CS307 Fall 2025 - Project Part I

李语尚 (12412308)
丁哲昊 (12412310)

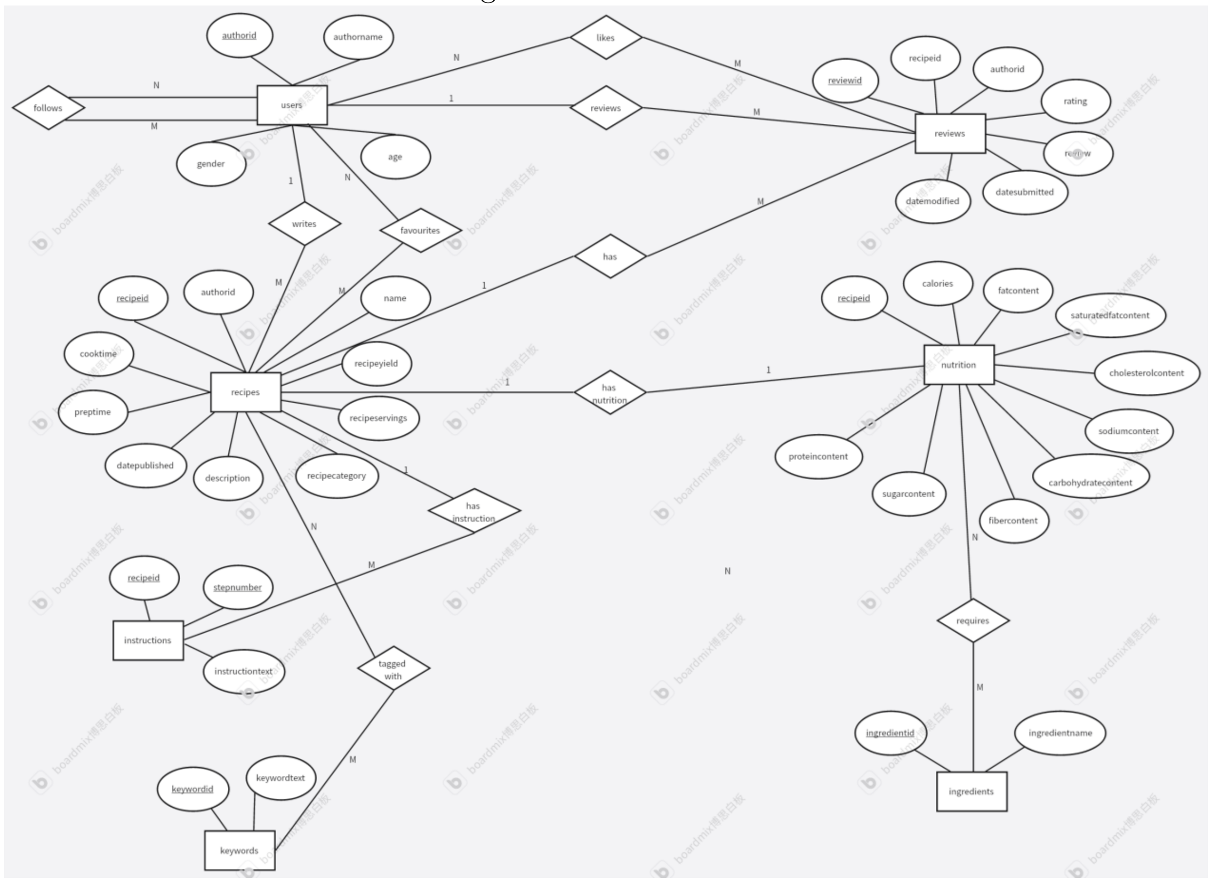
November 17, 2025

1 Group Information

- 李语尚 - 12412308
- 丁哲昊 - 12412310

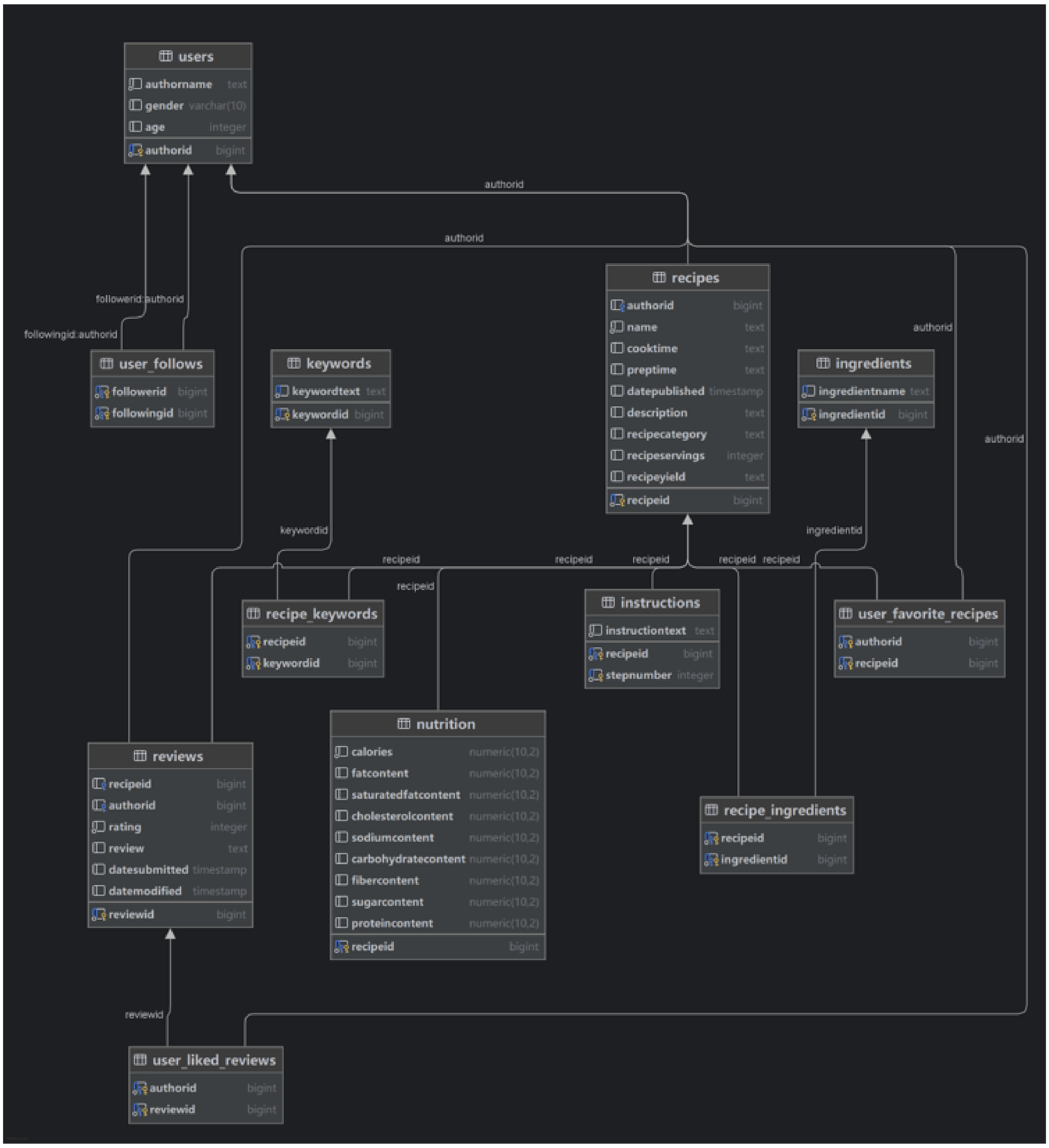
2 Task 1: E-R Diagram

We use boardmix to draw the E-R diagram.



3 Task 2: Database Design

3.1 E-R Diagram by Datagrip



3.2 Table Design Description

In order to model the relationships within the recipe and user ecosystem, and to ensure the database design meets the requirements of data integrity and expandability, we have designed the following tables, categorized into Core Entities, One-to-One/One-to-Many Relations, and Many-to-Many Junctions.

3.2.1 Core Entity Tables

users

- **authorid**: The unique identifier for the user, which serves as the primary key of this table.
- **authorname**: The name of the user.
- **gender**: The gender of the user.
- **age**: The age of the user.

recipes

- **recipeid**: The unique identification number for a recipe, serving as the primary key.
- **authorid**: A foreign key referencing `users.authorid`, linking the recipe to its creator. The `ON DELETE SET NULL` constraint ensures the recipe remains even if the user is deleted.
- **name**: The title of the recipe.
- **datepublished**: The timestamp when the recipe was published.
- **recipeservings**: The number of servings the recipe yields.

reviews

- **reviewid**: The unique identifier for a review, which is the primary key.
- **recipeid**: A foreign key referencing `recipes.recipeid`, indicating which recipe is being reviewed (`ON DELETE CASCADE`).
- **authorid**: A foreign key referencing `users.authorid`, indicating which user wrote the review (`ON DELETE CASCADE`).
- **rating**: The numerical rating given to the recipe (not null).
- **review**: The text content of the review.

3.2.2 One-to-One and One-to-Many Relation Tables

nutrition

- **recipeid**: Acts as both the primary key and a foreign key referencing `recipes.recipeid` (`ON DELETE CASCADE`), enforcing a one-to-one relationship with the recipes table.
- **calories**: The caloric content (not null).
- **fatcontent, proteincontent, sugarcontent, etc.**: Detailed nutritional information for the specific recipe.

instructions

- **Composite Primary Key**: `(recipeid, stepnumber)`, uniquely identifying each step for a given recipe.
- **recipeid**: A foreign key referencing `recipes.recipeid` (`ON DELETE CASCADE`).
- **stepnumber**: The order of the instruction step (1, 2, 3, …).
- **instructiontext**: The description of the cooking step.

3.2.3 Many-to-Many (M:M) Connection Tables

ingredients

- **ingredientid**: A self-increasing primary key.
- **ingredientname**: The name of the ingredient, which must be unique and not null.

recipe_ingredients

- **Composite Primary Key**: `(recipeid, ingredientid)`, linking a recipe to an ingredient.
- This table resolves the M:M relationship between recipes and ingredients.

keywords

- **keywordid**: A self-increasing primary key.
- **keywordtext**: The text of the keyword, which must be unique and not null.

recipe_keywords

- **Composite Primary Key**: `(recipeid, keywordid)`, linking a recipe to a keyword.
- This table resolves the M:M relationship between recipes and keywords.

user_favorite_recipes

- **Composite Primary Key**: `(authorid, recipeid)`, recording a user's favorite recipe.
- This table resolves the M:M relationship between users and recipes (Favorites).

user_liked_reviews

- **Composite Primary Key**: `(authorid, reviewid)`, recording which user liked which review.
- This table resolves the M:M relationship between users and reviews (Likes).

user_follows

- **Composite Primary Key**: `(followerid, followingid)`, recording a user following another user (a self-referencing M:M relationship).
- **followerid**: The ID of the user who is following.
- **followingid**: The ID of the user who is being followed.

3.3 Scalability

The tables we've designed exhibit strong scalability features, primarily due to the use of dedicated junction tables for many-to-many relationships and effective foreign key management:

Table users and recipes (Core Entities):

- The core entities are designed with simple primary keys (`authorid`, `recipeid`) and efficient data types (`bigint`), making lookups and direct access operations fast.
- The one-to-many relationship (one user publishes many recipes) is handled efficiently via `recipes.authorid`. New users and recipes can be added easily without complex table rewriting.

Junction Tables (e.g., recipe_ingredients, user_follows):

- Tables like `recipe_ingredients`, `recipe_keywords`, `user_favorite_recipes`, `user_liked_rev` and `user_follows` are designed with composite primary keys consisting of two `bigint` foreign keys.
- This structure efficiently models M:M relationships, allowing for an indefinite growth in connections (e.g., a recipe can have virtually unlimited ingredients, and a user can follow unlimited other users).
- Adding or deleting a new relationship (e.g., a user favoring a recipe) involves a simple insertion or deletion of a row into these junction tables, which is a fast, isolated operation, promoting high insertion throughput.

4 Task 3: Data Import

4.1 System Architecture

4.1.1 Project Structure

Our data import system is organized into task-specific directories:

```
DataBase/
   src/main/
      task1/
         1.png                    - ER diagram
      task2/
      2.png                     - ER diagram with postgres
   task3/                        - Task 3: Data Import Module
      Main.java                 - Entry point for batch insert method
      MainCompare.java          - Entry point for single insert method
      CsvDataImporter.java      - Batch insert import orchestrator
      CsvDataImporterCompare.java - Single insert import orchestrator
      DataReader.java           - CSV parsing utilities
      DataWriter.java           - Batch insert operations
      DataQuery.java            - Query utilities for validation
      database_schema.sql       - Database schema definition (12 tables)
   task4/                        - Task 4: Performance Testing Module
      PerformanceTest.java      - Main performance test orchestrator
      FileIOOperations.java     - File I/O operations for comparison
      BTreeIndex.java          - In-memory B-Tree index implementation
   common/                       - Shared Components
       ConnectionManager.java    - Database connection and transaction management
       DatabaseConfig.java       - Configuration loader (from db.properties)
       db.properties             - Database connection configuration
final_data/                      - Input CSV Data
   user.csv                      - User data (299,892 records)
   recipes.csv                   - Recipe data (928,283 records)
   reviews.csv                   - Review data (1,639,086 records)
postgresql-42.2.5.jar            - PostgreSQL JDBC driver
```

4.1.2 Key Components

1. Main Entry Points:

- `Main.java`: Orchestrates the batch insert import process
- `MainCompare.java`: Orchestrates the single insert import process for comparison

2. Import Orchestrators:

- `CsvDataImporter.java`: Implements optimized batch insert method
  - Uses `PreparedStatement.addBatch()` with batch size 1000
  - Implements in-memory caching for keywords and ingredients
  - Uses transaction management for atomicity
- `CsvDataImporterCompare.java`: Implements baseline single insert method
  - Uses `PreparedStatement.executeUpdate()` for each record
  - No batching or caching optimizations
  - Serves as baseline for performance comparison

3. Supporting Utilities (Task 3):

- `DataReader.java`: Handles CSV file reading and parsing
  - Custom CSV parser supporting quoted fields with escaped quotes

  - Parses comma-separated ID lists in various formats (`"""id1,id2"""`, `"id1,id2"`, `id1,id2`)
  - Data type conversion utilities (`parseLong`, `parseInteger`, `parseDouble`, `parseTimestamp`)

- `DataWriter.java`: Provides batch insert operations with configurable batch size (default: 1000)
- `DataQuery.java`: Provides query utilities for data validation and record counting

4. Shared Components (common/):

- `ConnectionManager.java`: Manages database connections and transactions
  - Handles connection lifecycle (creation, validation, closing)
  - Provides transaction management (`commit`, `rollback`, `setAutoCommit`)
  - Used by both Task 3 and Task 4
- `DatabaseConfig.java`: Loads database configuration from `db.properties`
  - Supports file-based configuration or default values
  - Provides connection parameters (host, port, database, user, password)
- `db.properties`: Database connection configuration file
  - Contains database connection settings
  - Shared between Task 3 and Task 4

4.1.3 Execution Steps

Prerequisites:

1. Execute `database_schema.sql` to create all 12 tables
2. Configure `db.properties` with database connection settings:

```
db.host=localhost
db.port=your database port
db.database=sustc_recipe_db
db.user=postgres
db.password=your_password
```

3. Place CSV files in `final_data/` directory

Compilation and Execution:

```
# Compile all Java files
javac -encoding UTF-8 -cp ".;postgresql-42.2.5.jar"
      src\main\task3\*.java src\main\common\*.java -d .

# Run batch insert method (optimized)
```

```
java -cp ".;postgresql-42.2.5.jar" main.Main

# Run single insert method (for comparison)
java -Xmx16g -cp ".;postgresql-42.2.5.jar" main.MainCompare
```

Note: The project is organized into separate task directories (`src/main/task3/` for Task 3, `src/main/task4/` for Task 4, `src/main/common/` for shared components) for clear separation of concerns and modularity.

4.2 Data Import Process

In this section, we describe our data import process step by step: how we read data from CSV files, clean and prepare the data, and finally import it into the database. We also compare two different import methods and draw conclusions about optimal import strategies.

4.2.1 Step 1: Reading Data from CSV Files

The import process begins by reading data from three main CSV files located in the `final_data/` directory:

- `user.csv` - Contains user information (AuthorId, AuthorName, Gender, Age, FollowerUsers, etc.)
- `recipes.csv` - Contains recipe information (RecipeId, AuthorId, Name, CookTime, Keywords, RecipeIngredientParts, etc.)
- `reviews.csv` - Contains review information (ReviewId, RecipeId, AuthorId, Rating, Review, etc.)

We use a custom CSV parser (`DataReader.java`) that handles:

- Quoted fields with escaped quotes (e.g., `"""value"""`)
- Comma-separated ID lists in various formats
- UTF-8 encoding for proper character handling

The parser reads all CSV records into memory as `List<Map<String, String>>`, where each map represents a row with column names as keys.

Raw Data Statistics:

- Users: 299,892 records read from `user.csv`
- Recipes: 928,283 records read from `recipes.csv`
- Reviews: 1,639,086 records read from `reviews.csv`

4.2.2 Step 2: Data Cleaning and Preparation

After reading the raw CSV data, we perform comprehensive data cleaning to ensure data quality before import:

1. **Duplicate Detection:** We use `Set<Long> seenIds` to track primary keys and skip duplicate records. For example, if multiple rows have the same `authorid`, only the first occurrence is kept.
2. **Null Value Handling:** Records with null primary keys or required fields are filtered out. For instance:
   - Users without `AuthorId` or `AuthorName` are skipped
   - Recipes without `RecipeId` are skipped
   - Reviews without `ReviewId` or `Rating` are skipped
3. **Field Normalization:** We use `DataReader.normalizeField()` to:
   - Trim leading and trailing whitespace
   - Convert empty strings to null
   - Handle special characters
4. **Data Type Conversion:** CSV string values are converted to appropriate database types:
   - `parseLong()` for ID fields (with null handling for invalid values)
   - `parseInteger()` for numeric fields (Age, Rating, etc.)
   - `parseDouble()` for decimal fields (Calories, FatContent, etc.)
   - `parseTimestamp()` for date fields (DatePublished, DateSubmitted, etc.)
5. **Complex Field Parsing:** We parse comma-separated ID lists (e.g., `FollowerUsers`, `FavoriteUsers`, `Likes`) with support for multiple formats:
   - Triple-quoted format: `"""id1,id2,id3"""`
   - Double-quoted format: `"id1,id2,id3"`
   - Unquoted format: `id1,id2,id3`
6. **Conditional Data Insertion:** Some records are conditionally inserted:
   - `nutrition` records are only inserted when `calories` is not null
   - M2M relationships with null foreign keys are skipped

Data Cleaning Results:

- Users: 299,892 valid records (100% retention, no duplicates)
- Recipes: 522,517 valid records (56% retention, 405,766 filtered due to duplicates/invalid data)

- Reviews: 1,401,983 valid records (85% retention, 237,103 filtered due to invalid data)

After cleaning, we prepare data structures for all 12 tables:

- Main entity tables: `users`, `recipes`, `reviews`, `nutrition`, `instructions`
- M2M base tables: `keywords`, `ingredients`
- M2M association tables: `recipe_keywords`, `recipe_ingredients`, `user_favorite_recipes`, `user_liked_reviews`, `user_follows`

4.2.3 Step 3: Database Import

The import process follows a three-phase approach to ensure data integrity and optimize performance:

Phase 1: M2M Base Tables

- Extract unique keywords and ingredients from recipe data
- Insert into `keywords` and `ingredients` tables
- Commit to ensure tables exist for subsequent operations
- Build in-memory caches (`keywordCache`, `ingredientCache`) for fast ID lookups

Phase 2: Main Entity Tables

- Insert `users` table (299,892 records)
- Insert `recipes` table (522,517 records)
- Insert `reviews` table (1,401,983 records)
- Insert `nutrition` table (486,050 records)
- Insert `instructions` table (1,147,650 records)

Phase 3: M2M Association Tables

- Use cached keyword/ingredient IDs to build `recipe_keywords` (2,313,507 records)
- Use cached ingredient IDs to build `recipe_ingredients` (3,711,215 records)
- Parse user relationship data to build `user_favorite_recipes` (1,251,900 records)
- Parse review like data to build `user_liked_reviews` (4,995,823 records)
- Parse follower data to build `user_follows` (1,663,578 records)

All insertions use `ON CONFLICT DO NOTHING` to ensure idempotent imports (safe to re-run). The entire process is wrapped in a single transaction with `autoCommit=false`, committing on success or rolling back on error.

Table 1: Record Counts for Each Table

Table Name	Number of Records
`users`	299,892
`recipes`	522,517
`reviews`	1,401,983
`nutrition`	486,050
`instructions`	1,147,650
`keywords`	309
`recipe_keywords`	2,313,507
`ingredients`	7,216
`recipe_ingredients`	3,711,215
`user_favorite_recipes`	1,251,900
`user_liked_reviews`	4,995,823
`user_follows`	1,663,578

4.2.4 Number of Data Entries

Table 1 shows the number of records imported into each table.

Total Records: Approximately 18.1 million records across 12 tables.

Import Statistics (Batch Insert Method):

- Total import time: ∼300 seconds (5.0 minutes, depending on system load)
- All tables successfully populated with data
- Data integrity verified through record count validation

4.3 Import Method Comparison

To understand the performance implications of different import strategies, we implemented and compared two methods:

4.3.1 Method 1: Single Insert (Baseline)

The first method (`CsvDataImporterCompare.java`) uses a straightforward approach:

- For each record, create a `PreparedStatement` with the insert SQL
- Execute `executeUpdate()` immediately for each record
- Each insert is a separate database round-trip

Implementation:

```
for (Map<String, Object> row : data) {
    PreparedStatement pstmt = conn.prepareStatement(sql);
    // Set parameters...
    pstmt.executeUpdate();  // Single insert
}
```

4.3.2 Method 2: Batch Insert (Optimized)

The second method (`CsvDataImporter.java`) uses batch processing:

- Create `PreparedStatement` once before the loop
- Use `addBatch()` to add multiple records to a batch
- Execute `executeBatch()` when batch size reaches 1000
- Use `setAutoCommit(false)` to wrap all inserts in a single transaction

Implementation:

```
PreparedStatement pstmt = conn.prepareStatement(sql);
conn.setAutoCommit(false);
for (Map<String, Object> row : data) {
    // Set parameters...
    pstmt.addBatch();  // Add to batch
    if (batchCount % 1000 == 0) {
        pstmt.executeBatch();  // Execute batch
    }
}
pstmt.executeBatch();  // Execute remaining
conn.commit();
```

4.3.3 Key Optimizations in Batch Method

1. **Pre-compiled SQL:** `PreparedStatement` is created once and reused, eliminating SQL parsing overhead for each insert
2. **Batch Processing:** Multiple inserts are combined into a single network round-trip, reducing network latency
3. **In-memory Caching:** `keywordCache` and `ingredientCache` eliminate ∼6 million database lookups when building M2M tables
4. **Transaction Management:** Single transaction reduces commit overhead

Table 2 shows the performance comparison between these two methods.

4.4 Performance Comparison Results

Table 2 shows the detailed performance comparison between the two methods for importing 18.1 million records across 12 tables.

Performance Improvement: The batch insert method is approximately **4.7× faster** than the single insert method for the complete dataset (300 seconds vs 1402.8 seconds).

Table 2: Data Import Performance Comparison

Table	Records	Batch Insert	Single Insert
`users`	299,892	∼15–20 s	∼15–20 s
`recipes`	522,517	∼25–30 s	∼30–40 s
`reviews`	1,401,983	∼40–50 s	∼80–100 s
`nutrition`	486,050	∼15–20 s	∼25–30 s
`instructions`	1,147,650	∼30–40 s	∼60–70 s
`recipe_keywords`	2,313,507	∼60–80 s	∼120–150 s
`recipe_ingredients`	3,711,215	∼80–100 s	∼200–250 s
`user_favorite_recipes`	1,251,900	∼30–40 s	∼80–100 s
`user_liked_reviews`	4,995,823	∼50–70 s	∼350–400 s
`user_follows`	1,663,578	∼30–40 s	∼100–120 s
Total	**18.1M**	**∼300 s (5.0 min)**	**1402.8 s (23.4 min)**

4.4.1 Performance Analysis

Network Overhead Reduction: The most significant factor is the reduction in network round-trips. For 18.1 million records:

- Single insert method: 18.1 million network round-trips (one per record)
- Batch insert method: ∼18,100 network round-trips (one per batch of 1000)
- Reduction: approximately 1000× fewer network operations

SQL Parsing Overhead:

- Single insert: Each `executeUpdate()` requires SQL parsing and compilation
- Batch insert: SQL is parsed once per batch, reused for 1000 inserts
- Reduction: approximately 1000× fewer parsing operations

Transaction Overhead:

- Single insert: Each insert may trigger implicit commit operations
- Batch insert: Single transaction for all inserts, one commit at the end
- Reduction: eliminates millions of commit operations

In-Memory Caching: Building `keywordCache` and `ingredientCache` eliminates ∼6 million database lookups when constructing M2M association tables. Without caching, each M2M relationship would require a database query to resolve keyword/ingredient IDs.

4.5 Conclusion

Based on our comprehensive comparison, we conclude that **batch insert is the optimal method for large-scale data imports**. The key findings are:

1. **Batch processing is essential:** Combining multiple inserts into batches reduces network overhead by approximately 1000×, which is the primary performance gain.

2. **Pre-compiled statements matter:** Reusing `PreparedStatement` objects eliminates redundant SQL parsing, providing significant overhead reduction.

3. **Transaction management is critical:** Using a single transaction with `autoCommit=false` eliminates millions of commit operations, further improving performance.

4. **In-memory caching accelerates M2M tables:** Building caches for frequently accessed lookup data eliminates millions of database queries.

5. **Practical impact:** For our dataset of 18.1 million records, batch insert completes in approximately 5 minutes, while single insert requires 23.4 minutes. This 4.7× speedup makes batch insert essential for production use.

Recommendation: For any data import operation involving more than a few thousand records, batch insert with appropriate batch size (1000 is optimal for our use case) should be the standard approach. The single insert method, while simpler to implement and understand, is impractical for production use with large datasets.

5 Task 4: Compare DBMS with File I/O

5.1 Test Environment and Data Organization

5.1.1 Test Environment

Tests were conducted on the following platform:

- **Operating System:** Windows 11
- **Database:** PostgreSQL 17.6
- **Programming Language:** Java JDK 17.0.4
- **CPU:** AMD Ryzen 9 7940HX
- **RAM:** 16GB

5.1.2 Test Data Organization

We use programmatic test data generation method implemented in `PerformanceTest.generateTestDat` to create test data with configurable sizes (5,000, 10,000, and 50,000 records).

The generated test data is stored in two formats for comparison:

- **Database:** PostgreSQL table `test_performance` with schema (`id, name, value, category`)
- **File I/O:** CSV files written to `test_data/` directory

Both formats use identical data to ensure fair comparison. The data is generated simultaneously for both storage methods.

5.1.3 Test Methods

We compare the following methods for data operations:

1. **Database Operations (DBMS):**
    - Insert operations: Single-threaded vs multi-threaded (2, 4, 8 threads) batch inserts
    - Query operations: Three types of SQL queries executed against PostgreSQL
2. **File I/O Operations:**
    - Write operations: Writing test data to CSV files
    - Query operations: Reading and parsing CSV files to perform searches
3. **In-Memory Operations:**
    - Linear search: Using `ArrayList` for sequential search
    - Indexed search: Using `BTreeIndex` for indexed search

5.1.4 Test Queries

Three types of SQL queries are tested to evaluate different database operations:

1. **Point Query:** `SELECT * FROM test_performance WHERE id = ?`
    - Tests primary key index lookup performance
    - Random `id` values generated for each query
2. **Range Query:** `SELECT * FROM test_performance WHERE value >= ? AND value <= ?`
    - Tests range scan performance
    - Random range boundaries generated for each query
3. **Complex Aggregation Query:** `SELECT category, COUNT(*), AVG(value), MAX(value) FROM test_performance WHERE category = ? AND value >= ? GROUP BY category`
    - Tests aggregation and grouping performance
    - Random `category` and `value` thresholds generated for each query

Each query type is executed 1000 times with randomly generated parameters, and the average execution time is calculated for statistical significance.

5.1.5 Source Code Architecture

The performance testing implementation is organized in `src/main/task4/`, with shared components in `src/main/common/`. The architecture consists of the following key components:

1. PerformanceTest.java (src/main/task4/): This is the main orchestrator class that coordinates all performance tests. Its key responsibilities include:

- **Test Data Generation:** `generateTestData(int size)` programmatically generates test data with configurable sizes (5k, 10k, 50k records). Each record contains `id`, `name`, `value`, and `category` fields with random values.
- **Insert Performance Testing:**
  - `testSingleThreadInsert()` - Tests single-threaded batch insert performance
  - `testMultiThreadInsert()` - Tests multi-threaded batch insert performance with configurable thread counts (2, 4, 8 threads)
- **Query Performance Testing:**
  - `testDatabaseQuery()` - Tests point query performance using `WHERE id = ?`
  - `testRangeQuery()` - Tests range query performance using `WHERE value >= ? AND value <= ?`
  - `testComplexQuery()` - Tests complex aggregation query performance with `GROUP BY`
- **Test Orchestration:**
  - `runFullPerformanceTest()` - Executes comprehensive performance tests including DBMS vs File I/O comparison
  - `runAdvancedPerformanceTest()` - Executes advanced tests with different data sizes, thread counts, and query types
- **Performance Measurement:** Uses `System.nanoTime()` for high-precision timing measurements

2. FileIOOperations.java (src/main/task4/): This class handles all File I/O operations for comparison with database operations. Its main methods include:

- `writeDataToFile()` - Writes test data to CSV files in `test_data/` directory. This simulates database insert operations and measures write performance.
- `searchFileForId()` - Performs point queries by reading and parsing CSV files line by line. Each query opens the file, reads sequentially, and searches for matching records. This represents the baseline File I/O query performance.
- `loadAllDataToMemory()` - Loads all CSV data into an `ArrayList` in memory for in-memory linear search comparison.
- `loadAllDataToBTree()` - Loads CSV data and builds a B-Tree index in memory for indexed search comparison.

3. BTreeIndex.java (src/main/task4/): This class implements an in-memory B-Tree index data structure for comparison with database indexing. Key features:

- **Data Structure:** Implements a B-Tree with configurable order (default order 3) for efficient key-value storage and retrieval
- **Operations:**
  - `put(key, value)` - Inserts a key-value pair into the B-Tree
  - `get(key)` - Searches for a value by key using O(log n) time complexity
  - `rangeQuery(minKey, maxKey)` - Performs range queries to find all values within a key range
- **Purpose:** Demonstrates the performance advantage of indexed data structures compared to linear search, providing a baseline for understanding database indexing benefits

Compilation and Execution:

```
# Compile all Java files
javac -encoding UTF-8 -cp ".;postgresql-42.2.5.jar"
      src\main\task4\*.java src\main\common\*.java -d .

# Run full performance test
java -cp ".;postgresql-42.2.5.jar" main.Main

# Run advanced performance test
java -cp ".;postgresql-42.2.5.jar" main.Main advanced
```

Project Organization: The source code is organized into task-specific directories (`src/main/task4/` for Task 4, `src/main/common/` for shared components) for clear separation of concerns and modularity.

5.2 Performance Comparison Results

5.2.1 Insert Performance

Table 3 shows single-threaded vs multi-threaded insert performance.

Table 3: Insert Performance Comparison

Data Size	Single-threaded	Multi-threaded (4)	Ratio
5,000 records	95 ms	162 ms	0.59×
10,000 records	75 ms	213 ms	0.35×
50,000 records	354 ms	272 ms	1.30×

5.2.2 Query Performance: DBMS vs File I/O

Table 4 compares database queries with File I/O operations.

Table 4: Query Performance: DBMS vs File I/O

Data Size	DBMS (ms)	File I/O (ms)	Speedup
5,000 records	0.1570	0.4458	2.84×
10,000 records	0.0721	0.5822	8.08×
50,000 records	0.0717	2.3654	33.00×

5.2.3 In-Memory Search Performance

Table 5 shows in-memory search performance (10,000 records, 1000 queries).

Table 5: In-Memory Search Performance

Method	Average Time (ms)
Database Query	0.0721
File I/O Query	0.5822
In-Memory (ArrayList)	0.0220
In-Memory (BTree)	0.0026

5.2.4 Thread Count Impact

Table 6 shows the impact of different thread counts (10,000 records).

Table 6: Thread Count Impact on Insert Performance

Thread Count	Time (ms)	Ratio vs Single
1 (Single-threaded)	75	1.00×
2 threads	107	0.70×
4 threads	102	0.74×
8 threads	130	0.58×

5.2.5 Query Type Comparison

Table 7 compares different query types (10,000 records, 1000 queries).

5.3 Performance Analysis and Insights

5.3.1 Major Performance Differences

1. Database vs File I/O Performance:

Database queries consistently outperform File I/O operations, with the performance advantage increasing dramatically as data size grows:

- **5,000 records:** Database is 2.84× faster (0.157 ms vs 0.446 ms)
- **10,000 records:** Database is 8.08× faster (0.072 ms vs 0.582 ms)
- **50,000 records:** Database is 33.00× faster (0.072 ms vs 2.365 ms)

Table 7: Query Type Performance Comparison

Query Type	**Average Time (ms)**
Point Query (Primary Key)	0.0501
Range Query	1.0738
Complex Aggregation Query	1.0248

Key Insight: The performance gap widens significantly with data size, demonstrating the superior scalability of database systems. This is primarily due to:

- **Indexing:** Primary key indexes enable O(log n) lookup time vs O(n) linear search in files
- **Query Optimization:** Database query planner optimizes execution paths
- **Buffering:** Database buffer pool caches frequently accessed data pages
- **File I/O Overhead:** File operations require full file reads and parsing for each query

2. Query Type Performance:
Different query types exhibit significantly different performance characteristics:

- **Point Query (Primary Key):** 0.050 ms - Fastest, uses direct index lookup
- **Range Query:** 1.074 ms - Slower, requires index range scan
- **Complex Aggregation:** 1.025 ms - Moderate, requires grouping and aggregation operations

Key Insight: Point queries are approximately 21× faster than range queries because they use direct index access (O(log n)) rather than range scans. Complex queries with aggregations perform similarly to range queries, both requiring full or partial table scans with additional processing.

3. In-Memory vs Database Performance:
In-memory operations provide the highest performance but require loading all data into memory:

- **Database Query:** 0.072 ms (10,000 records, 1000 queries)
- **File I/O Query:** 0.582 ms
- **In-Memory (ArrayList):** 0.022 ms - 3.3× faster than database
- **In-Memory (BTree):** 0.0026 ms - 27.7× faster than database

Key Insight: In-memory operations eliminate disk I/O overhead, providing dramatic performance improvements. BTree indexing provides 8.5× speedup over linear search (ArrayList), demonstrating the importance of appropriate data structures.

4. Multi-threading Impact:
Multi-threading shows mixed results depending on data size:

- **Small datasets (5k, 10k):** Multi-threading is slower (0.35–0.59×) due to thread overhead and synchronization costs
- **Large datasets (50k):** Multi-threading is faster (1.30×) due to better resource utilization and parallel I/O operations

Key Insight: Thread overhead and synchronization costs dominate for small datasets, making multi-threading counterproductive. However, for larger datasets (50k+ records), parallel processing benefits become significant as I/O and computation can be effectively parallelized, resulting in performance improvements.

5.3.2 Interesting Findings and Insights

1. **Database Performance Stability:** Database query performance remains remarkably stable across different data sizes (0.072–0.157 ms), demonstrating excellent scalability. This is due to efficient indexing and query optimization.
2. **File I/O Performance Degradation:** File I/O performance degrades significantly with data size (0.446 ms to 2.365 ms), as larger files require more time to read and parse. This highlights the scalability limitations of file-based approaches.
3. **Index Effectiveness:** The dramatic performance difference between point queries (0.050 ms) and range queries (1.074 ms) demonstrates the effectiveness of primary key indexes for exact lookups.
4. **Memory vs Disk Trade-off:** In-memory operations (BTree: 0.0026 ms) are 27.7× faster than database queries, but require loading all data into memory. This represents a classic trade-off between speed and memory usage.
5. **Scalability Advantage:** The performance advantage of databases over File I/O increases from 2.84× to 33.00× as data size grows, demonstrating superior scalability for large datasets.
6. **Thread Overhead:** For small datasets (5k, 10k), multi-threading introduces overhead that outweighs benefits, resulting in slower performance (0.35–0.59×). Only for larger datasets (50k+ records) does multi-threading provide performance gains (1.30×).

5.3.3 Practical Implications

Based on our comprehensive performance analysis, we draw the following practical conclusions:

1. **For Production Systems:** Database systems are essential for applications requiring frequent queries, especially as data size grows. The 33.00× performance advantage for 50k records makes databases the clear choice.
2. **For Small-Scale Applications:** File I/O may be acceptable for very small datasets ($<$ 5k records) where simplicity is prioritized over performance, but even then, databases provide 3× better performance.

3. **For High-Performance Requirements:** In-memory data structures (with appropriate indexing) provide the highest performance but require sufficient memory. This is suitable for frequently accessed, relatively static datasets.

4. **For Large-Scale Systems:** Database systems demonstrate superior scalability, maintaining consistent performance as data size increases, while file-based approaches degrade significantly.

5. **Query Type Selection:** Point queries should be preferred when possible, as they provide 21× better performance than range queries. Database schema design should prioritize primary key lookups for frequently accessed data.

5.3.4 Test Data Generation Method Benefits

Our programmatic test data generation approach provides significant advantages for performance testing:

- **Consistency:** Same data generation logic ensures identical test conditions for DBMS and File I/O comparisons, eliminating data-related variables
- **Scalability:** Easy to test different data sizes (5k, 10k, 50k, 100k+) without manual file preparation
- **Efficiency:** No need to store large test files; data is generated on-demand, reducing storage requirements
- **Reproducibility:** Fixed random seed enables consistent results across test runs, facilitating regression testing
- **Flexibility:** Can easily modify data distribution patterns (e.g., skewed distributions, different value ranges) for different test scenarios
- **Automation:** Fully automated generation enables comprehensive performance testing across different scales without manual intervention

This method ensures that both database and file I/O tests operate on identical datasets, providing fair and accurate performance comparisons that isolate the performance characteristics of the storage and query mechanisms themselves.