



**Course Name: Software and Data Engineering
Code Documentation**

**Topic: A performance comparison of SQL and NoSQL
databases**

Submitted by:

Akaash Chatterjee
Roll No: M24CSE002

Aman Saini
Roll No: M24CSE003

Bera Swaminath Ansuman Sabita
Roll No: M24CSE007

**Department of Computer Science and Engineering
Indian Institute of Technology Jodhpur
2024**

Contents

1	Code Documentation	2
1.1	bench.py	2
1.2	*_adapter_run.py	3
1.3	random_data_generator.py	6
1.4	visualize.py	6

Chapter 1

Code Documentation

Providing an overview of each Python script in the project directory, detailing its purpose, main functions, and expected input/output. These scripts are integral to benchmarking database performance on CRUD operations.

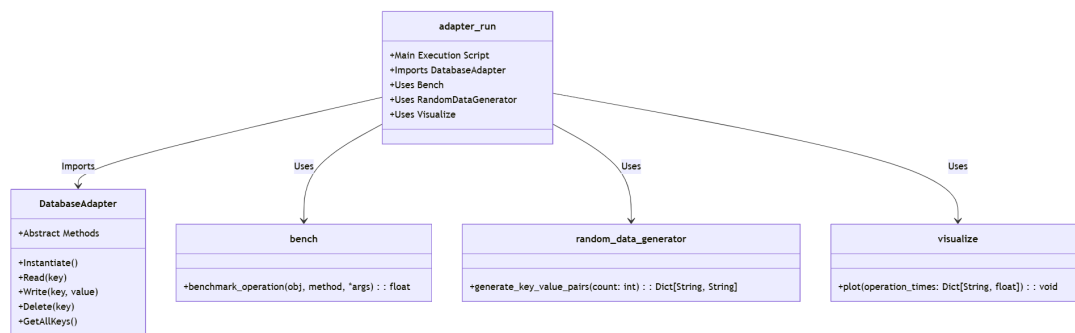


Figure 1.1: package structure and classes

1.1 bench.py

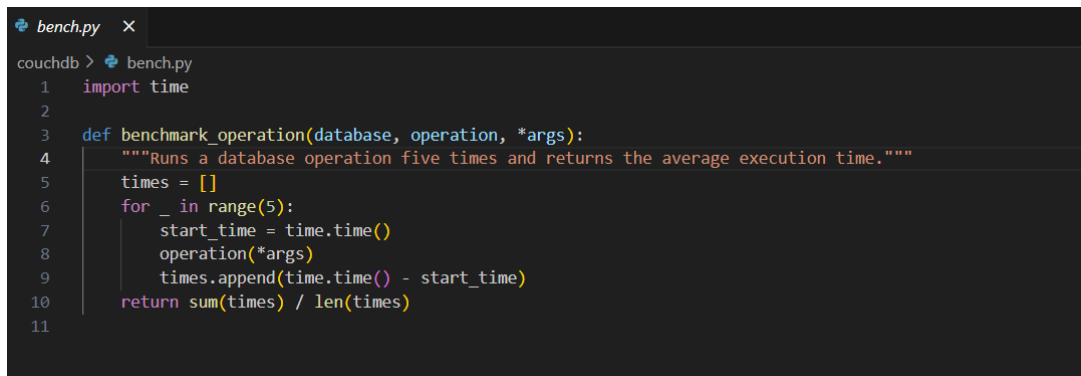
Purpose: `bench.py` is the main benchmarking script that measures the performance of CRUD operations on various databases. It runs each operation multiple times and calculates average execution times, allowing performance comparisons across databases.

Main Functions:

- `benchmark_operation(adapter, operation, *args)`: Executes a specified operation (e.g., instantiate, write, read, etc.) multiple times and calculates the average execution time.
- `run_benchmarks()`: Initializes database connections and runs the benchmark for each CRUD operation.

Expected Input: No direct input is needed. The script interacts with pre-configured database adapters to perform the benchmarking.

Expected Output: Prints the average execution time for each operation to the console, which can be redirected or saved for further analysis.

A screenshot of a code editor window titled 'bench.py'. The editor shows a Python script for benchmarking database operations. The script starts with 'import time'. It then defines a function 'benchmark_operation(database, operation, *args):' with a docstring: 'Runs a database operation five times and returns the average execution time.' The function initializes a list 'times = []', loops five times (range(5)), and for each iteration, it records the start time, performs the operation, and records the end time. Finally, it returns the average time calculated as 'sum(times) / len(times)'.

```
1 import time
2
3 def benchmark_operation(database, operation, *args):
4     """Runs a database operation five times and returns the average execution time."""
5     times = []
6     for _ in range(5):
7         start_time = time.time()
8         operation(*args)
9         times.append(time.time() - start_time)
10    return sum(times) / len(times)
11
```

1.2 *_adapter_run.py

Purpose: Adapter run files for e.g. `mysql_adapter_run.py` serves as a specialized adapter script for interacting with a MySQL database. It defines functions to perform CRUD operations, allowing MySQL to be benchmarked as a key-value store.

Main Functions:

- `connect_to_mysql()`: Establishes a connection to the MySQL database.
- `instantiate()`: Initializes a MySQL database or table for benchmarking.
- `write(key, value)`: Inserts or updates a key-value pair.
- `read(key)`: Fetches the value associated with a specified key.
- `delete(key)`: Deletes a key-value pair from the database.
- `get_all_keys()`: Retrieves all keys from the database.

Expected Input: No command-line input is required, but database connection parameters (e.g., host, username, password) may need to be specified within the script.

Expected Output: Prints or logs confirmation of each CRUD operation, useful for debugging and monitoring. Used in conjunction with `bench.py` to obtain performance metrics.

```

import couchdb
from bench import benchmark_operation
from random_data_generator import generate_key_value_pairs
from visualize import plot

class CouchDBDatabase:
    def __init__(self):
        # Connect to CouchDB server
        self.couch = couchdb.Server("http://admin:admin@127.0.0.1:5984/")
        self.db_name = "test_database"

        # Create or open the database
        if self.db_name in self.couch:
            self.db = self.couch[self.db_name]
        else:
            self.db = self.couch.create(self.db_name)

    def instantiate(self):
        """Clears all documents in the database to start with a fresh dataset."""
        for doc_id in list(self.db):
            self.db.delete(self.db[doc_id])

    def read(self, key):
        """Reads the value associated with the given key."""
        try:
            doc = self.db[key]
            return doc['value']
        except couchdb.ResourceNotFound:
            return None

```

```

    def read(self, key):
        """Reads the value associated with the given key."""
        try:
            doc = self.db[key]
            return doc['value']
        except couchdb.ResourceNotFound:
            return None

    def write(self, key, value):
        """Writes a key-value pair to the database or updates the value if the key exists."""
        if key in self.db:
            doc = self.db[key]
            doc['value'] = value
            self.db.save(doc)
        else:
            self.db[key] = {'value': value}

    def delete(self, key):
        """Deletes a key-value pair by key."""
        try:
            doc = self.db[key]
            self.db.delete(doc)
        except couchdb.ResourceNotFound:
            pass

    def get_all_keys(self):
        """Fetches all keys from the database."""
        return [doc.id for doc in self.db.view('_all_docs')]

data = generate_key_value_pairs(1000)
print("Sample data:", list(data.items())[:5])

# Initialize CouchDB adapter
couch_db = CouchDBDatabase()

```

```

# Step 5.1: Instantiate (clear all records)
instantiate_time = benchmark_operation(couch_db, couch_db.instantiate)
print(f"Instantiate Time: {instantiate_time:.5f} seconds")

# Step 5.2: Write (Insert all key-value pairs from generated data)
write_times = []
for key, value in data.items():
    write_times.append(benchmark_operation(couch_db, couch_db.write, key, value))
average_write_time = sum(write_times) / len(write_times)
print(f"Average Write Time: {average_write_time:.5f} seconds per record")

# Step 5.3: Read (Read all keys in the generated data)
read_times = []
for key in data.keys():
    read_times.append(benchmark_operation(couch_db, couch_db.read, key))
average_read_time = sum(read_times) / len(read_times)
print(f"Average Read Time: {average_read_time:.5f} seconds per record")

# Step 5.4: Delete (Delete all keys in the generated data)
delete_times = []
for key in data.keys():
    delete_times.append(benchmark_operation(couch_db, couch_db.delete, key))
average_delete_time = sum(delete_times) / len(delete_times)
print(f"Average Delete Time: {average_delete_time:.5f} seconds per record")

# Step 5.5: GetAllKeys (fetch all keys in one operation)
couch_db.instantiate() # Reset data
for key, value in data.items(): # Reinsert data to test GetAllKeys
    couch_db.write(key, value)
get_all_keys_time = benchmark_operation(couch_db, couch_db.get_all_keys)
print(f"GetAllKeys Time: {get_all_keys_time:.5f} seconds")

# Create a dictionary of operation times for plotting
operation_times = {
    "Instantiate": instantiate_time,
    "Write": average_write_time,
    "Read": average_read_time,
    "Delete": average_delete_time,
    "GetAllKeys": get_all_keys_time
}

plot(operation_times)

```

1.3 random_data_generator.py

Purpose: Generates random key-value pairs as sample data for benchmarking tests, ensuring realistic data for each database operation.

Main Functions:

- `generate_data(n)`: Generates `n` random key-value pairs, where keys are unique identifiers, and values are randomly generated strings or numbers.

Expected Input: Specify the number of key-value pairs to generate (`n`) as an argument within the script.

Expected Output: Returns a dictionary or list of key-value pairs, which can be used by the benchmarking scripts for database operations.

```
b > random_data_generator.py
import random
import string

def generate_key_value_pairs(num_pairs=1000):
    data = {}
    for i in range(num_pairs):
        key = f'k{i}'
        value = ''.join(random.choices(string.ascii_letters + string.digits, k=20))
        data[key] = value
    return data
```

Usage Example:

```
data = generate_data(100)  % Generates 100 random key-value pairs
```

1.4 visualize.py

Purpose: `visualize.py` generates a visual representation of the benchmarking results, such as a bar chart or line graph, which helps analyze and compare database performance.

Main Functions:

- `plot_results(results)`: Takes in a dictionary of results (e.g., average execution times for each operation) and plots them using a library like `matplotlib`.
- `save_figure(filename)`: Saves the generated plot as a PNG file, e.g., `Figure_1.png`.

Expected Input: A dictionary containing the results of each benchmark (e.g., `{"Instantiate": 0.1, "Write": 0.2, "Read": 0.05, "Delete": 0.03, "GetAllKeys": 0.04}`).

Expected Output: Displays a plot on the screen or saves it as `Figure_1.png` or a user-specified filename for inclusion in reports or presentations.

```

> visualize.py
import matplotlib.pyplot as plt

def plot(operation_times):
    # Plotting
    plt.figure(figsize=(10, 6))
    plt.bar(operation_times.keys(), operation_times.values(), color='skyblue')
    plt.xlabel("Operations")
    plt.ylabel("Average Time (seconds)")
    plt.title("CouchDB Operation Performance")
    plt.show()

```

Usage Example:

```

results = {
    "Instantiate": 0.1,
    "Write": 0.2,
    "Read": 0.05,
    "Delete": 0.03,
    "GetAllKeys": 0.04
}
plot_results(results)
save_figure("Figure_1.png")

```

This documentation provides a quick reference to the purpose and functions of each script in this project. By following these descriptions, users can understand how each component fits into the benchmarking experiment and how to use each script effectively.