Lecture 6:
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

- Lecture slides and other class information will be posted on Piazza (not Canvas).
  - Slides are posted under Resources→Lectures
  - Lecture Capture recordings are available to all students under Yuja.
    - That includes classes given over Zoom.
  - There's no Lecture Capture for Lab Sections.
- All Lab Sections (and Office Hours) normally meet In-Person, with rare exceptions announced on Piazza.
  - Some slides for Lab Sections have been posted on Piazza under Resources→Lab Section Notes
- Office Hours for TAs and me are posted in Syllabus and Lecture1, as well as in Piazza notice @7; that post also now includes hours for Group Tutors.
- Some suggestions about use (and non-use) of Generative AI systems such as ChatGPT were posted on Piazza in @41, with specific discussion about the Movies tables and the four Avatar queries in Lecture 4.

Important Notices

- Lab2 assignment was posted on Piazza under Resources→Lab2 on Wed, January 24. General Information about appears under Resources-->General Resources.
  - See Piazza announcement about Lab2, which is much more difficult than Lab1.
  - Lab2 is due **Tuesday, February 6**, by 11:59pm.
    - Late Lab Assignments will not be accepted; be sure that you post the correct file!
    - Be sure to do Lab assignments individually, but you can get help on concepts.
  - Your solution should be submitted via Canvas as a zip file.
    - Canvas will be used for both Lab Assignment submission and grading.
  - You will have enough information to complete Lab2 after Lecture on Mon, January 29.
  - A load file to help you test your Lab2 solution was posted under Resources→Lab2
    - We won't provide answers for your Lab2 queries on that test data, but in @55, we told you how many tuples should appear in the results of your query.
    - But testing can only prove that a solution is incorrect, not that it is correct.
  - Using ChatGPT to help understand CSE 180 concepts (or even ask what SQL queries you wrote actually do) is great …
    - … but if you do use systems such at ChatGPT to do Lab Assignments, you're mostly cheating yourself …
    - … because you can't use such systems on the Midterms or Final!
  - Lab2 has been (and will be) discussed at Lab Sections and Tutoring.

Important Notices

- Second Gradiance Assignment, CSE 180 Winter 2024 #2, was assigned on Tuesday, January 30.
  - It is due on Tuesday, February 6, by 11:59pm.
    - The day of the week on which Gradiance Assignments are due will vary.
  - There are 9 questions in this Gradiance Assignment.
    - Some of them may be difficult.
  - You should treat NULL as the **smallest** value for the ORDER BY question in Gradiance #2.
  - Website and Class Token for Gradiance are in the Syllabus and first Lecture.
    - If you forget your Gradiance password, you can recover it by providing your Gradiance login and email.
  - You may take each Gradiance Assignment as many time as you like.
    - Questions stay the same, but answers change.
    - Your Gradiance Assignment score is the score on your **Final** Gradiance submission before the deadline.
    - Gradiance does not allow late submissions, so no extensions are possible.
- Third Gradiance, CSE 180 Winter 2024 #3, will be assigned soon.

Very Important Notice

- The Winter 2024 CSE 180 Final is on Wednesday, March 20 from 8:00am – 11:00am in our usual classroom.
  - The Syllabus and First Lecture originally said that it was on Tuesday, March 19.
  - That's been fixed. Please adjust your calendars accordingly!

Important Notices: Midterm

CSE 180 Midterm is on **Wednesday, February 14,** **and it will be given in-person, except for students who receive Remote Exam permission due to serious illnesses.**

- Usual class time, 65 minutes (extended for DRC students).
  - Hope that all DRC students have submitted their forms for accommodations..
- **No** early/late Exams. **No** make-up Exams. **No** devices.
- **You may bring one double-sided 8.5" x 11" sheet to the Midterm, with anything that you want written or printed on it that you can read unassisted) …**
  - (Okay, you may bring two sheets with writing on only one side of each sheet.)
  - **But you must not use any other material or receive any help during the Midterm, whether you're in the classroom, remote, or in a DRC room**
  - As the Syllabus emphasize, Academic Integrity violations have serious consequences!
- Write your answers on the exam itself, readably, using **either ink or a #2 pencil**.
- We might assign seats to you as you enter the room.

Important Notices: Midterm (continued)

- If you need to take the Midterm Remotely due to serious illness, please send me an email whose subject is "Taking CSE 180 Midterm Remotely" justifying that request.
  - Send that email before 8:00pm on Tuesday, February 13, the day before the Midterm. I'll send you instructions before 10:30am on Wednesday, February 14.
  - Only students whose requests have been approved may take the Midterm Remotely.
- The CSE 180 Midterm from Winter 2023 was posted under Resources→Exams on Piazza on Wednesday, February 1.
  - We haven't covered all the material in that previous Midterm yet.
  - Also, last year's Midterm used that quarter's database schema; your schema is different.
  - Solution to Winter 2023 Midterm will be posted on Wednesday, February 7 … but I strongly suggest that you take it yourself first, rather than just reading the solution.
- **All past Lectures including this one, Lecture 6 (SQL4) will be included on the Midterm.**
  - **Moreover, the first half of Lecture 7 (Views) will be included on the Midterm. But the second half of Lecture 7 (Indexes) won't be.**
- At the end of the Midterm, you'll hand in your Midterm paper, and you'll show your UCSC ID.
  - Do not hand in your 8.5" x 11" sheet.
- **There will be Lab Sections and Tutoring during the rest of the week after the Midterm, on topics including:**
  - Lectures, Lab2 solution, Gradiance, Lab3.
  - Lab3 will be posted on Piazza after the Midterm.

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, DEFAULT and NULL

```
CREATE TABLE MovieStar (
    starName        CHAR(30),
    address         VARCHAR(255)  DEFAULT  'Hollywood',
    gender          CHAR(1) NOT NULL,
    birthdate       DATE
);
```

1. If a new row is inserted into MovieStar, and a value is specified for an attribute, then that attribute will receive that value.
2. If a new row is inserted into MovieStar, and no value is specified for an attribute, and that attribute has a DEFAULT (e.g., for address), then the value for that attribute will be that DEFAULT, which is 'Hollywood' for address.
3. If a new row is inserted into MovieStar, and no value is entered for an attribute, and the value of that attribute does not have a DEFAULT, and the value of that attribute is allowed to be NULL (e.g.,for birthdate), then the value for the attribute will be NULL.
4. If a new row is inserted into MovieStar, and no value is entered for an attribute, and the value of the attribute does not have a DEFAULT, and the value is not allowed to be NULL (e.g., for gender), then an error will be raised.

INSERT Statement with Subquery

Movies(MovieTitle, MovieYear, length, genre, studioName, producerC#)

Studio(studioName, address, presC#)

(Temporarily, assume that there isn't Referential Integrity for studioName.)

```
INSERT INTO Studio(studioName)
    SELECT DISTINCT m.studioName
    FROM Movies m
    WHERE m.studioName NOT IN
            (SELECT st.studioName
             FROM Studio st);
```

Why DISTINCT?

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

INSERT Statement with Subquery and Constants

Movies(movieTitle, movieYear, length, genre, studioName, producerC#)

Disney2018Movies(movieTitle, length)

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

Puts the Disney2018Movies into the Movies table, setting movieYear 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 all tuples from StarsIn where the name of the star was 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.movieTitle = s.movieTitle AND m.movieYear = 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 execName = 'Pres. ' || execName

WHERE cert# IN (SELECT presC# FROM Studio);

- 2nd line: concatenates the string 'Pres. ' with execName.

Another UPDATE with Subquery
(which might help you do Lab3)

- The UPDATE clause specifies the table being updated (just one table).
- A SQL UPDATE statement can also have a FROM clause.
- The UPDATE statement's WHERE clause can refer to both the updated table and the table(s) in the FROM clause.
- The UPDATE statement's SET clause can also refer to the updated table and the table(s) in the FROM clause, but attributes that are SET in the updated table must appear without the tuple variable.

Tickets(airlineID, flightNum, ticketID, custID, seatNum, paid)
NewTickets(airlineID, flightNum, ticketID, custID, seatNum);

```
UPDATE Tickets t
SET seatNum = newt.seatNum, paid = FALSE
FROM NewTickets newt
WHERE t.ticketID = newt.ticketID
    AND t.custID = newt.custID
    AND t.airlineID = newt.airlineID
    AND t.flightNum = newt.flightNum;
```

You may use tuple variables in the UPDATE, FROM and WHERE clauses of an UPDATE statement … but attributes that are SET in the updated table must appear without the tuple variable.

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. 6M is clearer (but it's not legal SQL).

MovieExec

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

Transactions in SQL

Instructor: Shel Finkelstein

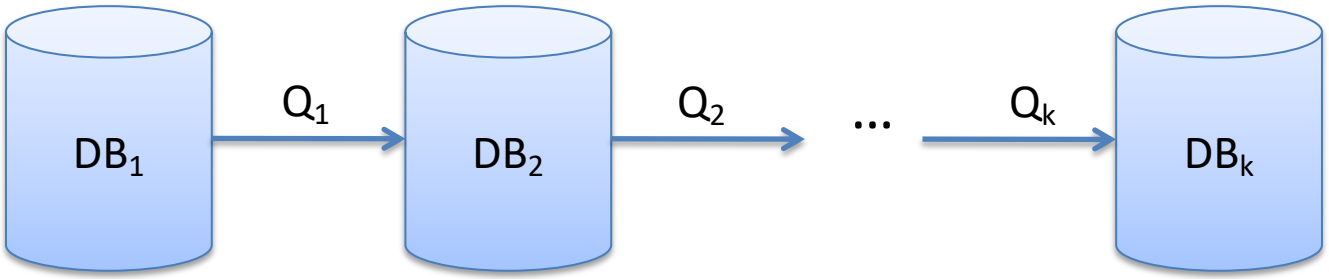
Transactions Reference:

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;

What happens in each of the following Scenarios?

- 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 occurs after withdrawing $100 from account 123.
- Scenario 4: Bank account 123 has $200, bank account 456 has $400, failure occurs 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.

- BEGIN TRANSACTION READ ONLY;
  - Tells the SQL system that the transaction is read-only.
  - SQL may take advantage of this knowledge to parallelize many read-only transactions.

- BEGIN TRANSACTION READ WRITE;
  - Tells SQL that the 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 exists and has at least $X in it.
   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.
- Initial values: A1: $100, A2: $200, A3: $300.
- What might be the (unexpected, undesireable) 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

```
BEGIN 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 REPEATABLE READ as their default isolation level.

- PostgreSQL accepts READ UNCOMMITTED, but when that's specified, it actually runs with Isolation Level READ COMMITTED.
  - Most other systems run READ UNCOMMITTED as READ UNCOMMITTED.

Other Isolation Levels

- BEGIN 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!

- BEGIN 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 reads of *different* data might return values that were committed by *different* transactions at *different* times.
  - Also, a second scan of a range (e.g., salary > 10000) might return "phantoms" not originally present in the scan.
    - Phantoms are new tuples inserted while the transaction was running.

- BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

What Do Isolation Levels Guarantee?

Isolation Level	Clean Reads	Repeatable Reads	Simultaneous Existence
READ UNCOMMITTED	N	N	N
READ COMMITTED	Y	N	N
REPEATABLE READ	Y	Y	N
SERIALIZABLE	Y	Y	Y

Simultaneous Existence of Data: With Repeatable Read (but without Serializability):

- Transaction execution might not be equivalent to a Serial execution schedule.
- For example, multiple data items read by a transaction might never have existed simultaneously in the database!

And then there's the **Phantom Problem**

Snapshot Isolation (SI)

- Read Committed, Repeatable Read and Snapshot Isolation are the most common Isolation Levels.
  - Better performance (response time, throughput) than Serializability, but worse Consistency.

- With SI, transaction reads data as it existed when transaction began (hence reads are repeatable).
  - However, transaction sees its own updates, as in Serializability.
- Conflicts on Writes are avoided; equivalent of Serializable on Writes …
  - … but not on Read/Write conflicts between transactions.

- If two transactions run with Isolation Level **Serializability**, and both Commit, then logically, one executed 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?