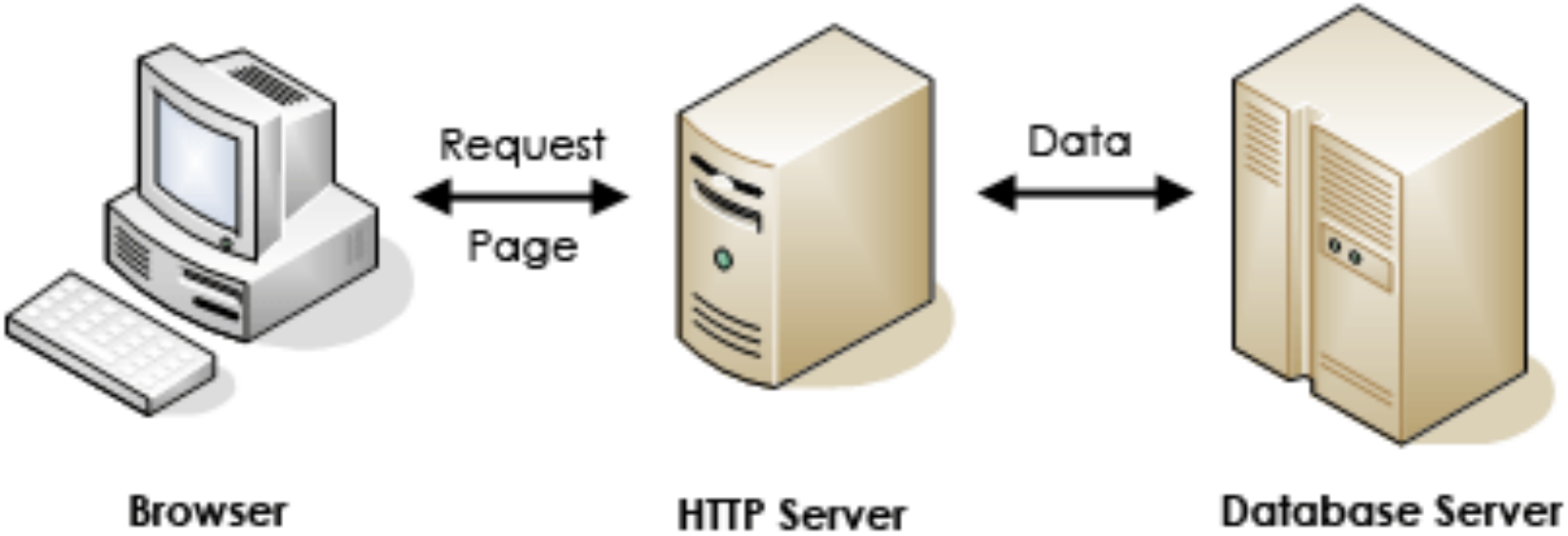
Databases

Three Tier Architecture



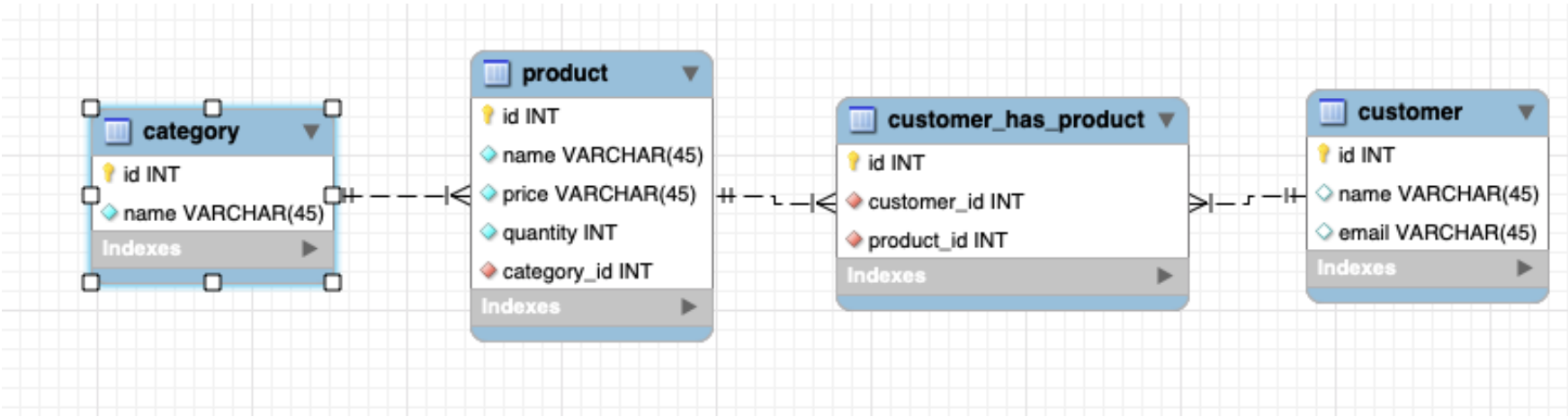
RDBMS

- Relational Database Management Systems
- A way of saving and accessing data on persistent (disk) storage.

Why Use an RDBMS

- Data Safety
  - data is immune to program crashes
- Concurrent Access
  - atomic updates via transactions
- Fault Tolerance
  - replicated dbs for instant fail-over on machine/disk crashes
- Data Integrity
  - aids to keep data meaningful
- Scalability
  - can handle small/large quantities of data in a uniform manner
- Reporting
  - easy to write SQL programs to generate arbitrary reports

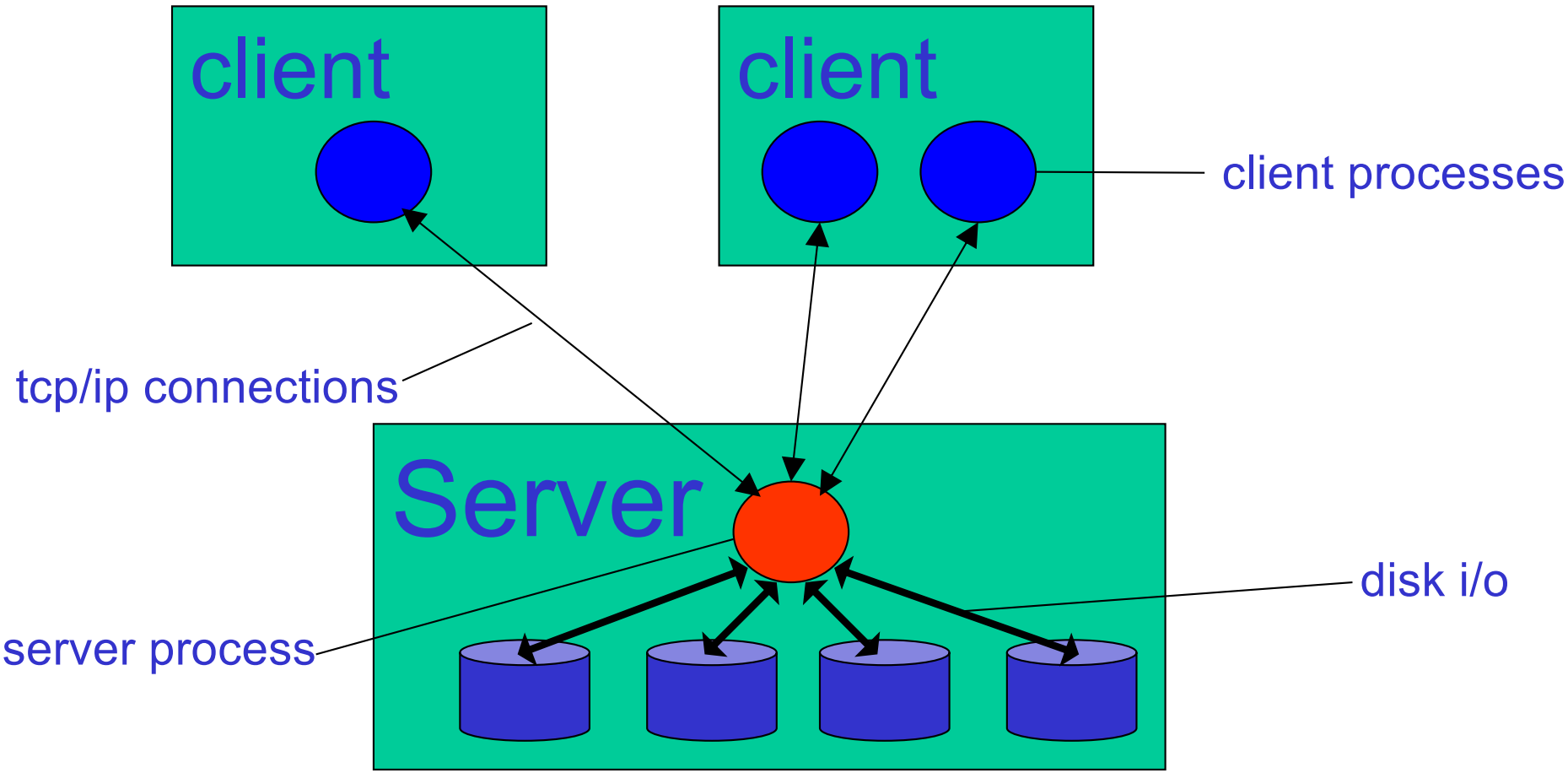
estore


category
id INT
name VARCHAR(45)
Indexes
product
id INT
name VARCHAR(45)
price VARCHAR(45)
quantity INT
category_id INT
Indexes
customer_has_product
id INT
customer_id INT
product_id INT
Indexes
customer
id INT
name VARCHAR(45)
email VARCHAR(45)
Indexes

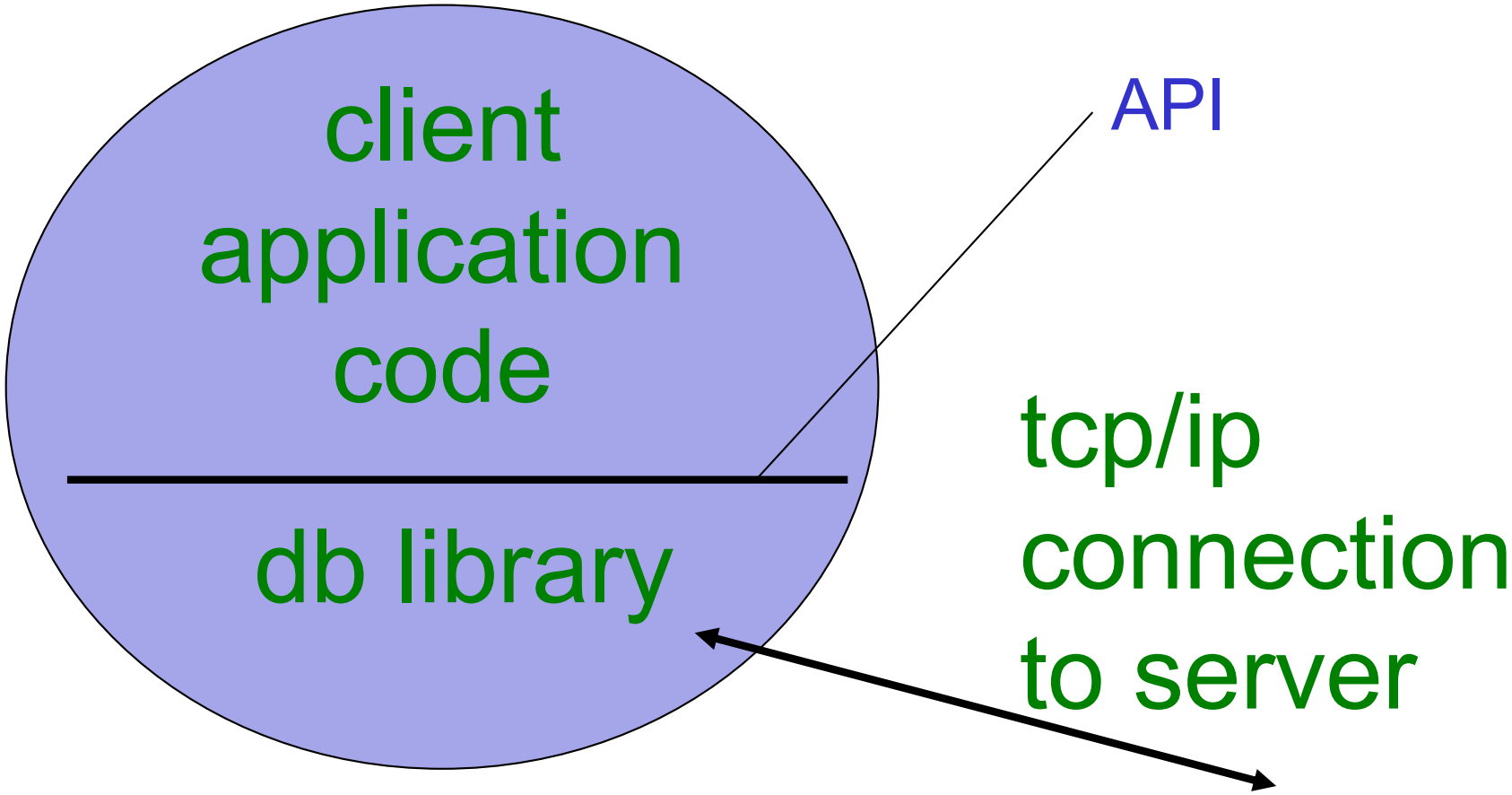
RDBMS Technology

- Client/Server Databases
  - Oracle, Sybase, MySQL, SQLServer
- Personal Databases
  - Access
- Embedded Databases
  - SQLite

Client/Server Databases



Inside the Client Process



MySQL Python Connector

- Standard SQL database access interface.
- Allows a Python program to issue SQL statements and process the results.
- Defines classes to represent constructs such as database connections, SQL statements, result sets, and database metadata.

API: Connection

```
import mysql.connector

db_config = {       'user': ’ece1779',
                    'password': ’some_password',
                    'host': '127.0.0.1',
                    'database’: estore'}

db = mysql.connector.connect(user=db_config['user'],
                             password=db_config['password'],
                              host=db_config['host'],
                               database=db_config['database'])

db.close()
```

API: Executing Queries

- A query can return many rows, each with many attributes
- Steps are

1 Send query to the database

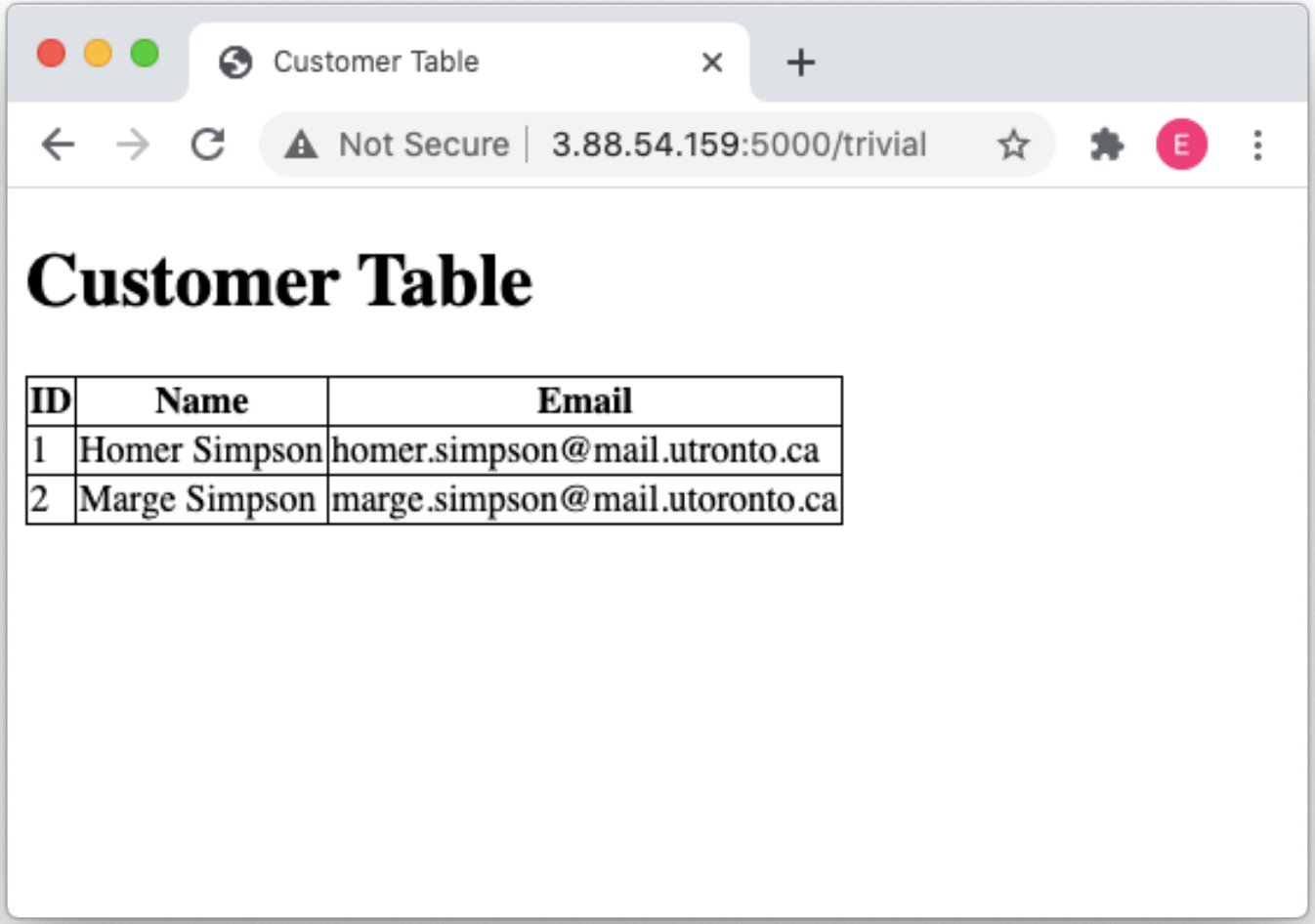
```
cursor = cnx.cursor()
query = 'SELECT * FROM customer'
cursor.execute(query)
```

2 Retrieve one row at a time

3 For each row, retrieve attributes

```
for row in cursor:
  print(row[0])
```

Trivial Example



Customer Table

ID	Name	Email
1	Homer Simpson	homer.simpson@mail.utronto.ca
2	Marge Simpson	marge.simpson@mail.utoronto.ca

trivial.py

```python
from flask import render_template
from app import webapp

import mysql.connector



@webapp.route('/trivial',methods=['GET'])
# Display an HTML list of all product.
def trivial():

    cnx = mysql.connector.connect(user='ece1779',
                                  password='secret',
                                  host='127.0.0.1',
                                  database='estore')

    cursor = cnx.cursor()
    query = "SELECT * FROM customer"
    cursor.execute(query)
    view = render_template("trivial.html",title="Customer Table", cursor=cursor)
    cnx.close()
    return view
```

trivial.html

```html
<!DOCTYPE html>
<html>
    <head>
        <title>{{title}}</title>
        <style>
            table, tr, td, th {
                border: 1px solid black;
                border-collapse: collapse;
            }
        </style>
    </head>
    <body>
        <h1>{{title}}</h2>
        <table>
        <thead>
                <th>ID</th><th>Name</th><th>Email</th>
        </thead>
        {% for row in cursor %}
            <tr>
                <td>{{ row[0]}}</td><td>{{ row[1]}}</td><td>{{ row[2]}}</td>
            </tr>
        {% endfor %}
        </table>
    </body>
</html>
```

CRUD Design Pattern

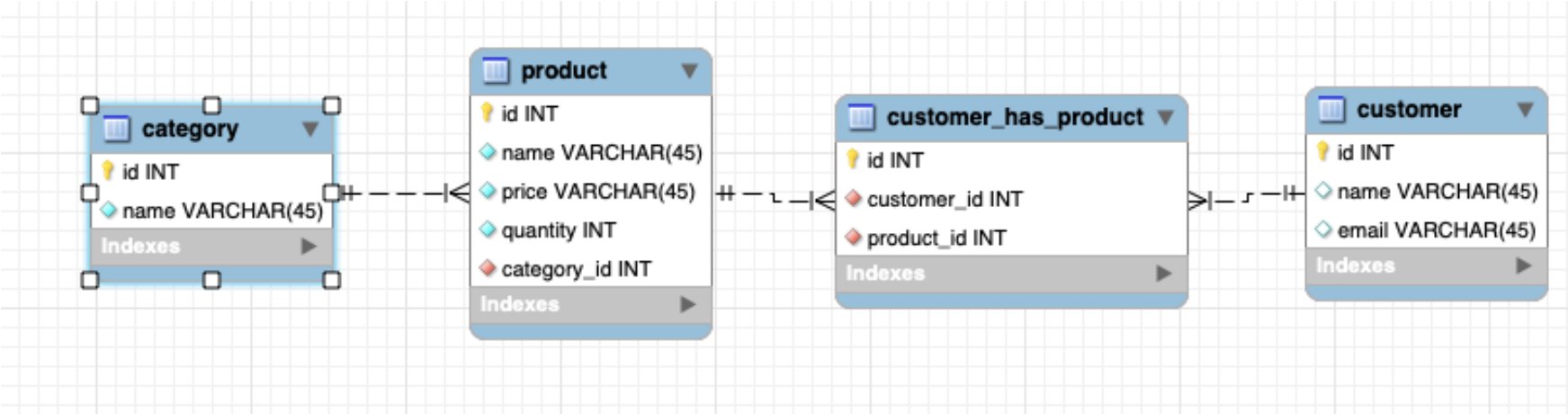
- Common design pattern for single table data manipulation
- Create, Read, Update, Delete (CRUD)

URL	Verb	Purpose
/courses/	GET	Display a list of courses.
/courses/[id]	GET	Display details about a specific course.
/courses/edit/[id]	GET	Display editable form populated with course data.
	POST	Save the form changes for a particular course.
/courses/create	GET	Display an empty HTML form that allows users to define a new course.
	POST	Create a new Course and save it.
/courses/delete/[id]	POST	Deletes the specified course

Transactions

- Definition: A transaction is a collection of DB modifications, which is treated as an atomic DB operation.
  - Transactions ensure that a collection of updates leaves the database in a consistent state (as defined by the application program); all updates take place or none do.
  - A sequence of **read** and **write** operations, terminated by a **commit** or **abort**
- Definition: Committed
  - A transaction that has completed successfully; once committed, a transaction cannot be undone
- Definition: Aborted
  - A transaction that did not complete normally
- Python Connector: By default in transactional mode: auto commit has been disabled, the method commit must be called explicitly; otherwise, database changes will not be saved.

Example: Buy a Product



Steps:

- Query *product* for availability
- If quantity > 0
  - Insert an entry into *customer_has_product*
  - Update entry in *product*
- Else
  - Fail