Transactions

Controlling Concurrent Behavior

Why Transactions?

- Database systems are normally being accessed by many users or processes at the same time.
  - Both queries and modifications.
- Unlike operating systems, which *support* interaction of processes, a DMBS needs to keep processes from troublesome interactions.

Example: Bad Interaction

- You and your domestic partner each take $100 from different ATM's at about the same time.
  - The DBMS better make sure one account deduction doesn't get lost.
- Compare: An OS allows two people to edit a document at the same time. If both write, one's changes get lost.

Transactions

- *Transaction* = process involving database queries and/or modification.
- Normally with some strong properties regarding concurrency.
- Formed in SQL from single statements or explicit programmer control.

ACID Transactions

- *ACID transactions* are:
  - *Atomic* : Whole transaction or none is done.
  - *Consistent* : Database constraints preserved.
  - *Isolated* : It appears to the user as if only one process executes at a time.
  - *Durable* : Effects of a process survive a crash.
- Optional: weaker forms of transactions are often supported as well.

COMMIT

- The SQL statement COMMIT causes a transaction to complete.
  - It's database modifications are now permanent in the database.

ROLLBACK

- The SQL statement ROLLBACK also causes the transaction to end, but by *aborting*.
  - No effects on the database.
- Failures like division by 0 or a constraint violation can also cause rollback, even if the programmer does not request it.

Example: Interacting Processes

- Assume the usual Sells(bar,beer,price) relation, and suppose that Joe's Bar sells only Bud for $2.50 and Miller for $3.00.
- Sally is querying Sells for the highest and lowest price Joe charges.
- Joe decides to stop selling Bud and Miller, but to sell only Heineken at $3.50.

Sally's Program

- Sally executes the following two SQL statements called (min) and (max) to help us remember what they do.

(max) SELECT MAX(price) FROM Sells
WHERE bar = 'Joes Bar';

(min) SELECT MIN(price) FROM Sells
WHERE bar = 'Joes Bar';

Joe's Program

- At about the same time, Joe executes the following steps: (del) and (ins).

```
(del)  DELETE FROM Sells
       WHERE bar = 'Joes Bar';
(ins)  INSERT INTO Sells
       VALUES('Joes Bar', 'Heineken', 3.50);
```

Interleaving of Statements

- Although (max) must come before (min), and (del) must come before (ins), there are no other constraints on the order of these statements, unless we group Sally's and/or Joe's statements into transactions.

Example: Strange Interleaving

- Suppose the steps execute in the order (max)(del)(ins)(min).

Joe's Prices:	{2.50,3.00}	{2.50,3.00}		{3.50}
Statement:	(max)	(del)	(ins)	(min)
Result:	3.00			3.50

- Sally sees MAX < MIN!

Fixing the Problem by Using Transactions

- If we group Sally's statements (max)(min) into one transaction, then she cannot see this inconsistency.
- She sees Joe's prices at some fixed time.
  - Either before or after he changes prices, or in the middle, but the MAX and MIN are computed from the same prices.

Another Problem: Rollback

- Suppose Joe executes (del)(ins), not as a transaction, but after executing these statements, thinks better of it and issues a ROLLBACK statement.
- If Sally executes her statements after (ins) but before the rollback, she sees a value, 3.50, that never existed in the database.

Solution

- If Joe executes (del)(ins) as a transaction, its effect cannot be seen by others until the transaction executes COMMIT.
  - If the transaction executes ROLLBACK instead, then its effects can *never* be seen.

Isolation Levels

- SQL defines four *isolation levels* = choices about what interactions are allowed by transactions that execute at about the same time.
- Only one level ("serializable") = ACID transactions.
- Each DBMS implements transactions in its own way.

Choosing the Isolation Level

- Within a transaction, we can say:

SET TRANSACTION ISOLATION LEVEL *X*

where *X* =

1. SERIALIZABLE
2. REPEATABLE READ
3. READ COMMITTED
4. READ UNCOMMITTED

/* Oracle allows only 1 and 3 and some similar to 2. */

Serializable Transactions

- If Sally = (max)(min) and Joe = (del)(ins) are each transactions, and Sally runs with isolation level SERIALIZABLE, then she will see the database either before or after Joe runs, but not in the middle.

Isolation Level Is Personal Choice

- Your choice, e.g., run serializable, affects only how *you* see the database, not how others see it.
- Example: If Joe Runs serializable, but Sally doesn't, then Sally might see no prices for Joe's Bar.
  - i.e., it looks to Sally as if she ran in the middle of Joe's transaction.

Read-Commited Transactions

- If Sally runs with isolation level READ COMMITTED, then she can see only committed data, but not necessarily the same data each time.
- Example: Under READ COMMITTED, the interleaving (max)(del)(ins)(min) is allowed, as long as Joe commits.
  - Sally sees MAX < MIN.

Repeatable-Read Transactions

- Requirement is like read-committed, plus: if data is read again, then everything seen the first time will be seen the second time.
  - But the second and subsequent reads may see *more* tuples as well.

Example: Repeatable Read

- Suppose Sally runs under REPEATABLE READ, and the order of execution is (max)(del)(ins)(min).
  - (max) sees prices 2.50 and 3.00.
  - (min) can see 3.50, but must also see 2.50 and 3.00, because they were seen on the earlier read by (max).

Read Uncommitted

- A transaction running under READ UNCOMMITTED can see data in the database, even if it was written by a transaction that has not committed (and may never).
- Example: If Sally runs under READ UNCOMMITTED, she could see a price 3.50 even if Joe later aborts.

Oracle Transactions

- A database transaction consists of one of the following:
  - DML statements that constitute one consistent change to the data
  - One DDL statement
  - One data control language (DCL) statement

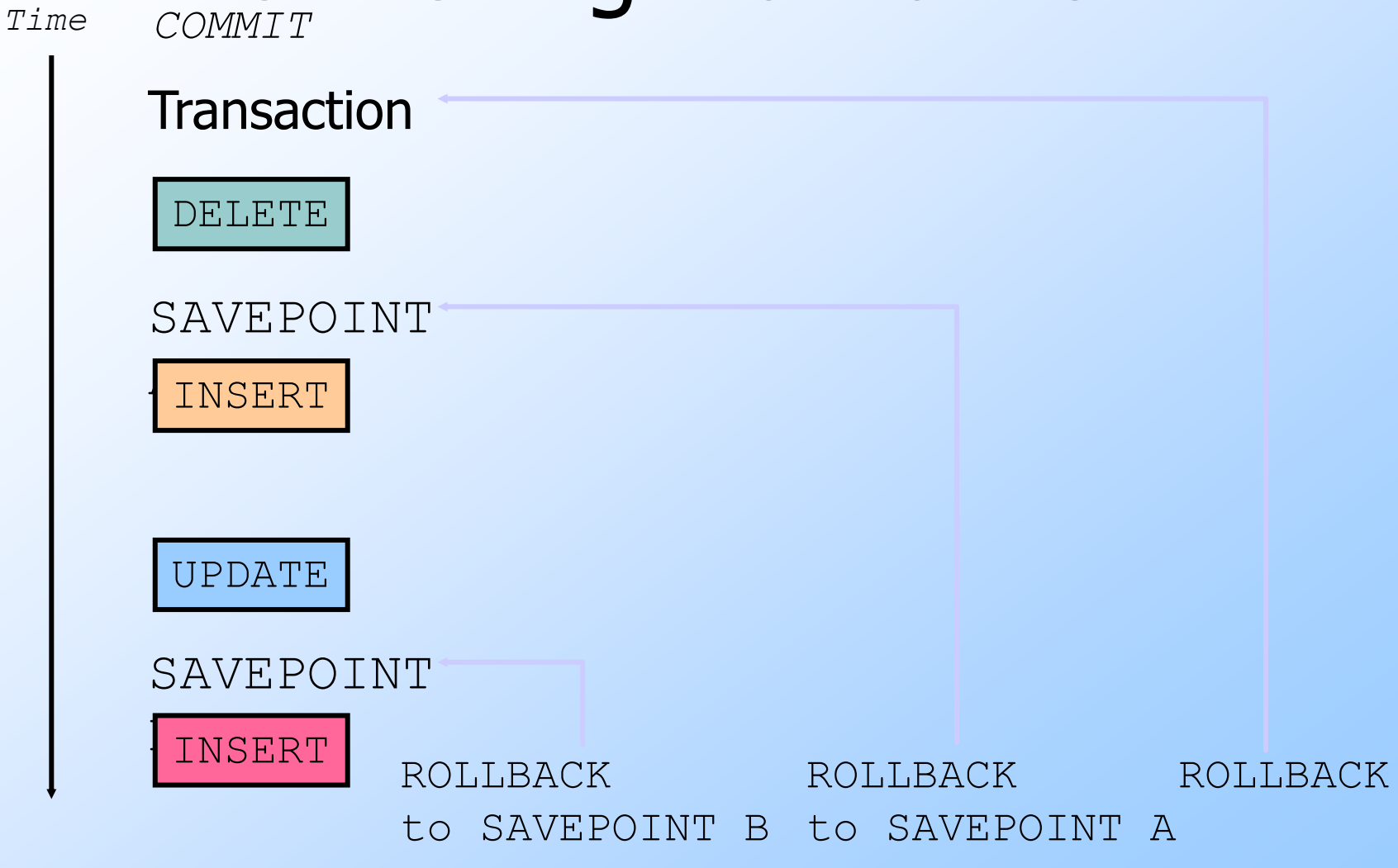
Database Transactions

- Begin when the first DML SQL statement is executed.
- End with one of the following events:
  - A `COMMIT` or `ROLLBACK` statement is issued.
  - A DDL or DCL statement executes (automatic commit).
  - The user exits SqlDeveloper.
  - The system crashes.

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.

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

Implicit Transaction Processing

- An automatic commit occurs under the following circumstances:
  - DDL statement is issued
  - DCL statement is issued
  - Normal exit from SqlDeveloper, without explicitly issuing `COMMIT` or `ROLLBACK` statements
- An automatic rollback occurs under an abnormal termination of SqlDeveloper or a system failure.

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 in 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 created.
```

- 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.
```

State of the Data After ROLLBACK

```
DELETE FROM test;        -- ups!, it's a mistake
25,000 rows deleted.

ROLLBACK;                -- correct the mistake
Rollback complete.

DELETE FROM test WHERE  id = 100; -- it's ok
1 row deleted.

SELECT * FROM   test WHERE  id = 100;
No rows selected.

COMMIT;                   -- make it permanent
Commit complete.
```

Statement-Level Rollback

- If a single DML statement fails during execution, only that statement is rolled back.
- The Oracle server implements an implicit savepoint.
- All other changes are retained.
- The user should terminate transactions explicitly by executing a `COMMIT` or `ROLLBACK` statement.

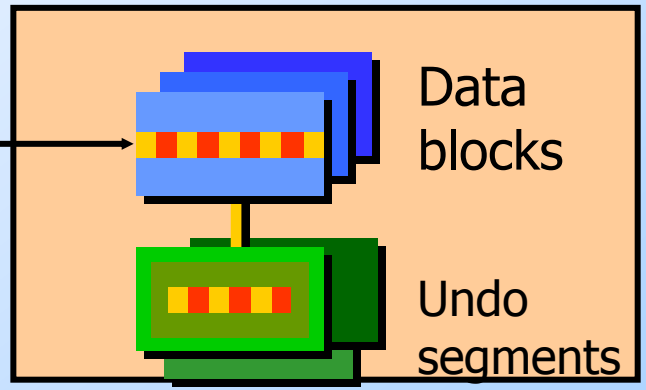
Read Consistency

- 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

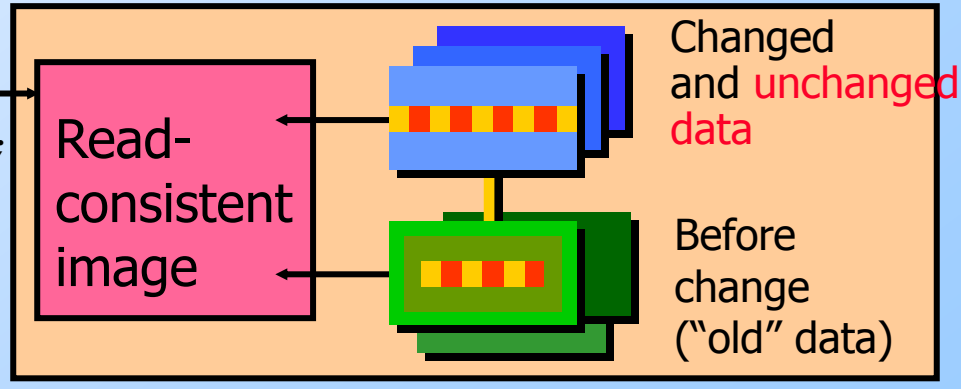
Implementation of Read Consistency

User A

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



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



User B