Transaction Control Language (TCL)

CE384: Database Design
Maryam Ramezani
Sharif University of Technology
maryam.ramezani@sharif.edu

Transactions

- A *transaction* is the DBMS's abstract view of a user program: a sequence of reads and writes.
- Concurrent execution of user programs is essential for good DBMS performance.
  - Increasing system **throughput** (# of completed transactions in any given time) by overlapping I/O and CPU operations
  - Increasing **response time** (time for completing a transaction) by avoiding short transactions getting stuck behind long ones
- A user's program may carry out many (in-memory) operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.

REVIEW: The ACID Properties (in a Nutshell)

- Atomicity: Either all actions of the transactions are executed or none at all.
- Consistency: Any transaction that starts executing in a consistent database state must leave it in a consistent state upon completion.
- Isolation: A transaction is protected from effects of concurrently running transactions.
- Durability: Effects of committed transactions must persist and overcome any system failure (system crash/media failure).

Consistency and Isolation

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions. The net effect is the same as serially executing the transactions one after the other.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
- *Issue:* Coping with effects of *interleaving* transactions (*Concurrency control*).

Atomicity and Durability

- A transaction might commit after completing all its actions, or it could terminate unsuccessfully:
  - It could abort (or be aborted by the DBMS) after executing some actions.
  - The system may crash while transactions are in progress.

- How does the DBMS achieve <u>atomicity</u> and <u>durability</u> of all transactions?
  - Atomicity: the DBMS logs all actions so that it can undo the actions of aborted transactions.
  - Durability: committed actions are written to disk or (in case of a crash) the system must redo actions of committed Xacts which were not yet written to disk.

- <u>Issue:</u> Coping with effects of crashes (Recovery).

Database Transactions

- Transactions give you more flexibility and control when changing data, and they ensure data consistency in the event of user process failure or system failure.
- A database transaction consists of one of the following:
  - DML statements which constitute one consistent change to the data
    - For example, a transfer of funds between two accounts should include the debit to one account and the credit to another account in the same amount. Both actions should either fail or succeed together; the credit should not be committed without the debit.
  - One DDL statement
  - One DCL statement

Database Transactions

When Does a Transaction Start and End?

A transaction begins when the first DML statement is encountered and ends when one of the following occurs:

- A COMMIT or ROLLBACK statement is issued
- A DDL statement, such as CREATE, is issued [In Oracle, MySQL,… Not in Postgres]
- A DCL statement is issued [In Oracle, MySQL,… Not in Postgres]
- A machine fails or the system crashes
- After one transaction ends, the next executable SQL statement automatically starts the next transaction.
- A DDL statement or a DCL statement is automatically committed and therefore implicitly ends a transaction.

Example

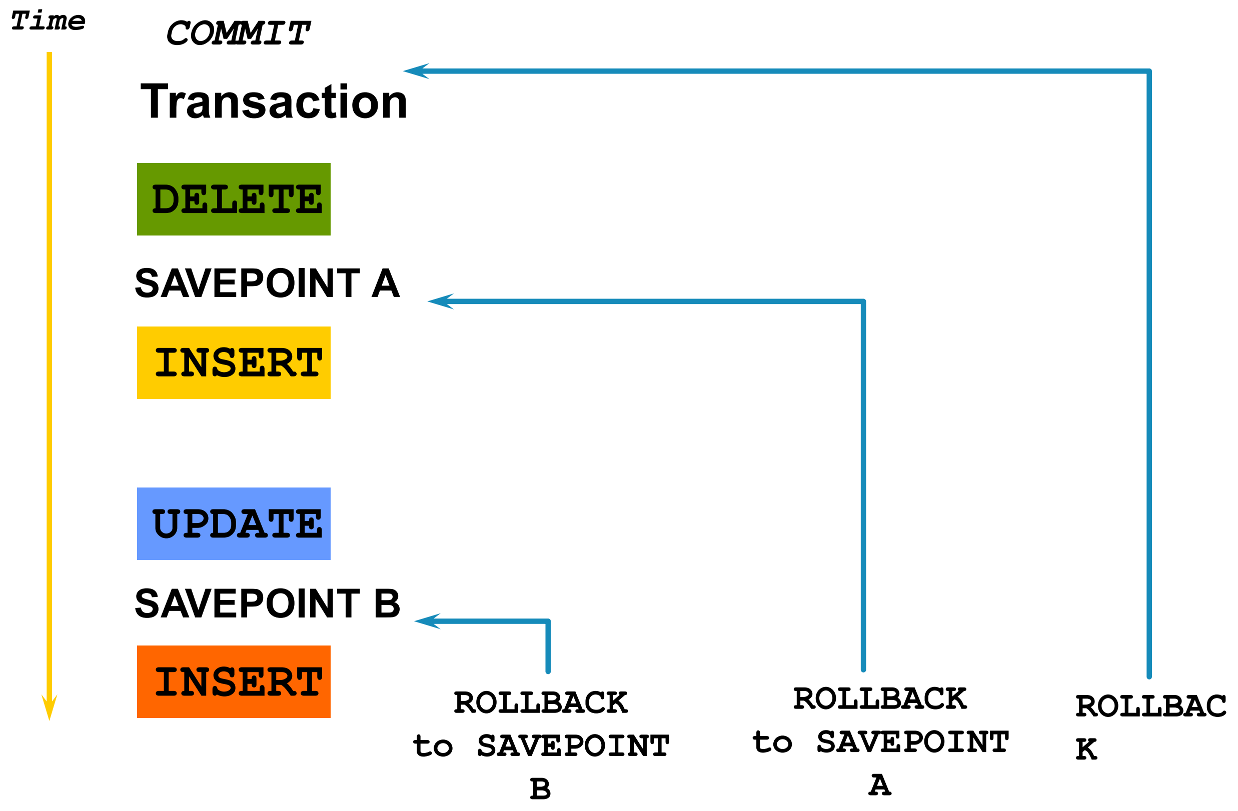
- \c HW2
- select * from accounts;
- Begin; update accounts set balance=222 where id=1;

Advantages of `COMMIT` and `ROLLBACK` Statements

With `COMMIT` and `ROLLBACK` statements, you can:

- Ensure data consistency
- Preview data changes before making changes permanent
- Group logically related operations

Controlling Transactions



Rolling Back Changes to a Marker

- Create a marker in a current transaction by using the `SAVEPOINT` statement.
- Roll back to that marker by using the `ROLLBACK TO SAVEPOINT` statement.
- If you create a second savepoint with the same name as an earlier savepoint, the earlier savepoint is deleted.

```
UPDATE...
SAVEPOINT update_done;
Savepoint created.
INSERT...
ROLLBACK TO update_done;
Rollback complete.
```

State of the Data Before COMMIT or ROLLBACK

- The previous state of the data can be recovered.
- The current user can review the results of the DML operations by using the SELECT statement.
- Other users *cannot* view the results of the DML statements by the current user.
- The affected rows are *locked*; other users cannot change the data within the affected rows.

State of the Data after COMMIT

- Data changes are made permanent in the database.
- The previous state of the data is permanently lost.
- All users can view the results.
- Locks on the affected rows are released; those rows are available for other users to manipulate.
- All savepoints are erased.

Committing Data

- Make the changes.

```
DELETE FROM employees
WHERE  employee_id = 99999;
1 row deleted.

INSERT INTO departments
VALUES (290, 'Corporate Tax', NULL, 1700);
1 row inserted.
```

- Commit the changes.

```
COMMIT;
Commit complete.
```

State of the Data After ROLLBACK

Discard all pending changes by using the ROLLBACK statement:

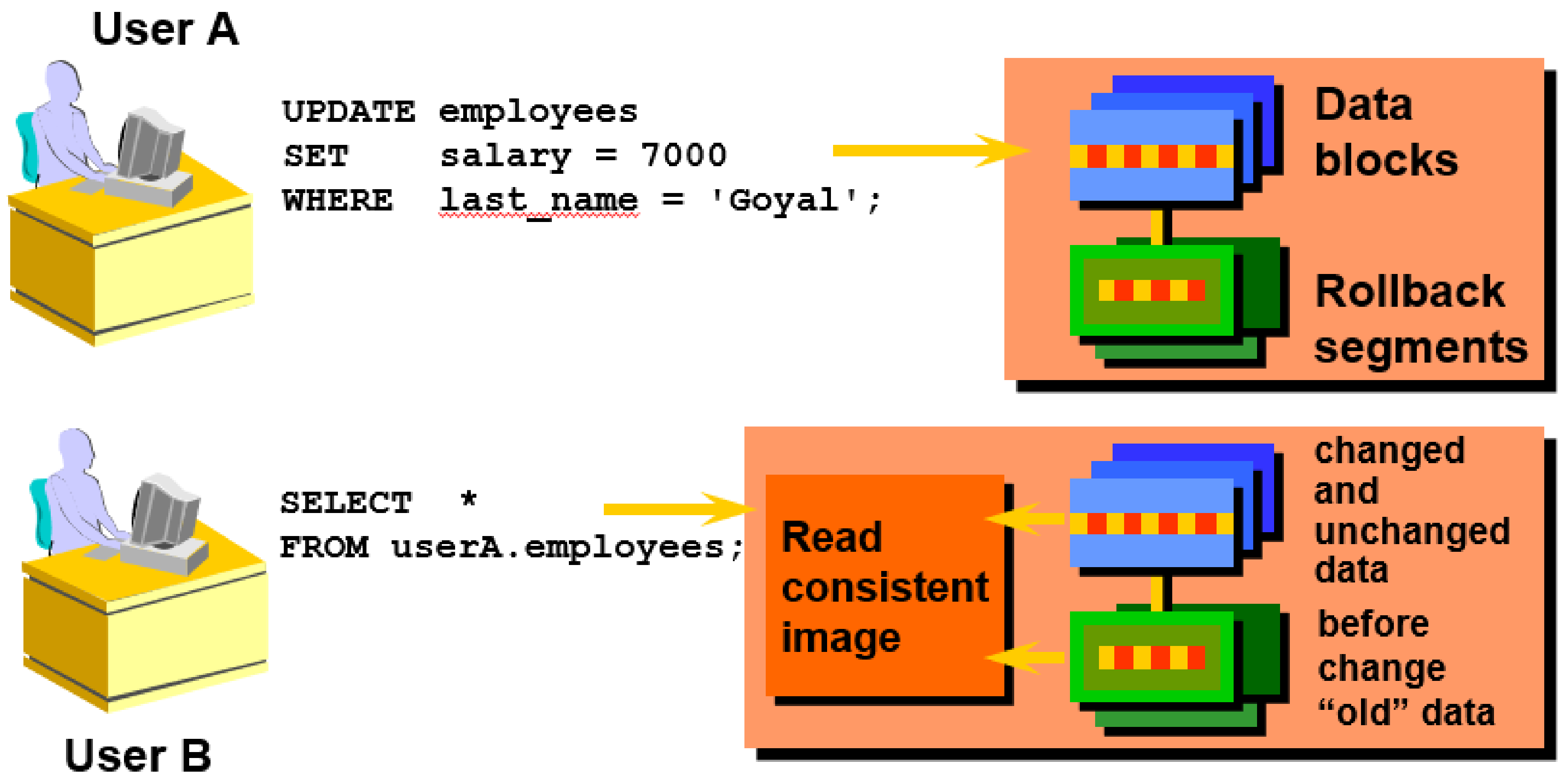
- Data changes are undone.
- Previous state of the data is restored.
- Locks on the affected rows are released.

```
DELETE FROM copy_emp;
22 rows deleted.
ROLLBACK;
Rollback complete.
```

Read Consistency

- Read consistency guarantees a consistent view of the data at all times.
- Changes made by one user do not conflict with changes made by another user.
- Read consistency ensures that on the same data:
  - Readers do not wait for writers.
  - Writers do not wait for readers.

Implementation of Read Consistency



```
UPDATE employees
SET    salary = 7000
WHERE  last_name = 'Goyal';
```

```
SELECT  *
FROM userA.employees;
```

Implementation of Read Consistency

- Implementation of Read Consistency
  - Read consistency is an automatic implementation. It keeps a partial copy of the database in undo segments.
  - When an insert, update, or delete operation is made to the database, the Oracle server takes a copy of the data before it is changed and writes it to a *undo segment*.
  - All readers, except the one who issued the change, still see the database as it existed before the changes started; they view the rollback segment's "snapshot" of the data.
  - Before changes are committed to the database, only the user who is modifying the data sees the database with the alterations; everyone else sees the snapshot in the undo segment. This guarantees that readers of the data read consistent data that is not currently undergoing change.
  - When a DML statement is committed, the change made to the database becomes visible to anyone executing a SELECT statement. The space occupied by the *old* data in the undo segment file is freed for reuse.
  - If the transaction is rolled back, the changes are undone:
    - The original, older version, of the data in the undo segment is written back to the table.
    - All users see the database as it existed before the transaction began.

Locking

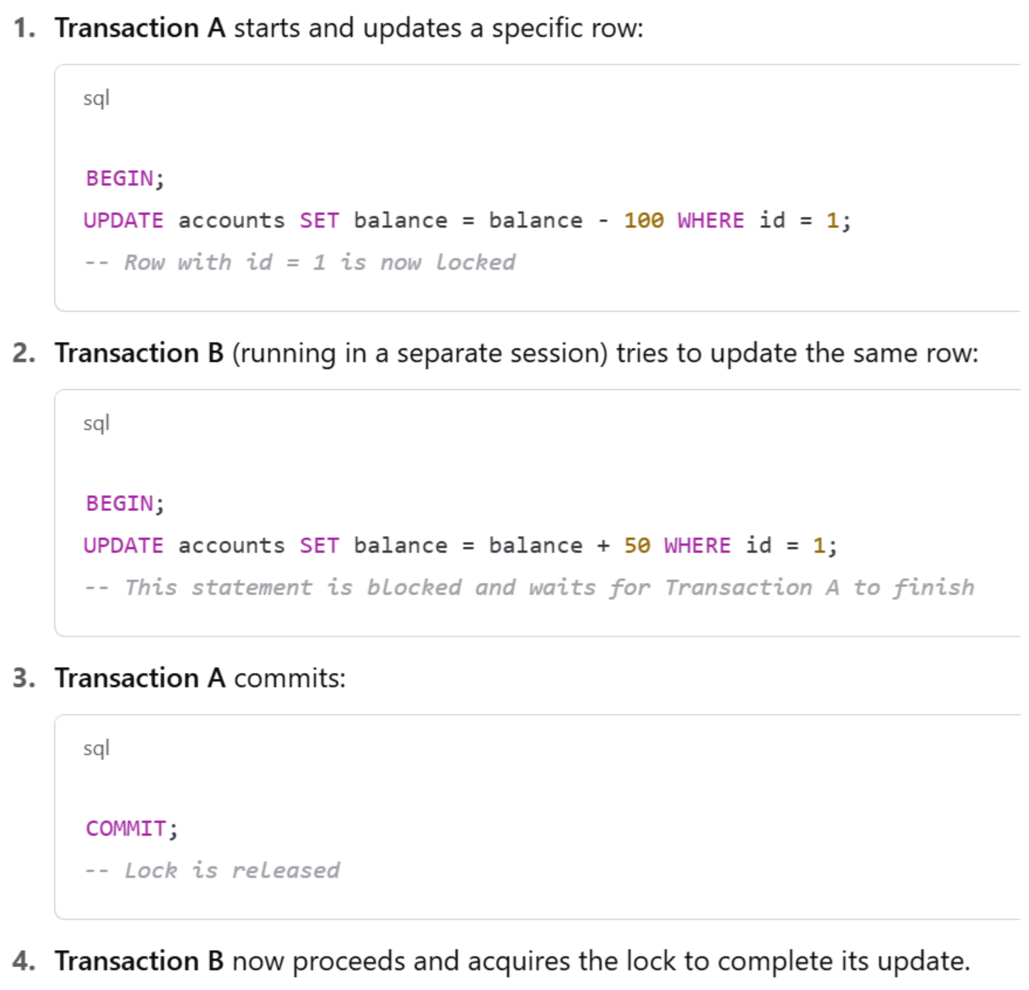
Locks are mechanisms that prevent destructive interaction between transactions accessing the same resource, either a user object (such as tables or rows) or a system object not visible to users (such as shared data structures and data dictionary rows).

In Postgres database, locks:

- Locks prevent conflicting operations between concurrent transactions.
- Locking is handled automatically by the system in most cases—no manual intervention is typically needed.
- PostgreSQL automatically applies the least restrictive lock necessary to maintain consistency and isolation.
- Locks are usually held for the duration of the transaction, unless explicitly released or the transaction ends (via COMMIT or ROLLBACK).
- There are two main types of locking:
- Implicit locking: Automatically managed by PostgreSQL (e.g., row-level locks during UPDATE).
- Explicit locking: Manually controlled by users (e.g., using LOCK TABLE, SELECT FOR UPDATE).

DBMS Lock Example

PostgreSQL uses **row-level locks** to prevent concurrent updates to the same row.



1. **Transaction A** starts and updates a specific row:

```sql
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
-- Row with id = 1 is now locked
```

2. **Transaction B** (running in a separate session) tries to update the same row:

```sql
BEGIN;
UPDATE accounts SET balance = balance + 50 WHERE id = 1;
-- This statement is blocked and waits for Transaction A to finish
```

3. **Transaction A** commits:

```sql
COMMIT;
-- Lock is released
```

4. **Transaction B** now proceeds and acquires the lock to complete its update.

User Lock Example

```
BEGIN;
SELECT * FROM accounts WHERE id = 2 FOR UPDATE;


Begin;
LOCK TABLE accounts IN ACCESS EXCLUSIVE MODE;
```

Implicit Locking

- High level of data concurrency:
  - DML: Table share, row exclusive
  - Queries: No locks required
  - DDL: Protects object definitions: DDL locks occur when you modify a database object such as a table.

- Two lock modes:
  - Exclusive: Locks out other users: An exclusive lock is acquired automatically for each row modified by a DML statement. Exclusive locks prevent the row from being changed by other transactions until the transaction is committed or rolled back. This lock ensures that no other user can modify the same row at the same time and overwrite changes not yet committed by another user.
  - Share: Allows other users to access: A share lock is automatically obtained at the table level during DML operations. With share lock mode, several transactions can acquire share locks on the same resource.

- Locks held until commit or rollback

Summary

Statement	Description
COMMIT	Makes all pending changes permanent
SAVEPOINT	Is used to rollback to the savepoint marker
ROLLBACK	Discards all pending data changes