# Redundancy

Database Administration
Lab Guide 3

2024/2025

Introduce redundancy in the toy benchmark to reduce I/O and computation. For each change, evaluate its the impact by (i) observing the query plan and (ii) running the benchmark for 1, 2, 4, 8, ... threads and plotting the corresponding scalability curve.

**Steps**

1. Use indexes[1] to improve the performance of the select queries:
   `CREATE INDEX [index_name] ON table_name [USING method] (col1 [, col2, ...]);`
2. Use a materialized view[2] to improve the performance for the top 10 operation. Implement an appropriate refresh strategy:
   `CREATE MATERIALIZED VIEW mat_view_name AS SELECT ...;`
   `REFRESH MATERIALIZED VIEW mat_view_name;`
3. Add the *recommended products* operation and optimize using indexes:
   - Follow the instructions in the appendix to prepare the weights.
   - Create a prepared statement to compute the 10 most similar products to a given product (id as argument), using the weights in the *Embedding* table.
   - Implement the `List<Integer> recommendedProducts()` method, which returns the list of identifiers of similar products. Update the `transaction()` method to include it with, e.g., 2% probability.

**Questions**

1. Are the indexes useful? Is clustering[3] the index useful?
2. Is it worth creating an index to optimize the query `SELECT * FROM Product WHERE price BETWEEN` $x$ `AND` $y$? Explain.
3. Is the materialized view useful? Which update strategy makes it useful?
4. How does each of these optimizations impact each operation alone?
5. What are the main advantages and disadvantages of introducing redundancy?

**Learning Outcomes** Recognize and explain the role of redundancy in relational databases, in particular, by contrasting it with implementation decisions in object-oriented programming.

[1] `https://www.postgresql.org/docs/17/indexes.html`
[2] `https://www.postgresql.org/docs/17/rules-materializedviews.html`
[3] `https://www.postgresql.org/docs/17/sql-cluster.html`

## Preparing the weights

- Install the `pgvector` extension:

```
docker exec -t postgres bash -c "
  apt update
  apt install -y git make gcc postgresql-server-dev-17
  git clone --branch v0.8.0 https://github.com/pgvector/pgvector.git
  cd pgvector
  make -j 8
  make install
"
```

- Enable the extension in the target database:

```
docker exec -t postgres psql -U postgres -d testdb \
  -c "create extension vector"
```

- Create the *Embedding* table and load the weights from the CSV file:

```
docker exec -t postgres psql -U postgres -d testdb \
  -c "create table embedding (ref varchar, embedding vector(1024))"

docker cp embedding.csv postgres:/

docker exec -t postgres psql -U postgres -d testdb \
  -c "copy embedding from '/embedding.csv' with csv header"
```

- Details about indexing in the `pgvector` extension can be found at `https://github.com/pgvector/pgvector?tab=readme-ov-file#indexing`.

## Helper scripts to run and plot

### Run (bash)

```
#!/bin/bash

CLIENTS=(1 2 4 8 16)
TIME=15
SCALE=16

mvn package -q || exit 1

# populate
echo "Populating"
java -jar target/benchmark-1.0-SNAPSHOT.jar \
    -d jdbc:postgresql://localhost/testdb -U postgres -P postgres \
    -s $SCALE -p
echo "clients,tps,rt" > results.csv

# run
for c in "${CLIENTS[@]}"; do
    echo "Running $c clients"
    java -jar target/benchmark-1.0-SNAPSHOT.jar \
        -d jdbc:postgresql://localhost/testdb -U postgres -P postgres \
        -s $SCALE -x -c $c -R $TIME -W 1 > out.txt
    tps=$(grep -Po '(?<=throughput \(txn/s\) = ).*' out.txt)
    rt=$(grep -Po "(?<=response time \(s\) = ).*" out.txt)
    echo "$c,$tps,$rt" >> results.csv
done
```

### Plot (python)

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import glob

dfs = []
for file in glob.glob('*.csv'):
    df = pd.read_csv(file)
    df['hue'] = file
    dfs.append(df)

df = pd.concat(dfs, axis=0, ignore_index=True)
ax = sns.lineplot(x='tps', y='rt', hue='hue', data=df, marker='o', sort=False)
ax.set_xlim(0)
ax.set_ylim(0)
plt.savefig('results.png', dpi=300)
```