Meal Planner – Indexing Documentation

Bharath Sadagopan

Overview: This document describes the indexes used in the Meal Planner Django application and explains which queries and reports benefit from those indexes. All indexes mentioned are either automatically created by Django or can possibly be manually added for performance optimization.

Database Tables: The application uses the following core models:

- User
- Ingredient
- Meal
- MealIngredient
- MealPlan
- MealPlanMeal

Table: Meals

Primary Indexes:

- `id` (Primary Key): auto-generated by Django
- `created_by_id` (Foreign key to User): auto-indexed by Django
- `total_calories`: used in range queries if indexed by a B+ tree(done manually)

Queries Supported:

- `Meal.objects.filter(created_by=some_user)`
    - Used to display meals created by a user
    - Benefits from index on **`created_by_id`**

Ex:

SELECT *

FROM Meals

WHERE created_by = “Johnny Appleseed”;

- `Meal.objects.filter(total_calories__range=(min, max))`
    - Used in reports for filtering meals by calorie range
    - Would benefit from index on **`total_calories`**

Ex:

SELECT *

FROM Meals

WHERE total_calories >= 300 AND total_calories <= 600;

Table: MealIngredient

Primary Indexes:

- `id` (Primary Key)
- `meal_id` (Foreign key to Meal): auto-indexed by Django
- `ingredient_id` (Foreign key to Ingredient): auto-indexed by Django

Queries Supported:

- `MealIngredient.objects.filter(meal=some_meal)`
  - Used to list all ingredients for a given meal
  - Benefits from index on **`meal_id`**

Ex:

SELECT i.name

FROM MealIngredient mi

JOIN Ingredient i ON mi.ingredient_id = i.id

WHERE mi.meal_id = 3;

- `MealIngredient.objects.filter(ingredient=some_ingredient)`
  - Used for ingredient reporting
  - Benefits from index on **`ingredient_id`**

Ex:

SELECT i.name

FROM MealIngredient mi

JOIN Ingredient i ON mi.ingredient_id = i.id

WHERE mi.ingredient_id = 4;

Table: MealPlan

Primary Indexes:

- `id` (Primary Key)
- `user_id` (Foreign key to User): auto-indexed by Django
- `date`: index to support reporting and filtering through clustered B+ tree(done manually)

Supporting Indexes:

```
class Meta:
   indexes = [
      models.Index(fields=['date']),
   ]
```

Queries Supported:

- `MealPlan.objects.filter(user=some_user)`
    - Used in user-specific views and reports
    - Benefits from index on **`user_id`**

Ex:

```
SELECT u.name, u.email

FROM MealPlan mp

JOIN User u ON mp.user_id = u.id

 WHERE mp.user_id = 5
```

- `MealPlan.objects.filter(date__range=(start_date, end_date))`
    - Used in calorie report and filtering
    - Benefits from index on **`date`**

Ex:

```
SELECT user

FROM MealPlan

WHERE data BETWEEN ‘2025-05-01’ AND ‘2025-05-03’;
```

Table: MealPlanMeal

Primary Indexes:

- `id` (Primary Key)
- `plan_id` (Foreign key to MealPlan): auto-indexed by Django
- `meal_id` (Foreign key to Meal): auto-indexed by Django

Queries Supported:

- `MealPlanMeal.objects.filter(plan=some_plan)`
    - Used to retrieve all meals under a plan
    - Benefits from index on **`plan_id`**

Ex:

SELECT mp.date

FROM MealPlanMeal mpm

JOIN MealPlan mp ON mpm.plan_id = mp._id

WHERE mpm.plan_id = 3;

- `MealPlanMeal.objects.filter(meal=some_meal)`
    - Used to trace a meal back to plans
    - Benefits from index on **`meal_id`**

Ex:

SELECT m.name, m.total_calories

FROM MealPlanMeal mpm

JOIN Meal m ON mpm.meal_id = m._id

WHERE mpm.meal_id = 2;

Table: Ingredient

Primary Indexes:

- `id` (Primary Key): auto indexed by Django
- `name`: would need to be manually indexed maybe using a hash if it is unique

Queries Supported:

- `Ingredient.objects.get(id=some_id)`
    - Used in forms or reports when matching by **id**

Ex:

SELECT name, calories_per_unit

FROM Ingredient

WHERE id = 6';

Table: User

Primary Indexes:

- `id` (Primary Key): auto indexed by Django

Queries Supported:

- `User.objects.get(id=some_id)`
    - Common for foreign key joins
    - Benefits from **default primary key index**

Ex:

SELECT name, email

FROM User

WHERE id = 24 AND id = 3;

Conclusion: The Meal Planner project effectively utilizes Django's default indexing on primary and foreign keys. Manual indexing on fields like **`date`** and **`total_calories`** is recommended to improve performance of commonly used filters and reporting queries. These indexes help optimize core operations like calorie tracking, user-specific views, and ingredient analysis. I could continue indexing on `Ingredient.name` or `User.email` for faster search and validation.