Database modifications

Instructor: Wang-Chiew Tan

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

Database modifications

- SQL statements for
  - *Inserting* tuples into a relation.
  - *Deleting* certain tuples from a relation.
  - *Update* values of certain components of certain existing tuples.
- Inserting, deleting, and updating are referred to as *modifications* operations.
- These operations do not return a result. They change the *state* of the database.

Insert operation

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 are created 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');

Movies(title, year, length, genre, studioName, producerC#)
Studio(name, address, presC#)

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

- Add to the relation all studioNames such that studioName does not occur in the list of names given by the Studio relation.

- The query must be completely evaluated before insertion occurs.
  - Why?
  - The SELECT clause of the previous query must be evaluated before tuples are inserted into Studio.
  - No side effect: Otherwise, new tuples added to Studio may affect the evaluation of the SELECT clause.

Deletions

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.

More examples

DELETE FROM MovieExec
WHERE netWorth < 10000000;

- Deletes all movie executives whose net worth is low, i.e., less than 10 million dollars.

Updates

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.

Key and Foreign Key Constraints

Instructor: Wang-Chiew Tan

Reference:
A First Course in Database Systems,
3rd edition, Chapter 7-7.3

Key and Foreign Key Constraints

- Declaring key and foreign key constraints:

Studio(name, address, presC#)
MovieExec(name, address, cert#, netWorth)

Assume cert# is the key of MovieExec relation. That is,

```
CREATE TABLE MoveExec(
  name VARCHAR(30),
  address VARCHAR(50),
  cert# INTEGER PRIMARY KEY,
  netWorth  INTEGER
);
```

Declaring Foreign Key Constraint

- Suppose presC# of Studio references cert# of MoveExec.
- For every tuple t of Studio, the value t.presC# occurs among the cert# of tuples in MovieExec. That is,

  S1: select distinct presC# from Studio;

  S2: select cert# from MovieExec;

  S1 is always contained in S2.

Declaring Foreign Key Constraint (cont'd)

```
CREATE TABLE Studio (
  name CHAR(30) PRIMARY KEY
  address VARCHAR(255)
  presC#   INTEGER,
  FOREIGN KEY (presC#) REFERENCES MovieExec(cert#)
);
```

General Syntax:

FOREIGN KEY (<*attributes*>) **REFERENCES** <*table*>(<*attributes*>)

Maintaining Referential Integrity

- What can go wrong?

  S1: select distinct presC# from Studio;

  S2: select cert# from MovieExec;

1) A new Studio tuple s is inserted and s.presC# does not occur in S2.
2) An existing Studio tuple s is updated and the new t.presC# does not occur in S2.
3) Delete a MovieExec tuple m and m.cert# occurs in S1.
4) Update a MovieExec tuple m and m.cert# occurs in S1.

In all cases above, the actions led to the violation of the invariant:

S1 is contained in S2.

- Actions 1 and 2:
  - The insertion (and resp. update) is rejected.

- Actions 3 and 4: Three options:
  - **a) The Default Policy**: Reject actions that violate the invariant.
  - **b) The Cascade Policy**: Changes to the referenced relation will be mimicked by the referencing relation.
    - Delete MovieExec tuple m => all tuples in Studio with presC# = m.cert# are deleted.
    - Update MovieExec tuple on m.cert# from value A to value B. All tuples in Studio with presC#=A will be modified to B.
  - **c) The Set-Null Policy**: Changes to the referenced relation will cause attributes of referencing relation to be set to NULL.
    - Delete MovieExec tuple m => all tuples in Studio with presC# = m.cert# are set to NULL.
    - Update MovieExec tuple on m.cert# from value A to value B. All tuples in Studio with presC#=A will be set to NULL.

```
CREATE TABLE Studio (
  name CHAR(30) PRIMARY KEY
  address VARCHAR(255)
  presC#   INTEGER,
  FOREIGN KEY (presC#) REFERENCES MovieExec(cert#)
    ON DELETE SET NULL
    ON UPDATE CASCADE
);
```

ON UPDATE SET NULL
ON DELETE CASCADE

Views and Indexes

Instructor: Wang-Chiew Tan

Reference:
A First Course in Database Systems,
3rd edition, Chapter 8-8.3

Virtual views

- Relations that are defined through the CREATE TABLE command have a physical footprint.
- Virtual views (or simply views) do not have a physical footprint. Defined by a query. However, views can be queried just like they are actual relations.

CREATE VIEW <*view-name*> **AS** <*view-definition*>;

Views

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

```
CREATE VIEW ParamountMovies AS
  SELECT title, year
  FROM Movies
  WHERE studioName = 'Paramount';
```

ParamountMovies(title, year)

Views (cont'd)

- Movies(title, year, length, genre, studioName, producerC#)
- MovieExec(name, address, cert#, netWorth)

```
CREATE VIEW MovieProd AS
  SELECT title, name
  FROM Movies, MovieExec
  WHERE producerC#=cert#;
```

MovieProd(title, name)

```
CREATE VIEW MovieProd(movieTitle, prodName) AS
  SELECT title, name
  FROM Movies, MovieExec
  WHERE producerC#=cert#;
```

Renames title and name to movieTitle and, resp. prodName.

Querying views

```
SELECT title
FROM ParamountMovies
WHERE year = 1979;
```

```
SELECT DISTINCT starName
FROM ParamountMovies, StarsIN
WHERE title = movieTitle AND year = movieYear;
```

Indexes in SQL

- An index on an attribute A is a data structure that makes it efficient to search tuples based on values of attribute A.
- An index on the year attribute will help find tuples with year < 2000 or year >= 1998 efficiently.

  CREATE INDEX YrIdx **ON** Movies(Year);

- Focus: implement indexes on large relations is of central importance to the implementation of a DBMS.
  - A central data structure for indexes is that of B+-trees.

Motivation for Indexes

SELECT *
FROM Movies
WHERE studioName='Disney' AND year=1990;

Naïve method:
Obtain the answers by scanning through the entire Movies relation.

If Movies is a big relation (e.g., millions of tuples) of which only 10 were made by Disney in 1990, then the naïve method is clearly very expensive.

If an index on year exists, then those 10 tuples can be easily retrieved.

Another example

SELECT name
FROM Movies, MovieExec
WHERE title=`Star Wars' AND producerC#=cert#;

Suppose there is an index on the title of Movies => all movies with title=`Star Wars' can be efficiently retrieved.

Suppose there is also an index on cert# of MovieExec => use this index to find the producerC# from the above result (i.e., movies with title=`Star Wars').

Note: no unnecessary tuples are accessed with these two indexes available.

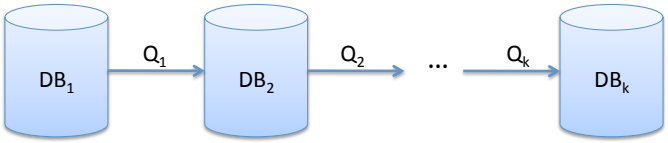
Transactions in SQL

Instructor: Wang-Chiew Tan

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

One-at-a-time semantics

- So far, we have learnt how to query or modify the database.
- SQL statements posed to the database system are executed one at a time.



Serializability

- Applications such as web services, banking, airline reservations demand high throughput on operations performed on the database.
  - Manage hundreds of sales transactions every second.
- Possible for two operations to simultaneously affect the same bank account of flight.
- These “simultaneous” operations must be handled with care.

Example of what could go wrong

Flights(fltNo, fltDate, seatNo, seatStatus)

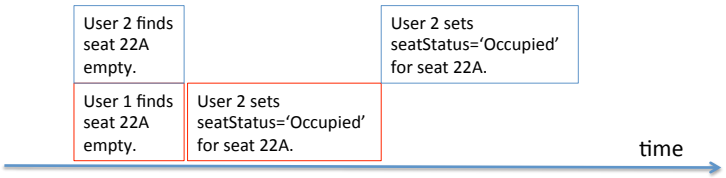
- User 1 issues the following query through a web interface.

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

- User 1 inspects the results and selects the seat, say 22A, through the web interface.

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

- User 2 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.



- Both believed that they have reserved seat 22A.
- Problem: SQL statements of both users are executed correctly but the global result is not correct.
- DBMS need to provide the illusion that the actions of user 1 and user 2 are executed *serially* (i.e., one at a time, with no overlap).

Another example of what could go wrong even with a single user

Accounts(acctNo, balance)

- User 1 transfers $100 from an account with acctNo=123 to an account with acctNo=456.
  1. Add $100 to account with acctNo=456.

     UPDATE Accounts

     SET balance = balance + 100

     WHERE acctNo=456;
  2. Subtract $100 from the account with acctNo=123.

     UPDATE Accounts

     SET balance = balance – 100

     WHERE acctNo=123;

Atomicity

- failure (e.g., network failure, power failure etc.) could occur after step 1.
  - If this happens, money has been deposited into account 456 but not withdrawn from account 123.
- The DBMS should provide a mechanism to ensure that groups of operations must be executed *atomically*.
  - That is, either all the operations in the group are executed to completion or none of the operations are executed.
  - Nothing in between.

Transactions

- A *transaction* is a group of operations that must be executed atomically.
- Operations of a transaction can be interleaved with operations of other transactions.
- However, by default, every transaction is serialized. The DBMS will execute a transaction in its entirety.

Transactions (cont'd)

- START TRANSACTION
  - Marks the beginning of a transaction, followed by one or more SQL statements.
- COMMIT
  - Ends the transaction. Changes to the database caused by the SQL statements within the transaction are committed (i.e., installed permanently) in the database.
  - Before commit, changes to the database caused by the SQL statements are not visible to other transactions.
- ROLLBACK
  - Causes the transaction to abort or terminate. I.e., any changes caused by the SQL statements within the transaction are undone or rolled back.

Example

BEGIN TRANSACTION
<SQL statement to check whether bank account 123 has >= $100>
If account 123 has <$100, ROLLBACK;
<SQL statement to add $100 to account 456>
<SQL statement to withdraw $100 from account 123>
COMMIT;

- Scenario 1: Suppose bank account 123 has $50.
- Scenario 2: Bank account 123 has $200, bank account 456 has $300.
- Scenario 3: Bank account 123 has $200, bank account 456 has $300, network failure after 2nd SQL statement.
- Scenario 4: Bank account 123 has $200, bank account 456 has $300, network failure after 3rd SQL statement 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 have serializability problems.

- 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.

Dirty Reads

- *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 that transfers money from account A to account B.

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

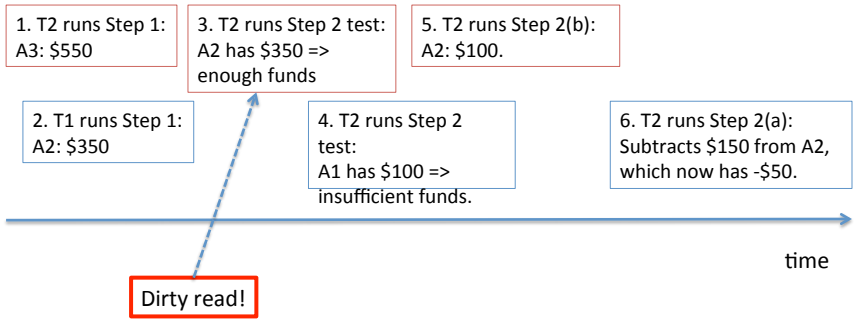
Dirty Reads (cont'd)

- Transaction T1: Transfers $150 from A1 to A2.
- Transaction T2: Transfers $250 from A2 to A3.
- A1: $100, A2: $200, A3: $300.

To fill in

Dirty Reads (cont'd)

- Transaction T1: Transfers $150 from A1 to A2.
- Transaction T2: Transfers $250 from A2 to A3.
- A1: $100, A2: $200, A3: $300.



- Allow dirty reads
  - More parallelism between transactions.
  - But may cause serious problems as previous example shows.
- Don't allow dirty reads
  - More overhead in the DBMS to prevent dirty reads.
  - Less parallelism, more time is spent on waiting for other transactions to commit or rollback.

Isolation levels

SET TRANSACTION READ WRITE
ISOLATION LEVEL READ UNCOMMITTED;

- First line: the transaction may write data.
- Second line: the transaction may run with isolation level "read uncommitted".
  - I.e., dirty reads are allowed.

Other isolation levels

- SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
  - Only clean reads, no dirty reads.
- SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
  - Repeated queries will retrieve the same answers (and *phantom* tuples), even though values may have changed in the meantime.
  - Phantom tuples are tuples that are newly inserted while the transaction is running.
- SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

END