Databases

Transactions and Concurrency Control

João R. Campos

Bachelor in Informatics Engineering
Department of Informatics Engineering
University of Coimbra
2025/2026

From Previous Lesson(s)…

- Relational Model & SQL
  - Relational Model
  - Relational Databases
  - Relational Query Languages: SQL
- Entity-Relationship Model
  - Database Design Process
  - Entity-Relationship Model
  - From E-R Diagrams to Relational Schemas
- Functional Dependencies and Database Normalization
  - Functional Dependencies
  - Normal Forms
  - Decomposition using Functional Dependencies

Outline

- Database Transactions and Properties
  - ACID Properties and Serializability
  - Transaction Management with SQL
  - Autocommit, Read Only Transactions, Savepoints
  - Transaction Log
- Concurrency Control
  - Lost Updates, Uncommitted Data, Inconsistent Retrievals
  - The Scheduler
- Locking for Concurrency Control
  - Database Lock, Table Lock, Page Lock, Row Lock

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**

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

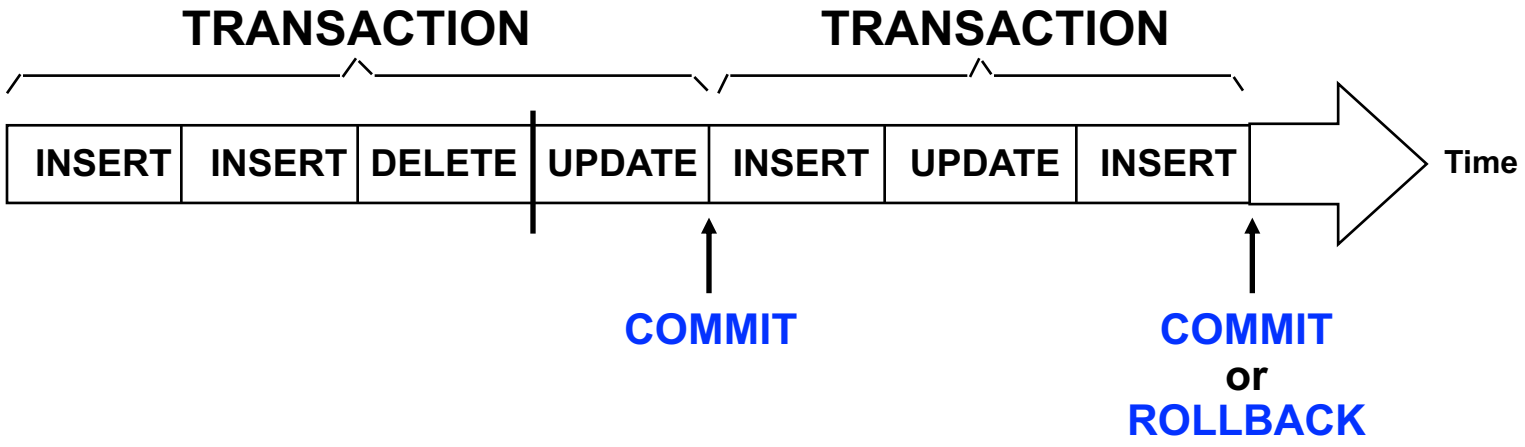
This class focuses mostly on **Chapter 10**

What Is a Transaction?

- A transaction is a logical unit of work that must be entirely completed or entirely aborted
  - All the SQL statements in the transaction must be completed successfully
  - If any of the SQL statements fail, the entire transaction is rolled back to the original database state that existed before the transaction started

- A successful transaction changes the database from one consistent state to another consistent state
  - State in which all data integrity constraints are satisfied
- Most real-world database transactions are formed by two or more database requests

What Is a Transaction?




```
insert into invoice values (1009, 10016,'18-Jan-2016', 256.99,
20.56, 277.55, 'cred', 0.00, 277.55);

insert into line values (1009, 1, '89-WRE-Q', 1, 256.99, 256.99);

update product set prod_qoh = prod_qoh - 1
where prod_code = '89-WRE-Q';

update customer set cust_balance = cust_balance + 277.55
where cust_number = 10016;

insert into acct_trans values (10007, '18-Jan-16', 10016,
'charge', 277.55);

commit;
```

What is the problem?

What Is Concurrency?

- When many transactions take place at the same time, they are called concurrent transactions
  - Concurrency is essential for assuring the performance of a DBMS
  - Various problems may occur with concurrent transactions:

- The most common scenario is having many users and programs accessing a DBMS at the same time many concurrent transactions
- Managing the execution of such transactions is called concurrency control

Why Transactions?
ACID Properties
Serializability
Transaction Management with SQL
Autocommit, Read Only Transactions, Savepoints
Transaction Log

DATABASE TRANSACTIONS AND PROPERTIES

Why Transactions?

T1 (50 from A to B)

```
read(A)          select…
A=A-50
write(A)         update…
read(B)          select…
B=B+50
write(B)         update…
```

T2 (10% from A to B)

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

- What happens if the transactions fail before completing?
- Is order important? Assume *A=100* and *B=200*:
  - What is the result if T1 is executed before T2?
  - What is the result if T2 is executed before T1?
  - What if the two transactions are executed simultaneously?

ACID Properties

- Atomicity
  - Requires that all operations (SQL requests) of a transaction be completed; if not, the transaction is aborted
- Consistency
  - Indicates the permanence of the database's consistent state
  - A transaction takes a database from one consistent state to another
- Isolation
  - The data used during the execution of a transaction cannot be used by a second transaction until the first one is completed
- Durability
  - Transaction changes that are done and committed, cannot be undone or lost, even in the event of a system failure

Serializability

- Ensures that the schedule for the concurrent execution of the transactions yields consistent results
  - The selected order of concurrent transaction operations leads to the same final state as if the transactions had been executed in a serial fashion
- Key property when multiple transactions are likely to be executed concurrently over the same data
  - If only a single transaction is executed, serializability is not an issue
- e.g., let's assume that the DBMS has three transactions (T1, T2 and T3) executing at the same time
  - Each individual transaction must comply with the ACID properties
  - At the same time, the schedule of such multiple transaction operations must exhibit the property of serializability

Transaction Management with SQL

- Transaction support is provided by two main SQL statements: COMMIT and ROLLBACK

- When a transaction is initiated, the sequence must continue through all SQL statements until one of the following four events occurs:
  - COMMIT statement is reached
    - All changes are permanently recorded within the database, and transaction ends
  - ROLLBACK statement is reached
    - All changes are aborted, database is rolled back, and transaction ends
  - End of a program is successfully reached
    - All changes are permanently recorded within the database: equivalent to COMMIT
  - Program is abnormally terminated
    - Database changes are aborted, database is rolled back: equivalent to ROLLBACK

usually... but you should not trust it

Example

```
update product
set prod_qoh = prod_qoh - 2
where prod_code = '1558-QW1';

update customer
set cust_balance = cust_balance + 87.98
where cust_number = '10011';

commit;
```

- Good programming practice dictates that we should include the COMMIT statement at the end of a transaction declaration

Different SQL Implementations

- Transaction begins implicitly when the first SQL statement is encountered
- Some DBMS (such as PostgreSQL) use transaction management statements to indicate the beginning of a new transaction:
  - e.g., BEGIN TRANSACTION
- Others allow assigning characteristics for the transactions as parameters to the BEGIN statement
  - e.g., Oracle uses the SET TRANSACTION statement to declare the start of a new transaction and its properties
- Read only transactions can only execute *select* statements
  - SET TRANSACTION READ ONLY

Autocommit: Be careful!

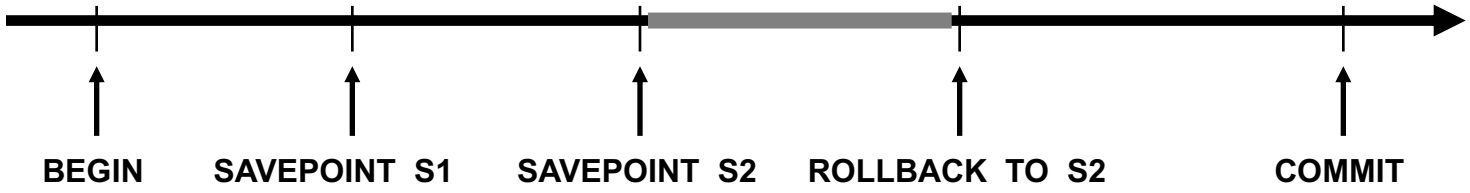
- Each individual database interaction (i.e., each SQL statement) will be executed in its own transaction that is implicitly committed

- Mode of operation of a database connection
  - Can normally be defined/changed at connection level
- Important: a SQL statement executed in autocommit mode cannot be rolled back

- By default, *psql* operates in autocommit mode
  - You can explicitly start a transaction with BEGIN TRANSACTION
  - Autocommit can be turned off using \set AUTOCOMMIT off

Savepoints in Transactions

- Special mark inside a transaction that allows all commands that are executed after it was established to be rolled back
  - This restores the transaction state to what it was at the time of the savepoint
- Do not end transactions; just save the changes performed
- Useful in long transactions
- After rolling back to a savepoint, the transaction remains active
- All savepoints are removed when the transaction ends



Savepoints in SQL

- Create a savepoint:
  - SAVEPOINT *name*
- Rollback to savepoint:
  - ROLLBACK TO SAVEPOINT *name*
- Release savepoint:
  - RELEASE SAVEPOINT *name*

```
begin;
insert into table1 values (1);
savepoint my_savepoint;
insert into table1 values (2);
rollback to savepoint my_savepoint;
insert into table1 values (3);
commit;
```

```
begin;
insert into table1 values (1);
savepoint my_savepoint;
insert into table1 values (2);
release savepoint my_savepoint;
insert into table1 values (3);
commit;
```

What is the result?

Transaction Log

- Used to keep track of all transaction operations that update the database
  - Used for recovery purposes!

- Includes the following info:
  - A record for the beginning of the transaction
  - For each transaction operation (SQL statement):
    - Type of operation (INSERT, UPDATE, DELETE)
    - Names of the objects affected (the name of the table)
    - The “before” and “after” values for the fields updated
    - Pointers to the previous and next transaction log entries for the same transaction
  - The ending (COMMIT) of the transaction

Simple Transaction Log: Illustration Only

trl_id	*trx_nur*	*prev_ptr*	*next_ptr*	*operation*	*table*	*row_id*	*attribute*	*before_value*	*after value*
341	101		352	begin					
352	101	341	363	update	product	1558-QW1	prod_qoh	25	23
363	101	352	365	update	customer	10011	cust_balance	525.75	615.73
365	101	363		commit					

DEMO #1



- Consider the database used in the PL classes

- TODO:
  - Open a session using *psql*
  - Insert a department and then rollback
  - Start a new transaction, insert a department, and then rollback
  - Start a new transaction, insert a department, and then commit
  - Open a second session with the same username
  - In the first session: start a transaction, insert a new department; in the second session: select the list of departments
  - In the first session: update the location of all departments to uppercase; in the second session: update the name of all departments to uppercase

Databases

Lost Updates
Uncommitted Data
Inconsistent Retrievals
The Scheduler

CONCURRENCY CONTROL

Concurrency Control

- Coordinating the simultaneous execution of transactions in a multiuser database system is known as concurrency control
  - The objective is to ensure the serializability of transactions
- Oriented towards preserving the isolation property of concurrently executing transactions

- Simultaneous execution of transactions over a shared database can create several data integrity and consistency problems
  - Lost updates
  - Uncommitted data
  - Inconsistent retrievals
  - (assuming no concurrency mechanisms are in place)

Lost Updates

- Occurs when two concurrent transactions, T1 and T2, are updating the same data element and one of the updates is lost (overwritten)
- Consider two transactions:
  - T1: purchase 100 units
  - T2: sell 30 units

Assuming prod_qoh=100, what will be the final value?

Time	T1	T2
1	`read(prod_qoh)`	
2	`prod_qoh = prod_qoh + 100`	
3	`write(prod_qoh)`	
4		`read(prod_qoh)`
5		`prod_qoh = prod_qoh - 30`
6		`write(prod_qoh)`

Lost Updates

- Occurs when two concurrent transactions, T1 and T2, are updating the same data element and one of the updates is lost (overwritten)
- Consider two transactions:
  - T1: purchase 100 units
  - T2: sell 30 units

Assuming prod_qoh=100, what will be the final value?

Time	T1	T2
1	read(*prod_qoh*)	
2		read(*prod_qoh*)
3	*prod_qoh* = *prod_qoh* + 100	
4		*prod_qoh* = *prod_qoh* - 30
5	write(*prod_qoh*)	
6		write(*prod_qoh*)

Uncommitted Data

- Occurs when two transactions, T1 and T2, are executed concurrently and T1 is rolled back after T2 has accessed the uncommitted data
- Consider two transactions:
  - T1: purchase 100 units – rolled back
  - T2: sell 30 units

Assuming prod_qoh=100, what will be the final value?

Time	T1	T2
1	`read(prod_qoh)`	
2	`prod_qoh = prod_qoh + 100`	
3	`write(prod_qoh)`	
4	`rollbak`	
5		`read(prod_qoh)`
6		`prod_qoh = prod_qoh - 30`
7		`write(prod_qoh)`

Uncommitted Data

- Occurs when two transactions, T1 and T2, are executed concurrently and T1 is rolled back after T2 has accessed the uncommitted data
- Consider two transactions:
  - T1: purchase 100 units – rolled back
  - T2: sell 30 units

Assuming prod_qoh=100, what will be the final value?

Time	T1	T2
1	read(*prod_qoh*)	
2	*prod_qoh* = *prod_qoh* + 100	
3	write(*prod_qoh*)	
4		read(*prod_qoh*) (read uncommitted data)
5		*prod_qoh* = *prod_qoh* – 30
6	rollbak	
7		write(*prod_qoh*)

Inconsistent Retrievals

- Occur when a transaction accesses data before and after one or more other transactions finish working with such data
- e.g., T1 calculates some summary (aggregate) function over a set of data while another transaction (T2) was updating the same data
  - The problem is that the transaction might read some data before it is changed and other data after it is changed

Time	T1	T2
1	SELECT SUM(PROD_QOH) FROM PRODUCT	UPDATE PRODUCT SET PROD_QOH = PROD_QOH + 10 WHERE PROD_CODE = 1546-QQ2
2		UPDATE PRODUCT SET PROD_QOH = PROD_QOH – 10 WHERE PROD_CODE = 1558-QW1
3		COMMIT;

The Scheduler

- Process that establishes the order in which the operations are executed within concurrent transactions.
  - Interleaves the execution of database operations to ensure serializability and isolation of transactions
- To determine the appropriate order, the scheduler bases its actions on concurrency control algorithms
- Not all transactions are serializable!
  - DBMS determines what transactions are serializable and proceeds to interleave the execution of the transaction's operations
  - Not serializable transactions are executed on a first-come, first-served basis

The Scheduler

- Goal is to create a serializable schedule of a transaction's operations
  - Interleaved execution of the transactions (T1, T2, T3, etc.) yields the same results as if the transactions were executed in serial order (one after another)
- Also makes sure that the computer's central processing unit (CPU) and storage systems are used efficiently
- Scheduler facilitates data isolation to ensure that two transactions do not update the same data element at the same time
- Conflict scenarios:
  - READ / READ - *no conflict*
  - READ / WRITE - *conflict*
  - WRITE / READ - *conflict*
  - WRITE / WRITE - *conflict*

Concurrency Control Methods

- Several methods have been proposed to schedule the execution of conflicting operations in concurrent transactions
  - Time stamping
  - Optimistic
  - Locking – most used, discussed in detail next!

Time Stamping Method

- Assigns a global, unique time stamp to each transaction
- Time stamp value produces an explicit order in which transactions are submitted to the DBMS
- Time stamps must have two properties:
  - Uniqueness: ensures that no equal time stamp values can exist
  - Monotonicity: ensures that time stamp values always increase
- The DBMS executes conflicting operations in time stamp order, thereby ensuring serializability of the transactions
- If two transactions conflict, one is stopped, rolled back, rescheduled, and assigned a new time stamp value
  - wait/die, wound/wait

Time Stamping Approaches

- wait/die
  - If the transaction requesting the lock is the older of the two transactions, it will wait until the other transaction is completed and the locks are released
  - If the transaction requesting the lock is the younger of the two transactions, it will die (roll back) and is rescheduled using the same time stamp

- wound/wait
  - If the transaction requesting the lock is the older of the two transactions, it will preempt (wound) the younger transaction by rolling it back
    - T1 preempts T2 when T1 rolls back T2
    - The younger, preempted transaction, is rescheduled using the same time stamp
  - If the transaction requesting the lock is the younger of the two transactions, it will wait until the other transaction is completed and the locks are released

Time Stamping Approaches: Summary

TRANSACTION REQUESTING LOCK	TRANSACTION OWNING LOCK	WAIT/DIE SCHEME	WOUND/WAIT SCHEME
T1 (11548789)	T2 (19562545)	• T1 waits until T2 is completed and T2 releases its locks.	• T1 preempts (rolls back) T2. • T2 is rescheduled using the same time stamp.
T2 (19562545)	T1 (11548789)	• T2 dies (rolls back). • T2 is rescheduled using the same time stamp.	• T2 waits until T1 is completed and T1 releases its locks.

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

Optimistic Method

- Based on the assumption that majority of operations do not conflict
- A transaction is executed without restrictions until it is committed

- Each transaction moves through two or three phases:
  - Read: transaction reads the database, executes the needed computations, and makes the updates to a private copy of the database values
  - Validation: the transaction is validated to ensure that the changes made will not affect the integrity and consistency of the database
    - If the validation test is positive, the transaction goes to the write phase
    - If the validation test is negative, the transaction is restarted, and the changes are discarded
  - Write: changes are permanently applied to the database

Locking Method

- A device that guarantees unique use of a data item in a particular transaction operation
  - A transaction requires a lock prior to data access
  - The lock is released after the operation's execution to enable other transactions to lock the data item for their own use

- The use of locks based on the assumption that conflict between transactions is likely is called pessimistic locking
- Lock manager is the component responsible for assigning and releasing locks

Database Lock
Table Lock
Page Lock
Row Lock
Field (attribute) Lock

LOCKING FOR CONCURRENCY CONTROL

Lock Granularity

- Indicates the level of lock to use
  - Different levels have different implications!

- Locking can take place at the following levels
  - Database
  - Table
  - Page
  - Row
  - Field (attribute)

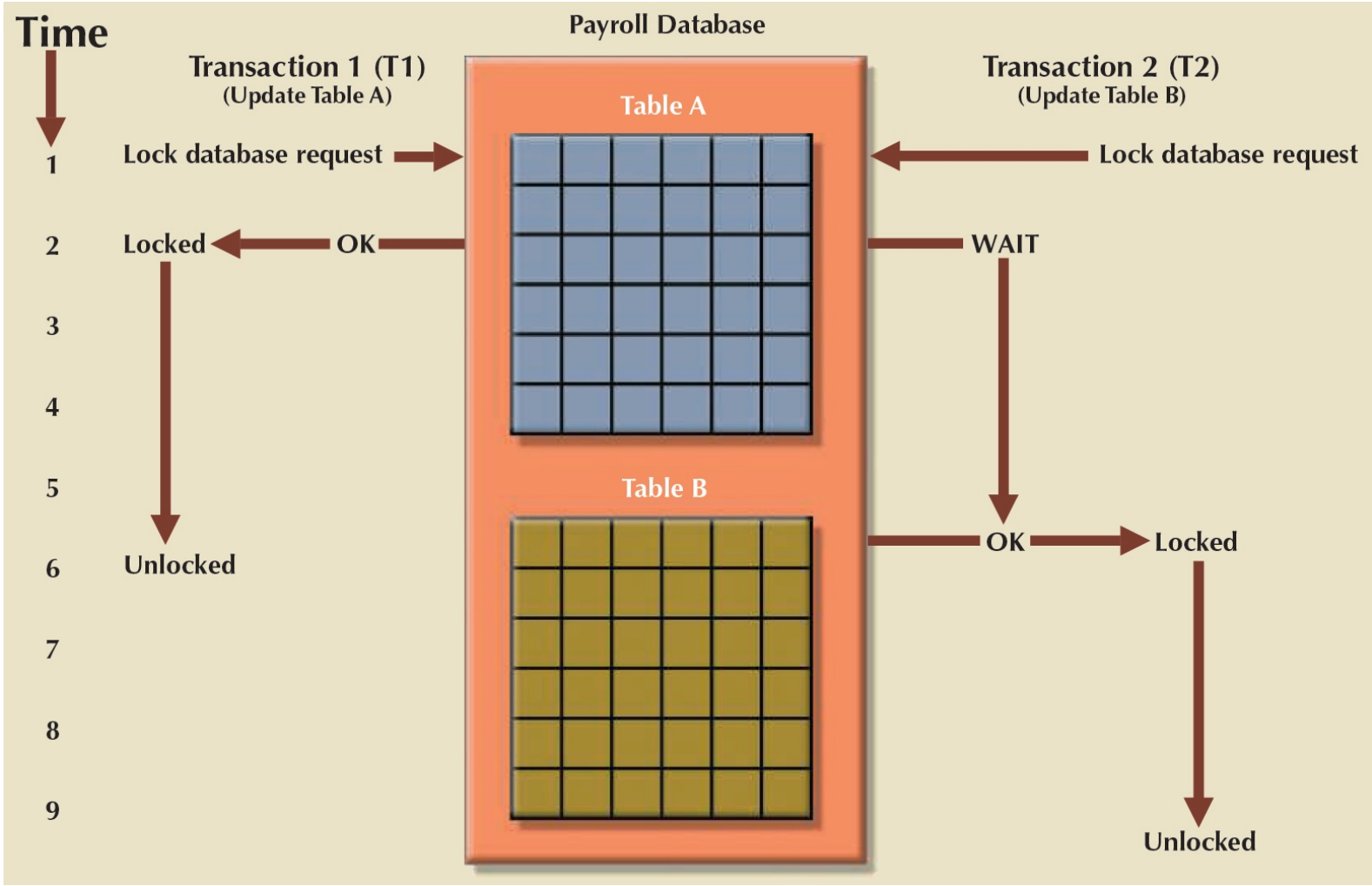
Database-Level Lock

- The entire database is locked!
  - Preventing the use of any tables in the database by transaction T2 while transaction T1 is being executed

- Good for batch processes
- Unsuitable for multiuser scenarios
  - Imagine how slow data access would be if thousands of transactions had to wait for the previous transaction to be completed

- Not implemented in PostgreSQL

Database-Level Lock



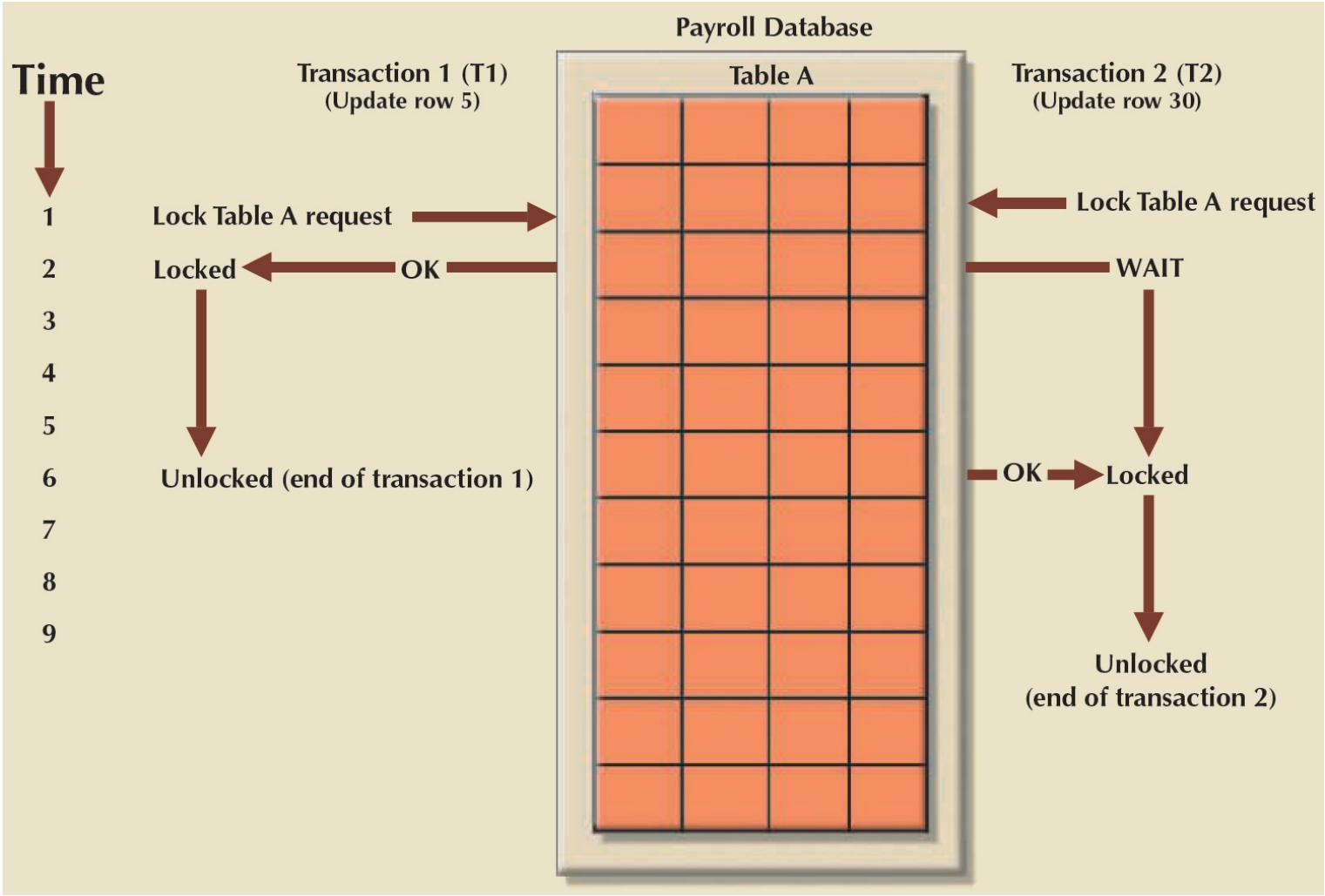
Source: C. Coronel, S. Morris, "Database Systems: Design, Implementation, and Management", Cengage Learning, 12th Edition, 2017.

Table-Level Lock

- The entire table is locked!
  - Preventing access to any row by T2 while T1 is using the table
  - If a transaction requires access to several tables, each table may be locked

- Two transactions can access the same database as long as they access different tables
  - Less restrictive than database-level locks
- Problem when many transactions are waiting to access the same table
  - Especially if the lock forces a delay when different transactions require access to different parts of the same table
    - i.e., when the transactions would not interfere with each other
  - Not suitable for multiuser scenarios

Table-Level Lock



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

Table-Level Lock: Example

Time	T1	T2
1	`begin transaction;`	
2		`begin transaction;`
3	`lock table dep;`	
4		`update dep set location=upper(location);`
5	`insert into dep values (50,'Sales','Coimbra');`	
6	`select * from dep;`	
7	`commit;`	
8		`select * from dep;`
9	`begin transaction;`	
10	`lock table dep;`	
11		`rollbak;`

When is this executed?

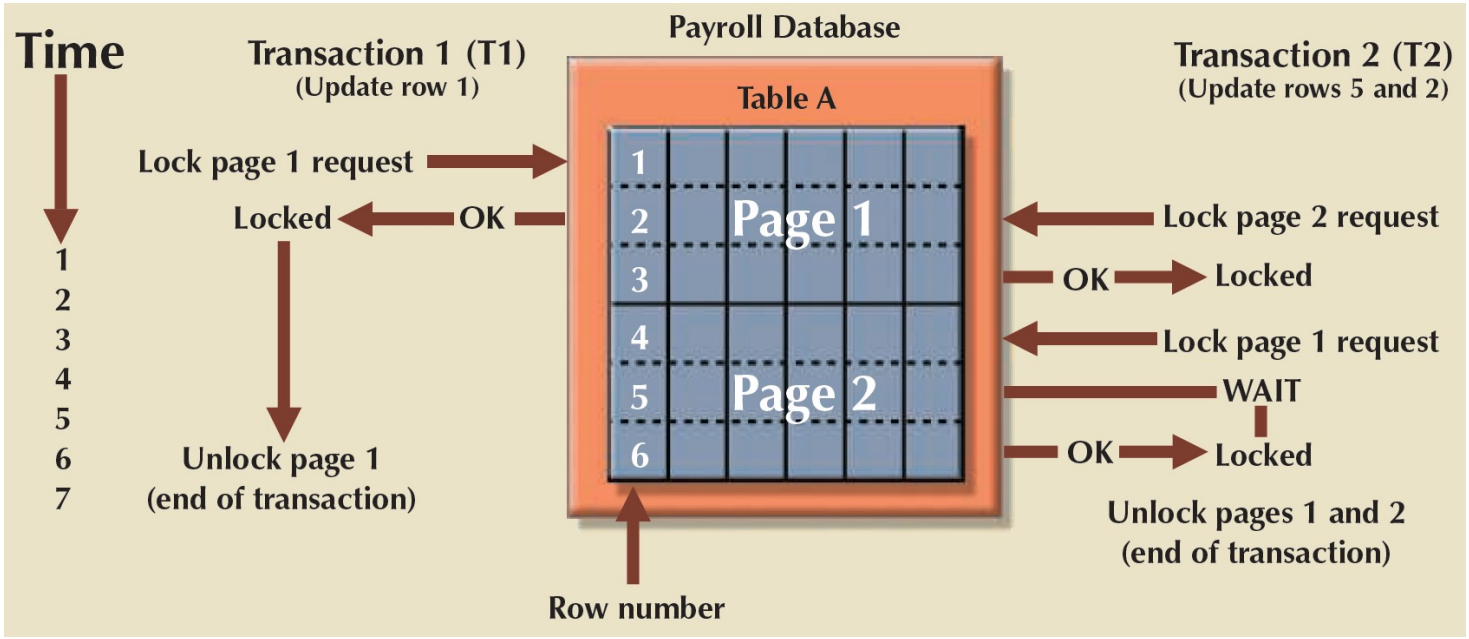
Page-Level Lock

- Locks are done over entire diskpages!
  - A diskpage, or page, is the equivalent of a diskblock, which can be described as a directly addressable section of a disk
    - A page has a fixed size, such as 4K, 8K, or 16K
  - For example, to write only 73 bytes to a 4K page, the entire 4K page must be read from disk, updated in memory, and written back to disk

- A table can span several pages, and a page can contain several rows of one or more tables
  - If a transaction is modifying a given row in a diskpage, then all rows in that page get locked

- Not to be managed by users!

Page-Level Lock



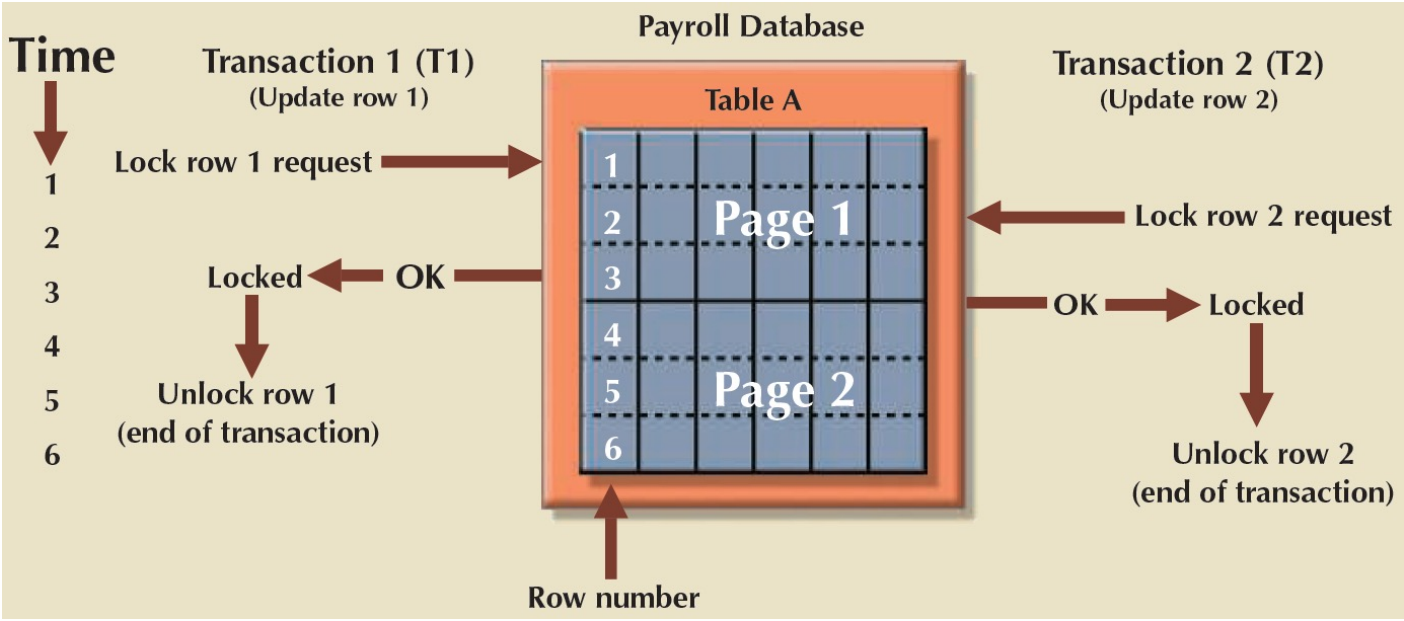
Source: C. Coronel, S. Morris, "Database Systems: Design, Implementation, and Management", Cengage Learning, 12th Edition, 2017.

Row-Level Lock

- Much less restrictive than the locks discussed earlier
- Allows concurrent transactions to access different rows of the same table
  - Thus, improving the availability of data
- Management requires high overhead because a lock exists for each row in a table of the database involved in a conflicting transaction
- Rows are automatically locked when they are modified
- Explicit row lock can be done using SELECT FOR UPDATE

```
select *
from emp
where emp_no = '1245'
for update;
```

Row-Level Lock



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

Row-Level Lock: Example

Time	T1	T2
1	`begin transaction;`	
2		`begin transaction;`
3		`update dep set location=upper(location) where dept_no=10;`
4	`update dep set location=upper(location) where dept_no=20;`	
5		`delete from dep where dept_no=20;`
6	`select * from dep;`	
7	`commit;`	
8		`commit;`

When is this executed?

Field-Level Lock

- Allows concurrent transactions to access the same row as long as they require the use of different fields (attributes) within that row
- Although field-level locking clearly yields the most flexible multiuser data access, it is rarely implemented
  - It requires an extremely high level of computer overhead and
  - The row-level lock is much more useful in practice
- Not implemented in PostgreSQL

Lock Types

- DBMS may use different lock types:
  - Binary: locked or unlocked (e.g., default LOCK TABLE)
  - Shared/exclusive: unlocked, shared (read), and exclusive (write)
    - e.g., SELECT vs UPDATE
    - shared locks allow read transactions to read the same data item concurrently
    - two transactions conflict only when at least one is a write transaction
    - strictly speaking, readers block writers (and vice-versa)
      - but DBMS typically allow shared with exclusive locks (i.e., readers don't block writers) to improve concurrency (e.g., SELECT and UPDATE)
      - this can lead to serialization issues, thus multiple isolation and lock levels exist
- Note: locks can be acquired implicitly (e.g., update) or explicitly (e.g., LOCK TABLE)

Will be further discussed in the next class

DEMO #2



- Consider the database used in the PL classes
- TODO:
  - Open two sessions using *psql*
  - S1: start a transaction, lock departments whose number is lower or equal to 20, insert a new department
  - S2: start a transaction, update the name of department 40 to uppercase
  - S2: update the location of department 10 to uppercase
  - S1: lock table *dep*
  - Rollback both transactions in S1 and S2
  - Start two new transactions (let's call them S3 and S4)
  - S3: insert a department (number 60)
  - S4: insert a department (number 70), insert a department (number 60)

Exercise #1

What happens at each instant T1, T2, …, T15?

What are the results of the queries at instants T7, T12, T13 e T15?

Time	Transaction A	Transaction B
1	`begin;`	`begin;`
2	`update physicians set name='Carlos' where reg_n=746586;`	
3		`select name from physicians where reg_n=768475;`
4	`commit;`	
5	`begin;`	
6	`update physicians set name='Pedro' where reg_n=746586;`	
7		`select name from physicians where reg_n=746586;`
8		`update physicians set name='Simão' where reg_n=365324;`
9	`update physicians set name='Mário' where reg_n=365324;`	
10		`commit;`
11		`begin;`
12		`select name from physicians where reg_n=365324;`
13	`select name from physicians where reg_n=365324;`	
14		`rollback;`
15	`select name from physicians where reg_n=365324;`	

Exercise #2

What happens at each instant?

What are the results of the queries?

Which user can unblock the final impasse?

tab1

name	*code*	*qty*
Joaquim	SS1	5
Pedro	SS5	4
Rui	SS3	5

Time	Transaction A	Transaction B	Transaction C
1	begin;	begin;	begin;
2	select * from tab1 where code='SS1';		set transaction read only;
3		select * from tab1 where code='SS5';	
4		update tab1 set qty= qty+5 where code='SS5';	
5		select * from tab1 where code='SS5';	select * from tab1 where code='SS5';
6	select * from tab1 where code='SS5';		
7		commit;	
8		begin;	
9		update tab1 set qty=qty+5 where code='SS3';	select * from tab1 where code='SS5';
10	select * from tab1 where code='SS1';		commit;
11			begin;
12	select * from tab1 where code='SS5';		select * from tab1 where code='SS5';
13			commit;
14			begin;
15			update tab1 set qty=qty+3 where code='SS1';
16	update tab1 set qty=qty+2 where code='SS1';		
17			update tab1 set qty=qty+2 where code='SS3';

Exercise #3

What is the result of:

```
select * from parts
where name = 'SS1';
```

- If executed in another session B at *time=4*
- If execute in another session B at *time=8*

parts

id_part	name	stock_qty	min_stock	order_quant
1982982	SS1	12	4	16
1827625	SS3	23	6	30
1827633	SS5	12	10	40
1982977	SS1	10	4	16

Time	Transaction A
1	`begin;`
2	

3	`update table parts`
	`set quant_stock = quant_stock - 2`
	`where id peça = 1982982;`

4	

5	`update table parts`
	`set quant_min = quant_min + 2`
	`where nome = 'SS1';`

6	
7	`commit;`
8	
9	`begin;`
10	
11	`insert into table parts` `values (1827669, 'SS9', 60, 10, 60);`
12	
13	`delete from parts` `where id_peça = 1827633;`
14	
15	`commit;`

Exercise #4

- What is the result of:

```
update emp_sup
set stock_qty = stock_qty - 1
where id_part = 1827669;
```

  - If executed in Session B at time=12

```
update emp_sup
set stock_qty = stock_qty - 2
where id_part = 1982977;
```

  - If executed in Session B at time=6

emp_sup

id_part	name	stock_qty	min_stock	order_quant
1982982	SS1	12	4	16
1827625	SS3	23	6	30
1827633	SS5	12	10	40
1982977	SS1	10	4	16

Time	Transaction A
1	`begin;`
2	

3	`update table parts`
	`set quant_stock = quant_stock - 2`
	`where id peça = 1982982;`

4	

5	`update table parts`
	`set quant_min = quant_min + 2`
	`where nome = 'SS1';`

6	
7	`commit;`
8	
9	`begin;`
10	
11	`insert into table parts` `values (1827669, 'SS9', 60, 10, 60);`
12	
13	`delete from parts` `where id_peça = 1827633;`
14	
15	`commit;`

Take-Away(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

Next Lesson(s)

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

Q&A



Databases

Transactions and Concurrency Control

João R. Campos

Bachelor in Informatics Engineering
Department of Informatics Engineering
University of Coimbra
2025/2026