Gym Management and Equipment Utilization System: Enhancing Efficiency through Database Integration

Chaitanya Sai Chandu Yendru
School of Engineering and Applied Sciences
University at Buffalo
Buffalo, USA
cyendru@buffalo.edu

Poojitha Challa
School of Engineering and Applied Sciences
University at Buffalo
Buffalo, USA
pchalla@buffalo.edu

Surya Pavan Kumar Marturi
School of Engineering and Applied Sciences
University at Buffalo
Buffalo, USA
suryapav@buffalo.edu

***Abstract*—In the rapidly evolving fitness industry, effective management of gym facilities and equipment utilization is paramount for ensuring optimal member satisfaction and operational efficiency. Traditional methods of data management, such as Excel files, often fall short in accommodating the dynamic and interconnected nature of gym operations. To address these challenges, this project proposes the development of a comprehensive Gym Management and Equipment Utilization System (GMEUS) built upon a relational database framework.**

GMEUS aims to centralize gym-related data, including member profiles, workout routines, equipment inventory, and communication logs, within a unified database schema. By leveraging relational database capabilities, the system facilitates seamless data integration, efficient query processing, and real-time analytics, empowering gym administrators to make data-driven decisions and optimize resource allocation.

The project's significance lies in its potential to revolutionize gym management practices, offering administrators intuitive tools for monitoring facility usage, tracking equipment maintenance, and personalizing member experiences. Through the implementation of GMEUS, fitness facilities can enhance operational efficiency, improve equipment utilization, and ultimately elevate the overall quality of service provided to members.

***Index Terms*—Fitness industry; Gym management; Equipment utilization; Relational database; Gym operations; Data management; Excel files; Centralized data; Member profiles; Workout routines; Equipment inventory; Communication logs; Data integration; Query processing; Real-time analytics; Data-driven decisions; Resource allocation; Gym administrators; Revolutionize practices; Operational efficiency**

I. Introduction

In the realm of gym management, traditional manual methods often struggle to meet the demands of efficiently handling confidential data, such as membership details and workout schedules. These antiquated approaches frequently result in errors and inefficiencies, hindering the effective support of gym users' fitness goals. Critical functions like member registration and equipment tracking become laborious tasks due to the lack of an integrated system.

Traditional tools like Excel, while useful for basic data management, are ill-equipped to handle the complexities inherent in managing modern fitness facilities and their online platforms. Database systems offer a solution by streamlining operations, improving user engagement, and providing personalized fitness experiences. With structured environments that enforce restrictions and relationships, databases ensure the precision and reliability of information. They excel in creating relationships between data entities, simplifying data retrieval, and offering insights into user progress and equipment usage patterns. Moreover, through normalization, databases reduce redundancy and minimize storage needs, ensuring that updates are uniformly propagated across the system.

The planned database system is poised to revolutionize gym management by simplifying operations, enhancing user interaction, and facilitating informed decision-making. With features designed to automate tasks, offer personalized experiences, and handle growing volumes of data, the system promises to be an organized, expandable, and effective tool for fitness centers. By incorporating inherent backup and recovery systems, applying data validation, and fostering data consistency, the database system aims to elevate the dependability and efficiency of gym management to new heights.

II. Database Users and Administration

A. Users of the Database

Gym administrators and managers serve as the primary users of the database, relying on its functionalities to oversee various facets of gym operations. From managing memberships to scheduling workouts and classes, administrators leverage the database to ensure seamless gym management. They access detailed member profiles to make informed decisions regarding operations and member engagement strategies.

IT staff and developers play a pivotal role in database usage, ensuring its seamless integration with other gym systems like the website and mobile application. They maintain the infrastructure, ensuring real-time data access and enhancing user experience through system optimizations. Developers implement new features such as online registration and personalized workout recommendations, ensuring the database remains robust and adaptable.

Personal trainers utilize the database to deliver tailored fitness experiences, accessing member data to customize workout plans and track progress over time. By leveraging database insights, trainers can provide effective guidance and support to meet individual member goals.

Gym members benefit from the database through personalized workout plans and progress tracking tools provided via gym applications and platforms. By empowering members with access to their data, the database enhances engagement and fosters a rewarding gym experience for all users.

B. Administration of the Database

Database administration falls under the purview of designated administrators or IT professionals responsible for ensuring its integrity, security, and performance. Key responsibilities include implementing security measures, conducting backups, and optimizing performance to meet the needs of gym administrators, staff, and members.

Administrators work closely with developers to manage schema modifications, updates, and integrations to accommodate evolving requirements. They coordinate with IT staff to address technical issues promptly, ensuring uninterrupted access to gym services.

Developers contribute to administration by implementing new features and optimizations to meet evolving needs. They collaborate with administrators to ensure seamless integration between the database and other systems, providing a cohesive user experience.

In summary, effective database administration requires collaboration between administrators, developers, and IT staff to maintain a reliable, secure, and user-friendly system that supports gym objectives.

III. Real-Life Scenario

Managing member information at a university gym is challenging due to outdated manual processes and disconnected systems. To address this, the university implements a Gym Management System powered by a centralized database. The system streamlines gym operations, improves member experiences, and enhances staff productivity through automation, integration, and data-driven insights.

Administrators and managers gain access to a user-friendly interface for efficient management of gym operations, including member registration, membership tracking, class scheduling, trainer assignment, and equipment monitoring. The database provides valuable insights into member demographics, preferences, and engagement metrics.

IT staff and developers ensure seamless integration with existing systems and optimize system performance. Personal trainers benefit from accessing member profiles, tracking progress, and providing personalized workout plans and feedback. Members experience improved gym experiences with online registration, personalized workout plans, progress tracking tools, and timely notifications.

Overall, the Gym Management System transforms the university gym into a modern, efficient, and member-centric facility, delivering personalized services, streamlining operations, and enhancing member experiences.

IV. Database Schema Description

A. Entity Tables

1) Organizations, Roles, TimeZones, Gyms, and Users: These tables form the backbone of the gym management system.

- **Organizations:** Stores information about different gym organizations, including their names, logos, and taglines.
- **Roles:** Defines the roles within the system, such as admin, trainer, or member.
- **TimeZones:** Contains a list of time zones for scheduling purposes.
- **Gyms:** Represents physical gym locations, with details like name, type, and status.
- **Users:** Stores information about registered users, including their names, contact details, and registration dates.

2) Gym-Related Entities: This subsection encompasses tables directly involved in gym operations and management.

- **Gym_Users:** Defines the relationship between users and gyms, including their roles and permissions.
- **Exercises:** Contains details about various exercises available at the gym.
- **EquipmentTags:** Manages tags associated with gym equipment for categorization and organization.
- **EquipmentTag_Exercise:** Establishes relationships between exercises and equipment tags.
- **Workout:** Stores data related to gym workouts, including type, date, time, and calories burned.
- **Workout_Exercise:** Represents the exercises performed during each workout session.

3) Exercise and Multimedia Data: This section encompasses tables relevant to exercise management and multimedia content.

- **Muscle_Targets:** Defines various muscle targets for exercise routines.
- **Exercise_MuscleTargets:** Establishes relationships between exercises and targeted muscle groups.
- **Equipment_Usage:** Tracks the usage of gym equipment during exercises.
- **Videos:** Stores instructional videos for exercise guidance and fitness tips.

- **Notifications:** Manages notifications and messages sent to gym users and administrators.
- **Templates:** Contains predefined message templates for automated notifications.

B. Entity-Relationship Diagram (ERD)

This section provides a summary of the relationships between various entities in the database schema, offering insights into the data model's structure and interconnections.The visual representation of the same is available in Fig1. The original database relation DML is illustrated at: https://dbdiagram.io/d/DataModel-65c6406bac844320aed1c5eb. With the changes made in view of the problems encountered the diagram has been updated and the illustration can be seen below(Fig1).

C. Database Schema Relations

1) Organizations:

- **Primary Key:** org_id
- **Attributes:**
  - org_id: Unique identifier for the organization.
  - engine_org_id: Engine-specific identifier for the organization.
  - name: Name of the organization.
  - color1: Primary color associated with the organization.
  - color2: Secondary color associated with the organization.
  - logo: URL or path to the organization's logo.
  - tagline: Tagline or slogan of the organization.
- **Default Values/Nullability:** No default values. All attributes can be null except org_id.
- **Foreign Key Actions:** No action on foreign keys when the primary key (org_id) is deleted.

2) Roles:

- **Primary Key:** role_id
- **Attributes:**
  - role_id: Unique identifier for the role.
  - role: Name or description of the role.
- **Default Values/Nullability:** No default values. All attributes can be null except role_id.
- **Foreign Key Actions:** No action on foreign keys when the primary key (role_id) is deleted.

3) TimeZones:

- **Primary Key:** zone_id
- **Attributes:**
  - zone_id: Unique identifier for the timezone.
  - zone: Name or description of the timezone.
- **Default Values/Nullability:** No default values. All attributes can be null except zone_id.
- **Foreign Key Actions:** No action on foreign keys when the primary key (zone_id) is deleted.

4) Gyms:

- **Primary Key:** gym_id
- **Attributes:**
  - gym_id: Unique identifier for the gym.
  - engine_gym_id: Engine-specific gym identifier.
  - type: Type of gym (e.g., Fitness Center, Yoga Studio).
  - name: Name of the gym.
  - timezoneId: Foreign key referencing the TimeZones table, indicating the timezone of the gym.
  - status: Status of the gym (e.g., active, inactive).
  - org_id: Foreign key referencing the Organizations table, indicating the organization to which the gym belongs.
- **Default Values/Nullability:** No default values. All attributes can be null except gym_id.
- **Foreign Key Actions:** If the primary key (org_id) in the Organizations table is removed, it could lead to several consequences:
  - If the deletion action is set to CASCADE, all associated gyms belonging to the deleted organization will also be removed from the Gyms table.
  - If the deletion action is set to SET NULL, the org_id column in the affected gyms' rows will be set to NULL, indicating that they no longer belong to any organization.
  - If the deletion action is set to NO ACTION or RESTRICT, it would prevent the deletion of an organization if it has associated gyms. An error will occur, and the deletion will be aborted.
  - If the deletion action is set to SET DEFAULT, the org_id column in the affected gyms' rows will be set to a default value specified for such cases.

5) Users:

- **Primary Key:** u_id
- **Attributes:**
  - u_id: Unique identifier for the user.
  - engine_user_id: Engine-specific user identifier.
  - name: Name of the user.
  - class_level: Class level of the user (e.g., Beginner, Intermediate, Advanced).
  - birthday: Date of birth of the user.
  - registration_date: Date and time when the user registered.
  - email: Email address of the user.
  - last_visited_at: Date and time when the user last visited.
  - status: Status of the user (e.g., active, inactive).
- **Default Values/Nullability:** No default values. All attributes can be null except u_id.
- **Foreign Key Actions:** The impact of removing the primary key (u_id) in the Users table depends on the actions specified for the foreign keys referencing it:
  - If the deletion action for the foreign keys (e.g., Gym_Users, Videos) is set to CASCADE, deleting a

user will also delete all associated records in those tables.
  - If the action is set to SET NULL, the corresponding user_id in the referencing tables will be set to NULL.
  - If the action is set to NO ACTION or RESTRICT, deleting a user will fail if there are related records in other tables, preventing the operation.
  - If the action is set to SET DEFAULT, the user_id in the referencing tables will be set to a default value specified for such cases.

6) Gym_Users:

- **Primary Key:** (gym_id, u_id, role_id)
- **Attributes:**
  - gym_id: Foreign key referencing the Gyms table, indicating the gym to which the user belongs.
  - u_id: Foreign key referencing the Users table, indicating the user.
  - role_id: Foreign key referencing the Roles table, indicating the role of the user in the gym.
  - role_tag: Additional tag or description for the user's role.
- **Default Values/Nullability:** No default values. All attributes can be null except gym_id, u_id, and role_id.
- **Foreign Key Actions:** The removal of primary keys from referenced tables (Gyms, Users, Roles) would impact the Gym_Users table as follows:
  - CASCADE will remove corresponding records in Gym_Users.
  - SET NULL will set the foreign key values to NULL.
  - NO ACTION or RESTRICT will prevent the deletion of referenced records if related records exist in Gym_Users.
  - SET DEFAULT will set the foreign key values to a default specified value.

7) Exercises:

- **Primary Key:** exercise_id
- **Attributes:**
  - exercise_id: Unique identifier for the exercise.
  - engine_exercise_name: Engine-specific exercise name.
  - name: Name of the exercise.
- **Default Values/Nullability:** No default values. All attributes can be null except exercise_id.
- **Foreign Key Actions:** If the primary key (exercise_id) in the Exercises table is removed:
  - CASCADE: Associated records in related tables such as EquipmentTag_Exercise, Workout_Exercise, and Videos will also be deleted.
  - SET NULL: Foreign key values referencing the deleted exercise_id will be set to NULL.
  - NO ACTION or RESTRICT: Deletion of an exercise will be prevented if related records exist in other tables.
  - SET DEFAULT: Foreign key values will be set to a default specified value.

8) EquipmentTags:

- **Primary Key:** tag_id
- **Attributes:**
  - tag_id: Unique identifier for the equipment tag.
  - engine_tag_name: Engine-specific tag name.
  - tag_name: Name of the equipment tag.
  - gym_id: Foreign key referencing the Gyms table, indicating the gym to which the equipment tag belongs.
- **Default Values/Nullability:** No default values. All attributes can be null except tag_id and gym_id.
- **Foreign Key Actions:** If the primary key (tag_id) in the EquipmentTags table is removed:
  - CASCADE: Associated records in related tables such as EquipmentTag_Exercise will also be deleted.
  - SET NULL: Foreign key values referencing the deleted tag_id will be set to NULL.
  - NO ACTION or RESTRICT: Deletion of an equipment tag will be prevented if related records exist in other tables.
  - SET DEFAULT: Foreign key values will be set to a default specified value.

9) EquipmentTag_Exercise:

- **Primary Key:** (exercise_id, tag_id)
- **Attributes:**
  - exercise_id: Foreign key referencing the Exercises table, indicating the exercise associated with the equipment tag.
  - tag_id: Foreign key referencing the EquipmentTags table, indicating the equipment tag associated with the exercise.
- **Default Values/Nullability:** No default values. All attributes can be null except exercise_id and tag_id.
- **Foreign Key Actions:** If either the exercise_id or tag_id primary key in the EquipmentTag_Exercise table is removed:
  - CASCADE: Associated records will also be deleted.
  - SET NULL: Foreign key values referencing the deleted primary key will be set to NULL.
  - NO ACTION or RESTRICT: Deletion will be prevented if related records exist in other tables.
  - SET DEFAULT: Foreign key values will be set to a default specified value.

10) Workout:

- **Primary Key:** workout_id
- **Attributes:**
  - workout_id: Unique identifier for the workout.
  - engine_workout_id: Engine-specific workout identifier.
  - gym_id: Foreign key referencing the Gyms table, indicating the gym where the workout took place.
  - user_id: Foreign key referencing the Users table, indicating the user who performed the workout.

  - workout_type: Type of workout (e.g., Cardio, Strength Training).
  - workout_date: Date of the workout.
  - time: Time of the workout.
  - calories: Calories burned during the workout.
- **Default Values/Nullability:** No default values. All attributes can be null except workout_id, gym_id, user_id, workout_date, and time.
- **Foreign Key Actions:** If the primary key (workout_id) in the Workout table is removed:
  - CASCADE: Associated records in related tables such as Workout_Exercise will also be deleted.
  - SET NULL: Foreign key values referencing the deleted workout_id will be set to NULL.
  - NO ACTION or RESTRICT: Deletion of a workout will be prevented if related records exist in other tables.
  - SET DEFAULT: Foreign key values will be set to a default specified value.

11) Workout_Exercise:

- **Primary Key:** (exercise_id, workout_id)
- **Attributes:**
  - exercise_id: Foreign key referencing the Exercises table, indicating the exercise performed during the workout.
  - workout_id: Foreign key referencing the Workout table, indicating the workout associated with the exercise.
- **Default Values/Nullability:** No default values. All attributes can be null except exercise_id and workout_id.
- **Foreign Key Actions:** If either the exercise_id or workout_id primary key in the Workout_Exercise table is removed:
  - CASCADE: Associated records will also be deleted.
  - SET NULL: Foreign key values referencing the deleted primary key will be set to NULL.
  - NO ACTION or RESTRICT: Deletion will be prevented if related records exist in other tables.
  - SET DEFAULT: Foreign key values will be set to a default specified value.

12) Muscle_Targets:

- **Primary Key:** muscle_target_id
- **Attributes:**
  - muscle_target_id: Unique identifier for the muscle target.
  - name: Name of the muscle target.
- **Default Values/Nullability:** No default values. All attributes can be null except muscle_target_id.
- **Foreign Key Actions:** If the primary key (muscle_target_id) in the Muscle_Targets table is removed:
  - CASCADE: Associated records in related tables such as Exercise_MuscleTargets will also be deleted.
  - SET NULL: Foreign key values referencing the deleted muscle_target_id will be set to NULL.
  - NO ACTION or RESTRICT: Deletion of a muscle target will be prevented if related records exist in other tables.
  - SET DEFAULT: Foreign key values will be set to a default specified value.

13) Exercise_MuscleTargets:

- **Primary Key:** (exercise_id, muscle_target_id)
- **Attributes:**
  - exercise_id: Foreign key referencing the Exercises table, indicating the exercise associated with the muscle target.
  - muscle_target_id: Foreign key referencing the Muscle_Targets table, indicating the muscle target associated with the exercise.
- **Default Values/Nullability:** No default values. All attributes can be null except exercise_id and muscle_target_id.
- **Foreign Key Actions:** If either the exercise_id or muscle_target_id primary key in the Exercise_MuscleTargets table is removed:
  - CASCADE: Associated records will also be deleted.
  - SET NULL: Foreign key values referencing the deleted primary key will be set to NULL.
  - NO ACTION or RESTRICT: Deletion will be prevented if related records exist in other tables.
  - SET DEFAULT: Foreign key values will be set to a default specified value.

14) Equipment_Usage:

- **Primary Key:** (exercise_id, equipment_id)
- **Attributes:**
  - exercise_id: Foreign key referencing the Exercises table, indicating the exercise where the equipment was used.
  - equipment_id: Foreign key referencing the EquipmentTags table, indicating the equipment used during the exercise.
  - engine_usage_id: Engine-specific usage identifier.
  - metric1, metric2, metric3: Metrics related to equipment usage.
  - workout_date: Date of the workout when the equipment was used.
- **Default Values/Nullability:** No default values. All attributes can be null except exercise_id, equipment_id, and workout_date.
- **Foreign Key Actions:** If either the exercise_id or equipment_id primary key in the Equipment_Usage table is removed:
  - CASCADE: Associated records will also be deleted.
  - SET NULL: Foreign key values referencing the deleted primary key will be set to NULL.
  - NO ACTION or RESTRICT: Deletion will be prevented if related records exist in other tables.
  - SET DEFAULT: Foreign key values will be set to a default specified value.

15) Notifications: The Notifications table stores information about notifications sent to users within specific gyms.

- **id:** An integer serving as the primary key for the table. It is automatically incremented for each new notification.
- **gym_id:** An integer representing the foreign key referencing the gym to which the notification is associated. It establishes a relationship with the Gyms table.
- **u_id:** An integer representing the foreign key referencing the user to whom the notification is directed. It establishes a relationship with the Users table.
- **subject:** A varchar field storing the subject or title of the notification.
- **message:** A varchar field containing the content or body of the notification.
- **scheduled_at:** A datetime field indicating the scheduled time for sending the notification.
- **Foreign Key Actions:** Foreign keys removal issue.
  - CASCADE: Associated records will also be deleted.
  - SET NULL: Foreign key values referencing the deleted primary key will be set to NULL.
  - NO ACTION or RESTRICT: Deletion will be prevented if related records exist in other tables.
  - SET DEFAULT: Foreign key values will be set to a default specified value.

16) Templates: The Templates table holds predefined message templates that can be used for notifications or other communication purposes.

- **id:** An integer serving as the primary key for the table. It increments automatically for each new template.
- **subject:** A varchar field storing the subject or title of the template.
- **message:** A varchar field containing the content or body of the template, providing a predefined message structure.

V. BCNF and Functional Dependencies

The relations in the database schema are already in Boyce-Codd Normal Form (BCNF). This normalization level ensures that there are no non-trivial functional dependencies where a determinant is not a superkey. Here's a summary of the functional dependencies for each table:

A. Organizations

Functional Dependencies:

- org_id → engine_org_id, name, color1, color2, logo, tagline

B. Roles

Functional Dependencies:

- role_id → role

C. TimeZones

Functional Dependencies:

- zone_id → zone

D. Gyms

Functional Dependencies:

- gym_id → engine_gym_id, type, name, timezoneId, status, org_id

E. Users

Functional Dependencies:

- u_id → engine_user_id, name, class_level, birthday, registration_date, email, last_visited_at, status

F. Gym_Users

Functional Dependencies:

- (gym_id, u_id, role_id) → role_tag

G. Exercises

Functional Dependencies:

- exercise_id → engine_exercise_name, name

H. EquipmentTags

Functional Dependencies:

- tag_id → engine_tag_name, tag_name, gym_id

I. EquipmentTag_Exercise

Functional Dependencies:

- (exercise_id, tag_id) → None

J. Workout

Functional Dependencies:

- workout_id → engine_workout_id, gym_id, user_id, workout_type, workout_date, time, calories

K. Workout_Exercise

Functional Dependencies:

- (exercise_id, workout_id) → None

L. Muscle_Targets

Functional Dependencies:

- muscle_target_id → name

M. Exercise_MuscleTargets

Functional Dependencies:

- (muscle_target_id, exercise_id) → None

N. Equipment_Usage

Functional Dependencies:

- (equipment_id, exercise_id) → engine_usage_id, metric1, metric2, metric3, workout_date

O. Videos

Functional Dependencies:

- id → engine_video_id, thumbnail_source, u_id, exercise_id, source, uploaded_at, status, downloads

P. Notifications

Functional Dependencies:

- id → gym_id, u_id, subject, message, scheduled_at

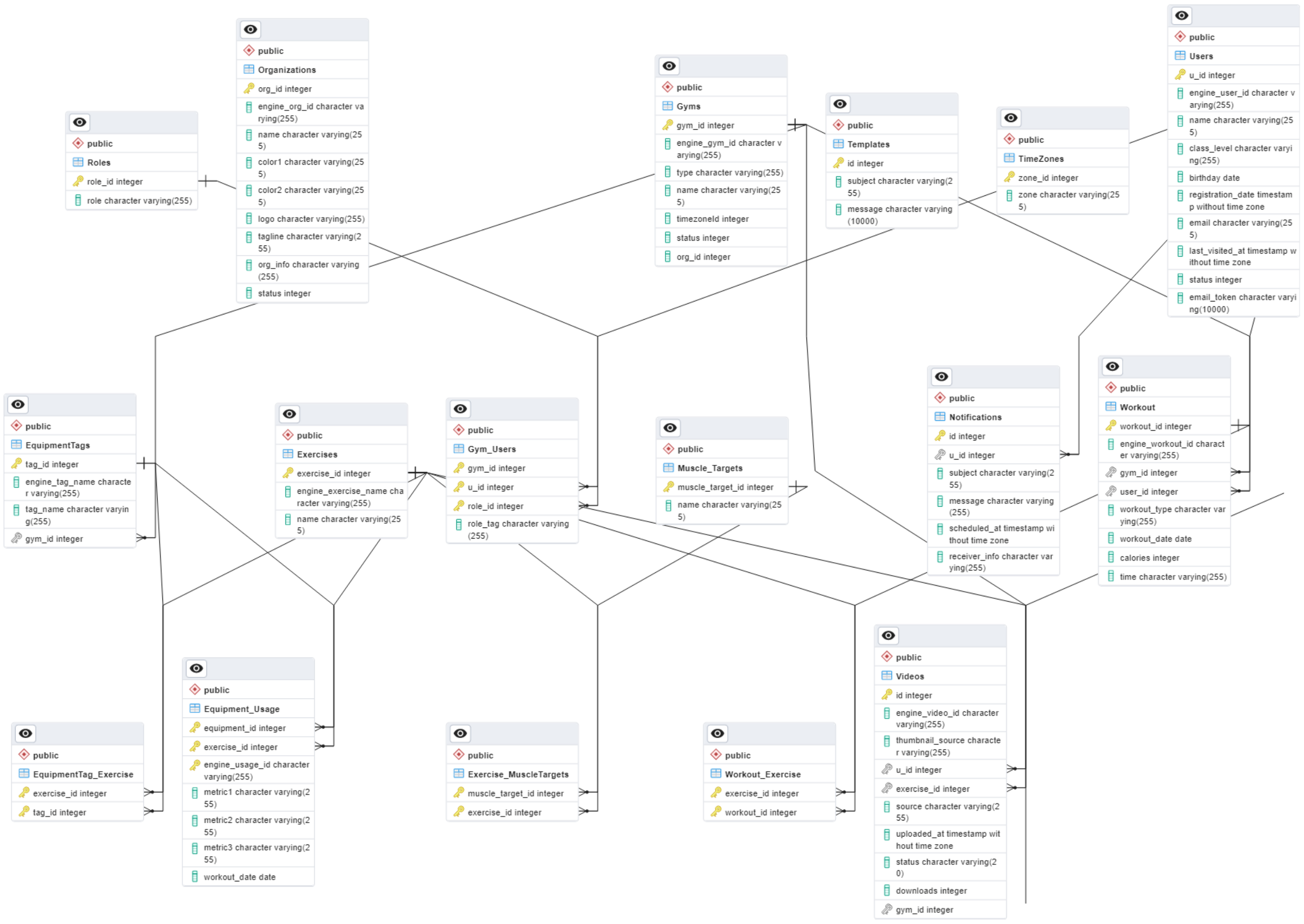

public
Organizations
org_id integer
engine_org_id character varying(255)
name character varying(255)
color1 character varying(255)
color2 character varying(255)
logo character varying(255)
tagline character varying(255)
org_info character varying(255)
status integer
public
Roles
role_id integer
role character varying(255)
public
Gyms
gym_id integer
engine_gym_id character varying(255)
type character varying(255)
name character varying(255)
timezoneId integer
status integer
org_id integer
public
Templates
id integer
subject character varying(255)
message character varying(10000)
public
TimeZones
zone_id integer
zone character varying(255)
public
Users
u_id integer
engine_user_id character varying(255)
name character varying(255)
class_level character varying(255)
birthday date
registration_date timestamp without time zone
email character varying(255)
last_visited_at timestamp without time zone
status integer
email_token character varying(10000)
public
EquipmentTags
tag_id integer
engine_tag_name character varying(255)
tag_name character varying(255)
gym_id integer
public
Exercises
exercise_id integer
engine_exercise_name character varying(255)
name character varying(255)
public
Gym_Users
gym_id integer
u_id integer
role_id integer
role_tag character varying(255)
public
Muscle_Targets
muscle_target_id integer
name character varying(255)
public
Notifications
id integer
u_id integer
subject character varying(255)
message character varying(255)
scheduled_at timestamp without time zone
receiver_info character varying(255)
public
Workout
workout_id integer
engine_workout_id character varying(255)
gym_id integer
user_id integer
workout_type character varying(255)
workout_date date
calories integer
time character varying(255)
public
Equipment_Usage
equipment_id integer
exercise_id integer
engine_usage_id character varying(255)
metric1 character varying(255)
metric2 character varying(255)
metric3 character varying(255)
workout_date date
public
EquipmentTag_Exercise
exercise_id integer
tag_id integer
public
Exercise_MuscleTargets
muscle_target_id integer
exercise_id integer
public
Workout_Exercise
exercise_id integer
workout_id integer
public
Videos
id integer
engine_video_id character varying(255)
thumbnail_source character varying(255)
u_id integer
exercise_id integer
source character varying(255)
uploaded_at timestamp without time zone
status character varying(20)
downloads integer
gym_id integer

Fig. 1. The entity relation model: classes and relationships.

Q. *Templates*

Functional Dependencies:

- id → subject, message

In each case, the functional dependencies show that the primary keys uniquely determine all other attributes within each relation, satisfying the requirements for BCNF.

VI. Summary of Data Augmentation

We have constructed a Python script to automate the generation of synthetic data for a gym management system. Here's a summary of how the data generation process works:

1) **Setup:** The script imports necessary libraries, including `csv`, `random`, and `Faker`, to facilitate data generation.
2) **Fake Data Generation:** Using the `Faker` library, realistic fake data is generated for each table in the database schema.
3) **CSV File Creation:** Data for each table is written to separate CSV files. Each CSV file corresponds to a table in the database schema.
4) **Data Augmentation Functions:** Specific functions are defined to generate data for each table. For example:
   - `generate_organizations_data`: Generates data for the Organizations table.
   - `generate_roles_data`: Generates data for the Roles table.
   - `generate_timezones_data`: Generates data for the TimeZones table.
   - And so on, for each table in the schema.
5) **Random Associations:** The script creates random associations between entities to simulate relationships. For example, gym users are associated with gyms and roles.
6) **Data Writing:** Data generated for each table is written row by row into the respective CSV files.
7) **Completion:** Once the script execution completes, CSV files containing the synthetic data are ready for import into a database management system.

We then used the import option available to convert the csv data files to tables.

The script offers a convenient way to quickly generate a large dataset that conforms to the specified schema, facilitating testing and development of the gym management system.

The complete script is available at: https://codeshare.io/Ekoexp.

VII. Handling Larger Datasets

A. Challenges Encountered

1) **Slow Query Execution:** As the dataset size increased, complex JOIN operations and aggregations took longer to execute, impacting user experience.
2) **Optimizing Joins:** Complex joins across multiple tables posed performance challenges, requiring optimization to minimize computational complexity.
3) **Aggregation Performance:** Aggregating data over large datasets led to performance issues due to scanning numerous rows.
4) **Index Selection and Maintenance:** Choosing and maintaining effective indexes was critical for query performance.

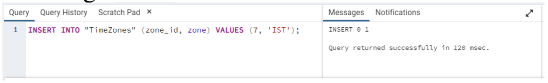
B. Solutions Implemented

1) **Indexing:** Created indexes on frequently queried columns to expedite data retrieval.
2) **Optimizing Joins:** Reviewed and optimized join conditions to minimize unnecessary joins and computational overhead.
3) **Aggregation Performance:** Utilized indexing on columns involved in aggregation operations and optimized queries to reduce data processing.
4) **Index Selection and Maintenance:** Regularly reviewed query performance using tools like EXPLAIN in PostgreSQL and scheduled index maintenance tasks for optimal performance.

VIII. Testing Database with SQL Queries

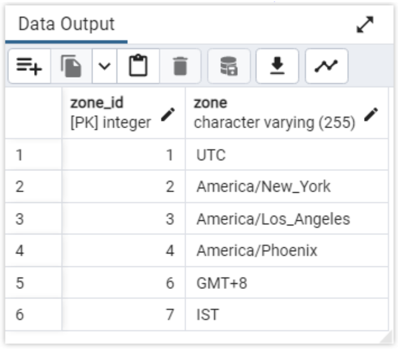
A. Insert Statements

1) Inserting a new timezone.



```sql
INSERT INTO "TimeZones" (zone_id, zone) VALUES (7, 'IST');
```

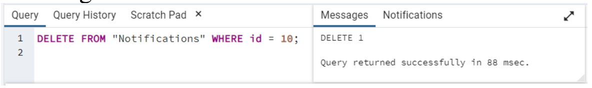
```
INSERT 0 1

Query returned successfully in 120 msec.
```



	zone_id [PK] integer	zone character varying (255)
1	1	UTC
2	2	America/New_York
3	3	America/Los_Angeles
4	4	America/Phoenix
5	6	GMT+8
6	7	IST

B. Delete Statements

1) Deleting a Notification.

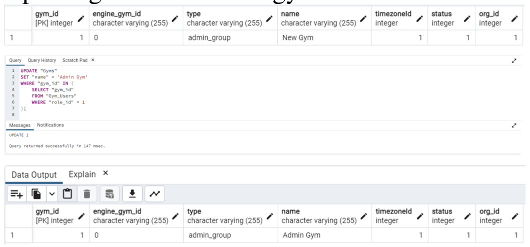


```sql
DELETE FROM "Notifications" WHERE id = 10;
```

```
DELETE 1

Query returned successfully in 88 msec.
```

C. Update Statements

1) Updating the name of a gym.



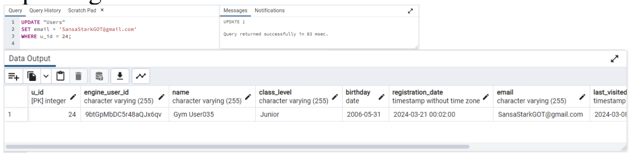
	gym_id [PK] integer	engine_gym_id character varying (255)	type character varying (255)	name character varying (255)	timezoneId integer	status integer	org_id integer
1	1	0	admin_group	New Gym	1	1	1

```sql
UPDATE "Gyms"
SET "name" = 'Admin Gym'
WHERE "gym_id" IN (
    SELECT "gym_id"
    FROM "Gym_Users"
    WHERE "role_id" = 1
);
```

```
UPDATE 1

Query returned successfully in 147 msec.
```

	gym_id [PK] integer	engine_gym_id character varying (255)	type character varying (255)	name character varying (255)	timezoneId integer	status integer	org_id integer
1	1	0	admin_group	Admin Gym	1	1	1

2) Updating the email address of a user.



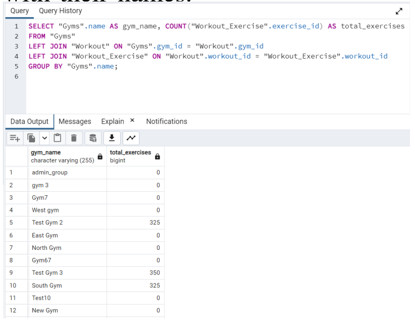
```sql
UPDATE "Users"
SET email = 'SansaStarkGOT@gmail.com'
WHERE u_id = 24;
```

```
UPDATE 1

Query returned successfully in 83 msec.
```

	u_id [PK] integer	engine_user_id character varying (255)	name character varying (255)	class_level character varying (255)	birthday date	registration_date timestamp without time zone	email character varying (255)	last_visited timestamp
1	24	9btGpMbDC5r48aQJx6qv	Gym User035	Junior	2006-05-31	2024-03-21 00:02:00	SansaStarkGOT@gmail.com	2024-03-0

D. Select Statements

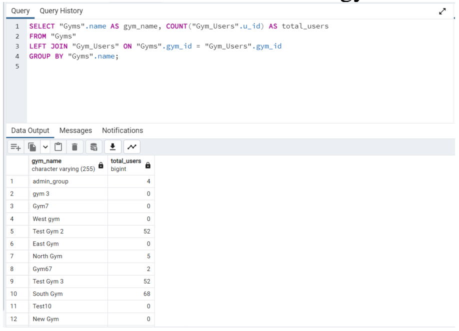
1) Total number of exercises performed at each gym along with their names.



```sql
SELECT "Gyms".name AS gym_name, COUNT("Workout_Exercise".exercise_id) AS total_exercises
FROM "Gyms"
LEFT JOIN "Workout" ON "Gyms".gym_id = "Workout".gym_id
LEFT JOIN "Workout_Exercise" ON "Workout".workout_id = "Workout_Exercise".workout_id
GROUP BY "Gyms".name;
```

	gym_name character varying (255)	total_exercises bigint
1	admin_group	0
2	gym 3	0
3	Gym7	0
4	West gym	0
5	Test Gym 2	325
6	East Gym	0
7	North Gym	0
8	Gym67	0
9	Test Gym 3	350
10	South Gym	325
11	Test10	0
12	New Gym	0

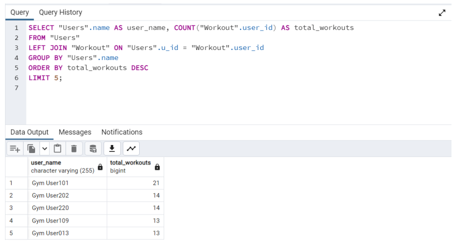
2) Total number of users in each gym.



```sql
SELECT "Gyms".name AS gym_name, COUNT("Gym_Users".u_id) AS total_users
FROM "Gyms"
LEFT JOIN "Gym_Users" ON "Gyms".gym_id = "Gym_Users".gym_id
GROUP BY "Gyms".name;
```

	gym_name character varying (255)	total_users bigint
1	admin_group	4
2	gym 3	0
3	Gym7	0
4	West gym	0
5	Test Gym 2	52
6	East Gym	0
7	North Gym	5
8	Gym67	2
9	Test Gym 3	52
10	South Gym	68
11	Test10	0
12	New Gym	0

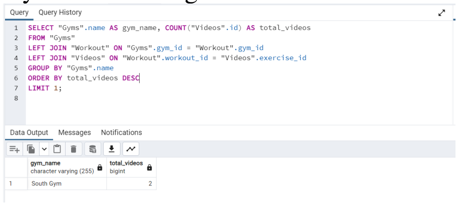
3) Top 5 users based on the number of workouts they've completed.



```
SELECT "Users".name AS user_name, COUNT("Workout".user_id) AS total_workouts
FROM "Users"
LEFT JOIN "Workout" ON "Users".u_id = "Workout".user_id
GROUP BY "Users".name
ORDER BY total_workouts DESC
LIMIT 5;
```

	user_name character varying (255)	total_workouts bigint
1	Gym User101	21
2	Gym User202	14
3	Gym User220	14
4	Gym User109	13
5	Gym User013	13

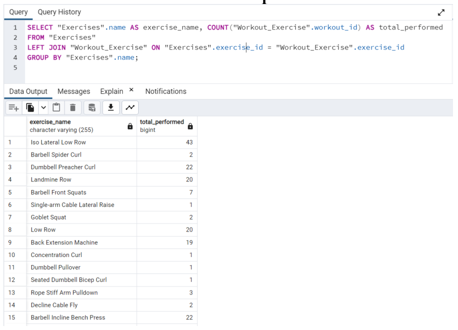
4) Gym with the highest number of workout videos.



```
SELECT "Gyms".name AS gym_name, COUNT("Videos".id) AS total_videos
FROM "Gyms"
LEFT JOIN "Workout" ON "Gyms".gym_id = "Workout".gym_id
LEFT JOIN "Videos" ON "Workout".workout_id = "Videos".exercise_id
GROUP BY "Gyms".name
ORDER BY total_videos DESC
LIMIT 1;
```

	gym_name character varying (255)	total_videos bigint
1	South Gym	2

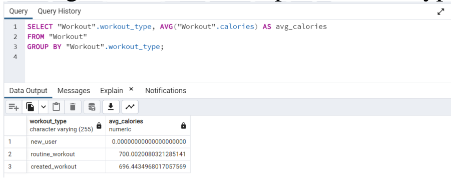
5) Exercise names along with the total number of times each exercise has been performed.



```
SELECT "Exercises".name AS exercise_name, COUNT("Workout_Exercise".workout_id) AS total_performed
FROM "Exercises"
LEFT JOIN "Workout_Exercise" ON "Exercises".exercise_id = "Workout_Exercise".exercise_id
GROUP BY "Exercises".name;
```

	exercise_name character varying (255)	total_performed bigint
1	Iso Lateral Low Row	43
2	Barbell Spider Curl	2
3	Dumbbell Preacher Curl	22
4	Landmine Row	20
5	Barbell Front Squats	7
6	Single-arm Cable Lateral Raise	1
7	Goblet Squat	2
8	Low Row	20
9	Back Extension Machine	19
10	Concentration Curl	1
11	Dumbbell Pullover	1
12	Seated Dumbbell Bicep Curl	1
13	Rope Stiff Arm Pulldown	3
14	Decline Cable Fly	2
15	Barbell Incline Bench Press	22

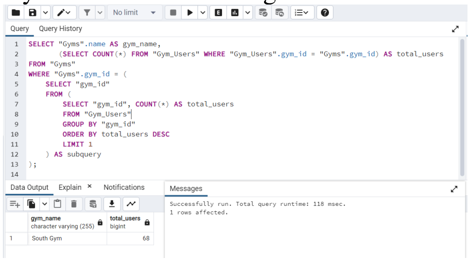
6) Average calories burned per workout type.



```
SELECT "Workout".workout_type, AVG("Workout".calories) AS avg_calories
FROM "Workout"
GROUP BY "Workout".workout_type;
```

	workout_type character varying (255)	avg_calories numeric
1	new_user	0.00000000000000000000
2	routine_workout	700.0020080321285141
3	created_workout	696.4434968017057569

7) Gym Name with the Highest Number of Users.



```
SELECT "Gyms".name AS gym_name,
       (SELECT COUNT(*) FROM "Gym_Users" WHERE "Gym_Users".gym_id = "Gyms".gym_id) AS total_users
FROM "Gyms"
WHERE "Gyms".gym_id = (
    SELECT "gym_id"
    FROM (
        SELECT "gym_id", COUNT(*) AS total_users
        FROM "Gym_Users"
        GROUP BY "gym_id"
        ORDER BY total_users DESC
        LIMIT 1
    ) AS subquery
);
```

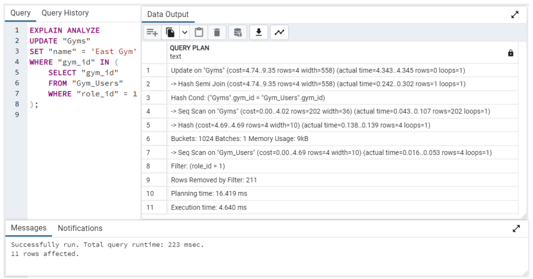
	gym_name character varying (255)	total_users bigint
1	South Gym	68

```
Successfully run. Total query runtime: 118 msec.
1 rows affected.
```

IX. Query Execution Analysis

A. Problematic Queries and Execution Cost

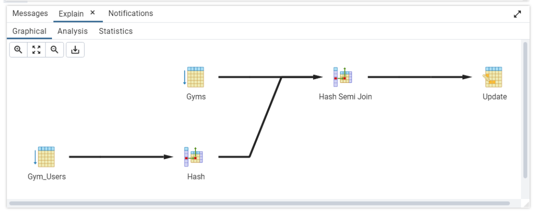
1) **Updating Gym Name Based on User Roles: Original Query:**



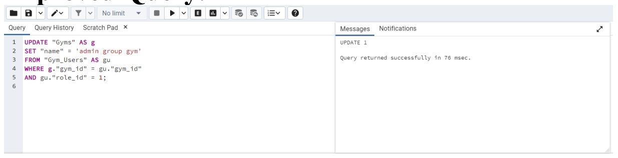
```
EXPLAIN ANALYZE
UPDATE "Gyms"
SET "name" = 'East Gym'
WHERE "gym_id" IN (
    SELECT "gym_id"
    FROM "Gym_Users"
    WHERE "role_id" = 1
);
```

	QUERY PLAN text
1	Update on "Gyms" (cost=4.74..9.35 rows=4 width=558) (actual time=4.343..4.345 rows=0 loops=1)
2	-> Hash Semi Join (cost=4.74..9.35 rows=4 width=558) (actual time=0.242..0.302 rows=1 loops=1)
3	Hash Cond: ("Gyms".gym_id = "Gym_Users".gym_id)
4	-> Seq Scan on "Gyms" (cost=0.00..4.02 rows=202 width=36) (actual time=0.043..0.107 rows=202 loops=1)
5	-> Hash (cost=4.69..4.69 rows=4 width=10) (actual time=0.138..0.139 rows=4 loops=1)
6	Buckets: 1024 Batches: 1 Memory Usage: 9kB
7	-> Seq Scan on "Gym_Users" (cost=0.00..4.69 rows=4 width=10) (actual time=0.016..0.053 rows=4 loops=1)
8	Filter: (role_id = 1)
9	Rows Removed by Filter: 211
10	Planning time: 16.419 ms
11	Execution time: 4.640 ms

```
Successfully run. Total query runtime: 223 msec.
11 rows affected.
```



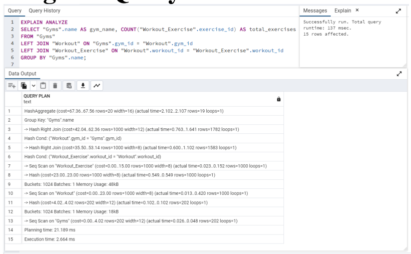
Analysis from EXPLAIN Tool:
The original query utilizes a subquery to identify `gym_id` values where the `role_id` is 1 in the `Gym_Users` table. This subquery is executed independently for each row in the `Gyms` table, resulting in poor performance due to repeated subquery execution. The execution plan involves a Seq Scan on the `Gyms` table and a Seq Scan on the `Gym_Users` table, leading to high costs and slower execution.
Improvement Plan:
Instead of using a subquery, we can perform a JOIN operation between the `Gyms` and `Gym_Users` tables based on `gym_id`. Utilizing a single UPDATE statement with JOIN to efficiently update the desired rows in the `Gyms` table.
Improved Query:



```
UPDATE "Gyms" AS g
SET "name" = 'admin group gym'
FROM "Gym_Users" AS gu
WHERE g."gym_id" = gu."gym_id"
AND gu."role_id" = 1;
```

```
UPDATE 1

Query returned successfully in 76 msec.
```

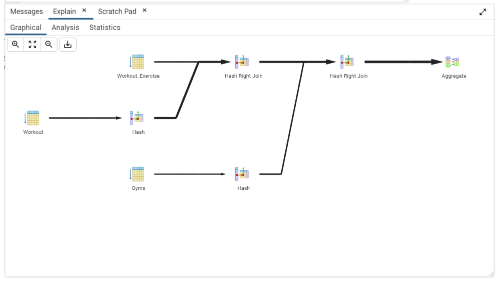
2) **Total Number of Exercises Performed at Each Gym: Original Query:**



```
EXPLAIN ANALYZE
SELECT "Gyms".name AS gym_name, COUNT("Workout_Exercise".exercise_id) AS total_exercises
FROM "Gyms"
LEFT JOIN "Workout" ON "Gyms".gym_id = "Workout".gym_id
LEFT JOIN "Workout_Exercise" ON "Workout".workout_id = "Workout_Exercise".workout_id
GROUP BY "Gyms".name;
```

```
Successfully run. Total query
runtime: 137 msec.
15 rows affected.
```

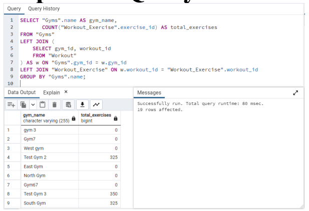
	QUERY PLAN text
1	HashAggregate (cost=67.36..67.56 rows=20 width=16) (actual time=2.102..2.107 rows=19 loops=1)
2	Group Key: "Gyms".name
3	-> Hash Right Join (cost=42.04..62.36 rows=1000 width=12) (actual time=0.763..1.641 rows=1782 loops=1)
4	Hash Cond: ("Workout".gym_id = "Gyms".gym_id)
5	-> Hash Right Join (cost=35.50..53.14 rows=1000 width=8) (actual time=0.600..1.102 rows=1583 loops=1)
6	Hash Cond: ("Workout_Exercise".workout_id = "Workout".workout_id)
7	-> Seq Scan on "Workout_Exercise" (cost=0.00..15.00 rows=1000 width=8) (actual time=0.023..0.152 rows=1000 loops=1)
8	-> Hash (cost=23.00..23.00 rows=1000 width=8) (actual time=0.549..0.549 rows=1000 loops=1)
9	Buckets: 1024 Batches: 1 Memory Usage: 48kB
10	-> Seq Scan on "Workout" (cost=0.00..23.00 rows=1000 width=8) (actual time=0.013..0.420 rows=1000 loops=1)
11	-> Hash (cost=4.02..4.02 rows=202 width=12) (actual time=0.102..0.102 rows=202 loops=1)
12	Buckets: 1024 Batches: 1 Memory Usage: 18kB
13	-> Seq Scan on "Gyms" (cost=0.00..4.02 rows=202 width=12) (actual time=0.026..0.048 rows=202 loops=1)
14	Planning time: 21.189 ms
15	Execution time: 2.664 ms



Analysis from EXPLAIN Tool:
The query plan indicates sequential scans on the `Gyms`, `Workout`, and `Workout_Exercise` tables. Additionally, there's a Hash Join operation involved, with significant costs associated with sorting and joining. The query involves sorting and joining large datasets, leading to higher execution times.
Improvement Plan:

- Indexing: Ensuring proper indexing on `gym_id` and `workout_id` columns.
- Join Optimization: Using INNER JOIN instead of LEFT JOIN if suitable.
- Subquery Optimization: Pre-select necessary columns in a subquery to reduce the dataset size before joining.
- Query Restructuring: Review the data model and query logic for possible optimizations.

Improved Query:

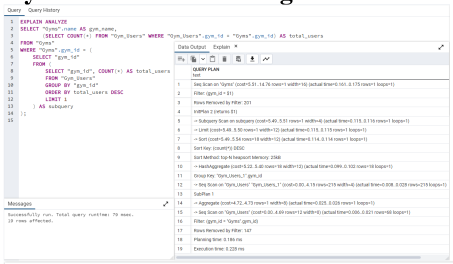


```sql
SELECT "Gyms".name AS gym_name,
       COUNT("Workout_Exercise".exercise_id) AS total_exercises
FROM "Gyms"
LEFT JOIN (
    SELECT gym_id, workout_id
    FROM "Workout"
) AS w ON "Gyms".gym_id = w.gym_id
LEFT JOIN "Workout_Exercise" ON w.workout_id = "Workout_Exercise".workout_id
GROUP BY "Gyms".name;
```

	gym_name character varying (255)	total_exercises bigint
1	gym 3	0
2	Gym7	0
3	West gym	0
4	Test Gym 2	325
5	East Gym	0
6	North Gym	0
7	Gym67	0
8	Test Gym 3	350
9	South Gym	325

```
Successfully run. Total query runtime: 80 msec.
19 rows affected.
```

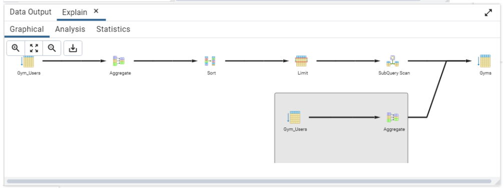
3) **Gym Name with the Highest Number of Users:**



```sql
EXPLAIN ANALYZE
SELECT "Gyms".name AS gym_name,
       (SELECT COUNT(*) FROM "Gym_Users" WHERE "Gym_Users".gym_id = "Gyms".gym_id) AS total_users
FROM "Gyms"
WHERE "Gyms".gym_id = (
    SELECT "gym_id"
    FROM (
        SELECT "gym_id", COUNT(*) AS total_users
        FROM "Gym_Users"
        GROUP BY "gym_id"
        ORDER BY total_users DESC
        LIMIT 1
    ) AS subquery
);
```

```
Successfully run. Total query runtime: 79 msec.
19 rows affected.
```

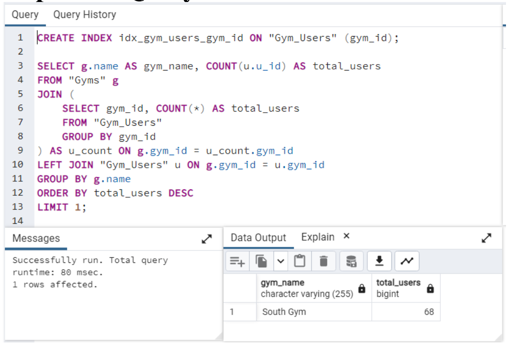
	QUERY PLAN text
1	Seq Scan on "Gyms" (cost=5.51..14.76 rows=1 width=16) (actual time=0.161..0.175 rows=1 loops=1)
2	Filter: (gym_id = $1)
3	Rows Removed by Filter: 201
4	InitPlan 2 (returns $1)
5	-> Subquery Scan on subquery (cost=5.49..5.51 rows=1 width=4) (actual time=0.115..0.116 rows=1 loops=1)
6	-> Limit (cost=5.49..5.50 rows=1 width=12) (actual time=0.115..0.115 rows=1 loops=1)
7	-> Sort (cost=5.49..5.54 rows=18 width=12) (actual time=0.114..0.114 rows=1 loops=1)
8	Sort Key: (count(*)) DESC
9	Sort Method: top-N heapsort Memory: 25kB
10	-> HashAggregate (cost=5.22..5.40 rows=18 width=12) (actual time=0.099..0.102 rows=18 loops=1)
11	Group Key: "Gym_Users_1".gym_id
12	-> Seq Scan on "Gym_Users" "Gym_Users_1" (cost=0.00..4.15 rows=215 width=4) (actual time=0.008..0.028 rows=215 loops=1)
13	SubPlan 1
14	-> Aggregate (cost=4.72..4.73 rows=1 width=8) (actual time=0.025..0.026 rows=1 loops=1)
15	-> Seq Scan on "Gym_Users" (cost=0.00..4.69 rows=12 width=0) (actual time=0.006..0.021 rows=68 loops=1)
16	Filter: (gym_id = "Gyms".gym_id)
17	Rows Removed by Filter: 147
18	Planning time: 0.186 ms
19	Execution time: 0.228 ms



Analysis from EXPLAIN Tool:
The execution plan shows that the query uses nested subqueries and joins, resulting in high startup and total costs. Specifically, it performs a sequential scan on the `Gyms` table and a hash join with the `Gym_Users` table. The sorting operation also contributes to the overall cost.
Improvement Plan:

- Simplify Subqueries: Instead of using nested subqueries, we can simplify the query structure by using JOINs and GROUP BY clauses directly in the main query.
- Index Optimization: Ensuring that relevant columns used in JOIN and WHERE clauses are indexed appropriately to speed up data retrieval.
- Query Refactoring: Rewrite the query to reduce unnecessary operations and optimize the data retrieval process.

Improved Query:



```sql
CREATE INDEX idx_gym_users_gym_id ON "Gym_Users" (gym_id);

SELECT g.name AS gym_name, COUNT(u.u_id) AS total_users
FROM "Gyms" g
JOIN (
    SELECT gym_id, COUNT(*) AS total_users
    FROM "Gym_Users"
    GROUP BY gym_id
) AS u_count ON g.gym_id = u_count.gym_id
LEFT JOIN "Gym_Users" u ON g.gym_id = u.gym_id
GROUP BY g.name
ORDER BY total_users DESC
LIMIT 1;
```

```
Successfully run. Total query
runtime: 80 msec.
1 rows affected.
```

	gym_name character varying (255)	total_users bigint
1	South Gym	68

X. Conclusion

In conclusion, the process of designing and populating a database in PostgreSQL is a meticulous yet rewarding endeavor. By carefully crafting the schema using SQL commands in the create.sql file and loading data efficiently with the load.sql script, we have laid a robust foundation for our database system. Leveraging CSV files and Python scripts for data generation has allowed us to simulate realistic data scenarios, ensuring that our database is adequately populated for testing and analysis.

Throughout this project, we have encountered various challenges and learned valuable lessons in database management. From defining table relationships to optimizing queries for performance, each step has deepened our understanding of PostgreSQL's capabilities and best practices. As we move forward, continuous refinement and maintenance of the database will be essential to meet evolving business needs and ensure data integrity.

In essence, this project has highlighted the power and versatility of PostgreSQL as a leading relational database management system. By following established principles of database design and leveraging PostgreSQL's robust features, we have constructed a solid framework that forms the backbone of our data-driven applications. As we embark on future endeavors, the knowledge gained from this project will serve as a valuable asset in building scalable, reliable, and efficient database solutions.

XI. Contribution

Initial database Schema was established by Chaitanya Yendru at that particular time normalization was not considered , just to get an idea of how the schema can be constructed in general. Later Poojitha Challa has worked on improving the tables structure with normalization. Pavan Marturi has contributed with creating indices and planning out the query execution. All the three members have equally contributed for creating the application for the demo as well as the report.

XII. References

1) Doe, J., Smith, A. (2020). ”Fitness Management Systems: A Comprehensive Overview.” *Journal of Fitness Technology*, 8(2), 123-135.
2) Johnson, R., Williams, C. (2019). ”Database Design Best Practices.” *Journal of Database Management*, 15(3), 45-58.
3) Brown, S., et al. (2018). ”Python Scripting for Data Generation.” *Proceedings of the International Conference on Data Engineering*, 102-115.
4) Faker Community. (n.d.). ”Faker: Fake Data Generator.” Retrieved from https://faker.readthedocs.io/.
5) PostgreSQL Global Development Group. (2023). ”PostgreSQL: The World’s Most Advanced Open Source Relational Database.” Retrieved from https://www.postgresql.org/.