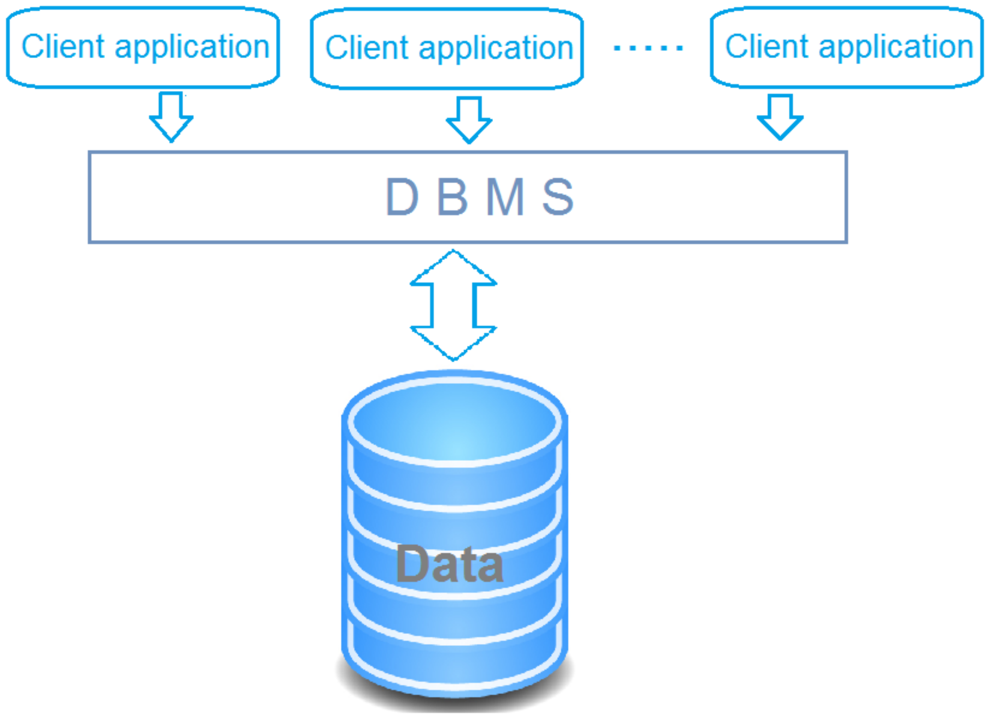
Motivation

Motivated by 2 requirements of DBMS:

- Persistent – Outlive the programs that create/access the data
- Safe – hardware/software failures, malicious users
- Multi-user – concurrently access to data (concurrency control)
- Convenient
  - Physical Data Independence; huge difference between physical representation of data on disk and the logical way of seeing and working with;
  - High level, declarative (what, not how) query languages (e.g. SQL)
- Efficient – thousands of operations (query/update) per second
- Reliable – 99.99999 % uptime

Concurrency

Concurrency = The ability of two processes to access the same data at the same time



Each client *i* issues a sequence of one or more SQL statements

Concurrent Access = Problems

Problems with concurrent access:

- By scope
    - Attribute / Tuple / Table / Multi-statement

- Read Phenomena
    - Dirty Reads
    - Non-repeatable Reads
    - Phantom Reads

Concurrent Access: Attribute level inconsistency

- Example: 2 users concurrently modify Students table:
  - User1:
    - `UPDATE Students SET Priority=Priority+1 WHERE ID=123456789`
  - User2:
    - `UPDATE Students SET Priority=Priority+0.5 WHERE ID=123456789`
- The computation `Priority = Priority+X` is decomposed into 3 basic operations by the system:
  - Step 1: the system fetches the value of field `Priority` from the database
  - Step 2: performs computation (increment the value with `X`)
  - Step 3: writes back the value of field `Priority` in the database
- Starting with `Priority=1`, what are the possible values
  - Sequential execution of the 2 statements
  - Inter-leaved execution of the 2 statements

Concurrent Access: Tuple level inconsistency

- Example: 2 users concurrently modify Students table:
  - User1:
    - `UPDATE Students SET Priority=1 WHERE ID=123456789`
  - User2:
    - `UPDATE Students SET Name='Popescu' WHERE ID=123456789`
- Similarly, each UPDATE statement is decomposed into 3 basic operations by the system:
  - Step 1: the system fetches the values of the whole tuple from the database
  - Step 2: performs computation (set a new value)
  - Step 3: writes back the whole tuple in the database
- What are the possible values
  - Sequential execution of the 2 statements
  - Inter-leaved execution of the 2 statements

Concurrent Access: Table level inconsistency

- Example: 2 users concurrently:
  - User1:

    -
      ```
      UPDATE Enrollments
      SET decision='Y'
      WHERE CNP IN (SELECT CNP FROM Students
                    WHERE TotalCredits>50)
      ```
  - User2:

    -
      ```
      UPDATE Students
      SET TotalCredits=TotalCredits+10
      WHERE MajorCode='INFO'
      ```

Concurrent Access: Multi-statement inconsistency

- Example: 2 users concurrently:
  - User1:
    - `INSERT INTO Archive`
      `SELECT * FROM Enrollments WHERE Decision = 'N';`
    - `DELETE FROM Enrollments WHERE Decision = 'N';`
  - User2:
    - `SELECT COUNT(*) FROM Enrollments;`
    - `SELECT COUNT(*) FROM Archive;`

Concurrent Access: Read Phenomena

Read Phenomena (by ISO SQL Standard 92)

- Dirty reads - a transaction reads uncommitted data
- Non-repeatable reads (aka inconsistent analysis) - A query might get different values in two separate reads in the same transaction
- Phantom reads - two SELECT statements using the same predicate in the same transaction return different set of rows

(come back later)

Concurrency

- The goal of concurrency is to execute sequence of SQL statements, so they appear to be running in isolation (avoid inconsistent and unexpected behavior).

- We **need** concurrency so that DBMS offers better performance.

Resilience to System Failures

- What happens if a system failure (hw/sw) happens
  - during a bulk load of data (from a large file, for example) into a database? => partially loaded
  - During executing sequence of commands altering multiple tables (see example in multi-statement consistency example)
  - Lots of updates (committing from cache memory to disks)

- The goal of resilience is to guarantee all-or-nothing execution!

- The solution for concurrency and resilience to failures is....

Resilience to System Failures

- What happens if a system failure (hw/sw) happens
  - during a bulk load of data (from a large file, for example) into a database? => partially loaded
  - During executing sequence of commands altering multiple tables (see example in multi-statement consistency example)
  - Lots of updates (committing from cache memory to disks)

- The goal of resilience is to guarantee all-or-nothing execution!

- The solution for concurrency and resilience to failures is....

TRANSACTIONS

Transactions

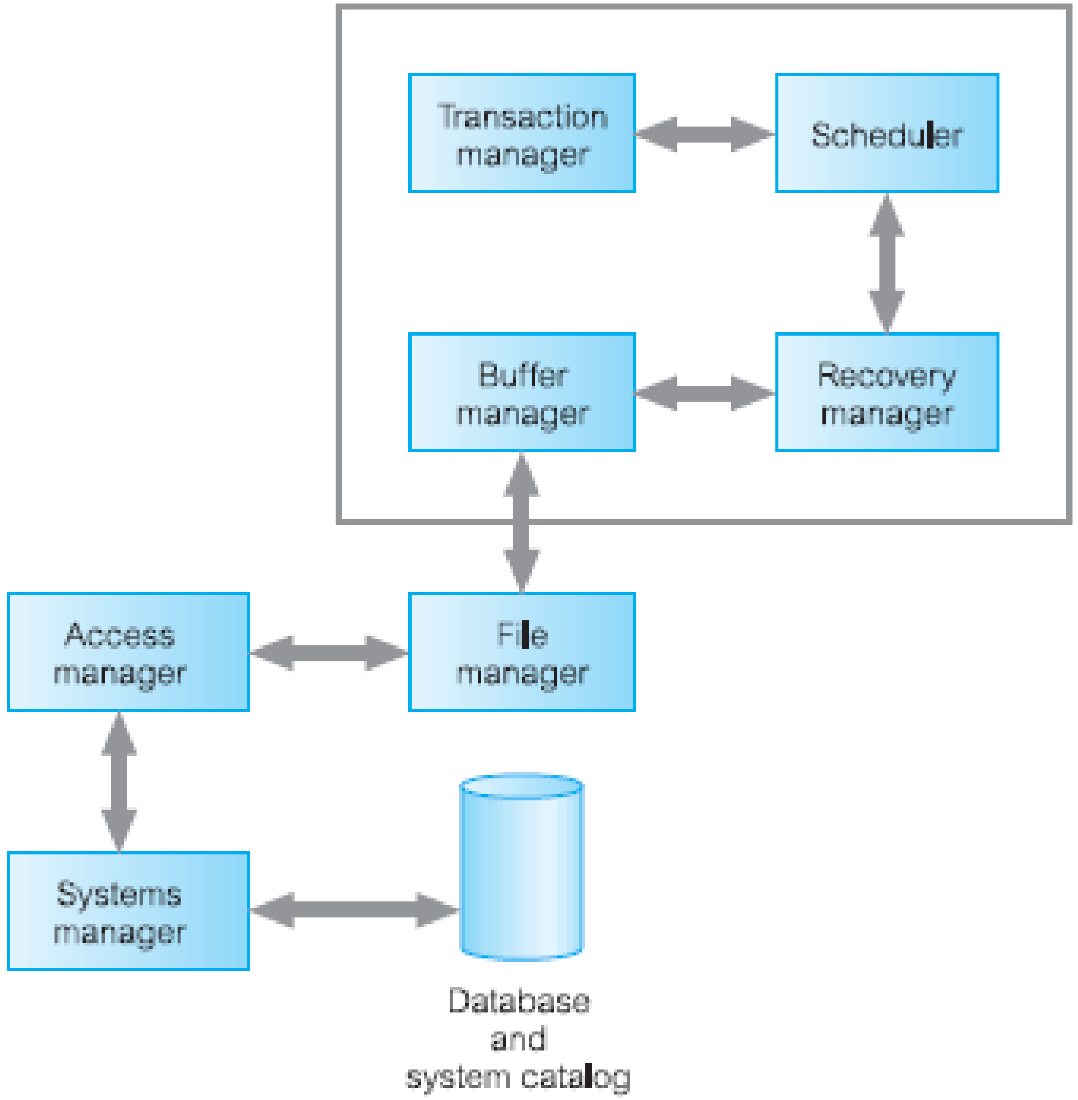
DEF: A transaction is a sequence of one or more SQL statements treated as a **unit** (a unit of work).

- They appear to run in isolation (concurrency)
- If the system fails, transaction's changes are reflected either entirely or not at all (resilience)
- **All or Nothing**

- When a transaction begins/ends?
    - Explicit
        - Begin transaction, commit, rollback commands
            - on `commit` transaction ends and a new one begins
    - `auto-commit` , which turns each SQL statement into a transaction
        - can not be explicitly rolled-back, but will automatically rollback in the event of a failure

- Transaction can have 2 outcomes: committed or aborted (rolled back).
    - Database remains in consistent state in either way.

DBMS Transaction Subsystem



Transaction properties

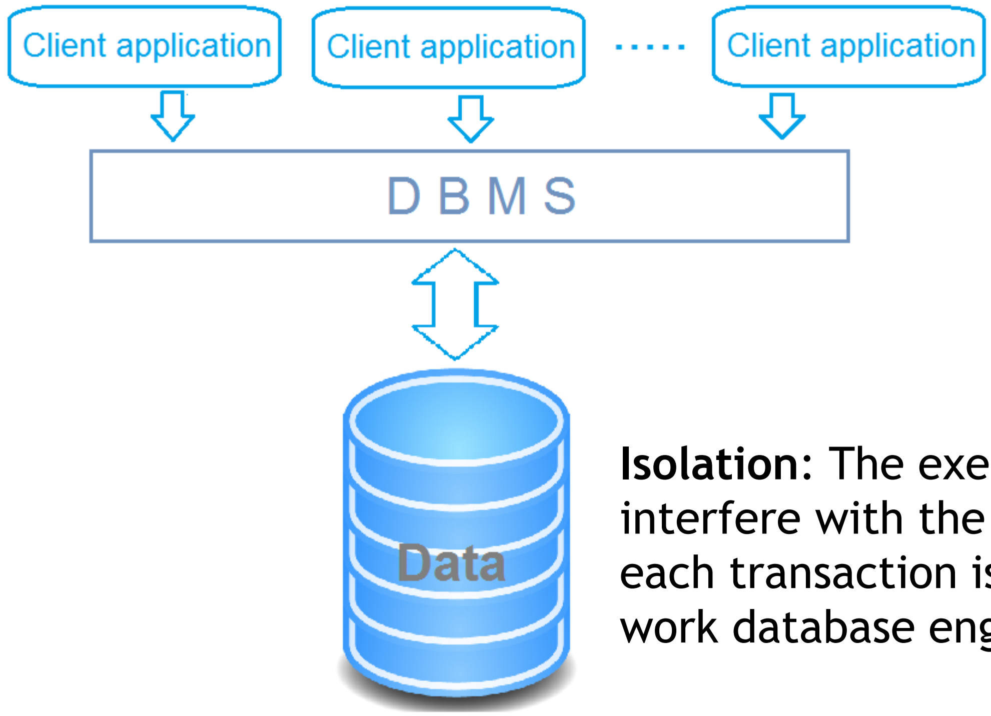
Atomicity,

Consistency,

Isolation,

Durability

Isolation



Each client *i* issues a sequence of transactions T_{i1},…T_{in}, each transaction being composed of multiple SQL statements

Isolation: The execution of one transaction should not interfere with the execution of other transactions, i.e. each transaction is executed as it is the only unit of work database engine is performing.

How? By locking portions of the database

Durability

A client issues a sequence of transactions T_1,…T_n, each transaction being composed of multiple SQL statements.

Durability guarantees if system crashes after transaction completes, all effects of transaction remain in the database.

How? Using logging.

Atomicity

A client issues a sequence of transactions T_1,…T_n, each transaction being composed of multiple SQL statements.

Atomicity : each transaction is "all or nothing", never left half-done. If there's a crash during the execution of statements composing a transaction, then the effects of executed statements are undone.

How? Using logging.

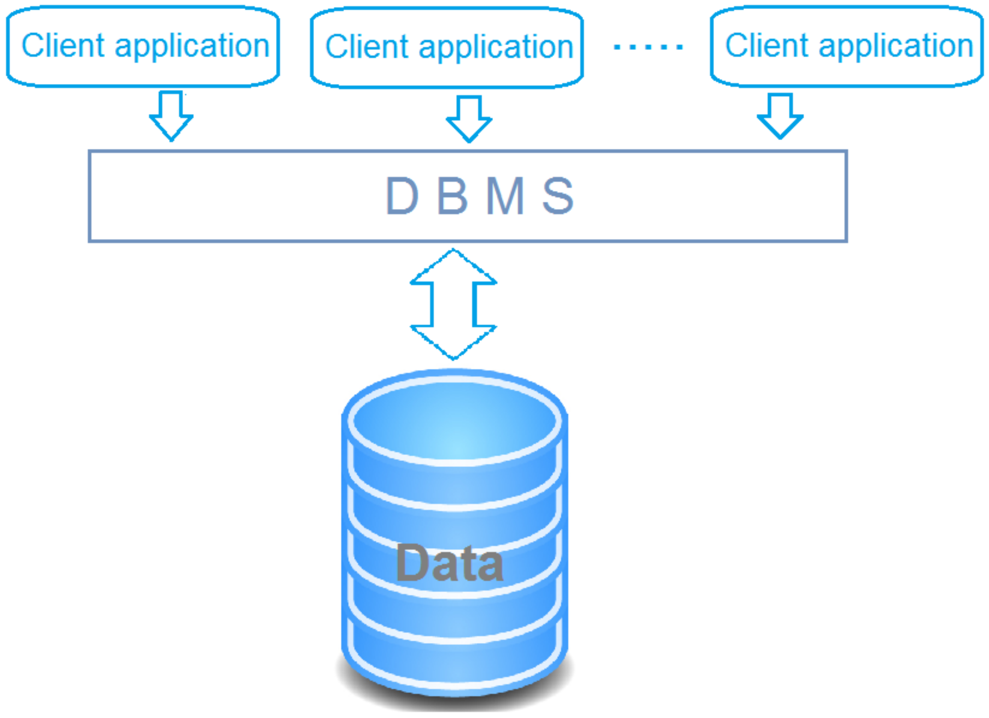
Remark: Application developers need to check the result returned by the DBMS for each executed transaction, and if there's an error re-execute the transaction.

Transaction abort / rollback = undo of partial effects of a transaction

Transaction rollback can be initiated

- by system when there's an error, or
- it can be issued by clients

Consistency



Each client *i* issues a serie of transactions $T_{i1}, \ldots T_{in}$, each transaction being composed of multiple SQL statements

Consistency – how *integrity constraints* defined on a database interact with transactions.

Each client can assume that all constraints hold when a transaction begins, and
Each client must guarantee that all constraints hold when a transaction finishes.

Consistency means that **database constraints always holds** regardless the execution order.

Locking, Blocking and Deadlocks

- These are the mechanisms that allow multiple processes (users) to access and operate on the data in a way that avoids inconsistencies in the data

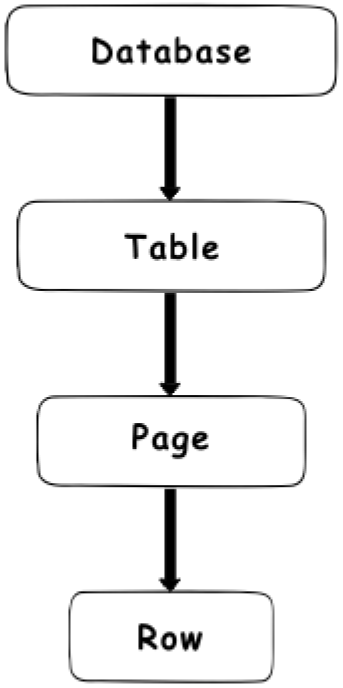
Locking

- Locking occurs when the database engine session takes ownership of a resource by requiring a **lock** in order to perform a particular operation (read/write) on that resource; the lock will stay in the effect until the database engine decides to release that lock

- Locking hierarchy - lock data resources at row / page / table / database level

- In order to allow a higher degree of concurrency, the db engine generally tries to acquire locks at the lowest level possible.

- There is always a shared lock on the database level that is imposed whenever a transaction is connected to a database. The shared lock on a database level is imposed to prevent dropping of the database or restoring a database backup over the database in use.



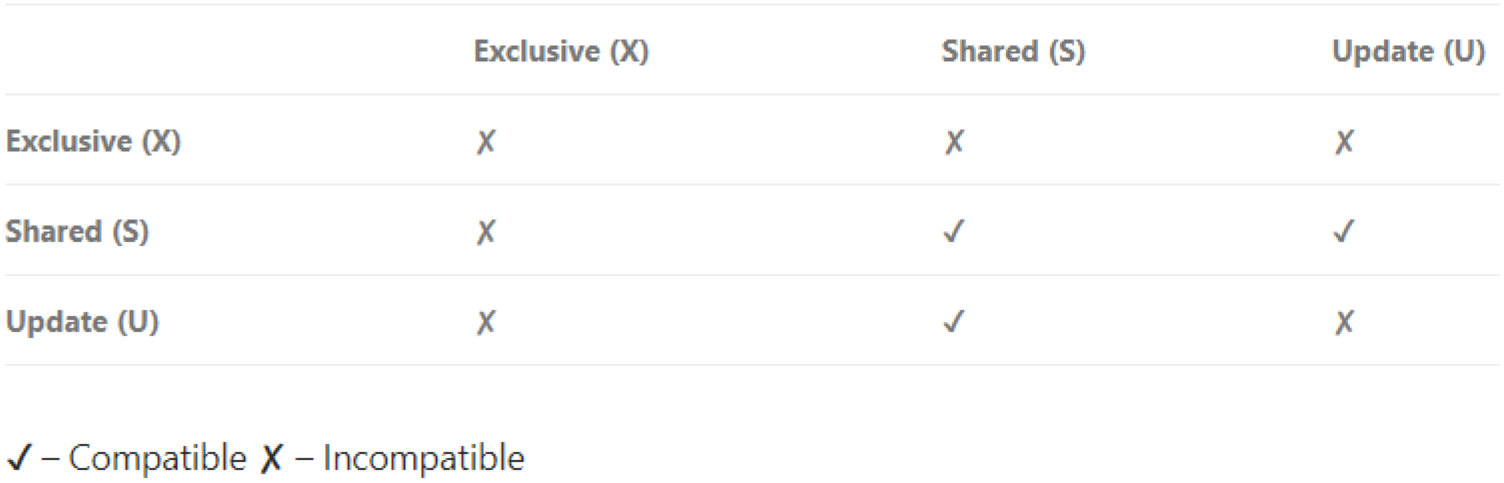
Source: https://www.sqlshack.com/locking-sql-server

Locking

- Lock modes:
    - Shared (S)
        - Used when reading the data
        - Many processes can hold a shared lock on a resource, but no process can acquire an exclusive lock unless it is the only one holding the shared lock
    - Exclusive (X)
        - Used when changing the data
        - Only one transaction can hold an exclusive lock
    - Update (U)
        - This is a hybrid lock: shared to find/read the data, exclusive to update it
    - Intent locks (IS/IX/IU + SIX/SIU/UIX)
        - Intent is a qualifier used in SQL Server; wants to take a shared/exclusive lock on a resource lower in the lock hierarchy; important for performance
- Lock duration:
    - The time span the database engine holds the lock
    - Depends on the lock mode and isolation level (discuss later)

Locking

- At row level, there are 3 lock modes that can be applied
  - Exclusive (X)
  - Shared (S)
  - Update (U)



	Exclusive (X)	Shared (S)	Update (U)
Exclusive (X)	✗	✗	✗
Shared (S)	✗	✓	✓
Update (U)	✗	✓	✗

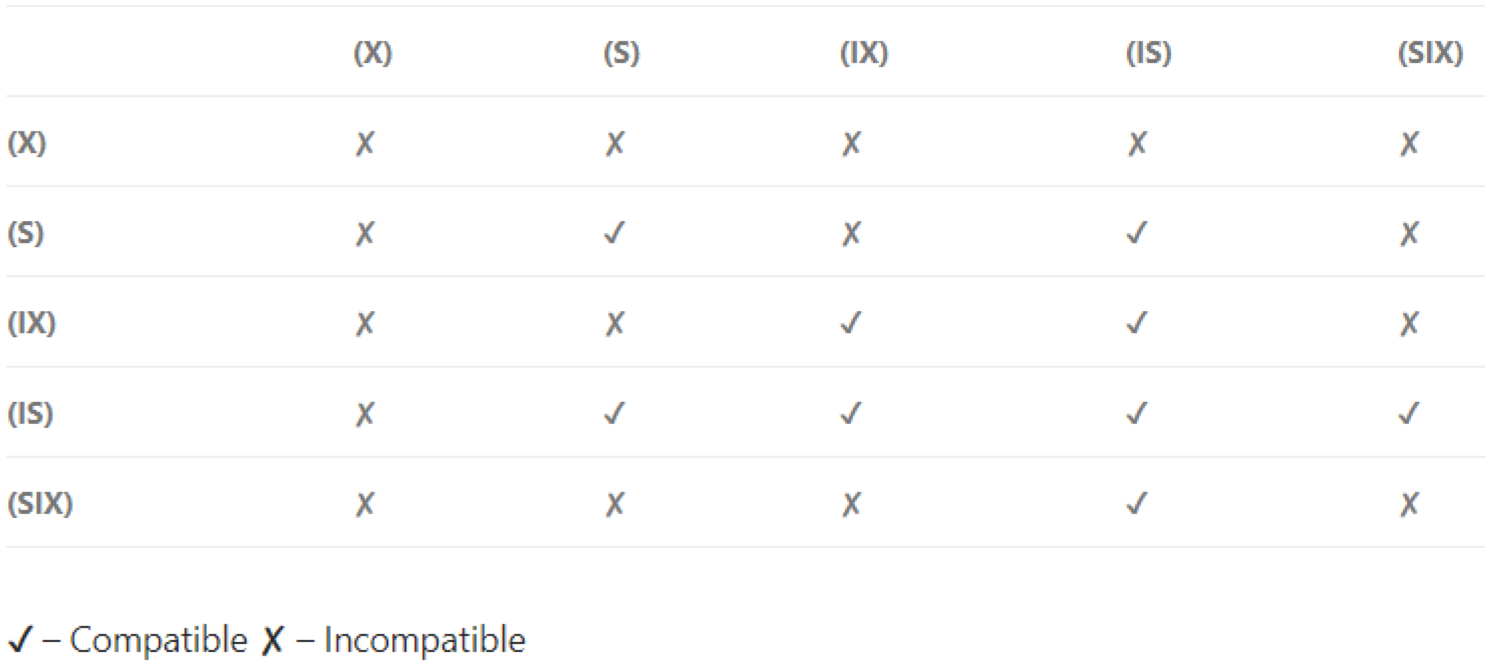
✓ – Compatible ✗ – Incompatible

- Compatible – co-exist simultaneously on the same resource
- When locks are incompatible => blocking

Source: https://www.sqlshack.com/locking-sql-server

Locking

- At table level, there are 5 lock modes that can be applied
  - Exclusive (X)
  - Shared (S)
  - Intent Exclusive (IX)
  - Intent Shared (IS)
  - Shared with intent exclusive (SIX)



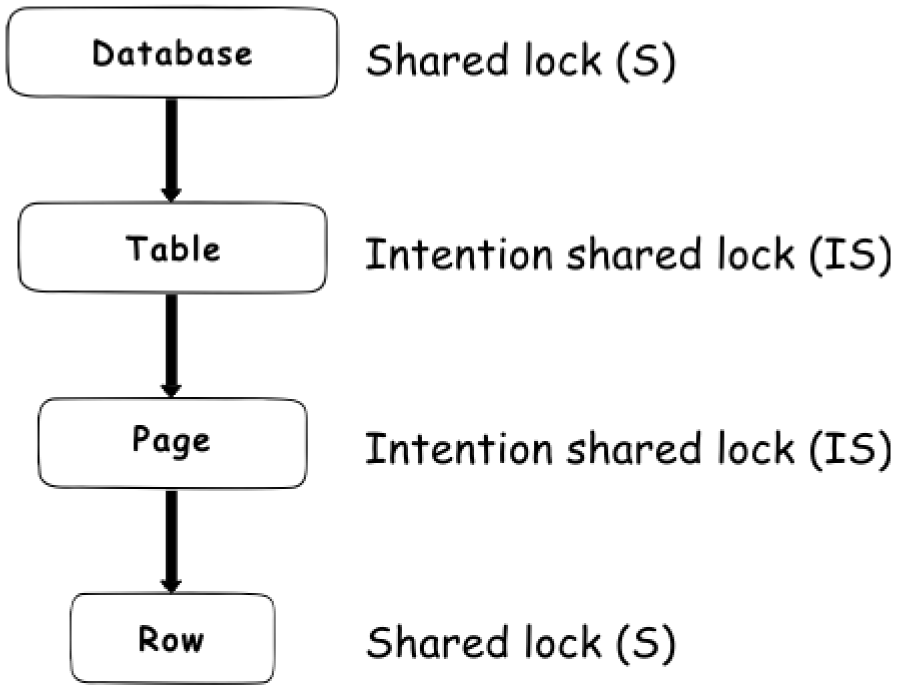
	(X)	(S)	(IX)	(IS)	(SIX)
(X)	✗	✗	✗	✗	✗
(S)	✗	✓	✗	✓	✗
(IX)	✗	✗	✓	✓	✗
(IS)	✗	✓	✓	✓	✓
(SIX)	✗	✗	✗	✓	✗

✓ – Compatible ✗ – Incompatible

Source: https://www.sqlshack.com/locking-sql-server

Locking

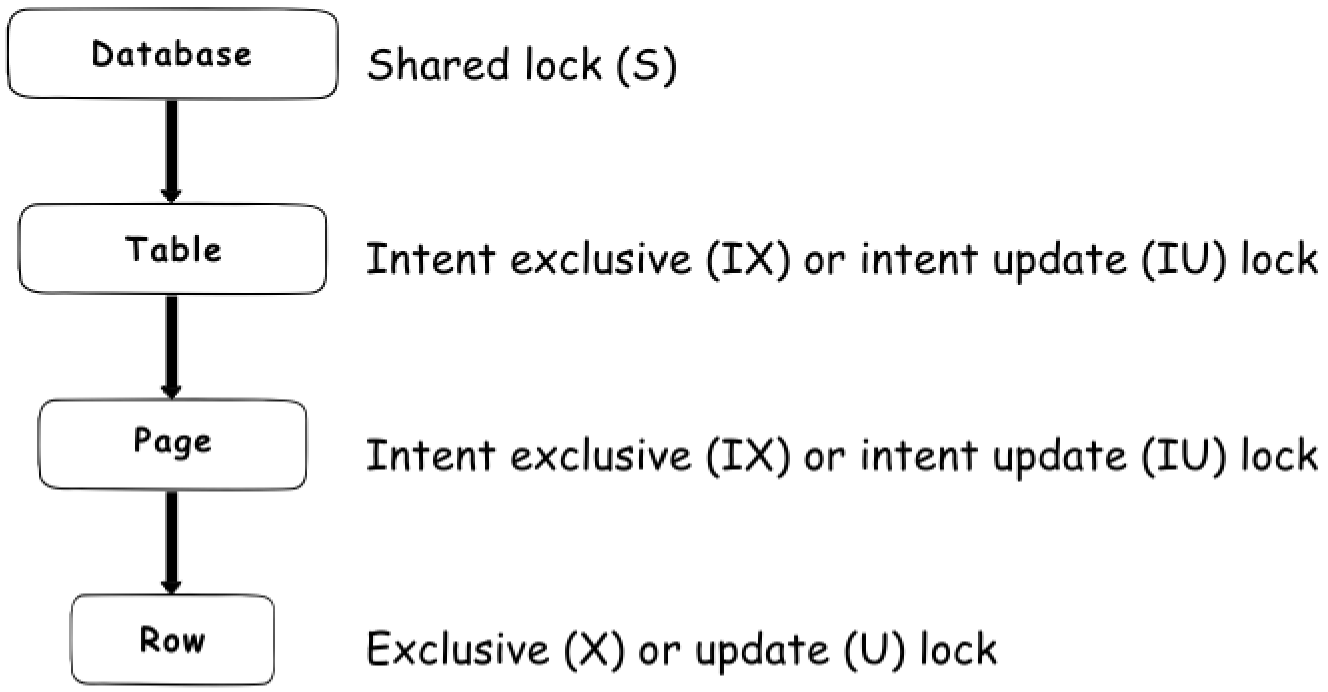
- SELECT statement (read data)



Source: https://www.sqlshack.com/locking-sql-server

Locking

- INSERT / UPDATE / DELETE statement (data manipulation)


Database
Shared lock (S)
Table
Intent exclusive (IX) or intent update (IU) lock
Page
Intent exclusive (IX) or intent update (IU) lock
Row
Exclusive (X) or update (U) lock

Source: https://www.sqlshack.com/locking-sql-server

Locking

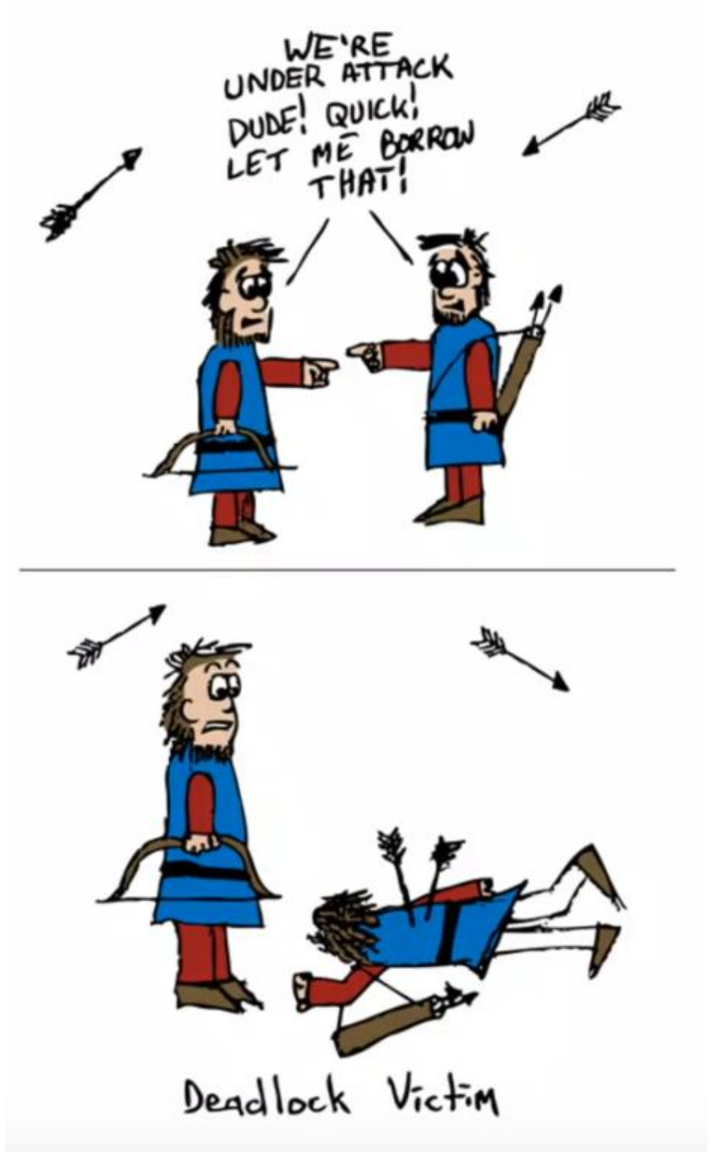
- Optimistic Locking is when you check if the record was updated by someone else before you commit the transaction.

- Pessimistic locking is when you take an exclusive lock so that no one else can start modifying the record.

Blocking

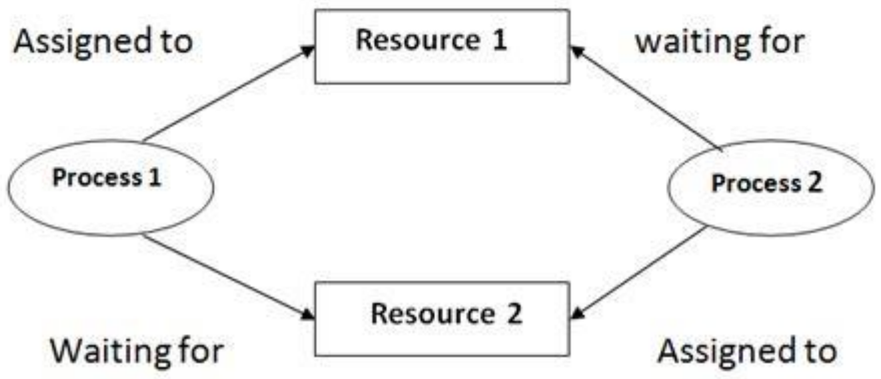
- Two sessions want concurrent access to the same resource
- No automatic limit to how long a process will wait for a resource
  - Use timeouts in your application code
- Blocking is normal, only an issue if excessive or long lasting

Deadlocks


WE'RE UNDER ATTACK DUDE! QUICK! LET ME BORROW THAT!
Deadlock Victim

Deadlocks

- When two sessions mutually block each other
- Can involve more than two processes trapped in a circular chain
- Db engine regularly checks for deadlocks (e.g. SQL Server, every 5 seconds)

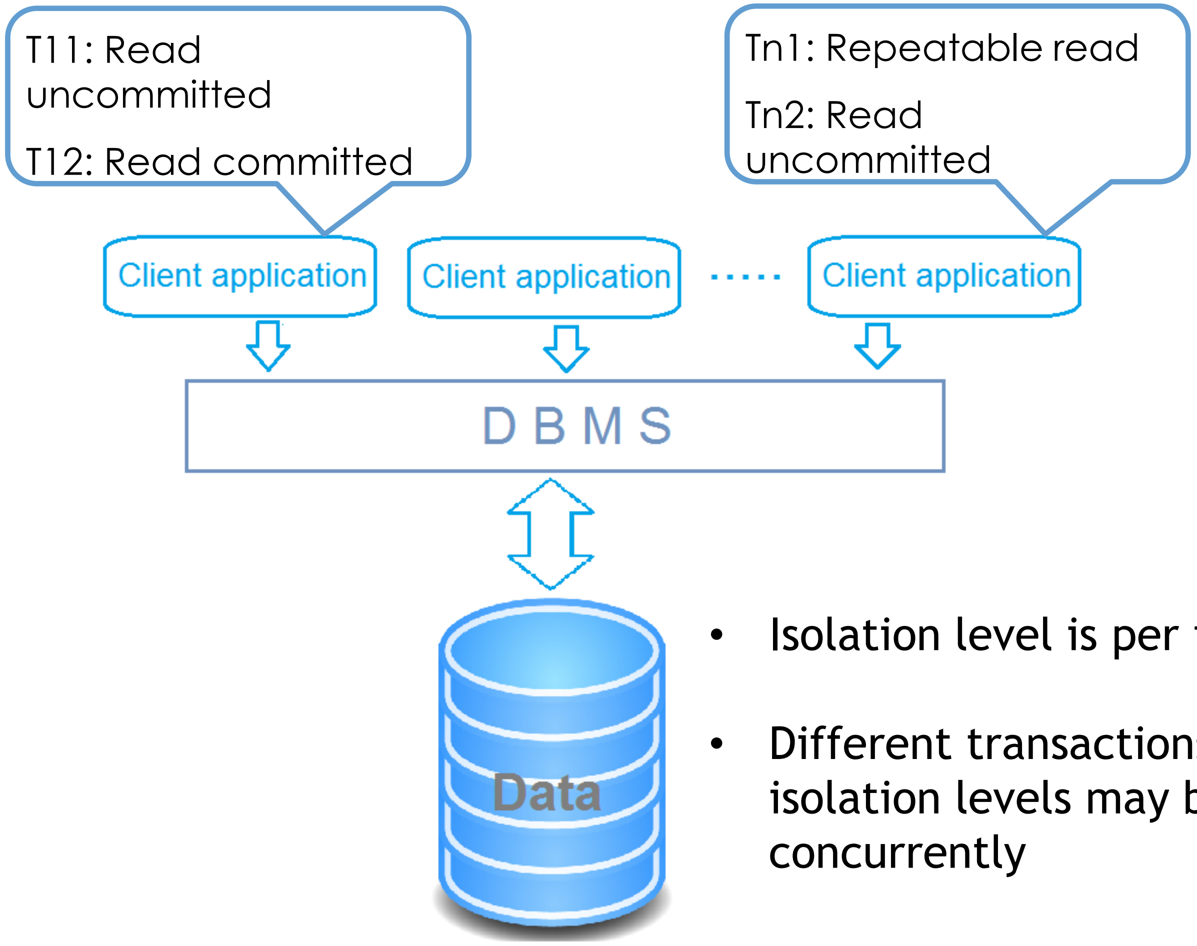


- When a deadlock is detected, a Victim is chosen, that session is rolled back (victim is always the process that is the least expensive to rollback) and the other one is committed.

Isolation levels

Isolation levels determine

- How sensitive a transaction is to other running transactions
- How long are the locks held
- How read phenomena are prevented



- Isolation level is per transaction
- Different transactions with different isolation levels may be executed concurrently

Read Phenomena

Read Phenomena (ANSI SQL 92 standard)

- Dirty reads
  - a transaction reads uncommitted data
- Non-repeatable reads (inconsistent analysis)
  - A query might get different values in two separate reads in the same transaction
- Phantom reads
  - two SELECT statements using the same predicate in the same transaction return different set of rows

Dirty Reads

DEF [‘Dirty’ data] A data item in the database is **‘dirty’** if it’s been written by a transaction that has not yet committed.

Example: two transactions executing concurrently

- T1: `UPDATE Students SET Priority=Priority+10 WHERE ID=123456789`
- T2: `SELECT AVG(Priority) FROM Students`

Non-repeatable Reads

Example: two transactions executing concurrently

- T1: `UPDATE Students SET Priority=Priority+10 WHERE TotalCredits < 50;`

- T2:
```
SELECT AVG(Priority) FROM Students;
SELECT MAX(Priority) FROM Students;
```

We read only committed values, but two consecutive reads (e.g. AVG and MAX in T2) MAY return different values, because another transaction (T1) might change the value in the meantime. Thus, AVG may be computed before T1 and MAX is computed after T1.

Phantom Reads

- T1: `INSERT INTO Students [10 tuples];`

- T2:
```
SELECT AVG(Priority) FROM Students;
SELECT AVG(Priority) FROM Students;
```

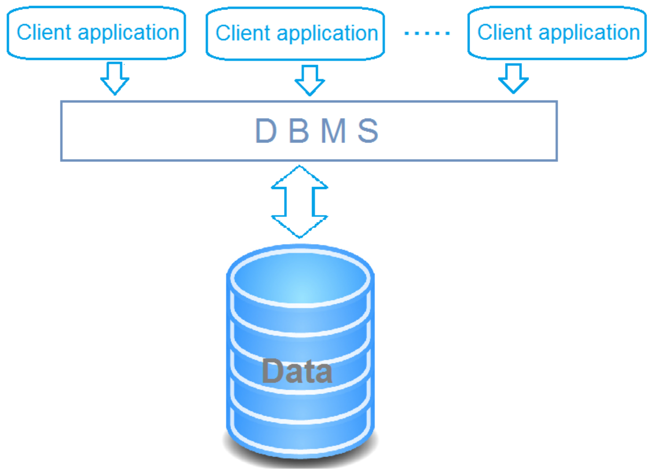
When a value is read it is locked and can't be modified, but when new tuples are inserted they are not locked so that they may appear in a second read of the relation. => 10 tuples are the phantom tuples!

- T1: `DELETE FROM Students [10 tuples];`

$\Uparrow$

Not allowed

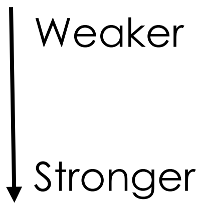
Isolation levels



Each client *i* issues a serie of transactions T_{i1},…T_{in}, each transaction being composed of multiple SQL statements.

There are 4 isolation levels in ANSI SQL 92 standard:

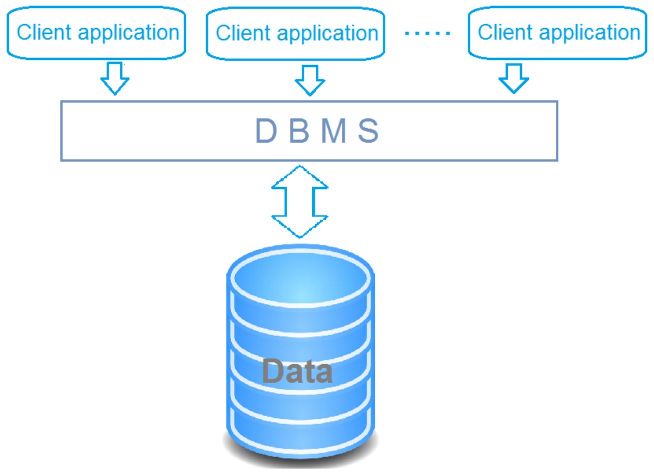
- Read Uncommitted
- Read Committed
- Repeatable Read
- Serializability



Weaker isolation levels allows

- Lower overhead,
- Increased concurrency,
- Lower consistency guarantees

Serializability



Each client *i* issues a serie of transactions T_{i1},…T_{in}, each transaction being composed of multiple SQL statements.

Serializability: operations within transactions may be interleaved across clients, but execution must be equivalent to some sequential (serial) order of all transactions.

=> T_{11}, T_{12}, T_{21}, T_{13}, T_{22}, …

Concurrent Access: Attribute level inconsistency

- Example: 2 users concurrently modify Students table:
  - `T1: UPDATE Students SET Priority=Priority+1 WHERE ID=123456789`
  - `T2: UPDATE Students SET Priority=Priority+0.5 WHERE ID=123456789`
- => equivalent sequential executions: T1; T2 or T2; T1. In either case, if we start with Priority=1, it’s correctly updated to 2.5 either way

Concurrent Access: Tuple level inconsistency

- Example: 2 users concurrently modify Students table:
  - `T1: UPDATE Students SET Priority=1 WHERE ID=123456789`
  - `T2: UPDATE Students SET Name='Popescu' WHERE ID=123456789`
- => equivalent sequential executions: T1; T2 or T2; T1. In either case, both fields will be correctly updated.

Concurrent Access: Table level inconsistency

- Example: 2 users concurrently:
  - `T1: UPDATE Enrollments SET decision='Y' WHERE CNP IN (SELECT CNP FROM Students WHERE TotalCredits>50)`
  - `T2: UPDATE Students SET TotalCredits=TotalCredits+10 WHERE MajorCode='INFO'`

- T1;T2 -> what result?
- T2;T1 -> what result?

Concurrent Access: Multi-statement inconsistency

- Example: 2 users concurrently:
  - `T1:`

    -
      ```
      INSERT INTO Archive
            SELECT * FROM Enrollments WHERE Decision
       = 'N';
      ```
    - `DELETE FROM Enrollments WHERE Decision = 'N';`
  - `T2:`
    - `SELECT COUNT(*) FROM Enrollments;`
    - `SELECT COUNT(*) FROM Archive;`
- T1;T2 -> what result?
- T2;T1 -> what result?

Exercise

Consider a relation R(A) containing two tuples {(2),(3)} and two transactions:

T1: Update R set A = A+1

T2: Update R set A = 2*A

Which of the following is NOT a possible final state of R?

a) 5, 6
b) 6, 8
c) 4, 6
d) 5, 7

Exercise

Consider a relation R(A) containing two tuples {(2),(3)} and two transactions:

T1: Update R set A = A+1

T2: Update R set A = 2*A

Which of the following is NOT a possible final state of R?

a) 5, 6 (correct)
b) 6, 8
c) 4, 6
d) 5, 7

Read Uncommitted

DEF: A transaction with **Read uncommitted** isolation level may perform dirty reads.

Example: two transactions executing concurrently

-
  ```
  T1: UPDATE Students SET Priority=Priority+10
      WHERE TotalCredits < 50
  ```

-
  ```
  T2: SELECT AVG(Priority) FROM Students
  ```

  - With Serializability, the transactions will be executed either T1; T2 or T2; T1.

Read Uncommitted

DEF: A transaction with **Read uncommitted** isolation level may perform dirty reads.

Example: two transactions executing concurrently

-
  ```
  T1: UPDATE Students SET Priority=Priority+10
      WHERE TotalCredits < 50
  ```

-
  ```
  T2: SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
      SELECT AVG(Priority) FROM Students;
  ```

- With Read Uncommitted, we don’t have exact consistency => only an approximation of Priority average, not the exact value.

Exercise

Consider a table R(A) containing {(1),(2)}. Suppose transaction

T1: UPDATE R SET A = 2*A
T2: SELECT AVG(A) FROM R

If transaction T2 executes using "read uncommitted", what are the possible values it returns?

a) 1.5, 2, 3
b) 1.5, 2.5, 3
c) 1.5, 3
d) 1.5, 2, 2.5, 3

Exercise

Consider a table R(A) containing {(1),(2)}. Suppose transaction

T1: UPDATE R SET A = 2*A
T2: SELECT AVG(A) FROM R

If transaction T2 executes using "read uncommitted", what are the possible values it returns?

a) 1.5, 2, 3
b) 1.5, 2.5, 3
c) 1.5, 3
d) 1.5, 2, 2.5, 3 (correct)

Read Committed

DEF: A transaction with **Read committed** isolation level may **NOT** perform dirty reads.

Example: two transactions executing concurrently

-
  ```
  T1: UPDATE Students SET Priority=Priority+10
      WHERE TotalCredits < 50
  ```

-
  ```
  T2: SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
      SELECT AVG(Priority) FROM Students;
  ```

- With Read Committed, we have an exact value of Priority average

Read Committed

Stronger, but Read Committed does not guarantee Serializability. It allows Non-repeatable reads.

Example: two transactions executing concurrently

-
  ```
  T1: UPDATE Students SET Priority=Priority+10
      WHERE TotalCredits < 50
  ```

-
  ```
  T2: SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
      SELECT AVG(Priority) FROM Students;
      SELECT MAX(Priority) FROM Students;
  ```

We read only committed values, but two consecutive reads (e.g., AVG and MAX in T2) MAY return two different values, because another transaction (T1) might change some values. Thus, AVG may be computed before T1 and MAX is computed after T1, with different inputs.

Lost Updates

Under Read Committed, lost updates may occur !

Lost update = two transactions edit the same record, but the transaction that is committed last will overwrite the changes made by the previous one

How to avoid lost updates

- Using Pessimistic locking
  - Repeatable Read isolation level => can potentially lead to deadlocks
  - Read Committed + UPDLOCK hint on the SELECT query that retrieves the rows for modifications

- Using Optimistic locking (SQL Server approach)
  - Snapshot isolation level (performance drawback due to tempdb usage and disk I/O)
  - Add a column of type ROWVERSION to your tables – this value is automatically checked by SQL Server before committing the change and if changed by another transaction then the record is not updated

Repeatable Read

DEF: A transaction with **Repeatable read** isolation level may **NOT** perform dirty reads and if an item is read multiple times it cannot change value.
With Repeatable Read, when a value is read it is *locked* and can’t be modified by other transactions.

-
  ```
  T1: UPDATE Students SET Priority=Priority+10
      WHERE TotalCredits < 50
  ```

-
  ```
  T2: SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
      SELECT AVG(Priority) FROM Students;
      SELECT MAX(Priority) FROM Students;
  ```

We read only committed values, and the two consecutive reads (in AVG and MAX in T2) return same values, because the transaction (T1) cannot be executed between the two SELECT statements in T2.

Repeatable Read

Example: the SELECT in T2 read different data

- `T1:  UPDATE Students SET Priority=Priority+10;`
  `     UPDATE Students SET TotalCredits=60 WHERE CNP=1..9;`

- `T2:  SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;`
  `     SELECT AVG(Priority) FROM Students;`
  `     SELECT AVG(TotalCredits) FROM Students;`

AVG(Priority) is computed before T1 and AVG(TotalCredits) is computed after T1 => this is not equivalent to none of T1;T2 nor T2;T1

Remark: We are not reading values multiple times! => 2nd condition holds! (we read different values, first time Priority, second time TotalCredits)

Repeatable Read

Example: Phantom reads

-
  ```
  T1:  INSERT INTO Students [10 tuples];
  ```

-
  ```
  T2:  SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
       SELECT AVG(Priority) FROM Students;
       SELECT MAX(Priority) FROM Students;
  ```

Repeatable read isolation level allows *a relation* to change value if read multiple times through **phantom tuples**.

-
  ```
  T1:  DELETE FROM Students [10 tuples];
  ```

Not allowed

Read-only Transactions

- Helps DBMS to optimize performance
- Independent of isolation level
- Example:

```
SET TRANSACTION READ ONLY;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT AVG(Priority) FROM Students;
SELECT MAX(Priority) FROM Students;
```

Lock Duration and Isolation Levels

- In Read Committed, a Shared lock is only held while the data is read, while an Exclusive lock is held until the end of the transaction

- In Repeatable Reads and Serialization, all locks are held until the end of transaction

Demo

Isolation levels summary

	Dirty reads	Non-repeatable reads	Phantom tuples
Read uncommitted	Y	Y	Y
Read committed	N	Y	Y
Repeatable read	N	N	Y
Serializable	N	N	N

- SQL Server default is Read Committed, Oracle / MySQL uses Repeatable Read
- There are vendor-specific isolation levels (e.g. Snapshot isolation in SQL Server etc.)

Transactions Summary

- Database engines use Transactions to manage units of work
- It uses locking, blocking and deadlocks to control access by concurrent transactions to the same data
- There are 4 isolation levels in ANSI SQL92 to manage the sensitivity of the application to other transactions and to determine how long locks are held for

Bibliography (recommended)

https://www.sqlshack.com/concurrency-problems-theory-and-experimentation-in-sql-server/ - Concurrency problems – theory and experimentation in SQL Server

https://www.sqlshack.com/locking-sql-server/ - All about locking in SQL Server

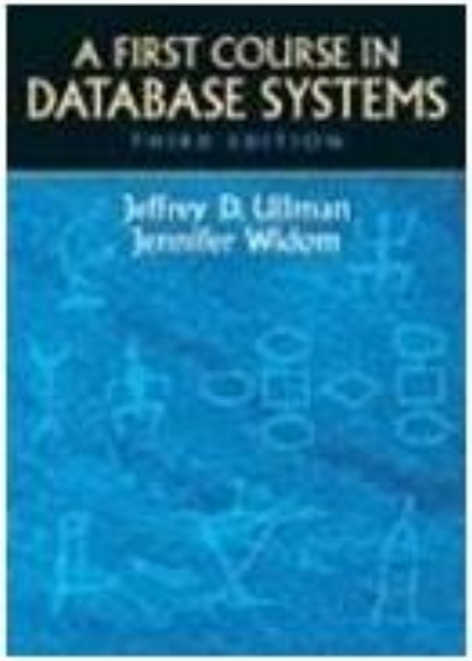
Understanding SQL Server Isolation Levels through Examples | Interface Technical Training (interfacett.com) – in SQL Server, including SQL Server’s row-versioning ones (snapshots)

https://web.archive.org/web/20200221041727/http://yrushka.com/index.php/sql-server/performance-tunning/key-range-locking-types-in-serializable-isolation-level/ - Key-Range Locks

Bibliography (recommended)

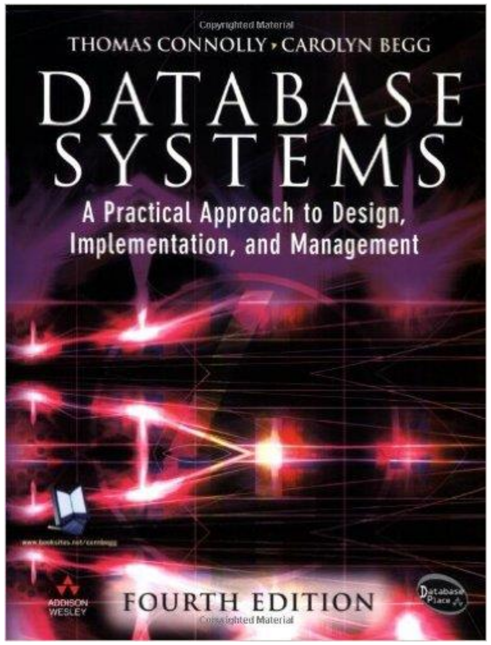
Teoria generala a bazelor de date, I. Despi, G. Petrov, R. Reisz, A. Stepan, Mirton, 2000

Cap 14.6 & 15.1



A First Course in Database Systems (3rd edition) by Jeffrey Ullman and Jennifer Widom, Prentice Hall, 2007

Chapter 6.5 - 6.6


THOMAS CONNOLLY • CAROLYN BEGG
DATABASE SYSTEMS
A Practical Approach to Design, Implementation, and Management
FOURTH EDITION
ADDISON WESLEY

Database Systems - A Practical Approach to Design, Implementation, and Management (4th edition) by Thomas Connolly and Carolyn Begg, Addison-Wesley, 2004

Chapter 6.5, 20