SQL Data Definition Language (DDL)

CSCI 220: Database Management and Systems Design

Today you will learn:

- How to implement a relational model using SQL

Database Changes

- How to implement your relational schema in the DBMS?
- Using the SQL DDL (Structured Query Language Data Definition Language)
- Create the Loan table:
```
CREATE TABLE loan (id INTEGER, amount MONEY)
PRIMARY KEY (id);
```
- Insert into the Loan table:
```
INSERT INTO loan (id, amount) VALUES (1, 100.00);
```

Future: Database Queries

- How to retrieve records from a database?
- Using the SQL DML (Structured Query Language Data Manipulation Language)
- Find the record for the customer with ID 111:

```
SELECT *
FROM loan
WHERE loan.id = 111;
```

- Supports sorting, queries across tables, computing averages, etc.
- Your SQL query tells the database what you want. The database (usually) retrieves the results as efficiently as possible.

Overview of DDL Operations

Operation	Statement
Create table	CREATE TABLE <name> (<field> <domain>, …)
Drop table	DROP TABLE <name>
Insert row into table	INSERT INTO <name> (<field names>) VALUES (<field values>)
Delete row from table	DELETE FROM <name> WHERE <condition>
Update rows in table	UPDATE <name> SET <field name> = <value> WHERE <condition>

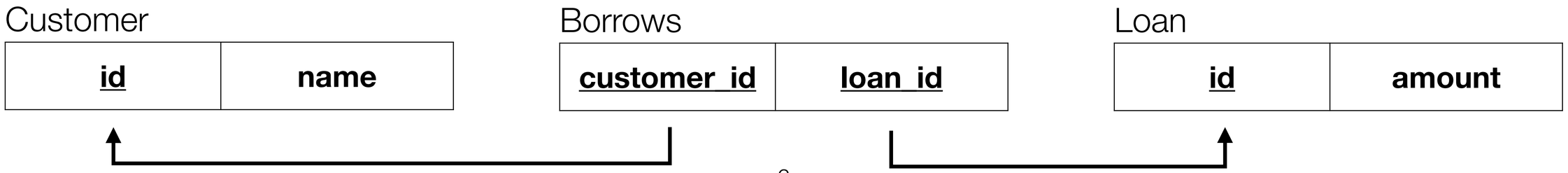
CREATE TABLE

- Create a table, specifying columns, and constraints:

- `CREATE TABLE customer (id INTEGER PRIMARY KEY, name TEXT);`

- `CREATE TABLE loan (id INTEGER PRIMARY KEY, amount MONEY);`

- ```
  CREATE TABLE borrows (
  customer_id INTEGER,
  loan_id INTEGER,
  PRIMARY KEY (customer_id, loan_id),
  FOREIGN KEY (customer_id) REFERENCES customer(id),
  FOREIGN KEY (loan_id) REFERENCES loan(id));
  ```


  ```

DROP TABLE

- Remove a table and its records:
- `DROP TABLE customer;`
- `DROP TABLE loan;`
- `DROP TABLE borrows;`

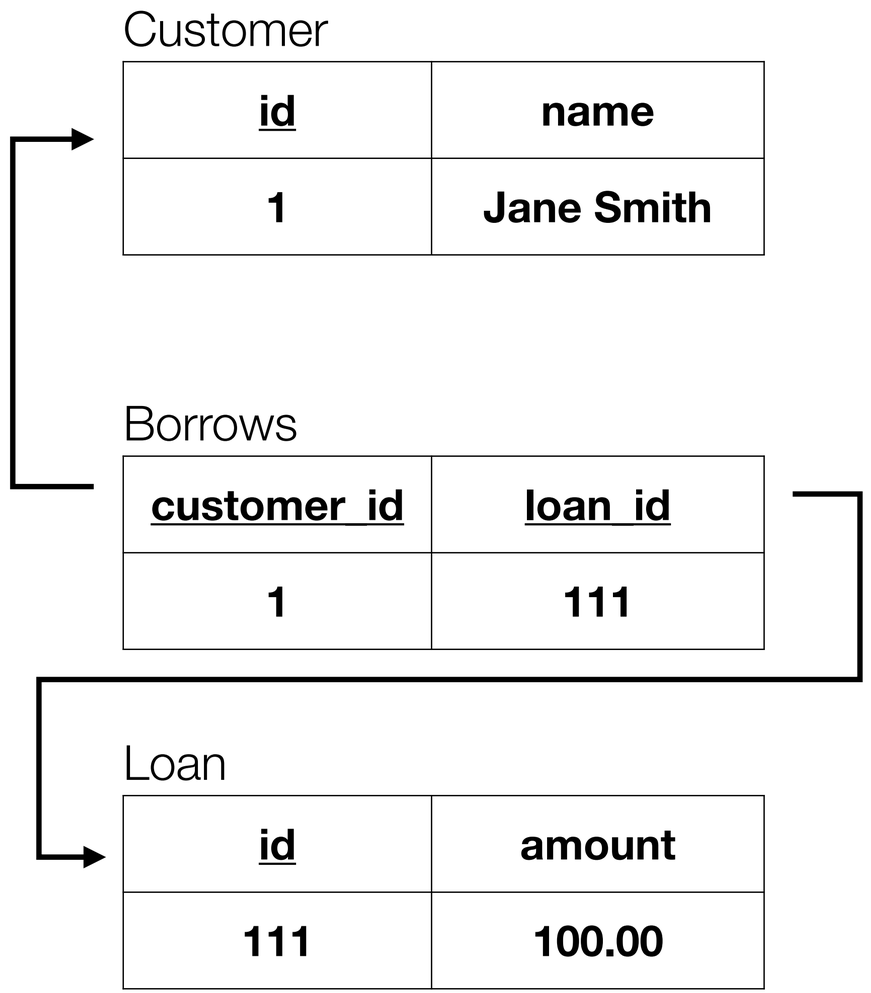
INSERT INTO

- Insert records into tables:

```
• INSERT INTO customer (id, name)
  VALUES (1, 'Jane Smith');

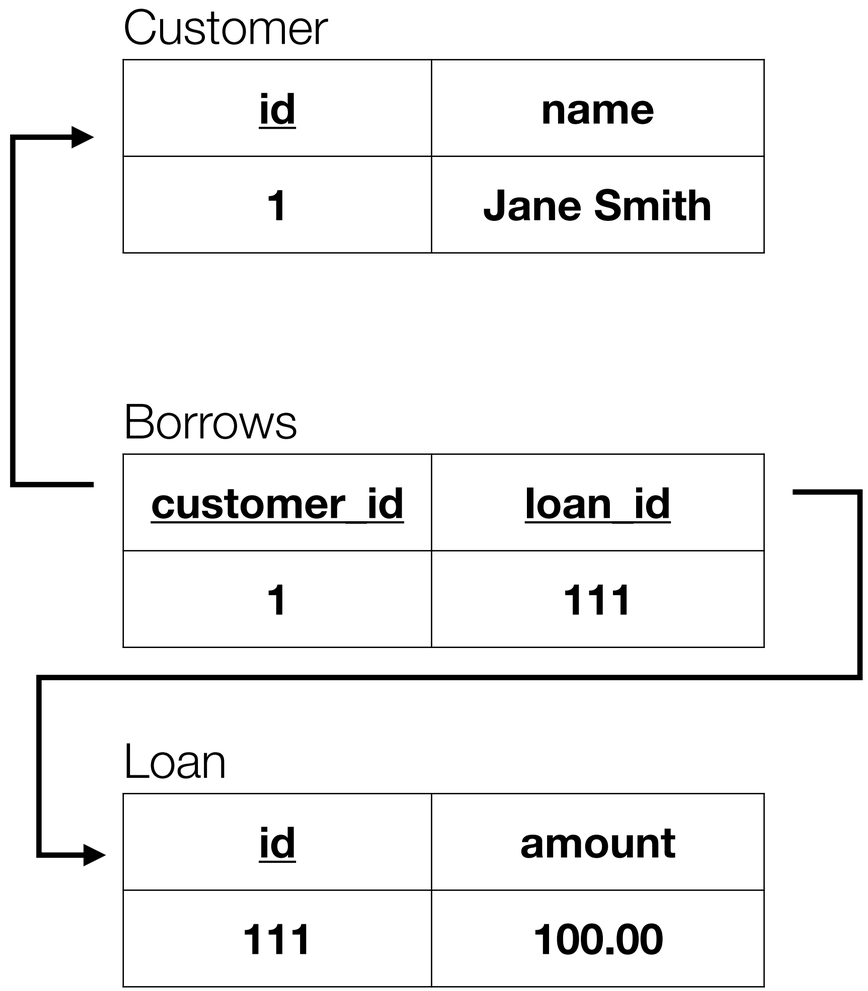
• INSERT INTO loan (id, amount)
  VALUES (111, 100.00);

• INSERT INTO
  borrows (customer_id, loan_id)
  VALUES (1, 111);
```



DELETE FROM

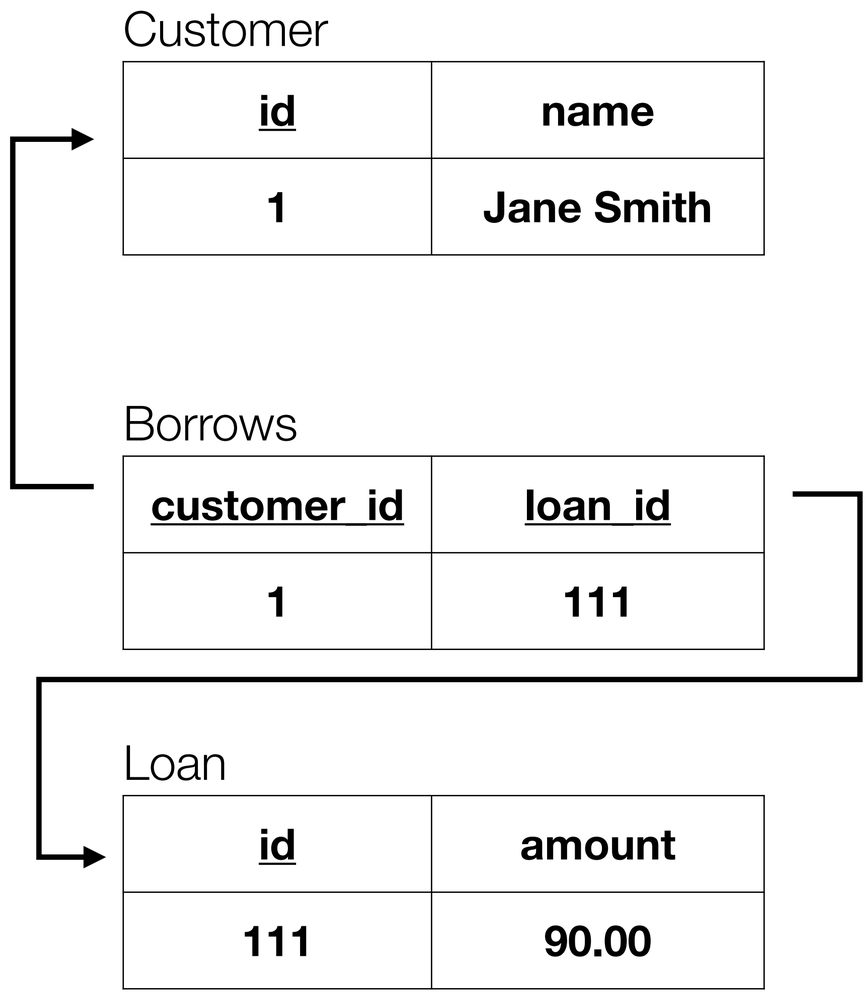
- Delete records from tables:
- `DELETE FROM borrows;`
- ```
  DELETE FROM borrows
  WHERE customer_id = 1;
  ```


  ```

UPDATE

- Update records:

- ```
  UPDATE loan
  SET amount = 90
  WHERE id = 111;
  ```


  ```

SELECT

- Useful to check which records you've inserted:
- `SELECT * FROM customer;`
- `SELECT * FROM borrows;`
- `SELECT * FROM loan;`
- Not our focus today. In another lecture, we'll see many advanced options.

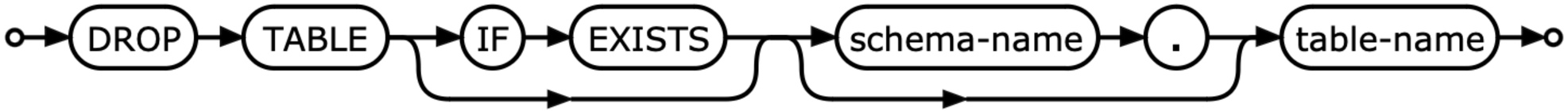
SQL vs DBMS Meta Commands

- SQL is (mostly) standardized
- DBMSs also offer meta commands, which are used to interact with the DBMS command-line. For example, in SQLite:
  - `.help`
  - `.tables`
  - `.read <my_file.sql>`

SQL Documentation

DROP TABLE

drop-table-stmt: hide



The DROP TABLE statement removes a table added with the CREATE TABLE statement. The name specified is the table name. The dropped table is completely removed from the database schema and the disk file. The table can not be recovered. All indices and triggers associated with the table are also deleted.

The optional IF EXISTS clause suppresses the error that would normally result if the table does not exist.

If foreign key constraints are enabled, a DROP TABLE command performs an implicit DELETE FROM command before removing the table from the database schema. Any

https://www.sqlite.org/lang_droptable.html

SQLite Quirks

- We will use SQLite in lab because it is easy to set up
- However, SQLite violates the SQL standard in significant ways
  - It doesn't check many constraints by default! For example:
  - "SQLite provides developers with the freedom to store content in any desired format, regardless of the declared datatype of the column." "As far as we know, SQLite is the only SQL database engine that supports this ***advanced*** capability."
  - …maybe because other DBMSs think it's a bad idea? 😉

SQLite Workarounds

- Until we switch to PostgreSQL, you should use workarounds to make SQLite behave like a typical DMBS:
  - Use the STRICT keyword to create STRICT tables which enforce types
  - Enforce foreign key constraints by running this statement:
    ```
    PRAGMA foreign_keys = ON;
    ```
- IMHO, this is absolute madness! We will use PostgreSQL for the course project.

SQLite Workarounds

```
$ sqlite3 test.db
sqlite> PRAGMA foreign_keys = ON;
sqlite> CREATE TABLE customer
   ...> (id INTEGER PRIMARY KEY, name TEXT) STRICT;
```

Troubleshooting SQL

PE

You

Why does this SQL statement give me a syntax error? CREATE TABLE borrows (ssn INTEGER, loan_id INTEGER,PRIMARY KEY (ssn, loan_id));

ChatGPT

The SQL statement you provided seems to be correctly formatted, but there might be an issue with the non-visible characters. It's possible that the line breaks or other hidden characters are causing the syntax error.

Try copying and pasting the following version into your SQL environment:

sql Copy code

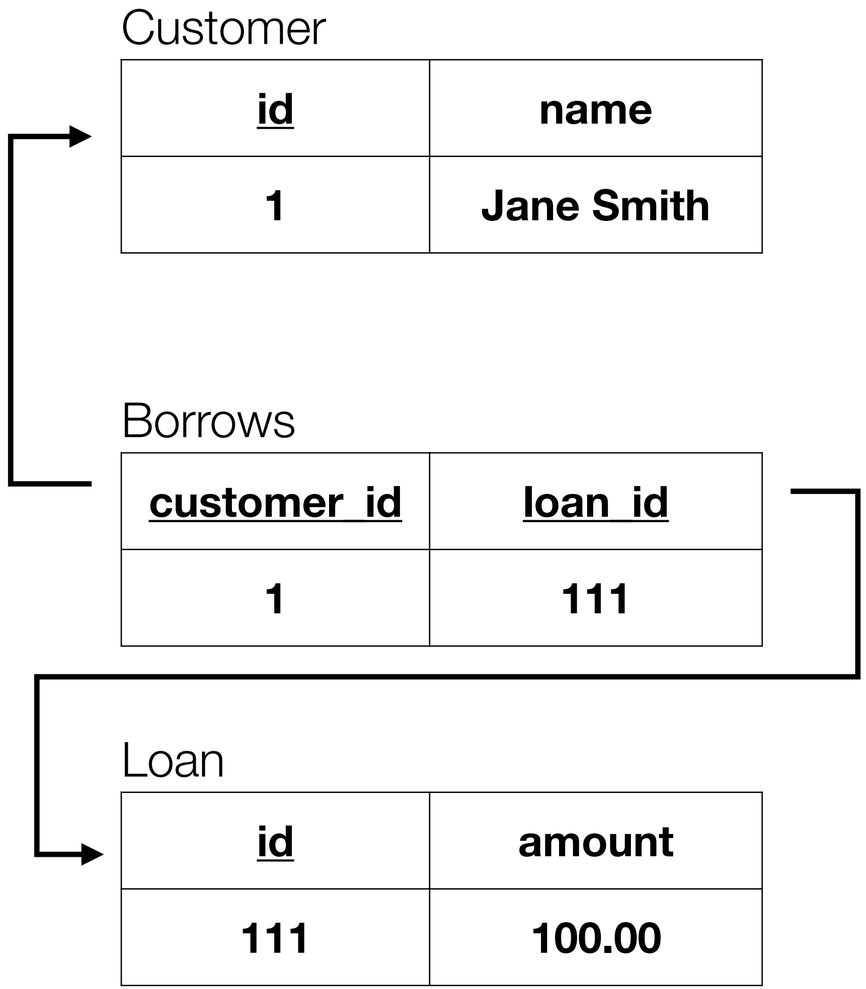
```sql
CREATE TABLE borrows (
    ssn INTEGER,
    loan_id INTEGER,
    PRIMARY KEY (ssn, loan_id)
);
```

Advanced SQL DDL

CSCI 220: Database Management and Systems Design

Practice Quiz: Constraints

- Give examples of operations that would violate each of these constraints:
  - Primary key constraint
  - Entity integrity constraint
  - Referential integrity constraint
  - Domain constraint



Today you will learn:

- How to use advanced SQL DDL features to enforce constraints

Constraints

- Primary key constraints
- Entity integrity constraints
- Referential integrity constraints
  - **New: Handling deletion**
- Domain constraints
  - **New: Custom checks**
  - **New: Custom domains**
- **New: Global constraints**

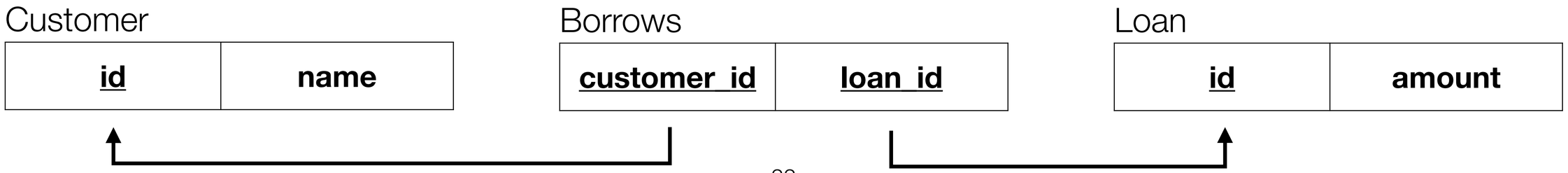
Review: CREATE TABLE

- Create a table, specifying columns, and constraints:

- `CREATE TABLE customer (id INTEGER PRIMARY KEY, name TEXT);`

- `CREATE TABLE loan (id INTEGER PRIMARY KEY, amount MONEY);`

- ```
  CREATE TABLE borrows (
  customer_id INTEGER,
  loan_id INTEGER,
  PRIMARY KEY (customer_id, loan_id),
  FOREIGN KEY (customer_id) REFERENCES customer(id),
  FOREIGN KEY (loan_id) REFERENCES loan(id));
  ```


  ```

Improvements

- Loan amounts must be non-negative and non-null
- Add (unique) email attribute
- Allow deletion of loans
- Only allow customers to hold a maximum of 5 loans

Add CHECK Constraints

- Ensure that loan amounts are non-negative and non-null

- ```
  ALTER TABLE loan
  ADD CONSTRAINT loan_amount_non_neg
  CHECK (amount >= '$0'::MONEY);
  ```

- ```
  ALTER TABLE loan
  ALTER COLUMN amount SET NOT NULL;
  ```

View CHECK Constraints

```
# \d loan
                  Table "public.loan"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           | not null |
 amount | money   |           | not null |
Indexes:
    "loan_pkey" PRIMARY KEY, btree (id)
Check constraints:
    "loan_amount_non_neg" CHECK (amount >= '$0.00'::money)
```

Remove CHECK Constraints

- ```
  ALTER TABLE loan
  DROP CONSTRAINT loan_amount_non_neg;
  ```

- ```
  ALTER TABLE loan
  ALTER COLUMN amount DROP NOT NULL;
  ```

Add Custom DOMAINs

- Ensure that loan amounts are non-negative and non-null

- ```
  CREATE DOMAIN balance AS MONEY
  NOT NULL
  CHECK (VALUE >= '$0'::MONEY);
  ```

- ```
  ALTER TABLE loan
  ALTER COLUMN amount TYPE balance;
  ```

View DOMAINs

```
django=# \dD
                                    List of domains
 Schema |  Name   | Type  | Collation | Nullable | Default |                Check
--------+---------+-------+-----------+----------+---------+-------------------------------
 public | balance | money |           | not null |         | CHECK (VALUE >= '$0.00'::money)
(1 row)
```

```
# \d loan
                  Table "public.loan"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           | not null |
 amount | balance |           |          |
```

Remove DOMAINs

- ```
  ALTER TABLE loan
  ALTER COLUMN amount TYPE money;
  ```

- ```
  DROP DOMAIN balance;
  ```

Add Columns

- Add an email column

- ```
  CREATE EXTENSION citext;
  ```

- ```
  CREATE DOMAIN email AS citext
  CHECK ( value ~ '^[a-zA-Z0-9.!#$%&''*+/=?^_`{|}~-]
  +@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\.
  [a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)*$' );
  ```

- ```
  ALTER TABLE customer
  ADD email email UNIQUE;
  ```

https://dba.stackexchange.com/a/165923
  ```

View Columns

```
django=# \d customer;
                 Table "public.customer"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           | not null |
 name   | text    |           |          |
 email  | email   |           |          |
Indexes:
    "customer_pkey" PRIMARY KEY, btree (id)
    "customer_email_key" UNIQUE CONSTRAINT, btree (email)
```

Remove Columns

- ALTER TABLE customer
  DROP email;

- DROP DOMAIN email;

Foreign Keys

- Borrows has FKs to customer and loan
- By default, deletions of referenced loans are rejected
- Other options:
  - Set FKs to NULL
  - Cascading deletion

```
django=# DELETE FROM loan WHERE id = 111;
ERROR:  update or delete on table "loan" violates foreign key constraint
"borrows_loan_id_fkey" on table "borrows"
DETAIL:  Key (id)=(111) is still referenced from table "borrows".
```

View Foreign Keys

```
django=# \d borrows
                  Table "public.borrows"
   Column    |  Type   | Collation | Nullable | Default
-------------+---------+-----------+----------+---------
 customer_id | integer |           | not null |
 loan_id     | integer |           | not null |
Indexes:
    "borrows_pkey" PRIMARY KEY, btree (customer_id, loan_id)
Foreign-key constraints:
    "borrows_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customer(id)
    "borrows_loan_id_fkey" FOREIGN KEY (loan_id) REFERENCES loan(id)
```

Modify Foreign Keys

- ```
  ALTER TABLE borrows
  DROP CONSTRAINT borrows_loan_id_fkey;
  ```

- ```
  ALTER TABLE borrows
  ADD CONSTRAINT borrows_loan_id_fkey
  FOREIGN KEY (loan_id) REFERENCES loan(id)
  ON DELETE CASCADE;
  ```

Create Triggers

- Only allow Customers to hold up to 5 Loans

```
CREATE OR REPLACE FUNCTION check_borrows_count()
RETURNS TRIGGER AS $$
BEGIN
  IF (SELECT COUNT(*) FROM borrows WHERE customer_id = NEW.customer_id) >= 5
THEN
    RAISE EXCEPTION 'A customer can hold a maximum of 5 loans';
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER before_insert_borrows
BEFORE INSERT ON borrows
FOR EACH ROW
EXECUTE FUNCTION check_borrows_count();
```

View Triggers

```
django=# \dft
                                List of functions
 Schema |         Name           | Result data type | Argument data types | Type
--------+------------------------+------------------+---------------------+------
 public | check_borrows_count | trigger          |                     | func
(1 row)

django=# \d borrows
                  Table "public.borrows"
   Column    |  Type   | Collation | Nullable | Default
-------------+---------+-----------+----------+---------
 customer_id | integer |           | not null |
 loan_id     | integer |           | not null |
...
Triggers:
    before_insert_borrows BEFORE INSERT ON borrows FOR EACH ROW EXECUTE FUNCTION
check_borrows_count()
```

Remove Triggers

- `DROP TRIGGER before_insert_borrows ON borrows;`
- `DROP FUNCTION check_borrows_count;`

Warning About Triggers

- Hard to implement correctly
  - We should have made the trigger run on INSERT **or UPDATE!**
- They can negatively affect performance
  - Triggers can implement global constraints (i.e., constraints which check an arbitrary number of rows and tables)

Constraint Enforcement

Constraint	Performance Cost	Explanation
Domain	Low	Type check
Entity Integrity	Low	NULL check
Referential Integrity	Low	Should use an index
(Primary) Key	Moderate	Maintain and use an index
Global	Low to High	Arbitrary checks

Miscellaneous SQL DDL

Create Views

- Use a view to calculate a customer's total debt conveniently

```
CREATE VIEW debt_view AS
SELECT customer.id, customer.name, SUM(amount) as debt
FROM customer
JOIN borrows ON customer.id = borrows.customer_id
JOIN loan ON borrows.loan_id = loan.id
GROUP BY customer.id;
```

Use Views

```
django=# \dv
          List of relations
 Schema |   Name     | Type | Owner
--------+------------+------+--------
 public | debt_view | view | django
(1 row)


django=# SELECT * FROM debt_view;
 id |    name     |    debt
----+-------------+------------
  1 | Jane Smith |    $100.00
  2 | Bill Gates | $10,000.00
(2 rows)
```

Remove Views

- `DROP VIEW debt_view;`

SERIAL

- How to assign unique identifiers to records?
  - For example: customer.id, loan.id, etc.
- In PostgreSQL:

```
CREATE TABLE customer (id SERIAL PRIMARY KEY, name TEXT);

INSERT INTO customer (id, name) VALUES (DEFAULT, 'Jane Smith');
INSERT INTO customer (name) VALUES ('John Smith');
```

- SQLite uses AUTOINCREMENT and ROWID to similar effect

INTO

- Copy data into a new table:

```
SELECT *
INTO customer_2024-1-1_bak
FROM customer
```

- Not supported by SQLite, but equivalent to:

```
CREATE TABLE customer_2024-1-1_bak AS
SELECT *
FROM customer
```