

INSERT, UPDATE, DELETE,  
TRANSACTIONS, CONSTRAINTS,  
AND REFERENTIAL INTEGRITY

# TODAY'S OBJECTIVES

- Inserts
- Deletes
- Updates
- Constraints and referential integrity
- Transactions

# INSERT

```
INSERT INTO table_name (column1, column2,  
..., column_n) VALUES (value1, value2,  
... value_n);
```

Example:

```
INSERT INTO client (client_id, first_name, last_name,  
active) VALUES (1, 'Molly', 'McButter', true);
```

# INSERT

```
INSERT INTO table_name VALUES (value1,  
value2, ... value_n);
```

Example:

```
INSERT INTO client VALUES (2, 'Johnny',  
'McMargarine', false);
```

# INSERT USING SELECT

```
INSERT INTO table_name (column1, column2,  
..., column_n) [SELECT STATEMENT]
```

Example:

```
INSERT INTO park_state  
SELECT park_id, 'CA' FROM park WHERE park_name = 'Disneyland';
```

# UPDATE

```
UPDATE table_name  
SET column1 = value1, column2 = value2,  
...  
WHERE condition;
```

Example:

```
UPDATE client SET last_name = 'Butters' WHERE  
client_id=2;
```

# UPDATE

**Can use subquery in WHERE clause.**

Example:

```
UPDATE client SET last_name = 'Butters' WHERE client_id =  
(SELECT client_id FROM client WHERE first_name='Johnny');
```

# DELETE

```
DELETE FROM table_name  
WHERE column=value;
```

Example:

```
DELETE FROM client WHERE client_id=2;
```



DELETE

**Don't forget the WHERE clause!!!**

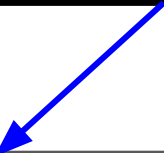
**Doing so can have *disastrous* consequences.**

# REFERENTIAL INTEGRITY

**What the heck is  
Referential Integrity?**

# REFERENTIAL INTEGRITY

```
DELETE FROM state WHERE state_abbreviation = 'TX';
```



city_id [PK] integer	city_name character varying (50)	state_abbreviation character (2)	population integer	area numeric (5,1)
1	Abilene	TX	123420	276.4
2	Akron	OH	197597	160.6
3	Albany	NY	96460	56.8
4	Albuquerque	NM	560513	487.4
5	Alexandria	VA	159428	38.8
6	Allen	TX	105623	70.2
7	Allentown	PA	121442	45.3
8	Amarillo	TX	199371	262.6
9	Anaheim	CA	350365	129.5
10	Anchorage	AK	288000	4420.1

# REFERENTIAL INTEGRITY

**Referential integrity** ensures that relationships between tables remain consistent.

- We enforce referential integrity and other rules by applying constraints to our tables.

# TABLE CONSTRAINTS

A **constraint** is associated with a table and defines properties that the column data must comply with.

<b>NOT NULL</b>	Requires data in column.
<b>UNIQUE</b>	Requires data in column to be unique.
<b>PRIMARY KEY</b>	Allows FKs to establish a relationship, and enforces NOT NULL and UNIQUE.
<b>FOREIGN KEY</b>	Enforces valid PK values, and limits deletion of the PK row if FK row exists.
<b>CHECK</b>	Specifies acceptable values that can be entered in the column.
<b>DEFAULT</b>	Provides a default value for the column.

# INTRODUCING TRANSACTIONS

# TRANSACTIONS

If we transfer money from one bank account to another and there's a failure depositing it after it withdraws, we wouldn't want our account to be out the money too and we would want the withdrawal from the original account to be reverted.

Even though there are two steps to the process (withdrawing and depositing), the process functions as a whole and must either complete successfully or be undone. In database-speak, this would be a **transaction**.

# THE ACID TEST

The ACID test can be used to determine whether a series of actions should be written as a transaction. A scenario implemented as a transaction should have the following characteristics:

- **Atomicity**: Within a transaction, a series of database operations all occur or none occur.
- **Consistency**: The completed transaction leaves things remaining in a consistent state at the end. Any rules in place before the transaction still pass after the transaction.
- **Isolation**: Ensures that the concurrent execution of a transaction results as if the operations were executed serially.
- **Durability**: Once a transaction has been committed it will remain so, even during a power loss, crash, or an error.



# TRANSACTIONS IN POSTGRES

- We can start a transaction by using **START TRANSACTION** before a set of SQL statements.
- Once we have completed all the statements successfully, we can **COMMIT** the transaction.
- If something goes wrong or we change our mind BEFORE committing, we can **ROLLBACK** the transaction.

# TRANSACTIONS IN POSTGRES

Example:

```
SELECT * FROM client; // Before we start we have 2 rows
START TRANSACTION; // Initiate transaction
INSERT INTO client VALUES (3, 'Howdy', 'Doodey', true); // Insert data
SELECT * FROM client; // We now have 3 rows but they have not been permanently saved
ROLLBACK; // If we use ROLLBACK, any changes from start of transaction will be reverted
SELECT * FROM client; // We now have 2 rows as we did before we inserted.
```

If we had used **COMMIT** rather than **ROLLBACK**, the data would have permanently saved. Once we use either **COMMIT** or **ROLLBACK**, the transaction is closed, so if we want to be able to **ROLLBACK** again, we must start a new transaction.