Extension to SQL: View, Triggers, Transaction

SS Chung
Views (Virtual Tables) in SQL

- Concept of a view in SQL
  - Single table derived from other tables
  - Considered to be a virtual table
Specification of Views in SQL

- **CREATE VIEW** command
  - Give Table (View) name, list of attribute names for the View (table), and a Query to specify the contents of the View

```
V1:  CREATE VIEW WORKS_ON1
     AS SELECT Fname, Lname, Pname, Hours
     FROM EMPLOYEE, PROJECT, WORKS_ON
     WHERE Ssn=Essn AND Pno=Pnumber;

V2:  CREATE VIEW DEPT_INFO(Dept_name, No_of_emps, Total_sal)
     AS SELECT Dname, COUNT (*), SUM (Salary)
     FROM DEPARTMENT, EMPLOYEE
     WHERE Dnumber=Dno
     GROUP BY Dname;
```
Altering Existing View

**Replace View** *Existing_View_Name* As

Select… From… Where…;

**Replace View** *Works_On1* As

Select Fname, Lname, Pno, Pname, Hours
From Employee, Works_On, Project
Where Ssn = Essn and Pno = Pnumber
Specification of Views in SQL (cont’d.)

- Specify SQL queries on a view
- View always up-to-date
  - Responsibility of the DBMS and not the user
- DROP VIEW command
  - Dispose of a view
Views

Students (sid, name, address, gpa)
Completed ( sid, course, grade)

A view is a query stored in the database
- Think of it as a table definition for future use

Example view definition:
CREATE VIEW GStudents AS
SELECT *
FROM Students
WHERE gpa >= 2.5

Views can be used like base tables, in any query or in any other view. Like a Macro. Different from Insert INTO.
Example view use: simpler queries

Suppose you want to retrieve good students who have completed CIS430.

**GStudents** (sid, name, address, gpa)

```sql
SELECT S.name, S.phone
FROM Gstudents S
INNER JOIN Completed C
ON S.sid = C.sid
WHERE C.course = 'CIS430';
```

It’s easier to write the query using the view.
View Processing

- View is a **virtual** table
- **How a view is defined:**
  
  ```sql
  CREATE VIEW V_ATL-FLT
  AS SELECT FLT#, AIRLINE, PRICE
  FROM FLT-SCHEDULE
  WHERE FROM-AIRPORTCODE = “ATL”;
  
  SELECT *
  FROM V_ATL-FLT
  WHERE PRICE <= 00200.00;
  
  SELECT FLT#, AIRLINE, PRICE
  FROM FLT-SCHEDULE
  WHERE FROM-AIRPORTCODE=“ATL”
  AND PRICE<00200.00;
  
  DROP VIEW ATL-FLT [RESTRICT|CASCADE];
  ```
Views for Security

This is the student table without the gpa field to hide gpa to any user in DB.

CREATE VIEW Sstudents AS
SELECT sid, name, address
FROM students
Views for Extensibility

- An old company’s database includes a table:
  Part (PartID, Name, Weight)
- Weight is stored in pounds
- The company is purchased by a new firm that uses metric weights
- The two databases, old and new, must be integrated and use Kg.
- But there’s lots of old software using pounds.
- Solution: views!
Views for extensibility (ctd)

Solution:

1. Base table with kilograms: NewPart for new integrated company

2. CREATE VIEW Part AS
   SELECT PartID, Name, 2.2046*Weight
   FROM NewPart;

3. Old programs still call the table “Part”
View Update and Inline Views

Clause WITH CHECK OPTION
- Must be added at the end of the view definition if a view is to be updated

In-line view (dynamic Table)
- Defined in the FROM clause of an SQL query
View Implementation, View Update, and Inline Views

- Complex problem of efficiently implementing a view for querying
- **Query modification** approach
  - Modify view query into a query on underlying base tables
  - Disadvantage: inefficient for views defined via complex queries that are time-consuming to execute
View Implementation

- **View materialization approach**
  - Physically create a temporary view table when the view is first queried
  - Keep that table on the assumption that other queries on the view will follow
  - Requires efficient strategy for automatically updating the view table when the base tables are updated
View Implementation (cont’d.)

- **Incremental update strategies**
  - DBMS determines what new tuples must be inserted, deleted, or modified in a materialized view table
Problem with views: update

- Views cannot always be updated unambiguously. Consider:

Emp (empid, ename, address, deptid)
Dept (deptid, dname)

CREATE VIEW EMPDEPT AS
SELECT ename, dname
FROM Emp INNERJOIN Dept ON Emp.deptid=Dept.deptid;

<table>
<thead>
<tr>
<th>EMPDEPT</th>
<th>ename</th>
<th>dname</th>
</tr>
</thead>
<tbody>
<tr>
<td>jim</td>
<td>shoe</td>
<td></td>
</tr>
<tr>
<td>joe</td>
<td>suit</td>
<td></td>
</tr>
</tbody>
</table>

I want to delete (jim, shoe) from EMPDEPT.
Can I do that?
View Update

View can be updated if

- It is defined on a single base table
- Using only Selection and Projection
- No Aggregates
- No DISTINCT
View Update and Inline Views

- Update on a view defined on a single table without any aggregate functions
  - Can be mapped to an update on underlying base table
- View involving joins can NOT be updated
  - Often not possible for DBMS to determine which of the updates is intended
Levels of Abstraction

- Physical Schema
  - ES 1
  - ES 2
  - ES 3

- Conceptual Schema

- Physical Schema
  - External view; user and data designer
  - Logical storage; data designer
  - Physical storage; DBA
The physical schema is a description of how the data is physically stored in the database. It includes:

- Where the data is located
- File structures
- Access methods
- Indexes

The physical schema is managed by the DBA.
Conceptual Schema

- The **conceptual schema**
  - a logical description of how the data is stored.
  - It consists of the schemas we have described with `CREATE TABLE` statements.

- It is managed by the data designer.
Each external schema is a combination of base tables and views, tailored to the needs of a single user. It is managed by the data designer and the user.
Data Independence

A database model possesses *data independence* if application programs are immune to changes in the conceptual and physical schemas.

Why is this important? Everything changes.

How does the relational model achieve logical (conceptual) data independence?

– Through views

– If the conceptual schema changes, a view can be defined to preserve existing applications
How does the relational model achieve physical data independence?

1. Conceptual level contains no physical info
2. SQL can program against the conceptual level
   - Earlier DBMSs (network, hierarchical) did not have these properties.
     - Their languages had physical properties embedded in them.

That is the primary reason for the success of the relational model.
Views: Summary

- A view is a stored query definition.
- Views can be very useful:
  - Easier query writing, security, extensibility.
- But views cannot be unambiguously updated.
- Three levels of abstraction in a relational DBMS:
  - Yields data independence: logical and physical.
Views vs Tables

<table>
<thead>
<tr>
<th>Creating</th>
<th>Create view V as (select * from A, B where ...)</th>
<th>Create table T as (select * from A, B where ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can be used</td>
<td>In any select query. Only some update queries.</td>
<td>It's a new table. You can do what you want.</td>
</tr>
<tr>
<td>Maintained as</td>
<td>1. Evaluate the query and store it on disk as if a table. 2. Don't store. Replace it with the query when referenced.</td>
<td>It's a new table. Stored on disk.</td>
</tr>
<tr>
<td>What if a tuple inserted in A?</td>
<td>1. If V is stored on disk, the stored table is automatically updated to be accurate. 2. If we are just replacing V with the query, there is no need to do anything.</td>
<td>T is a separate table; there is no reason why DBMS should keep it updated. If you want that, you must define a trigger to update T whenever A is updated.</td>
</tr>
</tbody>
</table>
Views vs Tables

- Views strictly supercede “create a table and define a trigger to keep it updated”

- Two main reasons for using them:
  - Security/authorization
  - Ease of writing queries
    - E.g. Collaborators table if you were asked to write a lot of queries about it.
    - The way we are doing it, the collaborators table is an instance of “creating table”, and not “creating view”
    - Creating a view might have been better.

- Perhaps the only reason to create a table is to force the DBMS to choose the option of “materializing”
  - That has efficiency advantages in some cases
  - Especially if the underlying tables don’t change
Specifying Constraints as Assertions and Actions as Triggers

- **CREATE ASSERTION**
  - Specify additional types of constraints outside scope of built-in relational model constraints

- **CREATE TRIGGER**
  - Specify automatic actions that database system will perform when certain events and conditions occur
Specifying General Constraints as Assertions in SQL

- **CREATE ASSERTION**
  - Specify a query that selects any tuples that violate the desired condition
  - Use only in cases where it is not possible to use `CHECK` on attributes and domains

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS ( SELECT * FROM EMPLOYEE E, EMPLOYEE M, DEPARTMENT D WHERE E.Salary>M.Salary AND E.Dno=D.Dnumber AND D.Mgr_ssn=M.Ssn ) );
```
CREATE TABLE Sailors
(seid INTEGER,
sname CHAR(10),
rating INTEGER,
age REAL,
PRIMARY KEY (sid),
CHECK
((SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT (B.bid) FROM Boats B) < 100 )

CREATE ASSERTION smallClub
CHECK
((SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT (B.bid) FROM Boats B) < 100 )
Assertions: Check over several tables

Employee( id, name, address, mgr, salary )

Any logical expression involving an SQL statement can be used to constrain tables in the database.

CREATE ASSERTION Salary_Mgr
CHECK (NOT EXISTS
  (SELECT *
   FROM Employee E,
   Employee M
   WHERE E.salary >
   M.salary AND
   E.mgr = M.id ))
Introduction to Triggers in SQL

- **CREATE TRIGGER** statement
  - Used to monitor the database
- Typical trigger has three components:
  - Event(s)
  - Condition
  - Action
Triggers (Active database)

- **Trigger**: A procedure that starts automatically if specified changes occur to the DBMS
  - Analog to a "daemon" that monitors a database for certain events to occur
- Three parts:
  - **Event**: activates the trigger
  - **Condition**: tests whether the triggers should run [Optional]
  - **Action**: what happens if the trigger runs
- **Semantics**:
  - When event occurs, and condition is satisfied, the action is performed.
Triggers (Not a constraint)

Three parts:

- **Event** (activates the trigger)
  - This will be an *insert, delete and/or update* to a table

- **Condition** (tests whether the triggers should run)
  - A *logical statement or a query*

- **Action** (what happens if the trigger runs)
  - Can execute queries, execute data-definition commands, transaction-oriented commands, and host-language procedures

**When** does the Action execute?

- Specified with **Event (BEFORE, AFTER)**
Event-Condition-Action (ECA)

- **Event** occurs in databases
  - e.g. addition of a new row, deletion of a row
- **Conditions** are checked
  - e.g. Is batch complete? Has student passed?
- **Actions** are executed if conditions are satisfied
  - e.g. send batch to supplier, congratulate student
Triggers – Event, Condition, Action

- **Events** could be:
  
  BEFORE | AFTER | INSERT | UPDATE | DELETE ON <tableName>
  
  e.g.: BEFORE INSERT ON Manager

- **Condition** is SQL expression or even an SQL query (query with non-empty result means TRUE)

- **Action** can be many different choices:
  
  - SQL statements
  - DDL and transaction-oriented statements like “commit”.
Trigger Syntax

```
CREATE TRIGGER <triggerName>
BEFORE|AFTER INSERT|DELETE|UPDATE
[OF <columnList>] ON <tableName>|<viewName>
[REFERENCING [OLD AS <oldName>]
  [NEW AS <newName>]]
[FOR EACH ROW]
( OR [FOR EACH STATEMENT] by default)
[WHEN (<condition>)]
Begin
<Trigger Action body>
End;
```
Syntax for creating triggers in SQL

- **Trigger name** - unique within one database schema
- **Timing** - depends on the order of controlled events (before or after or instead of)
- **Triggering event** - event which fires the trigger (E)
- **Filtering condition** - checked when the triggering event occurs (C)
- **Target** - table (or view) against which the trigger is fired; they should be both created within the same schema
- **Trigger Parameters** - parameters used to denote the record columns; preceded by colon
  - :new, :old for new and old versions of the values respectively
- **Trigger action** - SQL statements, executed when the trigger fires; surrounded by **Begin ... End** (A)
Syntax for Creating Statement Triggers

CREATE [OR REPLACE] TRIGGER trigger_name
  timing event1 [OR event2 OR event3]
  ON table_name
BEGIN
  SQL statements;
END;

The trigger body consisting of SQL statements will be executed only once according to the prescribed timing, when the event1 (event2, event3) occurs against the monitored table in question table_name.
Triggers: Example*

Assume one donation has been inserted to indiv, for simplicity

CREATE TRIGGER MYTRIG1
BEFORE INSERT ON Account
FOR EACH STATEMENT (---is default)
BEGIN
  IF (TO_CHAR (SYSDATE, 'dy') IN ('Sat', 'Sun'))
      OR TO_CHAR (SYSDATE, 'hh24:mi')
      NOT BETWEEN '08:00' AND '17:00')
  THEN
    RAISE_APPLICATION_ERROR(-20500, 'Cannot create new account now !!');
  END IF;
END;
Triggers vs Constraints

- Triggers are harder to understand
  - If multiple triggers apply, their order of execution is unspecified
  - One trigger can activate another, causing a chain of actions
  - A trigger can activate itself

- Triggers are more powerful than constraints
  - They can make changes to tables
  - They can take action before or after a data modification
Variations in DBMS
Example: Triggers in SQL Server

- An SQL Server **trigger** is a T/SQl procedure that is invoked when a specified database activity occurs.
- Triggers can be used to:
  - Enforce business rules
  - Set complex default values
  - Update views
  - Implement referential integrity actions
- SQL Server only supports **INSTEAD OF** and **AFTER** triggers:
  - A table may have one or more AFTER triggers.
  - AFTER triggers may **not** be assigned to views.
  - A view or table may have only one INSTEAD OF trigger for each triggering action.
- Triggers can roll back the transactions that caused them to be fired.
Types of SQL Triggers

How many times should the trigger body execute when the triggering event takes place?

- **Per statement:** the trigger body executes once for the triggering event. This is the default.
- **For each row:** the trigger body executes once for each row affected by the triggering event.

When the trigger can be fired

- Relative to the execution of an SQL DML statement (before or after or instead of it)
- Exactly in a situation depending on specific system resources (e.g. signal from the system clock, expiring timer, exhausting memory)
Example: Registering Operations

SQL> CREATE TRIGGER increase_salary_trg
2    BEFORE UPDATE OF sal
3    ON emp
4    BEGIN
5       INSERT INTO sal_hist(increased, changedOn)
6       VALUES ('YES', SYSDATE);
7    END;
8   /

Trigger name:     increase_salary_trg
Timing:           BEFORE executing the statement
Triggering event: UPDATE of sal column
Target:           emp table
Trigger action:   INSERT values INTO sal_hist table
Another Trigger Example

CREATE TRIGGER YoungSailorUpdate
AFTER INSERT ON SAILORS
REFERENCING NEW TABLE AS NewSailors
FOR EACH STATEMENT

INSERT INTO YoungSailors (sid, name, age, rating)
SELECT sid, name, age, rating
FROM NewSailors N
WHERE N.age <= 18
Syntax for Creating Row Triggers

CREATE [OR REPLACE] TRIGGER trigger_name
timing event1 [OR event2 OR event3]
ON table_name
[REFERENCING OLD AS old / NEW AS new]
FOR EACH ROW
[WHEN condition]
BEGIN
  SQL statements;
END

The trigger body consisting of SQL statements will be executed once for each row affected by event1 (event2, event3) in the table named table_name subject to the additional condition.
Example Trigger

Assume our DB has a relation schema:
Manager (Num, Name, salary)

We want to write a trigger that:
Ensures that any new manager inserted has:
salary >= 60000
Example Trigger

CREATE TRIGGER minSalary
BEFORE INSERT ON MANAGER
  for what context ?
BEGIN

  check for violation here

END;
Example Trigger

CREATE TRIGGER minSalary
BEFORE INSERT ON Manager
FOR EACH ROW
BEGIN

Check Violation of Minimum Manager Salary

END;
CREATE TRIGGER minSalary
BEFORE INSERT ON Manager
FOR EACH ROW
BEGIN
    IF (:new.salary < 60000)
    THEN RAISE_APPLICATION_ERROR (-20004, 'Violation of Minimum Manager Salary');
    END IF;
END;
END;
Details of Trigger Example

- **BEFORE INSERT ON** Manager
  - This trigger is checked before the tuple is inserted in the Target Table Manager

- **FOR EACH ROW**
  - specifies that trigger is performed for each row inserted

- **:new**
  - :new variable refers to the new tuple inserted

- **If (:new.salary < 60000)**
  - then an application error is raised and hence the row is not inserted; otherwise the row is inserted.
Example trigger

CREATE TRIGGER minSalary
BEFORE INSERT ON Manager
FOR EACH ROW
DECLARE temp int;    -- dummy variable for example
BEGIN
    IF (:new.salary < 60000) THEN
        RAISE_APPLICATION_ERROR (-20004, 'Violation of Minimum Manager Salary');
    END IF;
    temp := 10;        -- to illustrate declared variables
END;
Example Trigger Using Condition

```sql
CREATE TRIGGER minSalary
BEFORE INSERT ON Manager
FOR EACH ROW
WHEN (new.salary < 60000)
BEGIN
    RAISE_APPLICATION_ERROR (-20004, 'Violation of Minimum Manager Salary');
END;
```

- Conditions can refer to **old/new values of tuples** modified by the statement (Event) activating the trigger.
Triggers: REFERENCING

CREATE TRIGGER minSalary
BEFORE INSERT ON Manager
REFERENCING NEW as newTuple
FOR EACH ROW
WHEN (newTuple.salary < 60000)
BEGIN
  RAISE_APPLICATION_ERROR (-20004, 'Violation of Minimum Manager Salary');
END;
Details of Trigger Example

- **BEFORE UPDATE ON** `Emp`
  - This trigger is checked before the tuple is updated in the **Target Table** `Emp`

- **FOR EACH ROW**
  - specifies that trigger is performed for each row inserted

- **:new, :old**
  - **:new variable** refers to the **new tuple after update**
  - **:old variable** refers to the **old tuple before update**

- **If**: (`:new.salary < :old.salary`)
  - then an application error is raised and hence the row is not updated; otherwise the row is updated.
Example Trigger

Ensure that salary does not decrease

CREATE TRIGGER minSalary
BEFORE UPDATE ON Manager
REFERENCING OLD AS oldTuple
    NEW as newTuple
FOR EACH ROW WHEN (newTuple.salary < oldTuple.salary)
BEGIN
    RAISE_APPLICATION_ERROR (-20004, 'Salary Decreasing !!');
END;
Example Trigger

Ensure that salary does not decrease

CREATE TRIGGER minSalary
BEFORE UPDATE ON Emp
FOR EACH ROW

BEGIN
  IF (:new.salary < :old.salary) THEN
    RAISE_APPLICATION_ERROR (-20004, 'Violation of Salary Increase');
  END IF;
END IF;
END;
Row vs Statement Level Trigger

- **Row level**: activated once per modified tuple
- **Statement level**: activate once per SQL statement

- **Row level** triggers can access new data, statement level triggers cannot always do that (depends on DBMS).
- **Statement level** triggers will be more efficient if we do not need to make row-specific decisions
Event for Statement and Row Triggers

Example 1: Monitoring Statement Events

```
SQL> INSERT INTO dept (deptno, dname, loc)
2  VALUES (50, 'EDUCATION', 'NEW YORK');
```

Execute only once per event even if multiple rows affected

Example 2: Monitoring Row Events

```
SQL> UPDATE emp
2  SET sal = sal * 1.1
3  WHERE deptno = 30;
```

Execute for each row of the table affected by the event
Firing Sequence of Database Triggers on a Single Row

**DEPT table**

<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>DNAME</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
</tbody>
</table>

**Trigger Sequence**

- **BEFORE Statement trigger**
- **BEFORE Row trigger**
- **AFTER Row trigger**
- **AFTER Statement trigger**
Firing Sequence of Database Triggers on Multiple Rows

EMP table

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>7839</td>
<td>KING</td>
<td>30</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>30</td>
</tr>
<tr>
<td>7788</td>
<td>SMITH</td>
<td>30</td>
</tr>
</tbody>
</table>

- BEFORE statement trigger
- BEFORE row trigger
- AFTER row trigger
- BEFORE row trigger
- AFTER row trigger
- BEFORE row trigger
- AFTER row trigger
- AFTER statement trigger
Example: Calculating Derived Columns

```sql
SQL> CREATE OR REPLACE TRIGGER derive_commission_trg
2   BEFORE UPDATE OF sal ON emp
3   FOR EACH ROW
4   WHEN (new.job = 'SALESMAN')
5   BEGIN
6     :new.comm := :old.comm * (:new.sal/:old.sal);
7   END;
8 /
```

**Trigger name:** derive_commission_trg  
**Timing:** BEFORE executing the statement  
**Triggering event:** UPDATE of sal column  
**Filtering condition:** job = ‘SALESMAN’  
**Target:** emp table  
**Trigger parameters:** old, new  
**Trigger action:** calculate the new commission to be updated

Note: no (colon :) before new in WHEN
Trigger Execution order

1. Execute all BEFORE STATEMENT triggers
2. Disable temporarily all integrity constraints recorded against the table
3. Loop for each row in the table
   – Execute all BEFORE ROW triggers
   – Execute the SQL statement against the row and perform integrity constraint checking of the data
   – Execute all AFTER ROW triggers
4. Complete deferred integrity constraint checking against the table
5. Execute all AFTER STATEMENT triggers
Transactions

Address two issues:

- Access by multiple users
  - Remember the “client-server” architecture: one server with many clients
- Protection against crashes
Flight Reservation

get values for :flight, :date, :seat

EXEC SQL  SELECT occupied INTO :occ
    FROM  Flight
    WHERE fltNum = :flight AND fltdt= :date AND  fltSeat=:seat

if (!occ) {
    EXEC SQL UPDATE Flights
        SET occupied = 'true'
        WHERE fltNum= :flight AND fltdt= :date AND fltSeat=:seat
    /*  more code missing */
}
else /* notify customer that seat is not available */
Problem #1

Customer 1 - finds a seat empty
Customer 2 - finds the same seat empty
Customer 1 - reserves the seat.
Customer 2 - reserves the seat.

Customer 1 will not be happy.
Customer 2 overwrites Customer 1 Reservation

serializability
Bank Transfers

Transfer :amount from :account1 to :account2

EXEC SQL SELECT balance INTO :balance1
    FROM Accounts
    WHERE accNo = :account1

if (balance1 >= amount)
    EXEC SQL UPDATE Accounts
        SET balance = balance + :amount
        WHERE acctNo = :account2;
    EXEC SQL UPDATE Accounts
        SET balance = balance - :amount
        WHERE acctNo = :account1;
Transactions

The user/programmer can group a sequence of commands so that they are executed atomically and in a serializable fashion:

- **Transaction commit**: all the operations should be done and recorded.
- **Transaction abort**: none of the operations should be done.

In SQL:

- EXEC SQL COMMIT;
- EXEC SQL ROLLBACK;

*Easier said than done...*
**ACID Properties**

**Atomicity:** all actions of a transaction happen, or none happen.

**Consistency:** if a transaction is consistent, and the database starts from a consistent state, then it will end in a consistent state.

**Isolation:** the execution of one transaction is isolated from other transactions.

**Durability:** if a transaction commits, its effects persist in the database.
How Do We Assure ACID?

Concurrency control:

Guarantees consistency and isolation, given atomicity.

Logging and Recovery:

Guarantees atomicity and durability.

*If you are going to be in the logging business, one of the things that you'll have to do is learn about heavy equipment.*

-- Robert VanNatta

*Logging History of Columbia County*
Transactions in SQL

In “ad-hoc” SQL:
- Default: each statement = one transaction

In “embedded” SQL:
BEGIN TRANSACTION
[SQL statements]
COMMIT or ROLLBACK (=ABORT)
Transactions: Serializability

Serializability = the technical term for isolation

- An execution is **serial** if it is completely before or completely after any other transaction’s execution
- An execution is **serializable** if it equivalent to one that is serial
- DBMS can offer serializability guarantees
Serializability

Enforced with locks, like in Operating Systems!

But this is not enough:

User 1

LOCK A
[write A=1]
UNLOCK A

... ...

LOCK B
[write B=2]
UNLOCK B

User 2

LOCK A
[write A=3]
UNLOCK A

LOCK B
[write B=4]
UNLOCK B

What is wrong?
Serializability

Solution: two-phase locking
   – Lock everything at the beginning
   – Unlock everything at the end

Read locks: many simultaneous read locks allowed

Write locks: only one write lock lock allowed

Insert locks: one per table
Isolation Levels in SQL

1. “Dirty reads”
   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
2. “Committed reads”
   SET TRANSACTION ISOLATION LEVEL READ COMMITTED
3. “Repeatable reads”
   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
4. Serializable transactions (default):
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE