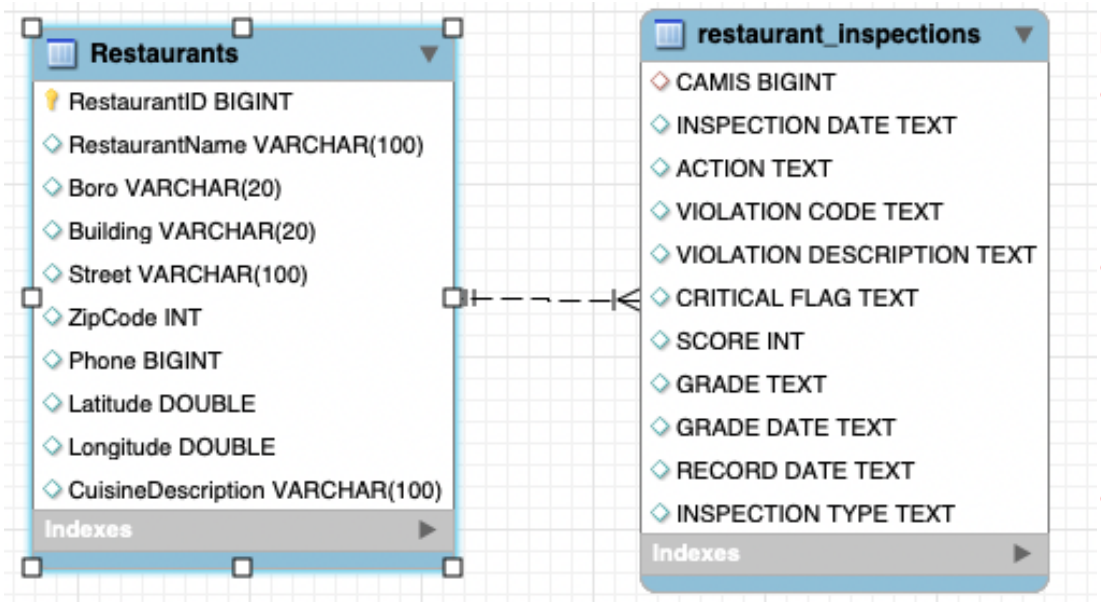
CS 61:
Database Systems

Advanced SQL

Adapted from https://www.mysqltutorial.org/ unless otherwise noted
Augmented by Claude Opus

Review: In our NYC health inspections schema, we left off with two tables

We can JOIN Restaurants with their inspections using RestaurantID = CAMIS



Restaurants

- Each row is data about one restaurant
- Can change name, cuisine, or move to new address -> just update one row in this table

restaurant_inspections

- Each row is data about one violation in one inspection of a restaurant
- We removed the duplicate columns about the restaurant's name, address, cuisine
- We just keep the CAMIS (ID) integer for each restaurant in this inspections table
- Now store one integer (4 bytes) vs. repeated restaurant name, address, boro, zip code, …

Agenda

1. Stored procedures and functions
2. Transactions
3. Triggers

NOTE: we can use variables in SQL, either by setting values directly or via query

Set variable value directly

```
3 •  SET @RestaurantID = 1111;
4 •  SELECT @RestaurantID;
5
6
7
```

100% 1:6

Result Grid | Filter Rows: Search

@RestaurantID
1111

Set value in query

```
7 •  SELECT RestaurantName, Boro INTO @RestaurantName, @Boro
8    FROM Restaurants
9    WHERE RestaurantID = @RetaurantID;
10
11 •  SELECT @RestaurantName, @Boro;
12
```

RestaurantID from previous query (value 1111)

100% 1:12

Result Grid | Filter Rows: Search | Export:

@RestaurantName	@Boro
Tim's Tasty Treats'	Manhattan

- No need to declare variable or type
- Format: @varname
- To see value use variable in SELECT statement

- Use SELECT columns INTO variables
- Can have multiple variables (name and boro above), but from only one row
- Use LIMIT 1 if query would return more than one row

Stored procedures and functions allow us to store business logic in the database

In the "bad old days" we embedded SQL directly into our application programs. This caused problems:

- What if multiple applications access the same database, how do we make sure they both implement the same business logic?
- How do we keep multiple applications following the same rules when changes occur?

Downsides:

If you use a lot of stored procedures and functions, tends to increase memory utilization

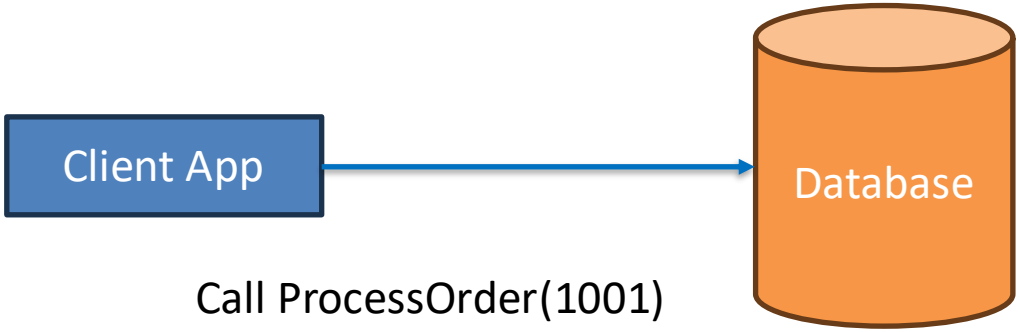
Stored procedures and functions allow us to move some business logic into the database itself

Also difficult to debug (no means to stop query execution and examine state)

- Now changes made in a single place
- Can make changes to logic and may not break applications

SQL is reasonably consistent across database vendors, but functions and stored procedures tend to be vendor-specific (our focus is MySQL)

Stored procedures and functions allow us to store business logic in the database



Call ProcessOrder(1001)
(one call)

Stored procedure
ProcessOrder(IN OrderID INT)

- Validate order
- Check inventory
- Update stock
- Create invoice
- Log transaction

Benefits:

- **Performance – precompiled**
- **Security – can control who can execute**
- **Reusability – write once, call from many apps**
- **Maintainability – can change logic in one place**
- **Reduced network traffic – on call vs. multiple**
- **Consistency – ensures same business logic applied**

Drawbacks

- **Debugging – harder to debug than app code**
- **Limited features – not as powerful as Python, Java, …**

Stored procedure can comprise many SQL statements

Database compiles stored procedure

Stored procedures allow us to save one or more SQL statements

Consider the following query

```
3    -- consider this query
4 •  SELECT * FROM Restaurants WHERE boro = 'Manhattan';
```

When you run this query from MySQL Workbench, database runs it and returns results as shown

- **If you run this query a lot, you might want to save it so you can easily run it again**
- **If you save it, the database can compile it for *possibly* faster execution**
- **Could use a view, but views have trouble with updates and deletes**
- **Stored procedures are *far* more powerful than views**

100% 1:12

Result Grid | Filter Rows: Search | Edit: | Export/Import: | Fetch rows:

RestaurantID	RestaurantName	Boro	Building	Street	Zip
1111	Tim's Tasty Treats	Manhattan	NULL	NULL	NUL
30191841	DJ REYNOLDS PUB AND RESTAU…	Manhattan	351	WEST 57 STREET	10
40359480	1 EAST 66TH STREET KITCHEN	Manhattan	1	EAST 66 STREET	10
40362264	P & S DELI GROCERY	Manhattan	730	COLUMBUS AVENUE	10
40362274	ANGELIKA FILM CENTER	Manhattan	18	WEST HOUSTON STR…	10
40362715	THE COUNTRY CAFE	Manhattan	60	WALL STREET	10
40363298	CAFE METRO	Manhattan	625	8 AVENUE	10
40363426	LEXLER DELI	Manhattan	405	LEXINGTON AVENUE	10
40363630	LORENZO & MARIA'S KITCHEN	Manhattan	1418	THIRD AVENUE	10

To create a stored procedure in MySQL, first change the delimiter

```
6    -- save query as stored procedure
7    DELIMITER $$
8    CREATE PROCEDURE GetManhattanRestaurants()
9    BEGIN
10       SELECT * FROM Restaurants WHERE boro = 'Manhattan';
11   END$$
12   DELIMITER ;
13
14   -- call stored procedure
15   CALL GetManhattanRestaurants();
```

- A stored procedure may have many commands separated by ;
- Temporarily change delimiter to something else ($$, //, etc) so MySQL knows the function is not done until it encounters the delimiter again
- Change delimiter back to ; at end

100% 13:15

Result Grid | Filter Rows: Search | Export: | Fetch rows:

RestaurantID	RestaurantName	Boro	Building	Street	ZipCode	Pho
1111	Tim's Tasty Treats	Manhattan	NULL	NULL	NULL	NULL
30191841	DJ REYNOLDS PUB AND RESTAU…	Manhattan	351	WEST 57 STREET	10019	212
40359480	1 EAST 66TH STREET KITCHEN	Manhattan	1	EAST 66 STREET	10065	212
40362264	P & S DELI GROCERY	Manhattan	730	COLUMBUS AVENUE	10025	212
40362274	ANGELIKA FILM CENTER	Manhattan	18	WEST HOUSTON STR…	10012	212
40362715	THE COUNTRY CAFE	Manhattan	60	WALL STREET	10005	347
40363298	CAFE METRO	Manhattan	625	8 AVENUE	10018	212
40363426	LEXLER DELI	Manhattan	405	LEXINGTON AVENUE	10174	212
40363630	LORENZO & MARIA'S KITCHEN	Manhattan	1418	THIRD AVENUE	10028	212

Then add your SQL, and change the delimiter back to a semicolon

```
6    -- save query as stored procedure
7    DELIMITER $$
8    CREATE PROCEDURE GetManhattanRestaurants()
9    BEGIN
10       SELECT * FROM Restaurants WHERE boro = 'Manhattan';
11   END$$
12   DELIMITER ;
13
14   -- call stored procedure
15   CALL GetManhattanRestaurants();
```

Create stored procedure and give it a name

- Can have several SQL commands between BEGIN and END statements
- Can call other stored procedures

Change command delimiter back to semicolon

Procedure is now stored as part of database

Use CALL to execute stored procedure

Same results as executing from MySQL Workbench directly

100% 13:15

Result Grid | Filter Rows: Search | Export: | Fetch rows:

RestaurantID	RestaurantName	Boro	Building	Street	ZipCode	Pho
1111	Tim's Tasty Treats	Manhattan	NULL	NULL	NULL	NULL
30191841	DJ REYNOLDS PUB AND RESTAU…	Manhattan	351	WEST 57 STREET	10019	212
40359480	1 EAST 66TH STREET KITCHEN	Manhattan	1	EAST 66 STREET	10065	212
40362264	P & S DELI GROCERY	Manhattan	730	COLUMBUS AVENUE	10025	212
40362274	ANGELIKA FILM CENTER	Manhattan	18	WEST HOUSTON STR…	10012	212
40362715	THE COUNTRY CAFE	Manhattan	60	WALL STREET	10005	347
40363298	CAFE METRO	Manhattan	625	8 AVENUE	10018	212
40363426	LEXLER DELI	Manhattan	405	LEXINGTON AVENUE	10174	212
40363630	LORENZO & MARIA'S KITCHEN	Manhattan	1418	THIRD AVENUE	10028	212

Call your stored procedure using the CALL command

```
6    -- save query as stored procedure
7    DELIMITER $$
8    CREATE PROCEDURE GetManhattanRestaurants()
9    BEGIN
10       SELECT * FROM Restaurants WHERE boro = 'Manhattan';
11   END$$
12   DELIMITER ;
13
14   -- call stored procedure
15   CALL GetManhattanRestaurants();
```

100% 13:15

Banks love stored procedures

- **Consistent business logic**
- **Secure – can control access**

On first call, MySQL looks up procedure name in the database catalog, compiles the code, places it in cache memory, and executes code

On subsequent calls, execute from cache

Multiple stored procedures in cache can use up memory quickly!

Each database user has its own cache!

Result Grid | Filter Rows: Search | Export: | Fetch rows:

RestaurantID	RestaurantName	Boro	Building	Street	ZipCode	Pho
1111	Tim's Tasty Treats	Manhattan	NULL	NULL	NULL	NULL
30191841	DJ REYNOLDS PUB AND RESTAU…	Manhattan	351	WEST 57 STREET	10019	212
40359480	1 EAST 66TH STREET KITCHEN	Manhattan	1	EAST 66 STREET	10065	212
40362264	P & S DELI GROCERY	Manhattan	730	COLUMBUS AVENUE	10025	212
40362274	ANGELIKA FILM CENTER	Manhattan	18	WEST HOUSTON STR…	10012	212
40362715	THE COUNTRY CAFE	Manhattan	60	WALL STREET	10005	347
40363298	CAFE METRO	Manhattan	625	8 AVENUE	10018	212
40363426	LEXLER DELI	Manhattan	405	LEXINGTON AVENUE	10174	212
40363630	LORENZO & MARIA'S KITCHEN	Manhattan	1418	THIRD AVENUE	10028	212

Stored procedures can take input and output variables (and input/output!)

```
20    DELIMITER $$
21    CREATE PROCEDURE GetRestaurantsByBoro(
22        IN BoroName VARCHAR(20),
23        OUT RestaurantCount INT)
24    BEGIN
25        SELECT * FROM Restaurants WHERE boro = BoroName;
26        SELECT count(*) INTO RestaurantCount
27            FROM Restaurants WHERE boro = BoroName;
28    END$$
29    DELIMITER ;
30
31    CALL GetRestaurantsByBoro('Manhattan',@BoroCount);
32    SELECT @BoroCount;
33
```

100% 15:31

Result Grid | Filter Rows: Search | Export: | Fetch rows:

RestaurantID	RestaurantName	Boro	Building	Street	ZipCode
30191841	DJ REYNOLDS PUB AND RESTAU…	Manhattan	351	WEST 57 STREET	10019
40359480	1 EAST 66TH STREET KITCHEN	Manhattan	1	EAST 66 STREET	10065
40362264	P & S DELI GROCERY	Manhattan	730	COLUMBUS AVENUE	10025
40362274	ANGELIKA FILM CENTER	Manhattan	18	WEST HOUSTON STR…	10012
40362715	THE COUNTRY CAFE	Manhattan	60	WALL STREET	10005
40363298	CAFE METRO	Manhattan	625	8 AVENUE	10018
40363426	LEXLER DELI	Manhattan	405	LEXINGTON AVENUE	10174

Parameters

- **Can have multiple params**
- **Give name and domain**
- **IN – input, value not changed outside stored procedure**
- **OUT – output, value returned**
- **INOUT – input and output variable**

Declare stored procedure local variables with:
DECLARE varName TYPE
Put at top of BEGIN/END
NOTE: no @ before variable name for local variables

Stored procedures can take input and output variables (and input/output!)

```
20    DELIMITER $$
21    CREATE PROCEDURE GetRestaurantsByBoro(
22        IN BoroName VARCHAR(20),
23        OUT RestaurantCount INT)
24    BEGIN
25        SELECT * FROM Restaurants WHERE boro = BoroName;
26        SELECT count(*) INTO RestaurantCount
27            FROM Restaurants WHERE boro = BoroName;
28    END$$
29    DELIMITER ;
30
31    CALL GetRestaurantsByBoro('Manhattan',@BoroCount);
32    SELECT @BoroCount;
33
```

100% 15:31

Result Grid | Filter Rows: Search | Export: | Fetch rows:

RestaurantID	RestaurantName	Boro	Building	Street	ZipCode
30191841	DJ REYNOLDS PUB AND RESTAU…	Manhattan	351	WEST 57 STREET	10019
40359480	1 EAST 66TH STREET KITCHEN	Manhattan	1	EAST 66 STREET	10065
40362264	P & S DELI GROCERY	Manhattan	730	COLUMBUS AVENUE	10025
40362274	ANGELIKA FILM CENTER	Manhattan	18	WEST HOUSTON STR…	10012
40362715	THE COUNTRY CAFE	Manhattan	60	WALL STREET	10005
40363298	CAFE METRO	Manhattan	625	8 AVENUE	10018
40363426	LEXLER DELI	Manhattan	405	LEXINGTON AVENUE	10174

- This stored procedure takes BoroName as input, returns the number of Restaurants in the boro (12,079) in RestaurantCount
- Also outputs table of matching restaurants (as shown)
- To not return table, comment out first SELECT
- Can see value of @BoroCount with SELECT @BoroCount

Stored procedures also have statements like a traditional programming language

```
DELIMITER $$
CREATE PROCEDURE RestaurantInspectionCategory (
    IN RestaurantID INT,
    OUT Classification VARCHAR(20)
)
BEGIN
    DECLARE inspectionAvg INT;

    SELECT AVG(Score) INTO inspectionAvg
        FROM restaurant_inspections
        WHERE CAMIS = RestaurantID;
    IF inspectionAvg IS NULL THEN
        SET Classification = 'Uknown';
    ELSEIF inspectionAvg < 30 THEN
        SET Classification = 'Good';
    ELSEIF inspectionAvg > 100 THEN
        SET Classification = 'Sketchy';
    ELSE
        SET Classification = 'Ok';
    END IF;
END $$

DELIMITER ;

-- test
CALL RestaurantInspectionCategory(1111, @class); -- Tim's Tasty Treat's (no inspections)
SELECT @class; -- Unknown
```

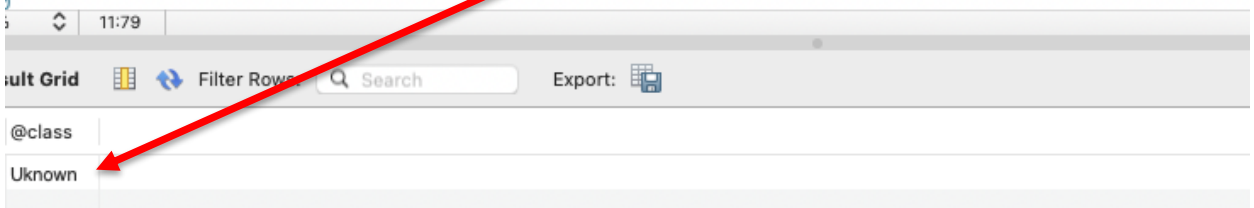
Input: RestaurantID (CAMIS)
Ouput: String based on avg inspection score

Local variables must be declared right after the BEGIN block with no @

Store avg inspection score for the restaurant parameter in inspectionAvg local variable

IF THEN ELSE block

Call with restaurant having no inspection scores (so inspectionAvg is NULL) returns Unknown



Stored procedures also have statements like a traditional programming language

Local variables

- Can declare local variables in stored procedures
- Cursors to get a results set (can iterate over)

Flow control

- CASE
- LOOP
- WHILE
- LEAVE (exits stored procedure)
- Structured error handling

We just scratched the surface today

- Stored procedures are not as capable as a traditional programming language
- But more capable than standard SQL

Practice

use nyc_inspections;

1. Create a stored procedure to return the min, max, avg, and count of inspection scores for a given restaurant ID
   - Hint, you'll need IN and OUT variables

2. Test your procedure on Morris Park Bake Shop at 1007 Morris Park Avenue

3. Double check your results are accurate!

Stored functions are like stored procedures but return one value

- Functions return one value
- Can be used anywhere a SQL expression can be used
- Can have parameters like stored procedures, but parameters can only be IN

```
40  CREATE FUNCTION IsBoro(
41      BoroName VARCHAR(20))
42  RETURNS Boolean
43  DETERMINISTIC
44  BEGIN
45      IF BoroName IN ('Manhattan','Brooklyn','Queens','Bronx','Staten Island') THEN
46          RETURN 1;
47      END IF;
48      RETURN 0;
49  END$$
50  DELIMITER ;
51
52  SELECT IsBoro('Queens') AS Queens, IsBoro('New Hampshire') AS NH;
53
54
```

100% 14:52

Result Grid Filter Rows: Search Export:

Queens	NH
1	0

Stored functions are like stored procedures but return one value

```
40  CREATE FUNCTION IsBoro(
41      BoroName VARCHAR(20))
42  RETURNS Boolean
43  DETERMINISTIC
44  BEGIN
45      IF BoroName IN ('Manhattan','Brooklyn','Queens','Bronx','Staten Island') THEN
46          RETURN 1;
47      END IF;
48      RETURN 0;
49  END$$
50  DELIMITER ;
51
52  SELECT IsBoro('Queens') AS Queens, IsBoro('New Hampshire') AS NH;
53
54
```

DETERMINISTIC means it will always return the same value for the same input

- **Allows database to cache results knowing they won't change**
- **"Assessment of the nature of a routine is based on the "honesty" of the creator"[1]**
- **Default is NOT DETERMINISTIC**
  - **example: uses date < CURDATE() or RAND()**

Functions must return a value in RETURN statement

100% 14:52

Result Grid Filter Rows: Search Export:

Queens	NH
1	0

[1] https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html

Practice

use nyc_inspections;

1. Create a function that classifies restaurants based on how many violations they have had during inspections
   Input: number of inspection violations
   Return:
   - 'Low' if fewer than 7 violations
   - 'Intermediate' if between 7 and 12 violation
   - 'High' if more than 12 violations

2. Use your function in a SELECT command to return each RestaurantName and its inspection classification (e.g., pass a count of violations to your function in the SELECT)
   Remember: CALL stored *procedures*, use stored *functions* in SELECT commands

Agenda

1. Stored procedures and functions
2. Transactions
3. Triggers

Transactions allow us to write multiple statements and treat them as atomic

Transactions

- A **transaction** consists of a sequence of query and/or update statements and is a "unit" of work
- The transaction must end with one of the following statements:
  - **COMMIT [work]** updates performed by the transaction become permanent in the database (durable)
  - **ROLLBACK [work]** updates performed by the SQL statements in the transaction are undone
- Atomic transaction
  - Either fully executed or rolled back as if it never occurred

MySQL example:

SET AUTOCOMMIT = 0;

START TRANSACTION;

<SQL Statements>

COMMIT (or **ROLLBACK**)

SET AUTOCOMMIT = 1;

In MySQL by default all statements executed immediately
Turn off auto commit

Begin atomic transaction

Commit makes updates permanent, if power failed before COMMIT statement, changes would not affect database (or could use ROLLBACK to cancel changes)

Turn autocommit back on

Example: ROLLBACK changes

```
187 •  SET AUTOCOMMIT = 0;
188 •  START TRANSACTION;
189 •      UPDATE Restaurants
190            SET RestaurantName = 'Tim''s Untasty Treats'
191            WHERE RestaurantID = 1111;
192 •  ROLLBACK;
193 •  SET AUTOCOMMIT = 1;
194
195 •  SELECT * FROM Restaurants WHERE RestaurantID = 1111;
196
```

Turn off auto commit

Start transaction, end with COMMIT or ROLLBACK

Update restaurant name to UNtasty Treats

ROLLBACK change

Database not changed

100% 18:195

Result Grid Filter Rows: Search Edit: Export/Import:

Restaur...	RestaurantName	Boro	Building	Street	ZipCode	Phone	Latitude	Longitude	CuisineDescription
1111	Tim's Tasty Treats	Manhattan	NULL	NULL	NULL	NULL	NULL	NULL	Fruits/Vegetables
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Restaurants 25

ction Output

One row affected by UPDATE

	Time	Action	Response
15...	13:09:21	START TRANSACTION	0 row(s) affected
15...	13:09:28	UPDATE Restaurants SET RestaurantName = 'Tim''s U...	1 row(s) affected Rows matched: 1 Changed: 1 Warnings: 0
15...	13:09:36	ROLLBACK	0 row(s) affected
15...	13:09:42	SET AUTOCOMMIT = 1	0 row(s) affected
15...	13:10:38	SELECT * FROM Restaurants WHERE RestaurantID = 111...	1 row(s) returned

Transactions allow us to write multiple statements and treat them as atomic

```
201 •   USE test_schema;
202 •   CREATE TABLE Accounts (
203         AccountID INT PRIMARY KEY,
204         Balance DECIMAL(10,2),  CHECK(Balance >=0)
205         );
206
207 •   INSERT INTO Accounts VALUES (1, 1000), (2, 2000);
208 •   SELECT * FROM Accounts;
```

100% 24:208

Result Grid Filter Rows: Search Edit: Ex

AccountID	Balance
1	1000.00
2	2000.00
NULL	NULL

Create Accounts table and add two accounts with balances $1000 and $2000

Transactions allow us to write multiple statements and treat them as atomic

Goal: transfer $500 from Account 1 to Account 2

	Account 1	Account 2
Start	Balance: $1000	Balance: $2000

Transactions allow us to write multiple statements and treat them as atomic

Goal: transfer $500 from Account 1 to Account 2

	Account 1	Account 2
Start	Balance: $1000	Balance: $2000
Subtract $500 from Account 1	**UPDATE** Accounts **SET** Balance = Balance – 500 **WHERE** AccountID=1; Balance: $500	

Transactions allow us to write multiple statements and treat them as atomic

Goal: transfer $500 from Account 1 to Account 2

	Account 1	Account 2
Start	Balance: $1000	Balance: $2000
Subtract $500 from Account 1	**UPDATE** Accounts **SET** Balance = Balance **–** 500 **WHERE** AccountID=1; Balance: $500	
Add $500 to Account2		**UPDATE** Accounts **SET** Balance = Balance **+** 500 **WHERE** AcountID=2; Balance: $2500

Transactions allow us to write multiple statements and treat them as atomic

Goal: transfer $500 from Account 1 to Account 2

	Account 1	Account 2
Start	Balance: $1000	Balance: $2000
Subtract $500 from Account 1	**UPDATE** Accounts **SET** Balance = Balance **–** 500 **WHERE** AccountID=Account1;	
	Balance: $500	
Add $500 to Account2		**UPDATE** Accounts **SET** Balance = Balance **+** 500 **WHERE** AcountID=Account2;
		Balance: $2500

What if database crashes here?
Account 1 debited, but Account 2 not credited
Loose $500!

What are possible outcomes?

- **No change to Account1 or Account2**
- **-500 to Account1, no change Account2**
- **-500 to Account1, +500 to Account2**

Wrap these commands in a transaction to ensure both complete (or neither complete)

Transactions allow us to write multiple statements and treat them as atomic

```
-- run both updates in a transaction
SET AUTOCOMMIT = 0;
START TRANSACTION;
    UPDATE Accounts
    SET Balance = Balance – 500
    WHERE AccountID=1;

    UPDATE Accounts
    SET Balance = Balance + 500
    WHERE AccountID=2;

COMMIT; -- commit both commands if successful
SET AUTOCOMMIT = 1;

SELECT * FROM Accounts;
```

Goal: transfer $500 from Account 1 to Account 2

Beginning balances

AccountID	Balance
1	500.00
2	2500.00

Changes are made to a redo log until a COMMIT or ROLLBACK is reached

- **If COMMIT – make changes permanent (durable)**
- **If ROLLBACK – undo changes in redo log**

If client disconnection – ROLLBACK
If power fails – ROLLBACK when power restored

Transactions allow us to write multiple statements and treat them as atomic

```
-- run both updates in a transaction
SET AUTOCOMMIT = 0;
START TRANSACTION;
    UPDATE Accounts
    SET Balance = Balance – 500
    WHERE AccountID=1;

    UPDATE Accounts
    SET Balance = Balance + 500
    WHERE AccountID=2;

COMMIT; -- commit both commands if successful
SET AUTOCOMMIT = 1;

SELECT * FROM Accounts;
```

Goal: transfer $500 from Account 1 to Account 2

Beginning balances

AccountID	Balance
1	500.00
2	2500.00

Ending balances

AccountID	Balance
1	0.00
2	3000.00

COMMIT makes changes permanent (durable)

Use stored procedure to make transaction automatically abort on any error

```
DELIMITER $$
CREATE PROCEDURE TransferMoney()
BEGIN
   -- 1. Declare the "Catch" logic first
   DECLARE EXIT HANDLER FOR SQLEXCEPTION
   BEGIN
      -- This block runs ONLY if an error occurs
      ROLLBACK; -- rollback on any error
      -- Optional: Log the error or send a custom message
   END;

   -- 2. The "Try" block: your actual logic
   START TRANSACTION;
   UPDATE Accounts -- Update 1
        SET Balance = Balance – 500
        WHERE AccountID=1;

        UPDATE Accounts -- Update 2
        SET Balance = Balance + 500
        WHERE AcountID=1;
   COMMIT; -- commit if both updates are successful
END $$
DELIMITER ;
```

Create "Catch" block that executes if there is an error in the transaction

MySQL has a conceptual try/catch in stored procedures

START TRANSACTION
Then multiple SQL statements
Followed by COMMIT
If all statements execute, changes saved, otherwise ROLLBACK in "Catch"

We will discuss transactions in more detail on Day 14, but this concept might be useful for your final project

Practice

use nyc_data;

Create a NewRestaurantInspection stored procedure that

- Makes a new entry in the Restaurants table
  - Take parameters:
    - RestaurantID
    - RestaurantName
- Makes an entry in the restaurant_inspections table
  - Take parameters:
    - InspectionDate
    - Score
- Make sure both commands succeed or neither command succeeds

- Test by inserting RestaurantID = 1, make sure both tables updated

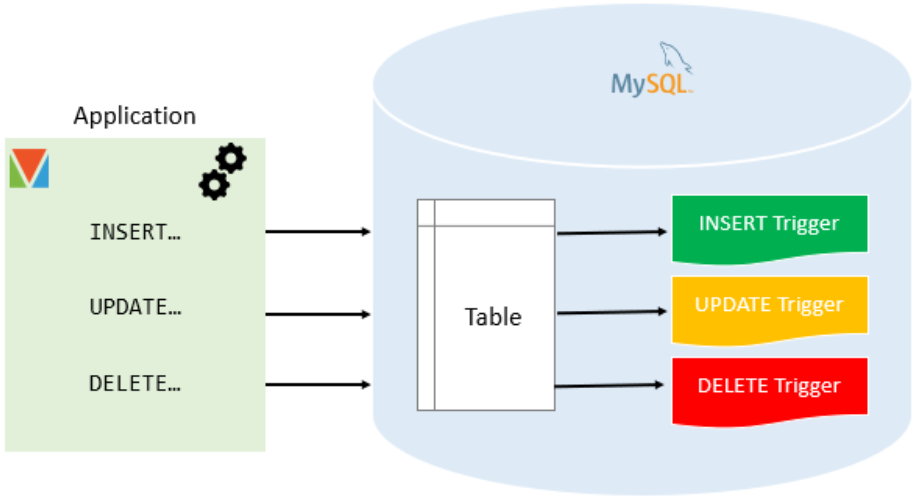
Agenda

1. Stored procedures and functions
2. Transactions
3. Triggers

Trigger fire in response to an event such as an INSERT, UPDATE, or DELETE on a table

A trigger is a stored program invoked automatically before or after an event such as:

- INSERT
- UPDATE
- DELETE



MySQL only supports row-level triggers

- If 100 rows inserted, updated, or deleted, trigger fires 100 times
- Other databases have statement-level triggers that fire once per statement

Like most things, triggers have pros and cons

Pros

- Triggers provide another way to check the integrity of data
- Triggers give an alternative way to run scheduled tasks:
  - No need to wait for scheduled cron jobs to run
  - Triggers are invoked automatically before or after a change is made to the data in a table
- Triggers can be useful for auditing the data changes in tables
  - Make an entry into an audit table when data is added, changed, or deleted
- Triggers can compute values on a change

Cons

- For simple validations, easier to use NOT NULL, UNIQUE, CHECK and FOREIGN KEY constraints
- Can be difficult to troubleshoot
  - Execute automatically in the database
  - May not be visible to client applications
- May increase processing overhead

Create trigger on inspection table changes to update statistics on Restaurant table

Goal: Keep avg score and count of inspections scores current in Restaurant table when Inspection table changes

What do we want to happen?

- Add columns to Restaurants for inspection count and average score
- Add new inspection to restaurant_inspections table for a specific restaurant
  - Increment count for that restaurant in Restaurants table
  - Re-compute average inspection score in Restaurants table
- Modify existing inspection in restaurant_inspections table for a specific restaurant
  - Re-compute average inspection score (keep count the same) in Restaurants table
- Delete inspection from restaurant_inspections table for a specific restaurant
  - Decrement count for that restaurant in Restaurants table
  - Re-compute average inspection score in Restaurants table
  - Set count and score to NULL if no inspections for this restaurant

Add columns for average score and inspection count in Restaurants

Goal: Keep avg score and count of inspections scores current in Restaurant table when Inspection table changes (e.g., if new Inspection entered, add one to count)

```
146     -- add InspectionAvg and InspectionCount to Restaurants
147 •   ALTER TABLE Restaurants ADD COLUMN InspectionAvg FLOAT;
148 •   ALTER TABLE Restaurants ADD COLUMN InspectionCount FLOAT;
149
150     -- notice there are two columns we created early to track the average inspection score and a count of inspection scores
151     -- for each restaurant.  These scores are currently NULL for each restaurant
152     -- we want to update these values when a new inspection is added, one is updated, or deleted
153 •   SELECT RestaurantID, RestaurantName, InspectionCount, InspectionAvg
154         FROM Restaurants; -- InspectionCount and InspectionAvgScores are all NULLs
155
```

Add columns for Avg Score and Count of inspections

Confirm NULLs

100% 17:154

Result Grid | Filter Rows: Search | Edit: | Export/Import: | Fetch rows:

Re...	RestaurantName	InspectionCou...	InspectionAvg
1111	Tim's Tasty Treats	NULL	NULL
30...	MORRIS PARK BAKE SHOP	NULL	NULL
30...	D.J. REYNOLDS	NULL	NULL
40...	RIVIERA CATERERS	NULL	NULL
40...	WILKEN'S FINE FOOD	NULL	NULL
40...	TASTE THE TROPICS ICE CREAM	NULL	NULL
40...	ASIA PLAZA CAFÉ	NULL	NULL

Fill new columns in Restaurants with avg score and count

Goal: Keep avg score and count of inspections scores current in Restaurant table when Inspection table changes (e.g., if new Inspection entered, add one to count)

```
156    -- can update counts and average score for all restaurants with the following command
157 •  UPDATE Restaurants r
158        SET InspectionCount = (SELECT count(*) from restaurant_inspections i WHERE i.CAMIS = r.RestaurantID),
159        InspectionAvg = (SELECT AVG(Score) from restaurant_inspections i WHERE i.CAMIS = r.RestaurantID);
160
161    -- check that update worked
162 •  SELECT RestaurantID, RestaurantName, InspectionCount, InspectionAvg
163        FROM Restaurants; -- InspectionCount and InspectionAvgScores are filled, Morris Park Bake Shop count 19, avg 19.6842 as seen before
164
```

Should have said COUNT(Score)!

Fill Avg Score and Count of inspections columns

100% 12:163

Result Grid | Filter Rows: Search | Edit: | Export/Import: | Fetch rows:

Re...	RestaurantName	InspectionCou...	InspectionAvg
1111	Tim's Tasty Treats	0	NULL
30...	MORRIS PARK BAKE SHOP	19	19.6842
30...	D.J. REYNOLDS	10	18.4
40...	RIVIERA CATERERS	4	10
40...	WILKEN'S FINE FOOD	20	23.65

Averages and counts

Tim's hasn't been inspected

Morris Park Bake Shop has 19 inspection results

Create trigger on Inspection table INSERT to update statistics on Restaurant table

Goal: Keep avg score and count of inspections scores current in Restaurant table when Inspection table changes (e.g., if new Inspection entered, add one to count)

```
-- create trigger on insert to restaurant_inspection table to keep count and avg score updated
-- in Restaurants table
DROP TRIGGER IF EXISTS UpdateInspectionStatsOnInsert;
DELIMITER $$
CREATE TRIGGER UpdateInspectionStatsOnInsert
    AFTER INSERT ON restaurant_inspections
    FOR EACH ROW
    BEGIN
        -- update both columns in Restaurants table
        UPDATE Restaurants
            SET InspectionAvg = (SELECT AVG(Score) FROM restaurant_inspections i WHERE i.CAMIS = NEW.CAMIS)
            WHERE RestaurantID = NEW.CAMIS;
        UPDATE Restaurants
            SET InspectionCount = (SELECT count(score) FROM restaurant_inspections i WHERE i.CAMIS = NEW.CAMIS)
            WHERE RestaurantID = NEW.CAMIS;
    END$$
DELIMITER ;
```

Give trigger a name

Can operate BEFORE or AFTER an INSERT, UPDATE, or DELETE on a specified table (restaurant_inspections)

SQL commands can reference the OLD or NEW values of an attribute

Now if a new inspection is inserted into the restaurant_inspections table, the avg score and count are BOTH updated in Restaurants table
Can do the same for UPDATES and DELETES

Create trigger on Inspection table DELETE to update statistics on Restaurant table

Goal: Keep avg score and count of inspections scores current in Restaurant table when Inspection table changes (e.g., if new Inspection entered, add one to count)

```
CREATE TRIGGER UpdateInspectionStatsOnDelete
    AFTER DELETE ON restaurant_inspections
    FOR EACH ROW
    BEGIN
        UPDATE Restaurants
            SET InspectionAvg = (SELECT AVG(Score) FROM restaurant_inspections i WHERE i.CAMIS = OLD.CAMIS)
            WHERE RestaurantID = OLD.CAMIS;
        UPDATE Restaurants
            SET InspectionCount = (SELECT IF (count(score) >0,count(score),NULL) FROM restaurant_inspections i WHERE i.CAMIS = OLD.CAMIS)
            WHERE RestaurantID = OLD.CAMIS;
    END$$

DELIMITER ;
```

Note: set count to NULL if no entries (not zero) using IF

If a row is deleted from restaurant_inspections, update Restaurant count and average score

NOTE: OLD refers to the row that was deleted

Create trigger on Inspection table UPDATE to update statistics on Restaurant table

Goal: Keep avg score and count of inspections scores current in Restaurant table when Inspection table changes (e.g., if new Inspection entered, add one to count)

```
-- create trigger on update to restaurant_inspection table to keep count and avg score updated
DROP TRIGGER IF EXISTS UpdateInspectionStatsOnUpdate;
DELIMITER $$
CREATE TRIGGER UpdateInspectionStatsOnUpdate
    AFTER UPDATE ON restaurant_inspections
    FOR EACH ROW
    BEGIN
        UPDATE Restaurants SET InspectionAvg = (SELECT IF (AVG(score) >0,AVG(score),NULL) FROM restaurant_inspections i WHERE i.CAMIS = NEW.CAMIS)
            WHERE RestaurantID = NEW.CAMIS;
        -- no need to change count on UPDATE
    END$$

DELIMITER ;
```

If a row is updated in restaurant_inspections, update Restaurant average score, but no need to update count

NOTE: NEW refers to the row that was updated

Test the triggers

```
217 •  INSERT INTO restaurant_inspections (CAMIS, Score) VALUES (1111,15); -- add inspection score for Tim's Tasty Treats
218 •  SELECT RestaurantID, RestaurantName, InspectionCount, InspectionAvg FROM Restaurants WHERE RestaurantName like 'Tim''s%'; -- count is 1, avg score is 15
```

100% 20:218

Result Grid Filter Rows: Search Edit: Export/Import:

Re...	RestaurantName	InspectionCou...	InspectionAvg
1111	Tim's Tasty Treats	1	15
NULL	NULL	NULL	NULL

Add inspection for Tim's Tasty Treats sets count =1 and avg score =15

Test the triggers

```
217 •  INSERT INTO restaurant_inspections (CAMIS, Score) VALUES (1111,15); -- add inspection score for Tim's Tasty Treats
218 •  SELECT RestaurantID, RestaurantName, InspectionCount, InspectionAvg FROM Restaurants WHERE RestaurantName like 'Tim''s%'; -- count is 1, avg score is 15
```

100% 20:218

Result Grid Filter Rows: Search Edit: Export/Import:

Re...	RestaurantName	InspectionCou...	InspectionAvg
1111	Tim's Tasty Treats	1	15
NULL	NULL	NULL	NULL

```
220 •  UPDATE restaurant_inspections SET Score = 20 WHERE CAMIS = 1111; -- update score to 20
221 •  SELECT RestaurantID, RestaurantName, InspectionCount, InspectionAvg FROM Restaurants WHERE RestaurantName like 'Tim''s%'; -- count is 1, avg score is 20
```

100% 25:221

Result Grid Filter Rows: Search Edit: Export/Import:

Re...	RestaurantName	InspectionCou...	InspectionAvg
1111	Tim's Tasty Treats	1	20
NULL	NULL	NULL	NULL

Updating score in restaurant_inspections updates avg score in Restaurants, keeps same count

Test the triggers

```
217 •  INSERT INTO restaurant_inspections (CAMIS, Score) VALUES (1111,15); -- add inspection score for Tim's Tasty Treats
218 •  SELECT RestaurantID, RestaurantName, InspectionCount, InspectionAvg FROM Restaurants WHERE RestaurantName like 'Tim''s%'; -- count is 1, avg score is 15
```

100% 20:218

Result Grid Filter Rows: Search Edit: Export/Import:

Re...	RestaurantName	InspectionCou...	InspectionAvg
1111	Tim's Tasty Treats	1	15
NULL	NULL	NULL	NULL

```
220 •  UPDATE restaurant_inspections SET Score = 20 WHERE CAMIS = 1111; -- update score to 20
221 •  SELECT RestaurantID, RestaurantName, InspectionCount, InspectionAvg FROM Restaurants WHERE RestaurantName like 'Tim''s%'; -- count is 1, avg score is 20
```

100% 25:221

Result Grid Filter Rows: Search Edit: Export/Import:

Re...	RestaurantName	InspectionCou...	InspectionAvg
1111	Tim's Tasty Treats	1	20
NULL	NULL	NULL	NULL

```
223 •  INSERT INTO restaurant_inspections (CAMIS, Score) VALUES (1111,10); -- add inspection score for Tim's Tasty Treats
224 •  SELECT RestaurantID, RestaurantName, InspectionCount, InspectionAvg FROM Restaurants WHERE RestaurantName like 'Tim''s%'; -- count is 2, avg score is 15 ((20+10)/2)
```

100% 29:224

Result Grid Filter Rows: Search Edit: Export/Import:

Re...	RestaurantName	InspectionCou...	InspectionAvg
1111	Tim's Tasty Treats	2	15
NULL	NULL	NULL	NULL

Inserting another inspection increase count to 2 and updates average score

Test the triggers

```
217 •  INSERT INTO restaurant_inspections (CAMIS, Score) VALUES (1111,15); -- add inspection score for Tim's Tasty Treats
218 •  SELECT RestaurantID, RestaurantName, InspectionCount, InspectionAvg FROM Restaurants WHERE RestaurantName like 'Tim''s%'; -- count is 1, avg score is 15
```

100% 20:218

Result Grid Filter Rows: Search Edit: Export/Import:

Re...	RestaurantName	InspectionCou...	InspectionAvg
1111	Tim's Tasty Treats	1	15
NULL	NULL	NULL	NULL

```
220 •  UPDATE restaurant_inspections SET Score = 20 WHERE CAMIS = 1111; -- update score to 20
221 •  SELECT RestaurantID, RestaurantName, InspectionCount, InspectionAvg FROM Restaurants WHERE RestaurantName like 'Tim''s%'; -- count is 1, avg score is 20
```

100% 25:221

Result Grid Filter Rows: Search Edit: Export/Import:

Re...	RestaurantName	InspectionCou...	InspectionAvg
1111	Tim's Tasty Treats	1	20
NULL	NULL	NULL	NULL

```
223 •  INSERT INTO restaurant_inspections (CAMIS, Score) VALUES (1111,10); -- add inspection score for Tim's Tasty Treats
224 •  SELECT RestaurantID, RestaurantName, InspectionCount, InspectionAvg FROM Restaurants WHERE RestaurantName like 'Tim''s%'; -- count is 2, avg score is 15 ((20+10)/2)
```

100% 29:224

Result Grid Filter Rows: Search Edit: Export/Import:

Re...	RestaurantName	InspectionCou...	InspectionAvg
1111	Tim's Tasty Treats	2	15
NULL	NULL	NULL	NULL

```
231 •  DELETE FROM restaurant_inspections WHERE CAMIS = 1111;
232 •  SELECT RestaurantID, RestaurantName, InspectionCount, InspectionAvg FROM Restaurants WHERE RestaurantName like 'Tim''s%'; -- count is null, avg score is null
```

100% 17:232

Result Grid Filter Rows: Search Edit: Export/Import:

Re...	RestaurantName	InspectionCou...	InspectionAvg
1111	Tim's Tasty Treats	NULL	NULL
NULL	NULL	NULL	NULL

Deleting all scores resets count and average to NULL

Practice

use nyc_data;

You're wondering if someone is paying off Health Inspectors to change inspection scores. You would like to log any changes to scores made in the Inspections table

1. Create an Audit table where we can log changes, include columns for:
   - The table that was changed (here always restaurant_inspections)
   - The CAMIS and InspectionDate of the row that was changed
   - The attribute that was changed (here always Score)
   - The score value before the change (e.g., score was a 5)
   - The score value after the change (e.g., score is now a 4)
   - The user that made the change (use the USER() function)
   - The date and time the change was made (look at CURRENT_TIMESTAMP())
2. Create a trigger that fires each time any score is updated in restaurant_inspections
3. To test, update InspectionID 26070 (Morris Park Bake Shop) from a score of 5 to a score of 4
4. Check your Audit table and confirm this change was logged
5. Are there any advantages to logging the change with a trigger vs. writing an entry into the Audit table with a user application?