CS307 Project1 Report

Group Information

12410922 李允臧

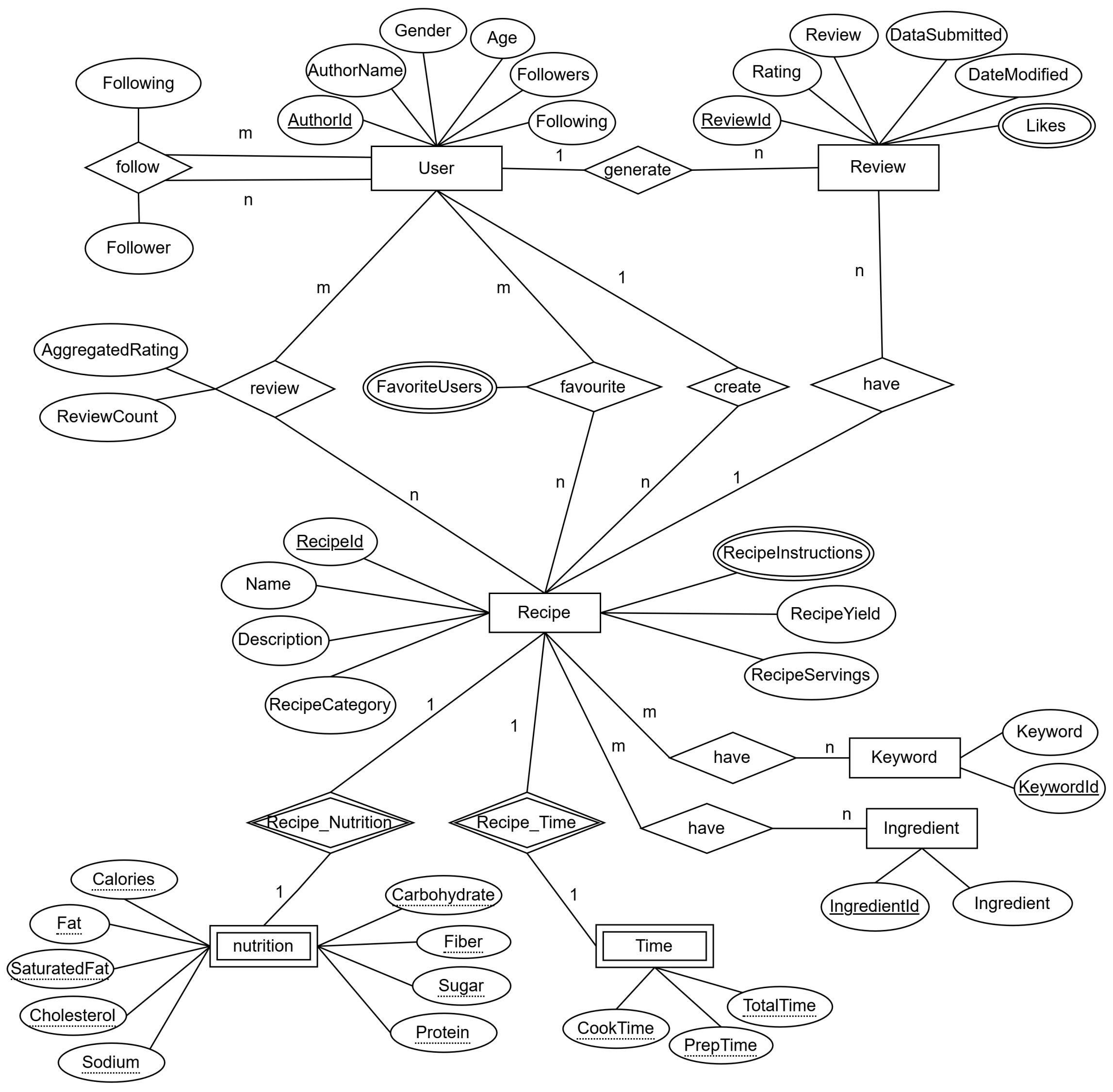
12412460 吴 桐

The source code is hosted on GitHub and will be released under the MIT License after the project deadline.

repo

TASK 1: ER Diagram

I use the tool **draw.io** to finish this **ER Diagram**



Entities:

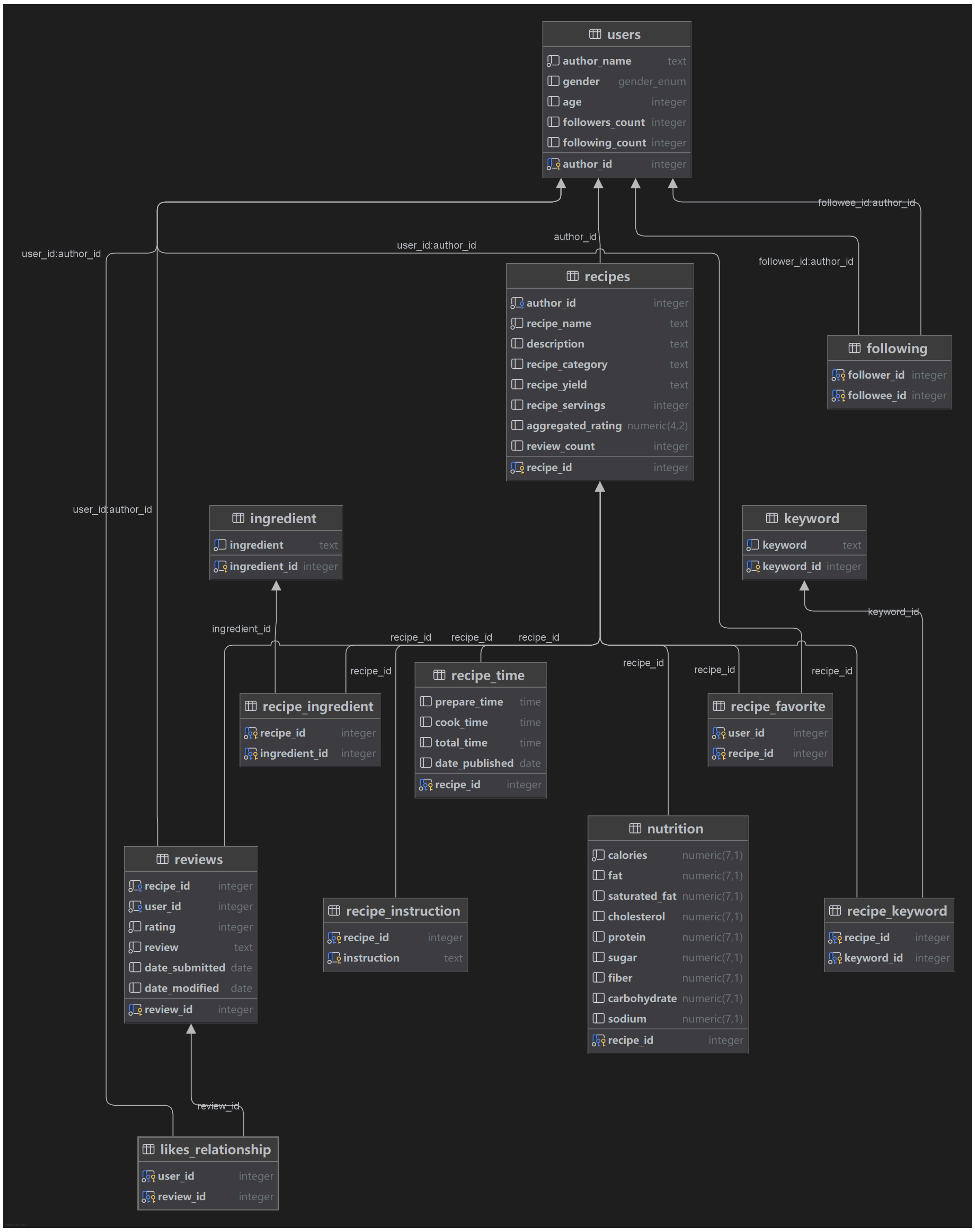
- **User**: Represents the users in the system
- **Review**: Comments on recipes
- **Recipe**: Information about a recipe
- **Keyword**: Keywords associated with the recipe
- **Ingredient**: Ingredients of the recipe
- **Nutrition**: Nutritional information of the recipe
- **Recipe_Nutrition**: A weak entity dependent on the Recipe, representing the nutritional components of the recipe
- **Recipe_Time**: A weak entity dependent on the Recipe, representing the preparation time related to the recipe

Relationship:

1. User **generate** Review: one-to-many
2. User **follows** User: many-to-many
3. User **favourites** Recipe: many-to-many
4. User **review** Recipe**:** many-to-many
5. User **create** Recipe**:** one-to-many
6. Recipe **have** Review: one-to-many
7. Recipe **have** Keyword: many-to-many
8. Recipe **have** Ingredient: many-to-many
9. Recipe_Nutrition: one-to-one
10. Recipe_Time: one-to-one

TASK 2: Database Design

In this TASK, I write DDL statements to create tables and columns according to the ER diagram. The SQL file is uploaded as attached file. Then I generate the database diagram by using the " **Show Visualization"** feature of Datagrip. The picture is shown below:



The following part is a brief description of the meaning of tables and columns. We **designed 13 tables** and will explain them separately. Finally, I will explain the reason that this design cater to all the requirements.

Description of tables and columns

1. `users` table stores all the users' information. It includes **author_id** as primary key, **author_name** as the name which is NOT NULL, **gender** which is an enum type and **age**.
2. `recipes` table stores the recipe information. It includes **recipe_id** as primary key, **author_id** which is the author of this recipe as foreign key reference to **author_id** in `users`,the **recipe_name** which is NOT NULL, the **description**, the **recipe_category** which is this recipe belong to,the **recipe_yield** which is the output of this recipe, the **aggregated_rating** which is the score obtained of this recipe, the **review_count** which is the reviewer number of this recipe, and **recipe_servings** indicating the number of servings.
3. `reviews` table stores the review information. It includes **review_id** as primary key, **recipe_id** as foreign key reference to `recipes` , **user_id** as foreign key reference to `users` , **rating** which is the score given to this recipe, **review** which is the detailed review content, **date_submitted** with type DATE which is the date of review submitted,**date_modified** which is the latest modification.
4. `following` table stores the follow relationships among users. It includes a composite primary key (**follower_id**, **followee_id**), both as foreign keys referencing `users(author_id)`, recording each directed follow relationship.
5. `recipe_favorite` table stores the user that favorites one recipe. It includes a composite primary key of(**user_id,recipe_id**), as foreign key to `users` and `recipes` , respectively.
6. `like_relationship` stores the user that likes one review. It includes a composite primary key of(**user_id,review_id**), as foreign key to `users` and `reviews` , respectively.
7. `keyword` stores a list of all possible keywords. It includes **keyword_id** as primary key, **keyword** as the content which is UNIQUE NOT NULL.
8. `recipe_keyword` stores the relationship between `recipes` and `keyword`, It includes a composite primary key of(**recipe_id,keyword_id**), as foreign key to `recipes` and `keyword` , respectively.
9. `ingredient` stores a list of all possible ingredients. It includes **ingredient_id** as primary key, **ingredient** as the content which is UNIQUE NOT NULL.
10. `recipe_ingredient` stores the relationship between `recipes` and `ingredient`, It includes a composite primary key of(**recipe_id,ingredient_id**), as foreign key to `recipes` and `ingredient`, respectively.
11. `nutrition` table stores the nutrition information for each recipe. It includes **recipe_id** as both the primary key and foreign key referencing `recipes(recipe_id)`, along with **calories**, **fat**, **carbohydrates**, **protein**, etc. Each numeric value uses a suitable type.
12. `recipe_time` stores the time information relevant to each recipe. It includes **recipe_id** as primary key and foreign key reference to`recipes`, **prepare_time** which is the time need to prepare with type interval, **cook_time** which is the time need to cook with type interval, **total_time** which is the sum of prepare and cook time with type interval, **date_published** which is the date of this recipe published.
13. `recipe_instruction` table stores detailed step-by-step cooking instructions. It includes **recipe_id** as foreign key referencing recipes and **instruction** as NOT NULL. The composite primary key (**recipe_id**, **instruction**) ensures no duplicate steps for the same recipe.

Compliance with Requirements and Normal Forms

The database design meets all the requirements and normal forms. It can manage all the information mentioned in the document. It contains primary key and foreign key which uniquely identify each row. Every table is not isolated and the database contains no circular links. What's more, each table has at least one NOT NULL column, and has suitable data type for columns.

The database design also satisfies the three normal forms.

- **1NF:** All attributes are atomic. Multivalued columns in the original dataset (such as ingredients, keywords, likes, following and instructions) are separated into relationship tables to remove repeating groups.
- **2NF:** Each non-key attribute fully depends on the whole primary key. Composite tables (`recipe_keyword`, `recipe_ingredient`) have no partial dependencies.
- **3NF:** No transitive dependencies exist among non-key attributes. Derived attributes like `review_count` are optional and can be omitted to avoid redundancy.

In summary, this design achieves data integrity, consistency, and scalability, satisfying the requirements of Task 2.

Data Preprocess

Before truly importing the data into the relational database designed in TASK 2, we preprocess the data in the three csv files and identified several data quality issues that might lead to incosistencies or meaningless. To ensure the correctness and reliablity of subsequent steps, it is necessary to preprocess the data and normalize it before importing into database. So This section focus on these three aspects of preprocessing: deleting confused or meaningless records, correcting inconsistent attributes using other fields within the same record, and resolving data type problems.

The first problem is confused or meaningless records. For example, in the `review.csv` file, there is a total number of **19 lines** that have a **redundent comma** between reviewId and recipeId, which means this record lack of AuthorId and will lead to confusion in the following steps. Therefore, the data preprocess stage must **delete these 19 lines** to avoid foreign key come to NULL. The final number of review records should be 1,401,963. Other confused or meaningless records are treated similarly.

The second problem is attributes that contradict other attributes within the same line. This issue may appear in `recipe.csv`. Some lines may have both CookTime and PrepareTime in correct form, but the TotalTime is either missing or inconsistent with the sum of two. The data preprocess procedures will treat CookTime and PrepareTime as correct and check the TotalTime. If the value is missing, it will be filled with the sum of other two times. However, if the value is inconsistent, we will use the sum of others to fill the blank instead of collisioned one. Similar procedures will also be used in the `followers_count` and `following_count` arrtibutes in the users table. However, in practical programming, we found that following relationship is quite complex with many conflicts. As a result, we treat following_users as correct version and process them when execute Java importing code.

The third problem includes data type errors. In `reviews.csv`, every RecipeId column has the incorrent data type with the attached ".0", but the IDs should be integer type. So the data preprocess in this step should convert them into integers and remove the zero in the end. Moreover, the time-relavant data follows ISO 8601 standard, with capital characters like P, T, H and so on. It should be the type `Time` or `Inteval` in SQL database, so we need to convert it into correct form as well. In our procedures, we decide to finish this convert step when inserting and parse it in Java program instead of preprocess stage.

In practical part, we used a python script to clean and preprocess row csv files and fixed all the problems mentioned previously. The source code will be attached as `data preprocess.py` file. There might be other strange data that haven't been covered, which will be processed in the following inserting part. The preprocess result is shown below:

```
D:\25summer_mathModel\test_code\.venv\Scripts\python.exe "D:\25summer_math
Fix reviews.csv with .0 id and extra comma

[clean_reviews] Fixed 19 lines with leading 'ID,,ID,'.
Fix time in recipes.csv

[fix_recipes_time] Checked 522517 records, fixed 24 inconsistent TotalTime
All preprocessing steps finished.

进程已结束，退出代码为 0
```

TASK 3 : Data Import

In this task, I implemented a Java program to import data from CSV files into the database which designed in Task 2.

Prerequisites

1. The database tables must already exist before running this script. (We will discuss a possible automated creation approach in the advanced part.)
2. The PostgreSQL **JDBC driver** must be added as a **project dependency**.

   In IntelliJ IDEA: open *File → Project Structure → Modules → Dependencies*, and add the JAR file `postgresql-42.2.5.jar`. The JAR file will be provided in the project directory.

Code Execution Steps

Step 1. Connect To the Database

In the `getConnection()` method, the class `org.postgresql.Driver` is loaded to establish a connection to the local PostgreSQL server.

The method uses the basic connection parameters — host, database name, user, password, and port — to build the JDBC URL and call `DriverManager.getConnection()`.

Once the dependency is correctly imported, the driver initializes the connection channel using the PostgreSQL protocol; otherwise, the connection attempt will fail. So please you have down import dependency properly in the prerequisites part.

Step 2. Read Data from CSV Files

The import program reads each CSV file using a lightweight line-based method. In `readOneCsvRecord()`, a `BufferedReader` retrieves one physical line at a time; empty lines are skipped, and `null` indicates the end of file.

After a complete line is obtained, the `splitCsvRecord()` method converts it into an array of fields. Instead of using `String.split(",")`, the method scans the line character by character. A boolean flag tracks whether the parser is currently inside quotation marks. Commas outside quotes are treated as field separators, while text inside quotes—including commas and escaped quotes—is preserved. This ensures that fields containing punctuation or quotation marks are handled safely.

These parsing steps standardize the incoming data and separate generic CSV handling from table-specific logic. All import methods in Step 3 reuse this shared parsing pipeline before converting the processed values into batched SQL insertions.

Step 3. Import Data

3.1 Import `users.csv`

The `importUsersCsv()` method reads the users information from *users.csv* and inserts it into the `users` table and the corresponding `following` relationship table.

After obtaining a parsed CSV record from Step 2, the program extracts fields such as AuthorId, AuthorName, Gender, Age, and the follower/following lists. The follower and following lists are parsed into individual user-to-user relations, which are then inserted into the `following` table.

A parameterized `PreparedStatement` is used for both tables to accelerate. For every valid record, parameters are assigned and inserted via `addBatch()`. Invalid IDs and malformed list fields are skipped to avoid insertion errors. When the batch size limit is reached, the statements are executed together using `executeBatch()`.

3.2 Import `recipes.csv`

The `importRecipesCsv()` method processes the complex *recipes.csv* file by distributing different fields into multiple tables according to the schema in Task 2.

A single recipe record is decomposed and inserted into the following tables:`recipes` , `nutrition`, `recipe_time`, `recipe_keyword`, `recipe_ingredient`, `recipe_instruction`, `recipe_favorite`, `keyword` and `ingredient`.Each table uses its own PreparedStatement.

Time-related fields (PrepTime, CookTime, TotalTime) are converted into SQL TIME values. The DatePublished field is parsed into DATE format with fallback handling for missing or invalid values.

For multi-value fields such as Keywords, Ingredients, Instructions, and Favorites, the program splits the string into lists and inserts each item into its corresponding relationship table. Keywords and ingredients are also inserted into their dictionary tables (`keyword`, `ingredient`) with `ON CONFLICT DO NOTHING` to guarantee uniqueness.

Invalid recipe IDs, empty fields, or unparseable values result in skipping that specific part or the entire record, depending on severity.

Similarly, For every valid record, parameters are assigned and inserted via `addBatch()`. When the batch size limit is reached, the statements are executed together using `executeBatch()`.

3.3 Import `reviews.csv`

The `importReviewsCsv()` method loads review data from *reviews.csv* and inserts it into `reviews` and `likes_relationship`.

Each CSV record is parsed into ReviewId, RecipeId, UserId, Rating, Review text, submission date, modification date, and Likes.

Since the database specifies `rating` as NOT NULL, any record with missing or invalid rating values is skipped.

Date fields are parsed into DATE type with safe null handling.

The Likes field is split and inserted as multiple rows into `likes_relationship`. Duplicate likes and invalid user IDs are automatically filtered using `ON CONFLICT DO NOTHING`. `PreparedStatement` batching is applied here as well to maintain fast insertion speed for the large review dataset.

3.4 Conclusion of import data

Here's the conclusion table for csv files, import method and the coressponding tables :

CSV File	Import Method	Affected Tables
users.csv	importUsersCsv()	users, following
recipes.csv	importRecipesCsv()	recipes, nutrition, recipe_time, recipe_keyword, recipe_ingredient, recipe_instruction, recipe_favorite, keyword, ingredient
reviews.csv	importReviewsCsv()	reviews, likes_relationship

This modular, table-aware import pipeline ensures correctness, clarity, and efficiency when handling all three large CSV files.

Step 4. Check the Import Correctness

After the data import process completed successfully, I verified the correctness by counting the total number of records in each entity table using SQL `COUNT(*)` queries to find the result table. You can find this SQL query code in the attached file `select_table_records_number.sql` or the project architecture `SQL/select_table_records_number.sql`.

The results are as follows:

Table name	Number of records (rows)
users	299,892
following	774,121
recipes	522,517
recipe_time	522,517
nutrition	522,517
reviews	1,401,963

Table name	Number of records (rows)
like-relationship	4,995,748
keyword	311
ingredient	7358
recipe_keyword	2,486,934
recipe_ingredient	4,003,863
recipe_favorite	2,588,000
recipe_instruction	3,429,015

These numbers in the **main tables** are consistent with the line counts of the original CSV files, and the `reviews` table is **19 lines less** than the csv file, which we delete it in data preprocess, indicating that the csv data have been successfully imported without loss or duplication.

Advanced Part

My test environment are shown below：

a. Hardware Specifications

CPU: AMD Ryzen 7 8745H (8 cores/16threads, with Radeon 780M Graphics)

RAM size: 24GB

Storage: 1TB NVMe SSD

b. Software Specifications

DBMS: PostgreSQL 17.6 on x86_64-windows, compiled by msvc-19.44.35213, 64-bit

Programming language: Java SE 23.0.2 (64-bit)

Compiler: Oracle JDK 23.0.2 (64-bit)

Operating System: Windows 11 Home Chinese Edition, Version 24H2

Database IDE version: DataGrip 2025.1.3

Programming IDE version: IntelliJ IDEA 2024.3.3 (Ultimate Edition)

1、Comparative Evaluation of Multiple Data Import Strategies

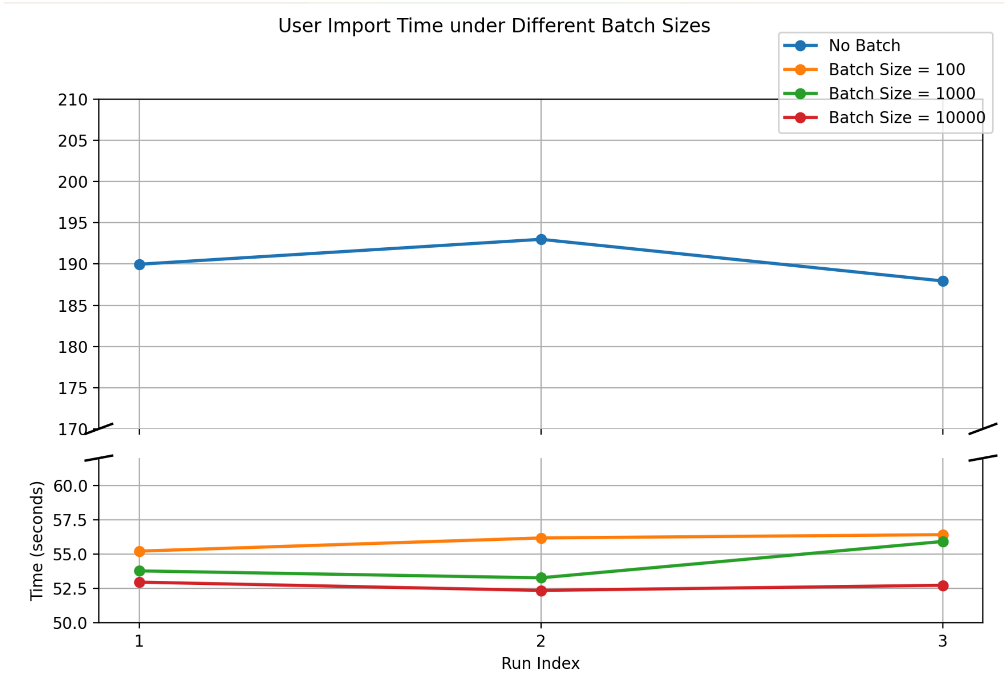
In the data importing part, I use some techniques to load data. In this part, I will try different strategies to evaluate the differences between import methods. Since the data source is large-scale, I will change the batch size and the use of preparedStatement to find their impact on the import time and programming efficiency.

In order to measure the import time in a constant way, I use a new class `StopWatch` to measure the execution time of program. When the import process starts, the timer records the start timestamp, and it records the end timestamp when the import finishes. Finally, it will calculate the time used and print it into console, so that we can get the experiment data. This method is stable and efficient, and it minimizes the influence of initialization and I/O on the timing results.

The first group of experiments focuses on batching techniques. I tried different method of a series of changing batch size, including a "NO BATCH" version, which means each line is inserted into the table individually. I execute every batch size seperately, and record the time they used. Repeat this procedure for 3 times to avoid some random disturb. The results are shown in the following table:

Method	Batch size	Average time	comparision
importUsersCsvNoBatch()	NO	190.307s	1×
importUsersCsvWith100Batch()	100 rows	55.947s	**3.402x faster**
importUsersCsvWith1000Batch()	1000 rows	53.836s	**3.535x faster**
importUsersCsvWith10000Batch()	10000 rows	51.693s	**3.681x faster**

And here is the picture for each runtime:



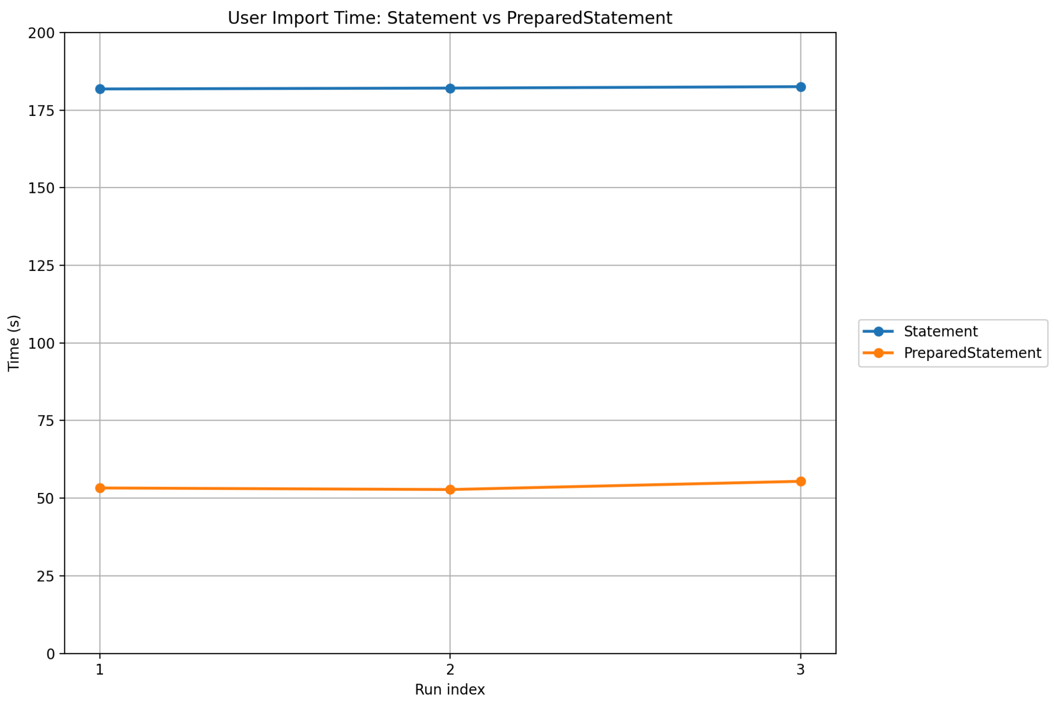
The results of the group of experiments demonstrate the importance of the batching technique in the data importing process. When batching is disabled, the import procedure becomes significantly slower, taking almost 3.5 times longer than the batched versions. When batching is enabled, the import process complete in roughly the same time, with only small improvement as the batch size increases.Therefore, the conclusion is that: **The batch technique is important and efficiency-friendly in large-scale data importing. Once a reasonable batch size is chosen and rows are grouped together, the size will not affect importing time significantly.**

The reason behind it is clearly to understand: Batching allows multiple insert operations to be grouped into a single request, which reduces communication overhead, minimizes repeated SQL parsing, and lowers the number of round-trip interactions with the database engine. So the import process becomes much more efficient as soon as batching is introduced.

The second group of experiments focuses on the use of `PreparedStatement`. In this part, I tried two importing strategies: one using `PreparedStatement` with a preprocessing SQL sentence, the other straightforward constructs row SQL statement for each row. The only difference between two methods is the SQL construction method, and other factors stay the same. These experiments are repeated for 3 times to reduce disturb and finally calculate the result, which is shown in the following table:

Method	Type	Average time	Comparison
importUsersWithoutPreparedStatement()	Statement	182.149s	1×
importUsersCsv()	PreparedStatement	53.836s	**3.383× faster**

And here is the picture for each runtime:



The results clearly show that the use of `PreparedStatement` brings a improvement in import speed and efficiency. When raw SQL strings are executed directly, the import becomes much slower because the database must parse and optimize every command separately. In contrast, the prepared version finishes in significantly less time and shows much more stable performance across repeated trials. Therefore, the conclusion is that **PreparedStatement technique is crucial for efficient large-scale imports, and removing it leads to performance degradation.**

2. Optimize import efficiency

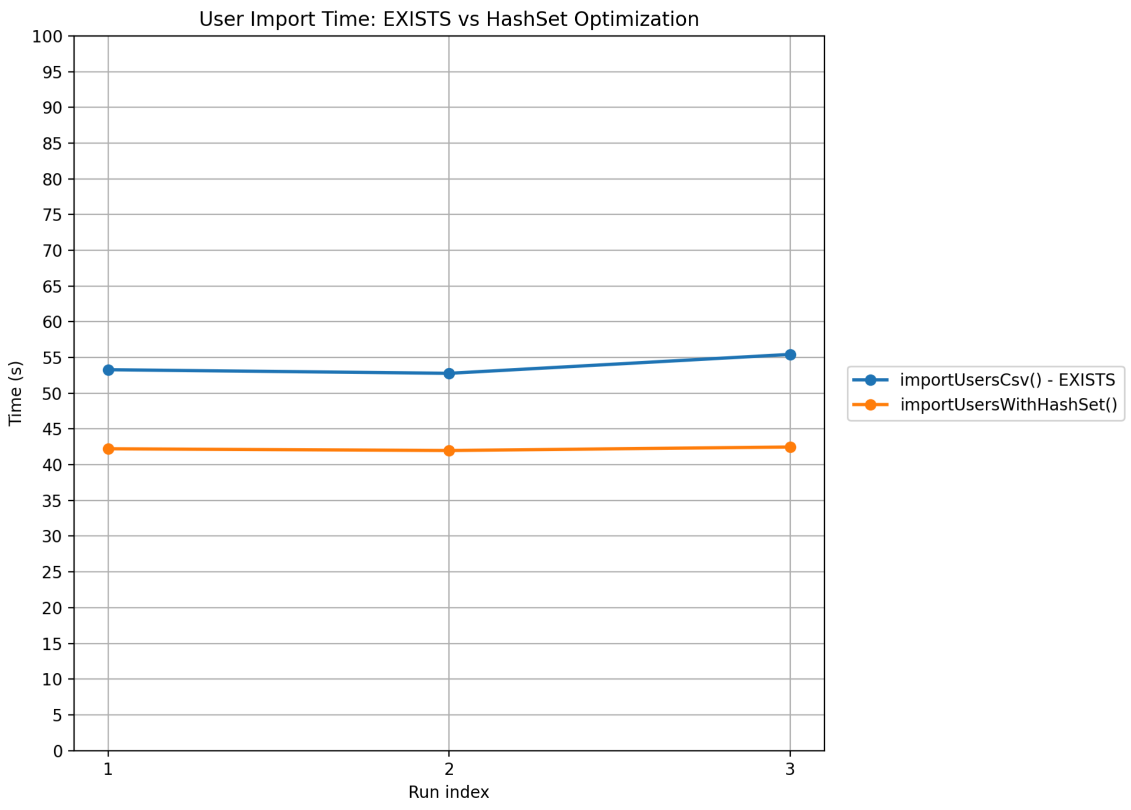
Since we have chosen a suitable batch size and apply PreparedStatement in the initial import version, the following part will focus on further efficiency optimization.

In the original version, we used `EXISTS` keyword in the PreparedStatement to check the existence of users, and then insert the following relationship, so that the foreign key won't conflict. This method is safe but might be time-consuming. In order to optimize the importing script, we must find a new way to double-check the existance.

Since the task is essentially checking whether a user ID exists, `HashSet` (or `HashMap`) is an appropriate data structure. We rewrote the `importUsersCsv` method, using simple `String SQL_FOLLOW` statement and leave the existence checking to the Java side. After that, in the stage of inserting to the `following` table, we read all the authorId from database, and store them in a hashSet. Finally, before the batch is executed, we only check whether the hashSet contains the certain user, instead of check them in the SQL statement. We use the batch size of 1000 rows and the other condition stay the same as comparison version (previously mentioned, with the average time of 53.836 seconds). These experiments are repeated for 3 times to reduce disturb and finally calculate the result, which is shown in the following table:

Method	Type	Average time	Comparison
importUsersCsv()	Use SQL EXISTS keyword	53.836s	1x
importUsersWithHashSet()	use HashSet	42.228s	**1.275× faster**

And here is the picture for each runtime:



The results demonstrate that replacing SQL `EXISTS` checks with a Java `HashSet` significantly improves the user import procedure. The original method using SQL `EXISTS` required an average of **53.836 seconds**, while the optimized version using `HashSet` took only **42.228 seconds** on average. This corresponds to an improvement of approximately **27%**.

The reason why this optimization works is checking foreign-key existence through SQL `EXISTS` requires PostgreSQL to do subqueries repeatedly. These repeated procedures significantly slow down the importing process. However, loading all valid user IDs into a Java `HashSet` allows each existence check to be completed in constant time **without contacting to the database**.Moreover, the application of hashSet filtered the redundency before insertion, which prevents unnecessary SQL statements from being executed. These factors result in observed approximately 27% improvement in overall import performance.

In summary, this optimization is particularly suitable when the reference table can be fully loaded into memory. Since all IDs are stored in a `HashSet`, the lookup cost remains constant even when the number of following relationships becomes large. For data at the scale of several hundred thousand records, memory consumption is negligible and the in-memory checking significantly reduces repeated SQL operations. Therefore, this method works well where the entire IDs set can be cached efficiently.

However, this optimization approach also have limitations. It may become less practical when the data grows to tens of millions of users, as the `HashSet` will require substantial memory. In such cases, the overhead of maintaining a large in-memory structure may offset the expected performance gain. Moreover, if the database schema or user table is frequently updated during import, the cached `HashSet` may be outdated and require refreshing. These factors limit this optimization for extremely large or dynamic datasets.

3. Automated Import

In the previous part, we first executed SQL DDL statements to create tables in DataGrip, and then executed import program in Java to complete this task. However, we can also integrate these two steps into a single automated workflow. That is, the input for the script is a given csv file, and the output is the well-built table in the database.

We use a new class `Automation` that extends the previous import class (You can view the detailed program in the attached files) , and we use "Edit Configuration" in IDEA to read the provided csv files. The argument should be correct file path and divided by a space. In this project architecture, the argument is `data/user.csv data/recipes.csv data/reviews.csv` . In this way we can access csv files by `args[]`. This approach also has flexibility for changing csv files.

Then we specify the path to the SQL DDL file and execute it. In this project architecture, the path is`"SQL/create_table_ddl_statements.sql"`. We write a method `runSchemaFromFile` to execute this SQL file and to create the necessary tables and types.

Finally, after the schema is created, we execute the import procedure, which stays the same as its parent class.

For the correctness check, we compare the row counts of each table with previous manual approach and they are all the same. This confirms that the automated import works as intended: Give this script 3 original csv files, it will create the schema and insert each record into it, producing a complete and consistent database.

4. Further Exploration for Import Efficiency

Beyond the optimizations already implemented (batching, PreparedStatement, and HashSet filtering), there are several practical directions that can further accelerate large-scale data import.

In the software-level, beyond the standard JDBC insertion, we can adopt PostgreSQL's native `COPY FROM STDIN` interface. This approach bypasses SQL parsing and, substantially reducing network round trips and transaction overhead. In practice, the import program can convert parsed records into COPY-formatted text stream and send it directly to the database through the `CopyManager` API. This method provides much higher throughput compared with batched `INSERT` operations. However, it also requires additional preprocessing effort and depends on PostgreSQL-specific APIs, making the implementation more complex than the standard JDBC workflow.

In the hardware-level, storage write speed directly impacts the import consuming time. Although the experiments were conducted on NVMe SSD, which offers higher write speed than traditional SATA SSD, there is still potential for further optimization by explore the interaction with underlying storage. In particular, loading operations are often bottlenecked by WAL calls rather than raw disk bandwidth. By allocating the WAL directory to a dedicated high-performance NVMe partition, enabling write-back caching, or increasing PostgreSQL parameters, the database can gain improvement in efficiency when importing data.

Task 4: Compare DBMS with File I/O

Basic Part

1. Test Environment

a. Hardware Specifications

- CPU Model: Intel Core Ultra 5 125H @ 3.60GHz
- Memory Capacity: 32GB
- Storage: 1 TB NVMe SSD

b. Software Specifications

- DBMS Version: PostgreSQL 17.6 (64-bit)
- Operating System: Windows 11
- Programming Language: Java
- Development Environment: Java 17 (JDK 17.0.4.1), javac 17.0.4.1
- JDBC Driver: postgresql-42.2.5.jar

2. Test Data Organization

Data Storage Description

- **user.csv**: Contains all user data (provided by the instructor), used for importing initial data into the DBMS `user` table and file system.
- **user_10000.csv**: Contains 10,000 rows of user data (created by the user), randomly selected 10,000 entries from `user.csv` with an added 3,000,000 to the `author_id` to prevent violations of primary key uniqueness. Used for DBMS `recipe` table insertion tests and file system `recipe.txt` insertion tests.
- **users table**: The table in the DBMS that stores user data (created by the user), containing data from `user.csv`, used for insert, update, query, and delete tests in the database system.
- **users.txt**: A file in the file system that stores all user data (created by the user), used for insert, update, query, and delete tests in the file system.

Data Organization in DBMS

In the PostgreSQL database, test data is organized in relational table structures. The specific implementation is as follows:

- **Table Structure Design**: The `users` table stores recipe information, which includes the following fields: `author_id` (primary key), `author_name`, `gender`, `age`, `followers_count`, `following_count`.
- **Data Types**: Different data types are used according to the field meaning, such as integer (`author_id`), text (`author_name`), and enum (`gender`), ensuring the accuracy of data types and optimization of storage space.

- **Constraints**: `author_id` as the primary key ensures the uniqueness of each record, allowing for fast location and querying.
- **Index Optimization**: The database automatically creates indexes for primary key fields, significantly improving query performance, especially for exact queries on the ID field.

Data Organization in File System

In the file system, test data is organized as text files. The specific implementation is as follows:

- **File Format**: Simple text files (e.g., `users.txt`) store all user records.
- **Record Separation**: Each record occupies one line, separated by a newline character (`\n`).
- **Field Separation**: Each record uses a semicolon (`;`) as a field separator, dividing different attributes.
- **Data Structure**: There are no explicit data type distinctions, all data is stored as strings, and type conversion is performed during program execution.
- **No Index Mechanism**: The file system itself does not provide index support, and queries require sequential scanning of the entire file.

SQL Statement Generation Method

All CRUD (Create, Read, Update, Delete) SQL statements are generated using a unified prepared statement method. The specific characteristics are as follows:

- **Parameterized Query Mode**: Placeholders (`?`) replace direct string concatenation, which is a widely used practice in various programming languages and database systems to prevent SQL injection attacks.
- **Dynamic Parameter Binding Mechanism**: Application variables are securely mapped to SQL parameters via type-safe parameter binding methods, ensuring correct and safe data type conversions.
- **Standardized SQL Statement Structure**: Standard SQL syntax is used to construct all CRUD operation statements, whether it is a `SELECT`, `INSERT`, `UPDATE`, or `DELETE`, all following the same prepared statement generation mechanism.
- **Error Handling Framework**: A unified exception handling mechanism is implemented to ensure that errors during SQL execution are correctly captured and meaningful error messages are returned.
- **Resource Lifecycle Management**: The standard connection-execute-close resource management pattern is followed to ensure that database connections and other resources are properly released after the operation is complete.
- **SQL Statement Reuse**: The prepared statement mechanism improves SQL execution efficiency by reducing redundant parsing and optimization overhead.

File Operation Mechanism

File operations are implemented using the standard Java I/O library:

- **Read Operation**: BufferedReader is used to read file content line by line, improving reading efficiency.
- **Write Operation**: FileWriter is used to write to the file, supporting append mode.

- **Temporary Files**: For delete and update operations, a temporary file strategy is employed to avoid conflicts while reading and writing simultaneously.
- **File Replacement**: After the operation is completed, the original file is deleted, and the temporary file is renamed to the original file name.

3. Test SQL Scripts and Program Source Code Description

The test involves 5 source code files: `DatabaseTest.java`, `FileTest.java`, `DatabaseOperations.java`, `FileOperation.java`, `DatabaseBetter.java`, and `user_10000.java`. Below is the functionality description of each file (the specific operation code is in the attachments):

- **DatabaseTest.java**:
  - Calls the DBMS test functions for insertion, query, update, and deletion, and prints the test results. Repeats the experiment five times, printing the result of each experiment and calculating the average.
- **FileTest.java**:
  - Calls the file test functions for insertion, query, update, and deletion, and prints the test results. Repeats the experiment five times, printing the result of each experiment and calculating the average.
- **DatabaseOperations.java**:
  - Interacts with PostgreSQL through JDBC, with functionalities including connection management and performing insert, update, query, and delete operations.
  - **Insert Operation (addUsers)**: Uses a parameterized SQL `INSERT INTO` statement to insert data.
  - **Query Operation (queryUsers)**: Uses a parameterized SQL `SELECT COUNT(*) FROM` statement to count matching records.
  - **Update Operation (updateUsersAge)**: Uses a parameterized SQL `UPDATE` statement to increment the age of matching users.
  - **Delete Operation (deleteUsers)**: Uses a parameterized SQL `DELETE FROM` statement to delete matching records.
- **FileManipulation.java**:
  - Manages user data through file I/O operations, with functionalities including file path management, data format, and CRUD operations for insert, query, update, and delete.

4. Performance Comparison Analysis

Operation Type	DBMS Test Time (ms)	DBMS Optimized Test Time (ms)	File I/O Test Time (ms)
Insert (insertTest)	1681	268	1844
Delete (deleteTest)	1556	118	9856
Query (selectTest)	1022	533	8945

Operation Type	DBMS Test Time (ms)	DBMS Optimized Test Time (ms)	File I/O Test Time (ms)
Update (updateTest)	2020	237	12043

Analysis:

1. When processing data (especially large-scale or complex datasets), using a DBMS such as PostgreSQL is far superior to managing data with a file system. The performance difference between DBMS and file I/O is significant. In conclusion, database systems have a clear advantage over direct file use when handling data.
2. For frequently operated attributes, introducing indexes, implementing transaction control, and using batch operations can significantly improve database performance.

Advanced1: High-Concurrency Performance Testing

Experiment Objectives

- Test PostgreSQL's response performance under different levels of concurrent load.
- Observe how throughput changes as the number of threads increases.
- Compare database performance with pure File I/O (sequential CSV scanning).

Experimental Environment

Same as Task 4 Basic Part.

Methodology

To cover the full range from low load to high load, this experiment uses the following thread counts:

1, 5, 10, 20, 50, 100

- **1:** Single-thread baseline
- **5–10:** Light concurrency (common in typical applications)
- **20–50:** Noticeable concurrent pressure (similar to API server scenarios)
- **100:** Required "high concurrency" level

Operations per thread: 5,000 query operations To reduce randomness, **each thread count is tested three times**, and the average result is used.

SQL Statement

A random-access approach is adopted to avoid cache effects. Each query randomly selects a `RecipeId` in the range **1–100000**: `SELECT RecipeId, Name FROM recipes WHERE RecipeId = ?;`

Code Explanation

DBMS:

A Java thread pool is used to simulate high-concurrency database access. In each round of testing, a fixed number of threads is created, and each thread independently executes multiple database queries. Queries use random `recipe_id` values to avoid caching effects and ensure fairness. The thread pool schedules tasks concurrently, generating a large amount of simultaneous requests to the database. After all threads finish execution, total execution time and throughput (QPS) are computed to evaluate PostgreSQL performance under various concurrency levels. (The Java thread-pool code is included in the appendix.)

File I/O:

A Java thread pool simulates high-concurrency file-reading scenarios. In each test round, a fixed number of threads is created, and each thread independently performs multiple file queries. Each query generates a random `recipe_id` and scans the preloaded CSV content to find the corresponding row. The CSV file is fully loaded into memory at startup to avoid repeated disk access. The thread pool schedules tasks concurrently so multiple threads scan the file at the same time. After all threads complete, the program calculates total execution time and throughput (QPS), which reflects File I/O performance under different concurrency levels.

(The Java thread-pool code is included in the appendix.)

Experimental Results

Threads	DBMS Avg QPS (ops/sec)	File I/O Avg QPS (ops/sec)
1	8,068	323
5	30,821	2,468
10	41,462	4,869
20	57,424	5,080
50	72,132	5,306
100	91,145	3,869

(Screenshots from each execution are included in the appendix.)

Conclusion

The experiment confirms that PostgreSQL scales well under multi-user concurrent query workloads and maintains high throughput even at high thread counts.

In contrast, pure File I/O—lacking indexing and caching mechanisms—hits a performance bottleneck quickly as concurrency increases.

These results demonstrate that database systems provide essential performance advantages in high-concurrency application scenarios, making them far superior to raw file-based data access.

Advanced2: Performance Comparison Between PostgreSQL and MySQL

1. Experiment Environment

- **MySQL Version**: 8.0.44
- **MySQL JDBC Driver**: mysql-connector-j-9.5.0.jar
- Other environmental settings are consistent with the basic part of Task 4

2. Data & Schema

Test Data Source:

For this experiment, the dataset `user.csv` is used, which contains about 300,000 rows of user data, provided by the course project. The `user.csv` file includes the following fields:

- `AuthorId`: Unique user identifier
- `AuthorName`: User's name
- `Gender`, `Age`: Basic attributes
- `Followers`, `Following`: User relationships
- `FollowerUsers`, `FollowingUsers` (ignored in this experiment)

Table Creation (Cross-database Compatible Version):

```sql
CREATE TABLE users (
AuthorId     INT PRIMARY KEY,
AuthorName   VARCHAR(100),
Gender       VARCHAR(10),
Age          INT,
Followers    INT,
Following    INT
);
```

3. Experiment Design

This experiment performs a comprehensive comparison of the two databases based on the following metrics:

- **Metrics**:
  1. Single INSERT
  2. Batch INSERT
  3. UPDATE
  4. Range Query (WHERE age BETWEEN)
  5. DELETE
- **Timing Method**:

  `System.nanoTime()` in Java is used for timing each operation.
- **Repetitions**:

  Each test is repeated 5 times.

The first few runs are affected by JVM JIT compilation, cache misses, etc., commonly known as "cold start" factors. To mitigate initial fluctuations and ensure reliable, reproducible results, the first two runs serve as JVM warm-ups. The average is taken from the last three runs.

- **Inter-Experiment Interval**:

  There is an interval of approximately 1 second between each experiment to avoid interference from the previous experiment's garbage collection (GC) and I/O flush.

4. Test and Results

All experiment codes and results are available in the appendix.

(1) INSERT Performance

Test data is generated using the following steps:

- Data is read and cleaned from the original `user.csv` file.
- To improve data processing efficiency and focus on query performance, the `FollowerUsers` and `FollowingUsers` columns, which are irrelevant to the query conditions, are removed.
- Random sampling is used to generate datasets of 10,000, 100,000, and 300,000 unique rows, resulting in the test files `user_10k.csv`, `user_100k.csv`, and `user_300k.csv`.

JDBC Single INSERT (10k rows)

Using a Java program, each INSERT is executed one at a time. After each insert, the database processes the transaction. Results from five repeated experiments are as follows:

Total Time (ms)	**1st**	**2nd**	**3rd**	**4th**	**5th**	**Average**
MySQL	12255	11371	12187	12744	11257	12029
PostgreSQL	1672	1554	1608	1577	1653	1613
Average Time (ms)	1st	2nd	3rd	4th	5th	Average
MySQL	1.178	1.090	1.172	1.226	1.072	1.156
PostgreSQL	0.138	0.127	0.134	0.129	0.135	0.133

PostgreSQL is faster and more stable, with smaller time fluctuations compared to **MySQL**, which shows significant variation between runs.

For tasks that involve frequent single-row inserts, **PostgreSQL is recommended** for better performance and lower latency.

JDBC Batch INSERT (batch size = 1000)

Using `addBatch()` and `executeBatch()`, multiple SQL statements are sent to the database for batch processing. The results are as follows (time in milliseconds):

Data Volume:10k	**1st**	**2nd**	**3rd**	**4th**	**5th**	**Average**
MySQL	284	269	253	282	241	259
PostgreSQL	157	154	155	165	147	156
Data Volume:100k	1st	2nd	3rd	4th	5th	Average

Data Volume:10k	1st	2nd	3rd	4th	5th	Average
MySQL	1654	1829	1980	1853	1816	1883
PostgreSQL	893	893	930	879	920	910
Data Volume:300k	1st	2nd	3rd	4th	5th	Average
MySQL	4875	5285	5022	4815	5240	5026
PostgreSQL	2698	2747	2505	2625	2811	2647

PostgreSQL outperforms **MySQL** in batch insert performance, especially when dealing with larger datasets.

For large data processing, **PostgreSQL's performance is more stable**, maintaining a lower insertion time, while **MySQL's insertion time increases significantly** as the data volume grows.

(2) UPDATE Performance

The performance of MySQL and PostgreSQL is compared for small-scale (1k records) and large-scale (100k records) update operations. The results are as follows (time in milliseconds):

Data Volume:1k	1st	2nd	3rd	4th	5th	Average
MySQL	703	696	837	830	761	809
PostgreSQL	398	436	455	456	369	427
Data Volume:100k	1st	2nd	3rd	4th	5th	Average
MySQL	13376	14770	14248	15847	15021	15039
PostgreSQL	3871	5224	5020	5047	4893	4987

PostgreSQL performs better than **MySQL** in both small and large-scale update operations, especially in large data updates.

(3) Query Performance (SELECT)

The task was to count users with ages between 20 and 40. Execution times and matching user counts were recorded for each database:

Time(ms)	1st	2nd	3rd	4th	5th	Average
MySQL	479	459	469	473	469	470
PostgreSQL	172	166	166	173	184	174

The results show that **PostgreSQL** performs significantly better for range queries, especially with large datasets.

(4) DELETE Performance (DELETE)

The DELETE operations for MySQL and PostgreSQL were compared by deleting users older than 40. Each delete operation used transactions and was rolled back after each execution to avoid modifying data. The results are as follows (time in milliseconds):

Time(ms)	1st	2nd	3rd	4th	5th	Average
MySQL	1717	1286	1226	1264	1298	1263
PostgreSQL	505	252	205	199	175	193

PostgreSQL performs far better than **MySQL**, especially in deleting large amounts of data. The delete speed is faster, and the response time is shorter.

5.Summary

- **PostgreSQL** consistently outperforms **MySQL** across all four operations (INSERT, Batch INSERT, UPDATE, DELETE), especially when handling large datasets. It offers greater stability and efficiency.
- **MySQL** can be competitive in certain scenarios but generally lags behind **PostgreSQL**, especially when handling complex queries and large data volumes.
- For projects requiring efficient data handling, especially frequent inserts, queries, and deletions, **PostgreSQL is the recommended database** due to its superior performance and lower latency.

Bonus

The report above explored the impact of index settings, transaction management, bulk import, parallel processing, and multi-thread optimization on improving database performance. The following will analyze the role of configuration tuning in improving database operation efficiency. The following SQL commands can be used to query memory-related configurations in PostgreSQL:

`SHOW shared_buffers;`

`SHOW work_mem;`

`SHOW maintenance_work_mem;`

`SHOW effective_cache_size;`

The default configuration is as follows:

- **shared_buffers**: 128MB
- **work_mem**: 4MB
- **maintenance_work_mem**: 64MB
- **effective_cache_size**: 4GB

shared_buffers: Determines the amount of memory PostgreSQL uses to cache table and index data. Larger buffers can improve the speed of accessing data blocks and reduce disk access.

work_mem: Defines the amount of memory allocated per query operation, including memory space for sorting, hash joins, etc. Increasing **work_mem** can reduce the frequency of temporary file creation for these operations, thus improving performance.

sort_buffer_size: Defines the memory buffer size for sorting operations. Increasing this value can speed up sorting operations and avoid the use of temporary files on disk.

maintenance_work_mem: Used for operations such as creating indexes, performing VACUUM, or REINDEX. Increasing **maintenance_work_mem** can significantly improve performance for these tasks.

effective_cache_size: Estimates the size of the operating system's cache, which influences the query optimizer's decisions. Increasing **effective_cache_size** tells the optimizer that more data might already be cached in the operating system, so it may choose to use index scans instead of sequential scans, thereby improving query efficiency.

It is important to note that when optimizing memory configurations, adjustments should be made based on the actual hardware environment and data characteristics. Overly high memory configurations can lead to wasted system resources, so the optimal configuration should be determined through testing and under real load conditions.

Achieving extremely high operational efficiency is not solely accomplished by one performance operation; it requires a combination of the performance optimizations mentioned above.