SQL (Part 4)
Database Modification Statements
and
Transactions in SQL

Instructor: Shel Finkelstein

- *Database Modification Reference: A First Course in Database Systems, 3rd edition, Chapter 6.5*
- *Transactions Reference: Transactions Reference: A First Course in Database Systems, 3rd edition, Chapter 6.6 – 6.7*

Important Notices

- Lab3 assignment is due on **Sunday, May 19**, by 11:59pm.
  - 3 weeks to do Lab3 because of Midterm.
  - Lab2 was/will be discussed at Lab Sections.
  - Your solution should be submitted via Canvas as a zip file.
    - Canvas is used for both Lab submission and grading.
    - Late Lab Assignments will not be accepted.
    - Be sure that you post the correct file!
  - Load file for Lab3 has been posted to Piazza.
    - You must use load file to do Lab3.
    - Helps with testing, but we won't post query solutions.
- The **third** Gradiance Assignment, "CMPS 182 Spring 2019 #3", was/will be posted on Monday, April 29, and is due **Monday, May 6**, by 11:59pm.

Important Notices

- **Reminder:** Midterm is on **Wednesday, May 8.**
  - **Includes material up to and including this Lecture**, but not Lecture 7.
  - No make-ups, no early Midterms, no late Midterms … and no devices.
  - You may bring a **single two-sided 8.5" x 11" sheet of paper** with as much info written (or printed) on it as you can fit and read unassisted.
    - No sharing of these sheets will be permitted.
  - "Practice Midterm" from Spring 2017 has beenposted on Piazza under Resources→Exams.
    - Solution will be posted there next week … but take it yourself first.
    - Some questions are on topics we haven't covered yet.
  - Hope that all requests for DRC accommodation have been submitted.
  - Piazza announcement will describe required seating pattern for Midterm.
- See Small Group Tutoring website for LSS Tutoring with Chandler Hawkins.

Database Modification Statements

Instructor: Shel Finkelstein

Database Modification Reference:
A First Course in Database Systems,
3rd edition, Chapter 6.5

Database Modification Statements

- SQL statements for:
  - *Inserting some* tuples into a relation
  - *Deleting* some tuples from a relation
  - *Updating* values of some columns of some existing tuples

- INSERT, DELETE, and UPDATE are referred to as *modification* operations.
  - They are Data Manipulation Language (DML) statements, as is SELECT.

- Modification operations change the *state* of the database.
  - They do not return a collection of rows or other values.
  - They may return errors/error codes.

Insert Statement with Values

INSERT INTO R(A_1, …, A_n)
    VALUES (v_1, …, v_n);

- A tuple (v_1, …, v_n) is inserted into the relation R, where attribute $A_i = v_i$ and default values (perhaps NULL) are entered for all missing attributes.

INSERT INTO StarsIn(movieTitle, movieYear, starName)
    VALUES (‘The Maltese Falcon’, 1942, ‘Sydney Greenstreet’);

- The tuple (‘The Maltese Falcon’, 1942, ‘Sydney Greenstreet’) will be added to the relation StarsIn.

INSERT INTO StarsIn
    VALUES (‘The Maltese Falcon’, 1942, ‘Sydney Greenstreet’);

INSERT Statement with Subquery

Movies(<u>title, year</u>, length, genre, studioName, producerC#)
Studio(<u>name</u>, address, presC#)

```
INSERT INTO Studio(name)
    SELECT DISTINCT studioName
    FROM Movies
    WHERE studioName NOT IN
            (SELECT name
             FROM Studio);
```

Why DISTINCT?

- Add to the relation Studio all the names that appear in the studioName column of Movies but do not already occur in the names in the Studio relation.

INSERT Statement with Subquery and Constants

Movies(title, year, length, genre, studioName, producerC#)

Disney2018Movies(title, length)

```
INSERT INTO Movies(title, year, length, studioName)
    SELECT dm.title, 2018, dm.length, 'Disney'
    FROM Disney2018Movies dm;
```

Puts the Disney2018Movies into the Movies table, setting year and studioName values using constants.

What will be the genre and producerC# values for the inserted movies?

DELETE Statement

```
DELETE FROM R
    WHERE <condition>;
```

```
DELETE FROM StarsIn
    WHERE movieTitle = 'The Maltese Falcon' AND
        movieYear = 1942 AND
        starName = 'Sydney Greenstreet';
```

- The tuple ('The Maltese Falcon', 1942, 'Sydney Greenstreet') will be deleted from the relation StarsIn.

- What if we wanted to delete tuples from StarsIn for all movies starring Sydney Greenstreet?

More DELETE Examples

```
DELETE FROM MovieExec
    WHERE netWorth < 10000000;
```

- Deletes all movie executives whose net worth is less than 10 million dollars.

```
DELETE FROM MovieExec
    WHERE cert# IN
        (SELECT m.producerC#
         FROM Movies m, StarsIn s
          WHERE  m.title = s.movieTitle AND m.year = s.movieYear
                    AND s.starName = 'Sydney Greenstreet');
```

- Deletes all movie executives who produced movies starring Sydney Greenstreet

DELETE: Careful

What does:

```
DELETE FROM MovieExec;
```

without a WHERE clause do?

Answer: Deletes all the tuples from MovieExec!!!

UPDATE Statement

UPDATE R
SET <new-value-assignments>
WHERE <condition>;

- <new-value-assignment> :-
  <attribute> = <expression>, …, <attribute> = <expression>

UPDATE Employees
SET salary = 85000, dept = ‘SALES’
WHERE SSnum=‘123456789’;

UPDATE Employees
SET salary = 25000
WHERE salary IS NULL;

UPDATE Employees
SET salary = salary * 1.1
WHERE salary > 100000;

UPDATE with Subquery

UPDATE R
SET <new-value-assignments>
WHERE <condition>;

- <new-value-assignment>:-
  <attribute> = <expression>, …, <attribute> = <expression>

UPDATE MovieExec
SET name = ‘Pres. ’ || name
WHERE cert# IN (SELECT presC# FROM Studio);

- 2nd line: concatenates the string ‘Pres. ’ with name.

Semantics of Modifications

- Database modification statements are completely evaluated on the old state of the database, producing a new state of the database.
  - What does this statement do? Is it deterministic or not?

```
UPDATE MovieExec e
   SET e.NetWorth = 6M
   WHERE NOT EXISTS (SELECT * FROM MovieExec e2
                     WHERE e2.Networth = 6M);
```

Should write 6000000,. This is clearer (but it's not legal SQL).

MovieExec

name	address	cert#	netWorth
S. Spielberg	X	38120	3M
G. Lucas	Y	43918	4M
W. Disney	Z	65271	5M

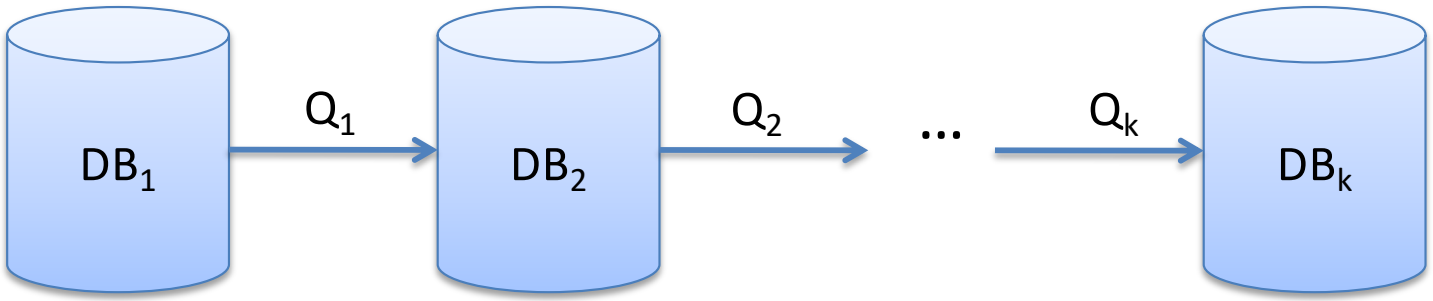
Transactions in SQL

Instructor: Shel Finkelstein

Transactions Reference:
A First Course in Database Systems,
3rd edition, Chapter 6.6 – 6.7

One-Statement-At-a-Time Semantics

- So far, we have learnt how to query and modify the database.
- SQL statements posed to the database system were executed one at a time, retrieving data or changing the data in the database.



Transactions

- Applications such as web services, banking, airline reservations demand high throughput on operations performed on the database.
  - Manage hundreds of sales transactions every second.
  - Transactions often involve multiple SQL statements.
  - Database are transformed to new state based on (multiple statement) transactions, not just single SQL statements.

- It's possible for two operations to simultaneously affect the same bank account or flight, e.g. two spouses doing banking transactions, or an automatic deposit during a withdrawal, or two people reserving the same seat.
  - These "concurrent" operations must be handled carefully.

ACID Transactions

- Atomicity
- Consistency
- Isolation
- Durability

Simple Example of What Could Go Wrong

Flights(fltNo, fltDate, seatNo, seatStatus, purchaser)

- Customer1 issues the following query via a web application.

```
SELECT seatNo
FROM Flights
WHERE fltNo=123 AND fltDate=DATE '2012-12-25'
    AND seatStatus='available';
```

- Customer1 inspects the results and selects a seat, say 22A.

```
UPDATE Flights
    SET seatStatus='occupied', purchaser='Customer1'
    WHERE fltNo=123 AND fltDate= DATE '2012-12-25' AND seatNo='22A';
```

Simple Example of What Could Go Wrong (continued)

- Customer2 is also looking at the same flight on the same day simultaneously and decides to choose seat 22A as well.
- Operations of query and update statements:

<< Draw on Board >>

- Both customers believe that they have reserved seat 22A.
- Problem: Each SQL statement of both users is executed correctly, but the overall result is not correct.
- However, a DBMS can provide the illusion that the actions of Customer1 and Customer2 are executed *serially* (i.e., one at a time, with no overlap).
  - **Serializability**

Another Example of What Could Go Wrong, Even with a Single User

Accounts(acctNo, balance)

- User1 wants to transfer $100 from an account with acctNo=123 to an account with acctNo=456.
  1. **Subtract $100 from the account with acctNo=123**

     UPDATE Accounts

     SET balance = balance – 100

     WHERE acctNo=123;
  2. **Add $100 to the account with acctNo=456**

     UPDATE Accounts

     SET balance = balance + 100

     WHERE acctNo=456;
- What if application or database fails after step 1, but before step 2?

Atomicity

- Failure (e.g., network failure, power failure etc.) could occur after step 1.
  - If this happens, money has been withdrawn from account 123 …
  - … but not not deposited into account 456.

- The DBMS should provide mechanisms to ensure that groups of operations are executed **atomically**.
  - That is, either **all** the operations in the group are executed to completion or **none** of the operations are executed.
  - All-or-nothing, no in-between

Transactions

- A *transaction* is a group of operations that should be executed atomically, all-or-nothing.

- Operations of a transaction can be interleaved with operations of other transactions.

- However, with an “isolation level” called *serializability*, the illusion is given that every transaction is executed one-by-one, in a serial order.
  - The DBMS will execute each transaction in its entirely or not at all, “without transactions interfering with each other”.

Transactions (cont'd)

- START TRANSACTION or BEGIN TRANSACTION (can be implicit)
  - Marks the beginning of a transaction, followed by one or more SQL statements.

- COMMIT
  - Ends the transaction. All changes to the database caused by the SQL statements within the transaction are committed (i.e., they are permanently there--**Durability**) and visible in the database.
  - All changes become visible at once (atomically).
  - Before commit, changes to the database caused by the SQL statements are visible to this transaction, but are not visible to other transactions.

- ROLLBACK
  - Causes the transaction to abort or terminate. Any changes made by SQL statements within the transaction are undone ("rolled back").

Example Using Informal Syntax

BEGIN TRANSACTION

<SQL statement to check whether bank account 123 has >= $100>

If there is no account 123, then ROLLBACK;

If account 123 has < $100, then ROLLBACK;

<SQL statement to withdraw $100 from account 123>

<SQL statement to add $100 to account 456>

If there is no account 456, then ROLLBACK;

COMMIT;

- Scenario 1: Suppose bank account 123 has $50.
- Scenario 2: Bank account 123 has $200, bank account 456 has $400.
- Scenario 3: Bank account 123 has $200, bank account 456 has $400, failure after withdrawing $100 from account 123.
- Scenario 4: Bank account 123 has $200, bank account 456 has $400, failure after depositing $100 to account 456, but before COMMIT

Read-Only Transactions

- In the previous examples, each transaction involved a read, then a write.
- If a transaction has only read operations, it is less likely to impact serializability.

- SET TRANSACTION READ ONLY;
  - Stated *before* the transaction begins.
  - Tells the SQL system that the next transaction is read-only.
  - SQL may take advantage of this knowledge to parallelize many read-only transactions.

- SET TRANSACTION READ WRITE;
  - Tells SQL that the next transaction may write data, in addition to read.
  - Default option if not specified; often (usually) not specified.

Dirty Reads (Read Uncommitted)

- *Dirty data* refers to data that is written by a transaction but has not yet been committed by the transaction.
- A *dirty read* refers to the read of dirty data written by another transaction.

- Consider the following transaction T that transfers an amount of money ($X) from one account to another:

1. Add $X to Account 2.
2. Test if Account 1 has $X.
   a) If there is insufficient money, remove $X from Account 2.
   b) Otherwise, subtract $X from Account 1.

Dirty Reads (cont'd)

- Transaction T1: Transfers $150 from A1 to A2.
- Transaction T2: Transfers $250 from A2 to A3.
- Initially: A1: $100, A2: $200, A3: $300.
- What might be the (unexpected, unwanted) result with Dirty Reads if execution of T1 and T2 happens to interlace in a certain way?

<< To discuss, and write on board >>

Should Transactions Allow Dirty Reads?

- **Allow** Dirty Reads
  - More parallelism between transactions.
  - But may cause serious problems, as previous example shows.

- **Don't Allow** Dirty Reads
  - Less parallelism, more time is spent on waiting for other transactions to commit or rollback.
  - More overhead in the DBMS to prevent dirty reads.
  - Cleaner semantics.

Isolation levels

```
SET TRANSACTION READ WRITE
    ISOLATION LEVEL READ UNCOMMITTED;
```

- First line: The transaction may write data (that's the default).
- Second line: The transaction can run with isolation level "Read Uncommitted", allowing Dirty Reads.

- Default Isolation Level depends on system.
  - Most systems run with READ COMMITTED or SNAPSHOT ISOLATION.

Other Isolation Levels

- SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
  - Only clean (committed) reads, no dirty reads.
  - But you might read data committed by *different* transactions.
    - You might not even get the same value even when you read same data a second time during a single transaction!

- SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
  - Repeated queries of a tuple during a transaction will retrieve the same value, even if its value was changed by another transaction.
    - But *different* data reads might return values that were committed by *different* transactions at different times.
  - Also, a second scan of a range (e.g., salary>10000) may return "phantoms" not originally present in the scan..
    - Phantoms are tuples newly inserted while the transaction is running.

- SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Isolation Levels

Isolation Level	dirty reads	non-repeatable reads	phantoms
READ UNCOMMITTED	Y	Y	Y
READ COMMITTED	N	Y	Y
REPEATABLE READ	N	N	Y
SERIALIZABLE	N	N	N

Snapshot Isolation (SI)

- SI and Read Committed are most commonly used Isolation Levels.
  - Better performance (response time, throughput) than Serializability
- Transaction reads data as it existed when transaction began (repeatable).
  - As usual, transaction also sees its own updates
- Conflicts on Writes are avoided; equivalent of Serializable on Writes.
  - … but not on Read/Write interactions between transactions
- Example: Two transactions that are running under **Serializability** change both A and B. If both Commit, then one ran logically after the other.
- **SI Example**: A is supposed to be less than B. Initially A is 0 and B is 100.
  - T1 reads original A and B values, and changes A to 60.
  - T2 reads original A and B values, and changes B to 20.
  - Transactions T1 and T2 both maintained the consistency condition "A < B" … but what are the final values of A and B?
  - Could this happen with Serializability?