Databases

Transactions and Concurrency Control

João R. Campos

Bachelor in Informatics Engineering
Department of Informatics Engineering
University of Coimbra
2024/2025

From Previous Lesson(s)…

- ACID Properties
- Serializability
- Commit, Rollback, Savepoints
- Autocommit, Read Only Transactions
- Concurrency Control
- Table Lock
- Row Lock

- Programmer must specify the start and end of transactions

Outline

- Locking for Concurrency Control
  - Lock Types
  - Two-Phase Locks
  - Deadlocks
- Transaction Isolation
  - Read Uncommitted
  - Read Committed
  - Repeatable Read
  - Serializable

Register your presence at UCStudent

These slides use the following book as reference: Abraham Silberschatz, Henry F. Korth and S. Sudarshan, "Database System Concepts", McGraw-Hill Education, Seventh Edition, 2019.

This class focuses mostly on **Chapter 17 & 18**

These slides use the following book as reference: Carlos Coronel, and Steven Morris, "Database Systems: Design, Implementation, and Management", Cengage Learning, 12th Edition, 2017.

This class focuses mostly on **Chapter 10**

Lock Types
Two-Phase Locking
Deadlocks

LOCKING FOR CONCURRENCY CONTROL: LOCK TYPES

Lock Types

- Regardless of the level of granularity of the lock, the DBMS may use different lock types or modes:
  - Binary
  - Shared/exclusive

- Note: locks can be acquired implicitly or explicitly!

Binary Lock

- A binary lock has two states: locked (1) or unlocked (0)
  - No transaction can use an object that is locked by other transaction
  - If an object is unlocked, any transaction can lock the object for use
  - e.g., default LOCK TABLE
- Every operation requires the affected object to be locked
- A transaction must unlock the object after its termination
  - A transaction requires a lock and unlock operation for each accessed data item

- Binary locking is too restrictive!
  - Two transactions cannot read the same object even though neither transaction updates the database, and therefore, no concurrency problems can occur

Shared/Exclusive Lock

- An exclusive lock exists when access is reserved specifically for the transaction that locked the object (e.g., UPDATE)
  - Must be used when the potential for conflict exists
- A shared lock exists when concurrent transactions are granted read (e.g., SELECT) access on the basis of a common lock
  - Produces no conflict as long as all the concurrent transactions are read-only

- A shared lock is issued when a transaction wants to read data from the database and no exclusive lock is held on that data item
- An exclusive lock is issued when a transaction wants to update (write) a data item and no locks are currently held on that data item

Shared/Exclusive Lock - Conflicts

- Using the shared/exclusive locking concept, a lock can have three states: unlocked, shared (read), and exclusive (write)
- Two transactions conflict only when at least one is a write transaction
- As two read transactions can be safely executed at once, shared locks allow read transactions to read the same data item concurrently
  - If transaction T1 has a shared lock on data item X and transaction T2 wants to read data item X, T2 may also obtain a shared lock on data item X

Shared/Exclusive Lock - Conflicts

- If transaction T2 updates data item X, an exclusive lock is required by T2 over data item X
  - The exclusive lock is granted if and only if **no other locks** are held on the data
  - Mutual exclusive rule: only one transaction at a time can own an exclusive lock on an object
- If a shared/exclusive lock is already held on data item X by transaction T1, an exclusive lock cannot be granted to T2 (waits)
- A shared lock will block an exclusive (write) lock
  - Hence, decreasing transaction concurrency
  - Strictly speaking, **readers block writers** (and vice-versa)

Shared/Exclusive Lock - Conflicts

- But! DBMS typically allow shared with exclusive locks to improve concurrency
  - **i.e., readers don't block writers!** (e.g., SELECT and UPDATE)
  - various techniques, Multi-Version Concurrency Control (MVCC), Snapshot Isolation (SI), multiple versions of a rows are maintained
  - this can lead to serialization issues, thus multiple isolation and lock levels exist

Explicit Table Locking - PostgreSQL

```
LOCK TABLE table_name IN lock_mode MODE;
```

- PostgreSQL implements eight different table locking modes!
  - Modes have different purposes and conflict with different lock modes

- Let's take a look to examples of four key modes:
  - ACCESS EXCLUSIVE
  - EXCLUSIVE
  - SHARE
  - ACCESS SHARE

ACCESS EXCLUSIVE

- The holder is the only transaction accessing the table in any way
  - Extremely restrictive!
- Default lock mode for LOCK TABLE when the mode is not explicitly defined

Time	T1	T2
1	`begin transaction;`	`begin transaction;`
2	`lock table dep` `in access exclusive mode;`	
3		`select * from dep;`
4	`...`	
5	`...`	
6	`commit;`	
7		

EXCLUSIVE

- Only reads from the table can proceed in parallel with a transaction holding this lock mode
- Does not allow other transactions to lock the table in any mode, except ACCESS SHARE

Time	T1	T2
1	`begin transaction;`	`begin transaction;`
2	`lock table dep in exclusive mode;`	
3		`select * from dep;`
4		`update dep...`
5	`...`	
6	`commit;`	
7		

SHARE

- Protects against concurrent data changes
- Other transactions cannot change data or lock exclusive
- Allows concurrent reads and other transactions to lock in share mode

Time	T1	T2
1	`begin transaction;`	`begin transaction;`
2	`lock table dep in share mode;`	
3		`lock table dep in share mode;`
4	`select * from dep;`	
5	`update dep...`	
6		`update dep...`
7		

What happens?

ACCESS SHARE

- Prevents other transactions from acquiring ACCESS EXCLUSIVE
- The SELECT command acquires a lock of this mode on referenced tables

Time	T1	T2
1	`begin transaction;`	`begin transaction;`
2	`select * from dep;`	
3		`lock table dep` `in access exclusive mode;`
4	`update dep...`	
5	`...`	
6	`commit;`	
7		

Explicit Row Locking - PostgreSQL

- Explicit row lock can be done using SELECT FOR UPDATE

```
select …
...
for update;
```

Time	T1	T2
1	`begin transaction;`	`begin transaction;`
2	`select * from dep where ndep=10 for update;`	
3		`update dep set local=upper(local) where ndep=20;`
4		`update dep set local=upper(local) where ndep=10;`
5	`...`	
6	`commit;`	
7		

Implicit Locking - PostgreSQL

- Implicit locking happens when rows are accessed and/or modified by DML commands
  - Some DDL commands also lead to implicit locking
  - e.g., ALTER TABLE requires a SHARE UPDATE/ROW EXCLUSIVE lock, which protects a table against concurrent schema changes

- SELECT implicitly acquires an ACCESS SHARE lock
  - Thus, the table cannot be locked ACCESS EXCLUSIVE by other transaction
- INSERT, UPDATE and DELETE implicitly acquires an ROW EXCLUSIVE lock
  - No other transactions can change the rows locked or acquire a lock on the entire table (except ACCESS SHARE)

DEMO #1

- Open two sessions using *psql* (DB from PL classes)

Time	T1	T2
1	begin transaction;	
2		begin transaction;
3	lock table dep in access exclusive mode;	
4		select * from dep;
5	commit;	
6	begin transaction;	
7		lock table dep in exclusive mode;
8	select * from dep;	
9	update dep set local=upper(local);	
10		rollback;
11		begin transaction;
12		update dep set nome=upper(nome);
13	select * from dep;	
14	rollback;	
15		rollback;

DEMO #1

16	begin transaction;	
17	select * from dep;	
18		begin transaction;
19		update dep set nome=upper(nome);
20		lock table dep in access exclusive mode;
21	update dep set nome=lower(nome);	
22	rollback;	
23		rollback;
24	begin transaction;	
25	select * from dep where ndep=10 for update;	
26		begin transaction;
27		update dep set nome=upper(nome) where ndep=20;
28		update dep set nome=upper(nome) where ndep=10;
29	rollback;	
30		commit;

Problems of Locking

- Locks prevent serious data inconsistencies, but they can lead to two major problems:
  - The resulting transaction schedule might not be serializable
  - The schedule might create deadlocks
    - Occurs when two transactions wait indefinitely for each other to unlock data

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Figure 18.4 Schedule 1.

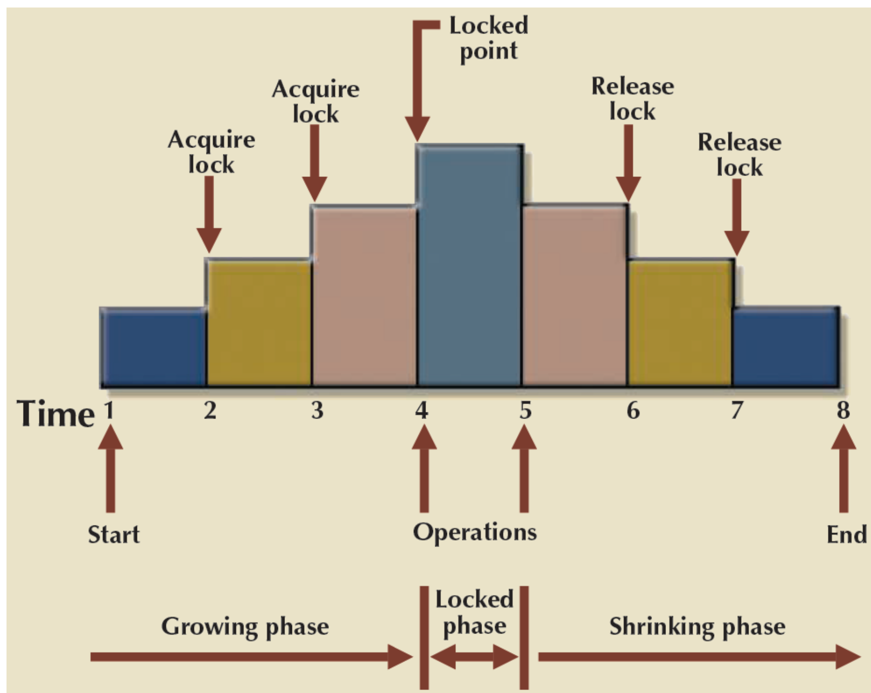
Two-Phase Locking

- Two-phase locking protocol ensures serializability
- Defines how transactions acquire and relinquish locks
- Two-phase locking guarantees serializability, but it does not prevent deadlocks

- The two phases are:
  1. A growing phase, in which a transaction acquires all required locks without unlocking any data
     - Once all locks have been acquired, the transaction is in its locked point
  2. A shrinking phase, in which a transaction releases all locks and cannot obtain a new lock

Two-Phase Locking - Rules

- Governed by the following rules:
  - Two transactions cannot have conflicting locks
  - No unlock operation can precede a lock operation in the same transaction
  - No data is affected until all locks are obtained
    - i.e., until the transaction is in its locked point



Source: C. Coronel, S. Morris, "Database Systems: Design, Implementation, and Management", Cengage Learning, 12th Edition, 2017.

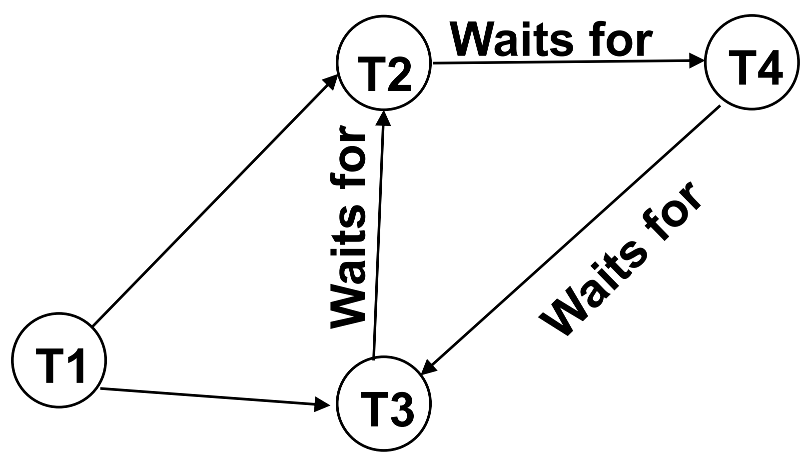
Deadlocks

- Occurs when two (or more) transactions wait indefinitely for each other to unlock data
- Only happens when transactions obtain exclusive locks on a data item
  - No deadlock condition can exist among shared locks
- Can be solved by deadlock detection and prevention
  - We study 3 possible strategies next

Deadlock Detection

- The DBMS periodically tests for deadlocks
- If a deadlock is found, the "victim" transaction is aborted (rolled back) and the other transaction continues

- *Wait-For-Graph*



Deadlock Prevention

- A transaction requesting a new lock is aborted when there is the possibility that a deadlock can occur
- If the transaction is aborted, all changes made by this transaction are rolled back and all locks obtained by the transaction are released
- The transaction is then rescheduled for execution
- Avoids the conditions that lead to deadlocking

Deadlock Avoidance

- The transaction must obtain all the locks it needs before it can be executed
- Various approaches:
  - Correct programming of transaction
  - Impose an ordering of all data items
- Avoids the rolling back of conflicting transactions by requiring that locks be obtained in succession
- The serial lock assignment required in deadlock avoidance increases action response times (another problem, what data to lock?)

- Concurrency decreases as locking increases

Read Uncommitted
Read Committed
Repeatable Read
Serializable

TRANSACTION ISOLATION

Transaction Isolation

- Refer to the degree to which transaction data is "protected or isolated" from other concurrent transactions
  - Isolation levels are described based on what data other transactions can see (read) during execution, i.e., the type of "reads" that a transaction allows or not
- Types of read operations *(issues)*:
  - Dirty read: a transaction can read data that is not yet committed
  - Nonrepeatable read: a transaction reads a given row at time t_1, and then it reads the same row at time t_2, yielding different results
  - Phantom read: a transaction executes a query at time t_1, and then it runs the same query at time t_2, yielding additional rows that satisfy the query
  - Serialization anomaly: the result of the commit is inconsistent with all possible orderings of running the transactions one at a time
- Isolation levels: read uncommitted, read committed, repeatable read, serializable

Why Different Isolation Levels?

- The goal is to increase transaction concurrency
- Levels go from the least restrictive (read uncommitted) to the more restrictive (serializable)
- The higher the isolation level the more locks (shared and exclusive) are required to improve data consistency
  - At the expense of transaction concurrency performance

- Isolation level defined in the transaction statement (PostgreSQL):
  - BEGIN TRANSACTION ISOLATION LEVEL *isolation_level*
- Setting the transaction mode:
  - SET TRANSACTION ISOLATION LEVEL *isolation_level*

Read Uncommitted

- Allows reading uncommitted data from other transactions
- Database does not place any locks on the data, which increases transaction performance but at the cost of data consistency
- Not implemented by PostgreSQL!
  - READ UNCOMMITTED is treated as READ COMMITTED!

Read Committed

- Transactions can only read committed data
- Default mode of operation for most databases

Time	T1	T2	
1	`begin transaction;`	`begin transaction isolation level read committed;`	
2	`update dep set local='Lisboa' where ndep=20;`		
3		`select local from dep where ndep=20;`	Mealhada
4	`commit;`		
5		`select local from dep where ndep=20;`	Result?

Repeatable Read

- Ensures that queries return consistent results
  - i.e., only sees data committed before the transaction began **(PostgreSQL only after data is read)**

Time	T1	T2
1	`begin transaction;`	`begin transaction isolation level repeatable read;`
2		`select local from dep where ndep=20;` Mealhada
3	`update dep set local='Lisboa' where ndep=20;`	
4	`commit;`	
5		`select local from dep where ndep=20;` Result?

Serializable

- Most restrictive level
  - Does not allow dirty reads, nonrepeatable reads, or phantom reads
- Emulates serial transaction execution for all committed transactions
  - As if transactions had been executed one after another, serially
  - Restrictive, can lead to false positives (e.g., table scan on a small table creates predicate locks on the entire table

In Summary…

	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read Uncommitted	Allowed, but not in PostgreSQL	Possible	Possible	Possible
Read Committed	Not possible	Possible	Possible	Possible
Repeatable Read	Not possible	Not possible	Allowed, but not in PostgreSQL	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

DEMO #2

- Consider the database used in the PL classes
- TODO:
  - Open two sessions using *psql*, and execute:

Time	T1	T2
1	`begin transaction isolation level serializable;`	`begin transaction isolation level serializable;`
2	`select * from dep;`	
3	`update dep set local=upper(local) where ndep=30;`	
4		`select * from dep;`
5		`update dep set local=upper(local) where ndep=40;`
6	`commit;`	
7		`commit;` What will happen?

Take-Away(s)

- Lock Types
  - Binary, Shared/exclusive
  - Access exclusive, exclusive, share, access share
- Explicit locking, implicit locking
- Two-Phase locks and deadlocks
- Transaction isolation
  - Read uncommitted, read committed, repeatable read, serializable

Next Lesson(s)

- Database Application Development
- Database Application Architectures
- Security in Database Applications
- REST API: Basic Concepts and Examples
- Object / Relational Mapping (ORM)

- PL/pgSQL

Where Are We in the Course?

- Relational Model
- Structured Query Language (SQL)
- Entity-Relationship Model
- Functional Dependencies
- Database Normalization
- Transactions and Concurrency Control

What is Missing?

- Development of Database Applications
- PL/pgSQL

- Data Storage and Indexing
- Database Performance Tuning, Query Execution, and Indexing
- Database Administration and Security
- Data Warehouses and OLAP
- Big Data Storage and NoSQL Databases

Q&A

Databases

Transactions and Concurrency Control

João R. Campos

Bachelor in Informatics Engineering

Department of Informatics Engineering
University of Coimbra

2024/2025