# MSiA-413 Introduction to Databases and Information Retrieval

## Lecture 16
## Triggers, Introduction to Transactions

Instructor: Nikos Hardavellas

Slides adapted from S. Tarzia, A. Silberschatz, H.F. Korth, S. Sudarshan

## Last Lecture

- Recursive Queries on Networks
  - Powerful queries to express arbitrarily long chains of dependent queries
- Views
  - Temporary relations that are not part of the conceptual model
- `WITH` statement
  - Creates scoped views available only to the query issuing the `WITH` clause
- Set comparison and existential operators
  - `SOME`, `ANY`, `ALL`, `EXISTS`

Part I

Triggers



# Adding integrity constraints

SalesOrders.sqlite

- *New rule: no more than 4 employees in the same office (i.e., with same area code)*

```
SELECT EmpAreaCode, COUNT(*) AS NumEmployeesAtOffice
FROM Employees
GROUP BY EmpAreaCode;
```

| EmpAreaCode | NumEmployeesAtOffice |
|---|---|
| 206 | 1 |
| 210 | 1 |
| 253 | 2 |
| 425 | 4 |
| 515 | 1 |

- Inserting new employee at area code 425 should fail

```
INSERT INTO Employees
VALUES (800, "Nikos", "Hardavellas", "2233 Tech Dr",
        "Evanston", "IL", "60208", 425, "491-2270");
```

# How to enforce the rule? Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes

specify when and what to do



# Adding a trigger

SalesOrders.sqlite

- *New rule: no more than 4 employees in the same office (i.e., with same area code)*

```
CREATE TRIGGER Max4EmployeesPerOffice
BEFORE INSERT
ON Employees
FOR EACH ROW
BEGIN
  SELECT CASE
       WHEN (SELECT COUNT(*)
             FROM Employees
             WHERE EmpAreaCode = new.EmpAreaCode) >= 4
       THEN RAISE(FAIL, "Error: max 4 employees per office")
       END;
END;
```

*Trigger name*

*When to "fire"*

*On which table*

*Run trigger per row*

*Refer to "new row"*

*Trigger actions*

new. = the row that the insert statement attempts to insert

*Raise an exception & print error*

## Insert with the trigger defined

SalesOrders.sqlite

- Inserting new employee at area code 425 should fail

```
INSERT INTO Employees
VALUES (800, "Nikos", "Hardavellas", "2233 Tech Dr",
        "Evanston", "IL", "60208", 425, "491-2270");
```

```
Error: max 4 employees per office: INSERT INTO Employees
VALUES (800, "Nikos", "Hardavellas", "2233 Tech Dr",
"Evanston", "IL", "60208", 425, "491-2270")
```

- Triggers are supported by SQLite, **but not** DB Browser for SQLite
  - trigger bug report, github: "*We practically don't handle triggers at all in our application because they are complicated to parse*"
- Use the `sqlite3` command-line interface if in trouble



## Trigger Events

- The triggering event can be `insert`, `delete`, `update`, or `update of`

```
CREATE TRIGGER trigger_name
BEFORE INSERT ON table
...
```

```
CREATE TRIGGER trigger_name
BEFORE DELETE ON table
...
```

```
CREATE TRIGGER trigger_name
BEFORE UPDATE ON table
...
```

```
CREATE TRIGGER trigger_name
BEFORE UPDATE OF column1, column2, ... ON table
...
```

# Trigger Timing

- The trigger can fire `before`, `after`, or `instead of` the triggering event

```
CREATE TRIGGER trigger_name
BEFORE INSERT ON table
...
```

Typical use: add integrity constraints

```
CREATE TRIGGER trigger_name
INSERT ON table
...
```

Default is `BEFORE`

```
CREATE TRIGGER trigger_name
AFTER INSERT ON table
...
```

Typical use: perform additional actions

```
CREATE TRIGGER trigger_name
INSTEAD OF INSERT ON table
...
```



# INSTEAD OF Triggers

take another action than the user specified action

- Changes the statement to execute
- Example: instead of modifying a view, modify the main table

```
CREATE TABLE Customers(ID INTEGER PRIMARY KEY,
                       Name varchar(100),
                       Addr varchar(100) );


CREATE VIEW CustomerView
AS SELECT ID, Addr FROM Customers;


CREATE TRIGGER CustomerViewChange
INSTEAD OF UPDATE OF Addr ON CustomerView
BEGIN
  UPDATE customers SET Addr = new.Addr
  WHERE ID = new.ID;
END;
```

# Referencing attributes of old/new rows

- Use `old.` and `new.` to refer to the old/new rows of the `insert`, `update`, or `delete` statement that fired the trigger
- **`INSERT`**: `new.` references are valid Reference all the rows with new values
- **`UPDATE`**: `new.` and `old.` references are valid
- **`DELETE`**: `old.` references are valid Reference all the rows with original values
- Example:

```
CREATE TRIGGER AddrChange
AFTER UPDATE OF Addr ON Customers
BEGIN
  UPDATE orders SET Addr = new.Addr
  WHERE ID = old.ID;
END;
```

Refers to the row after the update

Refers to the row before the update



# Trigger execution granularity

- Defines how often the trigger will execute
- Example:

```
CREATE TRIGGER TriggerName
BEFORE INSERT
ON table
[FOR EACH ROW | FOR EACH STATEMENT]
BEGIN
...
END;
```

Execute trigger once for each row inserted by the triggering statement

Execute trigger once for the triggering statement

- SQLite implements only per-row triggers, hence this clause is optional

# Trigger event filtering

- Execute trigger only when certain conditions are satisfied
- Example:

```
CREATE TRIGGER TriggerName
BEFORE INSERT
ON table
WHEN condition
BEGIN
...
END;
```

similar to WHERE clause

- SQLite implements only per-row triggers, hence this clause is optional



# Raising exceptions

- Notify the caller that an error has occurred
  - Actions: print an error message, return an error to the application if needed
- `RAISE` is an **expression**, not a statement
  - Must be within a `SELECT`, `CASE`, or any other statement accepting expressions

```
CREATE TRIGGER Max4EmployeesPerOffice
BEFORE INSERT ON Employees
FOR EACH ROW
BEGIN
 SELECT CASE
      WHEN (SELECT COUNT(*)
            FROM Employees
            WHERE EmpAreaCode = new.EmpAreaCode) >= 4
      THEN RAISE(FAIL, "Error: max 4 employees per office")
      END;
END;
```

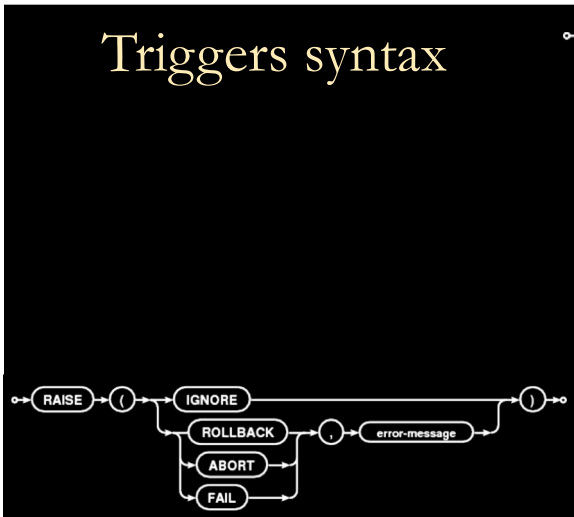
Conflict resolution algorithm

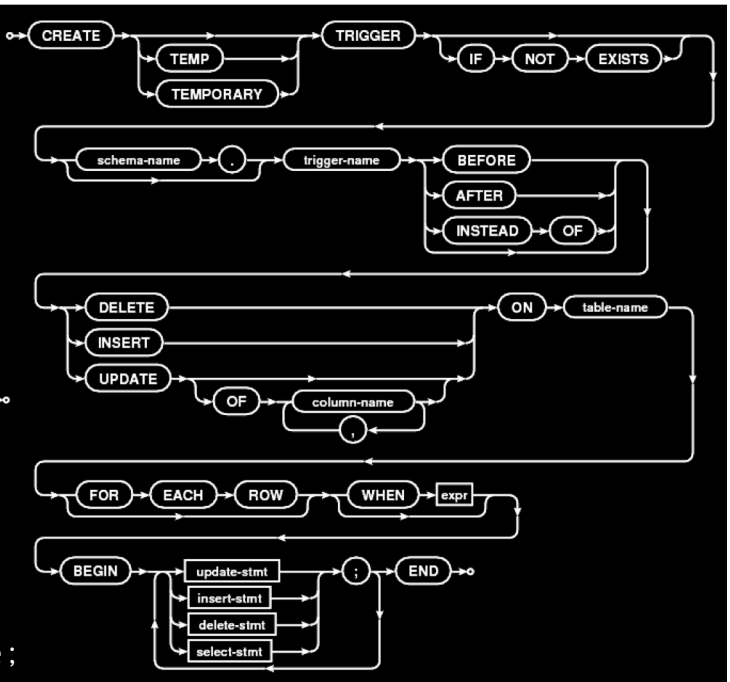
# Conflict resolution algorithms

- `FAIL`: stop processing the rest of the current SQL statement
  - Do not undo any prior changes — don't undo what didn't fail
  - If it fails on the 100th row, the actions taken due to the previous 99 rows are preserved
  - The transaction remains active (if within one)
- `ABORT`: stop processing the rest of the current SQL statement and abort
  - **Undo** any prior changes made by the **current SQL statement**
  - Changes caused by prior SQL statements within the same transaction are preserved
  - The transaction remains active
- `ROLLBACK`: stop processing the rest of the current SQL statement and rollback
  - **Undo** any prior changes made by **all SQL statements** in the transaction
  - **End the transaction**
  - If not within a transaction, `ROLLBACK` and `ABORT` are the same
- `IGNORE`: skip the one row that violates the constraint — skip the one that violates
  - Continue processing subsequent rows as if nothing went wrong
  - Do not return an error to the application



# Triggers syntax




- Drop with:

```
DROP TRIGGER trigger_name;
```

# Part II
# Transactions



## Transaction Concept

unit of work - single atomic element - happens or not

- A transaction is a unit of program execution that accesses and possibly updates various data items
- Example: transaction to transfer $50 from account A to account B:

```
1. read(A)
2. A := A – 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)
```

- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

# Atomicity requirement

- Transaction to transfer $50 from account A to account B:

```
1. read(A)
2. A := A – 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)
```

transfer needs to happen in entirety or not at all

- What if the transaction fails at step 5 ?
  - Money will be "lost" leading to an inconsistent database state
  - Failure could be due to software or hardware
- The system should ensure that updates of a partially executed transaction are not reflected in the database



# Durability requirement

- Transaction to transfer $50 from account A to account B:

```
1. read(A)
2. A := A – 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)
```

- The updates to the database by the transaction must persist even if there are software or hardware failures
- Once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place) the state of the database should always reflect that

# Consistency requirement

- Transaction to transfer $50 from account A to account B:

```
1. read(A)
2. A := A – 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)
```

check sum integrity constraint before and after transaction

- In above example: the sum of A and B is unchanged
- In general, consistency requirements include
  - Explicit integrity constraints, e.g., primary keys, foreign keys, unique values
  - Implicit integrity constraints, e.g., balances minus loans must equal cash-in-hand
- A transaction must see a consistent database
  - During transaction execution the database may be temporarily inconsistent
  - When the transaction completes successfully the database must be consistent



# Isolation requirement

- Transaction to transfer $50 from account A to account B:

```
   User 1                          User 2
1. read(A)
2. A := A – 50
3. write(A)
                        read(A), read(B), print(A+B)
4. read(B)
5. B := B + 50
6. write(B)
```

- User 2 should not be allowed to see the temporarily inconsistent database
  - The sum A+B should not be incorrect, otherwise money appear to be "lost"
- Provide the illusion that transactions execute **serially**, i.e., one after the other
  - User 1 fully executes his transaction, then User 2 fully executes his transaction
  - ...or the other way around

# ACID properties

- **Atomicity**. Either all operations of the transaction are properly reflected in the database, or none are
- **Consistency**. The execution of a transaction in isolation preserves the consistency of the database
- **Isolation**. Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions
  - Intermediate results must be hidden from the outside world
  - For every pair of transactions $T_i$ and $T_j$, it appears to $T_i$ that
    - Either $T_i$ finished execution before $T_j$ started, or
    - $T_j$ finished execution before $T_i$ started
- **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures



# Transaction state

- Active: the initial state; the transaction stays in this state while it is executing
- Partially committed: after the final statement has been executed
- Failed: after the discovery that normal execution can no longer proceed
- Aborted: after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.
  - Two options after it has been aborted:
    - Restart the transaction (can be done only if no internal logical error )
    - Kill the transaction
- Committed: after successful completion
- Transactions begin implicitly or explicitly
  - Ended by commit work or rollback work
- Default on most databases: each SQL statement commits automatically

temporary state

# Example: atomically delete rows

SalesOrders.sqlite

```
select * from order_details;
```

| | OrderNumber | ProductNumber | QuotedPrice | QuantityOrdered |
|---|---|---|---|---|
| 1 | 1 | 1 | 1200 | 2 |
| 2 | 1 | 6 | 635 | 3 |
| 3 | 1 | 11 | 1650 | 4 |
| 4 | 1 | 16 | 28 | 1 |
| 5 | 1 | 21 | 55 | 3 |
| 6 | 1 | 26 | 121.25 | 5 |
| 7 | 1 | 40 | 174.6 | 6 |
| 8 | 2 | 27 | 24 | 4 |
| 9 | 2 | 40 | 180 | 4 |
| 10 | 3 | 1 | 1164 | 5 |

Commit transaction

Begin transaction

```
savepoint transaction1;
delete from order_details where orderNumber=1;
delete from order_details where orderNumber=2;
select * from order_details;
commit;

select * from order_details;
```

| | OrderNumber | ProductNumber | QuotedPrice | QuantityOrdered |
|---|---|---|---|---|
| 1 | 3 | 1 | 1164 | 5 |
| 2 | 3 | 6 | 615.95 | 5 |
| 3 | 3 | 11 | 1650 | 1 |



# Example: rollback attempt to atomically delete rows

SalesOrders.sqlite

```
select * from order_details;
```

| | OrderNumber | ProductNumber | QuotedPrice | QuantityOrdered |
|---|---|---|---|---|
| 1 | 3 | 1 | 1164 | 5 |
| 2 | 3 | 6 | 615.95 | 5 |
| 3 | 3 | 11 | 1650 | 1 |
| 4 | 3 | 16 | 28 | 2 |

Rollback and end transaction

Begin transaction

```
savepoint transaction2;
delete from order_details where orderNumber=3;
delete from order_details where orderNumber=4;
select * from order_details;
rollback;

select * from order_details;
```

| | OrderNumber | ProductNumber | QuotedPrice | QuantityOrdered |
|---|---|---|---|---|
| 1 | 3 | 1 | 1164 | 5 |
| 2 | 3 | 6 | 615.95 | 5 |
| 3 | 3 | 11 | 1650 | 1 |
| 4 | 3 | 16 | 28 | 2 |

# Named transactions

- `BEGIN ... END` is another way to denote a transaction
  - `END` and `COMMIT` are the same: complete and exit the transaction
  - `ROLLBACK`: undo all changes and cancel the transaction
    - Subsequent SQL statements are not part of a transaction
  - `BEGIN` cannot be used within a transaction (i.e., no nesting)
- `SAVEPOINT` starts a transaction that is named and can be nested
  - `COMMIT`: commits all outstanding transactions and leaves transaction stack empty
  - `ROLLBACK`: undo all changes and cancel the transaction
  - `ROLLBACK TO TransactionName`:
    - Undo all changes until the beginning of `TransactionName`, and
    - Enter a new transaction with the name `TransactionName`
  - `RELEASE TransactionName`:
    - Remove all savepoints back to and including the savepoint named `TransactionName`
    - Cannot rollback to these savepoints anymore
    - No write back of modifications; `COMMIT` does that



# Example: nested named transactions

`SalesOrders.sqlite`

```
select * from order_details;
```

| | OrderNumber | ProductNumber | QuotedPrice | QuantityOrdered |
|---|---|---|---|---|
| 1 | 3 | 1 | 1164 | 5 |
| 2 | 3 | 6 | 615.95 | 5 |
| 3 | 3 | 11 | 1650 | 1 |
| 4 | 3 | 16 | 28 | 2 |

```
T1: savepoint transaction1;
T1: delete from order_details where orderNumber=3;
T1: savepoint transaction2;
T2: delete from order_details where orderNumber=4;
T2: select * from order_details;
```

transaction 2 still active,

```
T2: rollback to transaction2;
T2: select * from order_details;
```

justremove previous statements

T1 still not terminated

```
T2: rollback to transaction1;
T1: rollback;
```

rollback removes everything including transaction name

```
select * from order_details;
```

| | OrderNumber | ProductNumber | QuotedPrice | QuantityOrdered |
|---|---|---|---|---|
| 1 | 3 | 1 | 1164 | 5 |
| 2 | 3 | 6 | 615.95 | 5 |
| 3 | 3 | 11 | 1650 | 1 |
| 4 | 3 | 16 | 28 | 2 |



release transaction 2