MLDS-413 Introduction to Databases and Information Retrieval

Lecture 17
Triggers
Introduction to Transactions

Instructor: Nikos Hardavellas

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

Last Lecture

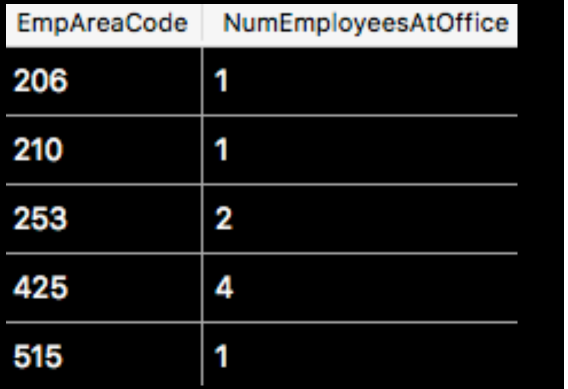
- Datetime functions
  - Powerful functions to manipulate date and time
- Windowing
  - Constrain aggregators to a moving window of rows
- `OVER` and `WINDOW` statements
  - Define a window partition, order, and frame

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
- Complicated syntax
  - DB Browser for SQLite comment on github bug report, Aug 20, 2018: *"We practically don't handle triggers at all in our application because they are complicated to parse"*
  - Support is there now

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

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");
```

```
Execution finished with errors.
Result: Error: max 4 employees per office
At line 5:
INSERT INTO Employees
VALUES (800, "Nikos", "Hardavellas", "2233 Tech Dr",
"Evanston", "IL", "60208", 425, "491-2270");
```

Trigger Events

- The trigger event can be an insert, delete, update, or update of <cols>

```
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

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

```
CREATE VIEW CustomerView
AS SELECT CustomerID, CustAreaCode FROM Customers;


SELECT CustAreaCode FROM customerview WHERE CustomerID = 1001;

425


UPDATE CustomerView SET CustAreaCode = 314
WHERE CustomerID = 1001;

Execution finished with errors.
Result: cannot modify CustomerView because it is a view
At line 18:
UPDATE CustomerView SET CustAreaCode = 314
WHERE CustomerID = 1001
```

INSTEAD OF Triggers

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

```
CREATE VIEW CustomerView
AS SELECT CustomerID, CustAreaCode FROM Customers;


CREATE TRIGGER CustomerViewChange
INSTEAD OF UPDATE OF CustAreaCode ON CustomerView
BEGIN
   UPDATE customers SET CustAreaCode = new.CustAreaCode
   WHERE CustomerID = new.CustomerID;
END;


UPDATE CustomerView SET CustAreaCode = 314
WHERE CustomerID = 1001;

SELECT CustAreaCode FROM Customers WHERE CustomerID = 1001;

314
```

Referencing attributes of old/new rows

- Example: data integrity in order $ between `Orders` and `Order_Details`
- Without a trigger

```
SELECT OrderTotal FROM Orders WHERE OrderNumber=522;
4.99
SELECT QuotedPrice FROM Order_Details WHERE OrderNumber=522;
4.99


UPDATE Order_Details SET QuotedPrice=5.99
WHERE OrderNumber=522;


SELECT OrderTotal FROM Orders WHERE OrderNumber=522;
4.99
SELECT QuotedPrice FROM Order_Details WHERE OrderNumber=522;
5.99
```

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
  `UPDATE`: `new.` and `old.` references are valid
  `DELETE`: `old.` references are valid
- Example trigger for `Orders` and `Order_Details`

```
CREATE TRIGGER PriceChange
AFTER UPDATE OF QuotedPrice ON Order_Details
BEGIN
   UPDATE Orders
   SET OrderTotal = OrderTotal
       + old.QuantityOrdered * (new.QuotedPrice - old.QuotedPrice)
   WHERE OrderNumber = old.OrderNumber;
END;
```

`new` refers to the row after the update

`old` refers to the row before the update

Referencing attributes of old/new rows

- Example: data integrity in order $ between `Orders` and `Order_Details`
- With the trigger

```
SELECT OrderTotal FROM Orders WHERE OrderNumber=522;
4.99
SELECT QuotedPrice FROM Order_Details WHERE OrderNumber=522;
4.99


UPDATE Order_Details SET QuotedPrice=5.99
WHERE OrderNumber=522;


SELECT OrderTotal FROM Orders WHERE OrderNumber=522;
5.99
SELECT QuotedPrice FROM Order_Details WHERE OrderNumber=522;
5.99
```

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 each triggering statement

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

Trigger event filtering

- Execute trigger only when certain conditions are satisfied
- WHEN condition can access new. and old. rows
- Example: the previous trigger for Orders and Order_Details can be modified to detect updates of QuotedPrice

```
CREATE TRIGGER PriceChange
AFTER UPDATE ~~OF QuotedPrice~~ ON Order_Details
WHEN old.QuotedPrice <> new.QuotedPrice
BEGIN
  UPDATE Orders
  SET OrderTotal = OrderTotal
      + old.QuantityOrdered * (new.QuotedPrice -
old.QuotedPrice)
  WHERE OrderNumber = old.OrderNumber;
END;
```

Undefined behavior and `BEFORE` triggers

- Rules for `BEFORE UPDATE` and `BEFORE DELETE` triggers
  - If trigger modifies or deletes a row that was to have been updated or deleted
    → the subsequent update or delete operation is undefined
  - If trigger modifies or deletes a row
    → `AFTER` triggers that would have otherwise run on those rows may/may not run

- Rules for `BEFORE INSERT` triggers
  - If `rowid` is not explicitly set to an integer
    → `NEW.rowid` is undefined

- Because of the behaviors described above, programmers are encouraged to prefer `AFTER` triggers over `BEFORE` triggers when the triggers change data

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 a **function** (i.e., part of an **expression**), not a statement
  - Must be within a **`SELECT`**, **`CASE`**, or any other statement accepting expressions

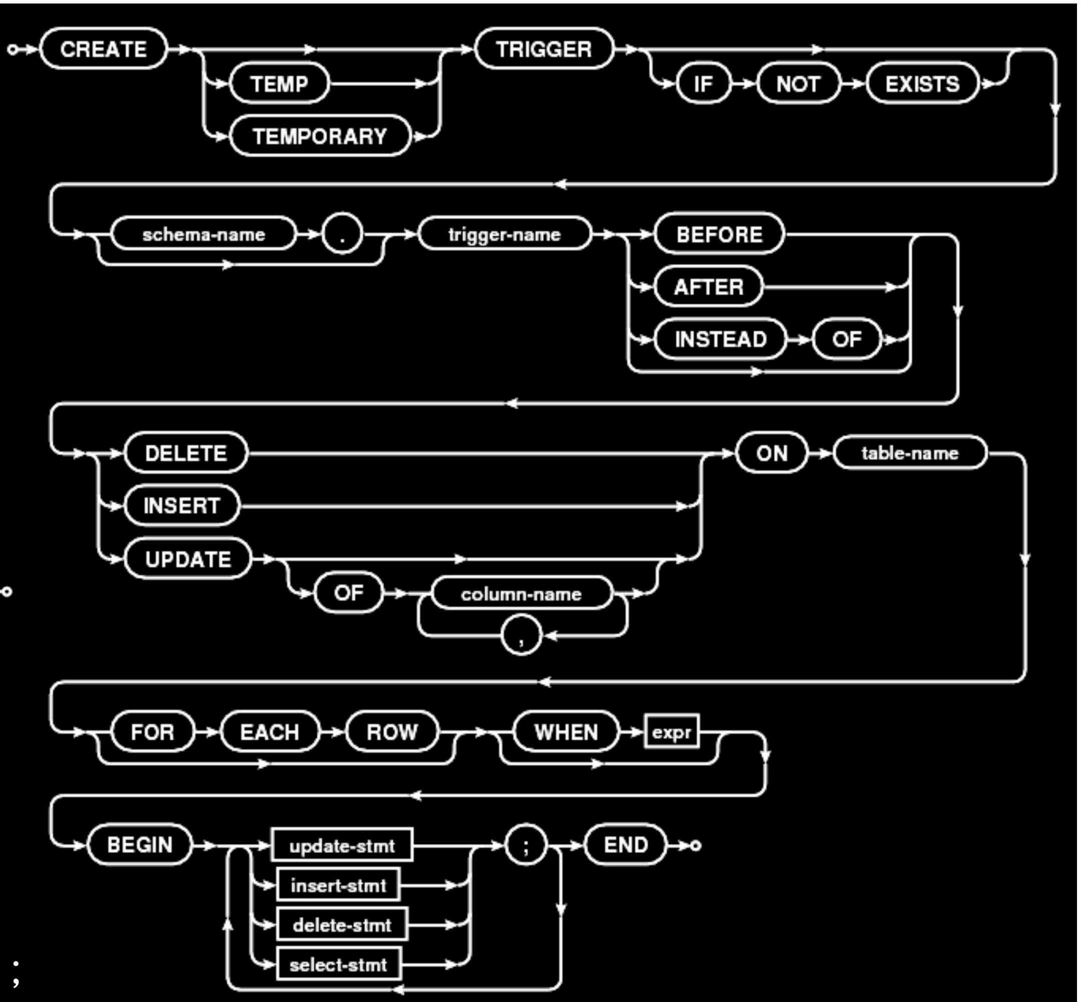
```
CREATE TRIGGER Max4EmployeesPerOffice
BEFORE INSERT ON Employees
FOR EACH ROW
BEGIN
 SELECT CASE
      WHEN (SELECT COUNT(*)
            FROM Employees
            WHERE EmpAreaCode = new.EmpAreaCode) >= 4
      THEN RAISE(FAIL, "Error: max 4 employees per office")
      END;
END;
```

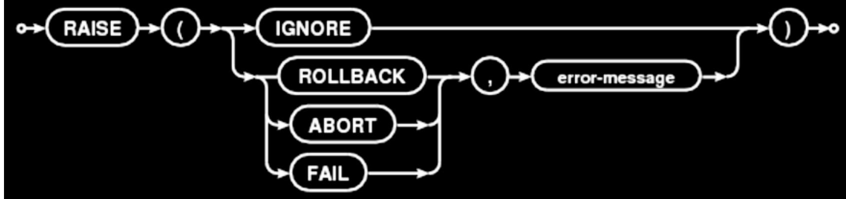
Conflict resolution algorithm

Conflict resolution algorithms

- `FAIL`: stop processing the rest of the current SQL statement
  - Do not undo any prior changes
  - 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
  - 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;
```

Triggers in Postgres

- PostgreSQL supports triggers similarly to SQLite

```
CREATE [CONSTRAINT] TRIGGER name
  {BEFORE | AFTER | INSTEAD OF} {event [OR ...]}
  ON table_name
  [FROM referenced_table_name]
  [NOT DEFERRABLE | [DEFERRABLE] [INITIALLY IMMEDIATE | INITIALLY DEFERRED]]
  [REFERENCING { {OLD | NEW} TABLE [AS] transition_relation_name } [ ... ]
]
  [FOR [EACH] { ROW | STATEMENT } ]
  [WHEN (condition)]
  EXECUTE PROCEDURE function_name (arguments)
```

```
where event can be one of:

  INSERT
  UPDATE [ OF column_name [, ... ] ]
  DELETE
  TRUNCATE
```

```
CREATE FUNCTION function_name() RETURNS trigger AS $function_name$
BEGIN ... END
```

Transition Tables

- `new/old` iterate over rows only
- Can reference new/old rows of a *statement trigger* through transition tables

```
CREATE TRIGGER my_trigger
  BEFORE INSERT ON orig_table
  REFERENCING OLD TABLE AS old_tbl NEW TABLE AS new_tbl
  FOR EACH ROW
  EXECUTE PROCEDURE func();


CREATE FUNCTION func() RETURNS trigger AS $func$
BEGIN ... END
```

Constraint Triggers in Postgres

```
CREATE CONSTRAINT TRIGGER my_trigger
  ...
```

- Must be `AFTER ROW` triggers
- The timing of the trigger firing can be adjusted
  - at the end of the statement causing the triggering event, or
  - at the end of the containing transaction (deferred)

- `SET CONSTRAINTS`
  - set constraint check timing for the current transaction
  - `SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }`

(Non) Deferrable Constraint Triggers in Postgres

- `NOT DEFERRABLE`
  - Will be checked immediately after every command

- `DEFERRABLE`
  - Checking can be postponed until the end of the transaction

- `INITIALLY IMMEDIATE`
- `INITIALLY DEFERRED`
  - For deferrable constraints
  - Specify the default time to check the constraint
  - `INITIALLY IMMEDIATE`: check after each statement. This is the default.
  - `INITIALLY DEFERRED`: checked only at the end of the transaction
  - The constraint check time can be altered with the `SET CONSTRAINTS` command

Part II
Introduction to Transactions

Transaction Concept

- 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 such as hardware failures, system crashes, query failures (e.g., triggers, conflicts)
  - 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)
```

- 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)
```

- 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 example: atomic updates - commit

```
SELECT OrderTotal FROM Orders WHERE OrderNumber IN (100, 101);
3835.68
1380.64
```

Start a transaction

```
BEGIN TRANSACTION;
UPDATE Orders SET OrderTotal=1.99 WHERE OrderNumber=100;
UPDATE Orders SET OrderTotal=1.99 WHERE OrderNumber=101;
SELECT OrderTotal FROM Orders WHERE OrderNumber IN (100, 101);
1.99
1.99
```

… check things; satisfied all is in order …

```
COMMIT TRANSACTION;
```

make transaction updates persistent & end trans.

```
SELECT OrderTotal FROM Orders WHERE OrderNumber IN (100, 101);
1.99
1.99
```

Transaction example: atomic updates - rollback

```
SELECT OrderTotal FROM Orders WHERE OrderNumber IN (100, 101);
3835.68
1380.64
```

Start a transaction

```
BEGIN TRANSACTION;
UPDATE Orders SET OrderTotal=1.99 WHERE OrderNumber=100;
UPDATE Orders SET OrderTotal=1.99 WHERE OrderNumber=101;
SELECT OrderTotal FROM Orders WHERE OrderNumber IN (100, 101);
1.99
1.99
```

... check things; realized you made a mistake ...

```
ROLLBACK TRANSACTION;
```

undo transaction updates & and transaction

```
SELECT OrderTotal FROM Orders WHERE OrderNumber IN (100, 101);
3835.68
1380.64
```

Transaction state

- Active: the initial state; the transaction stays in this state while it is executing
- Failed: after the discovery that normal execution can no longer proceed
- Aborted: after the transaction has been rolled back and the database is 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
- Partially committed: after the final statement has been executed
- Committed: after successful completion

Named transactions

- `BEGIN ... END` is one 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 the transaction
  - `BEGIN` cannot be used within a transaction (i.e., no nesting)
- `SAVEPOINT` starts a transaction that is named and can be nested

Savepoints are similar to snapshots

- `SAVEPOINT TransactionName`
  - Create a new "mark" named `TransactionName` in the transaction timeline
  - "checkpoint" the database, i.e., takes a logical snapshot of it
- `ROLLBACK TO TransactionName`
  - Rewind the timeline back to a point just after the `TransactionName` mark
  - "restore the checkpoint"
- `RELEASE TransactionName`
  - Erase marks from the timeline without actually making any changes to the database
  - ...true for nested savepoints only
- `COMMIT`
  - Commits all outstanding transactions and leaves transaction stack empty
- `ROLLBACK`
  - Undo all changes and cancel all outstanding transactions

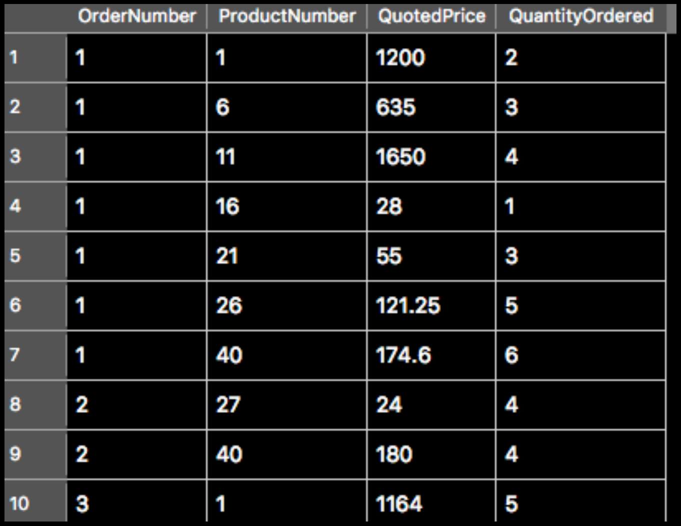
Savepoints

- `SAVEPOINT TransactionName`
  - Starts a transaction that is named and can be nested
- `ROLLBACK TO TransactionName`
  - Undo all changes until the beginning of `TransactionName`, and
  - Cancel all intervening savepoints, and
  - Restart the transaction with the name `TransactionName`
- `RELEASE TransactionName`
  - Remove all savepoints back to and including `TransactionName`
  - Cannot rollback to these savepoints anymore
  - If `TransactionName` is inner savepoint: no write back of modifications; `COMMIT` does that
  - If `TransactionName` is outermost savepoint, so that the transaction stack becomes empty, then `RELEASE` is the same as `COMMIT`
- `COMMIT`: commits all outstanding transactions and leaves transaction stack empty
- `ROLLBACK`: undo all changes and cancel all outstanding transactions

Example: atomically delete rows

SalesOrders.sqlite

```
select * from order_details;
```



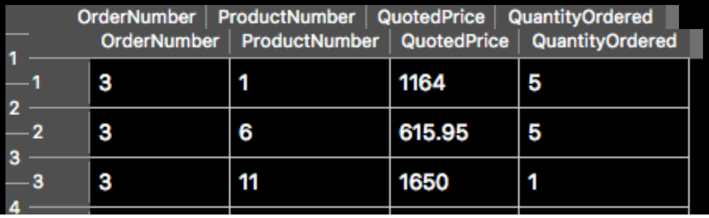
	OrderNumber	ProductNumber	QuotedPrice	QuantityOrdered
1	1	1	1200	2
2	1	6	635	3
3	1	11	1650	4
4	1	16	28	1
5	1	21	55	3
6	1	26	121.25	5
7	1	40	174.6	6
8	2	27	24	4
9	2	40	180	4
10	3	1	1164	5

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;
```

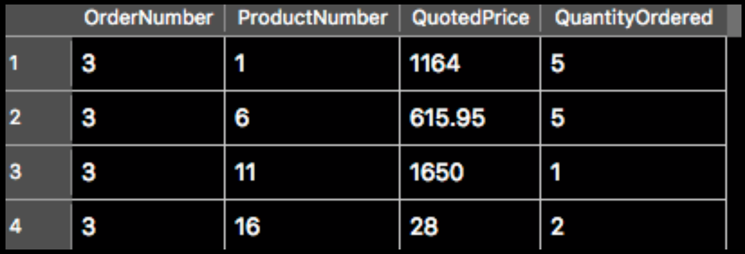


	OrderNumber	ProductNumber	QuotedPrice	QuantityOrdered
1	3	1	1164	5
2	3	6	615.95	5
3	3	11	1650	1

Example: rollback attempt to atomically delete rows

`SalesOrders.sqlite`

```
select * from order_details;
```



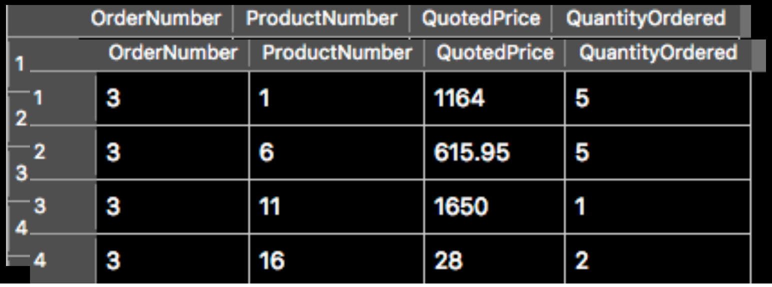
	OrderNumber	ProductNumber	QuotedPrice	QuantityOrdered
1	3	1	1164	5
2	3	6	615.95	5
3	3	11	1650	1
4	3	16	28	2

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;
```

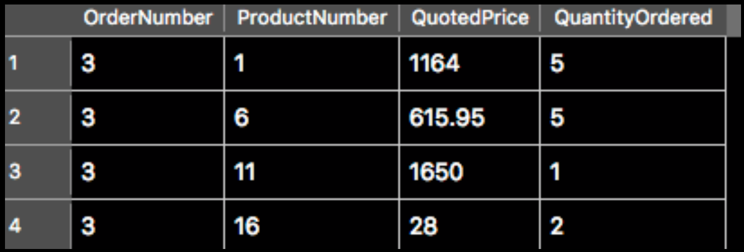


	OrderNumber	ProductNumber	QuotedPrice	QuantityOrdered
1	3	1	1164	5
2	3	6	615.95	5
3	3	11	1650	1
4	3	16	28	2

Example: nested named transactions

SalesOrders.sqlite

```
    select * from order_details;
```



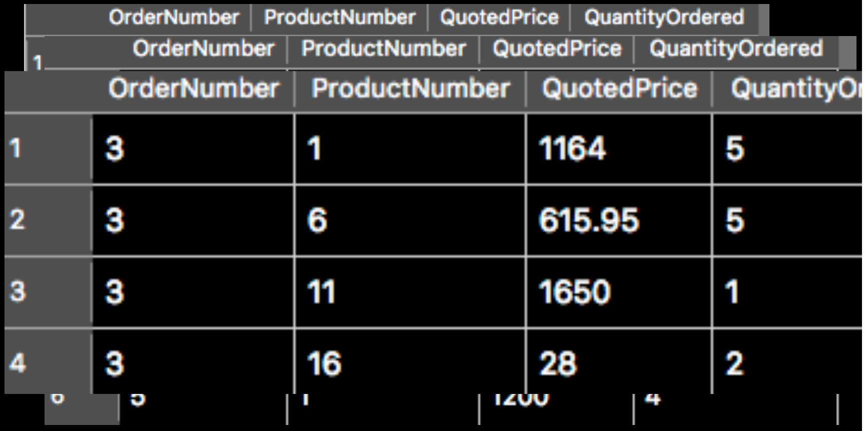
	OrderNumber	ProductNumber	QuotedPrice	QuantityOrdered
1	3	1	1164	5
2	3	6	615.95	5
3	3	11	1650	1
4	3	16	28	2

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

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

T2:X    rollback to transaction1;
T1:X  rollback;

  select * from order_details;
```



	OrderNumber	ProductNumber	QuotedPrice	QuantityO
1	3	1	1164	5
2	3	6	615.95	5
3	3	11	1650	1
4	3	16	28	2

Transaction initiation

- Transactions can begin explicitly
  - `[BEGIN | SAVEPOINT] .. .[END | COMMIT | ROLLBACK | RELEASE]`
- Transactions can begin implicitly
  - Default on most databases: each SQL statement is wrapped in its own transaction
    - Transaction begins at SQL statement start
    - Transaction commits at the statement's final ";"
  - Starting up a server connection → transaction begin
    - pgAdmin 4: Configure → Query Tools → Options → Autocommit / Autorollback
  - Starting up DB Browser for SQLite → transaction begin

  - Do `SELECT` statements require a transaction?
    - At the usual isolation level, a `SELECT`
      - Should not read uncommitted writes
      - Should not read writes from transactions that commit while the `SELECT` is running

Transactions in Postgres, MySQL

- PostgreSQL support transactions similarly to SQLite
- This includes support for `SAVEPOINT`