Query Methods are like SELECT

From the example:

Basic Queries

- `Superhero.objects.all()` : Retrieves all superhero objects from the database.
- `Superhero.objects.filter(name="Superman")` : Filters superheroes with the name "Superman."
- `Superhero.objects.filter(name="Superman").first()` : Gets the first superhero object with the name "Superman."
- `Superhero.objects.filter(name="Spider-Man").first().secret_identity` : Fetches the secret identity of the first "Spider-Man" object.
- `Superhero.objects.filter(name="Superman").first().enemies.all()` : Retrieves all enemies related to the first "Superman" object.
- `Superhero.objects.filter(name="Spider-Man").first().powers.all()` : Gets all powers associated with the first "Spider-Man" object.
- `Superhero.objects.exclude(name="Wonder Woman")` : Retrieves all superhero objects except those named "Wonder Woman."
- `Superhero.objects.order_by("name")` : Orders all superhero objects by their names in ascending order.

Aggregations and Counts

- `Superhero.objects.count()` : Counts the total number of superhero objects.
- `Superhero.objects.aggregate(Count("name"))` : Counts the number of distinct superhero names.
- `Superhero.objects.aggregate(Min("name"))` : Finds the superhero with the alphabetically first name.
- `Superhero.objects.aggregate(Max("name"))` : Finds the superhero with the alphabetically last name.
- `Superhero.objects.aggregate(Min("name"),Max("name"))` : Combines the previous two queries.
- `Superhero.objects.filter(name__contains="man").count()` : Counts superheroes whose names contain "man."

Complex Filtering

- `Superhero.objects.filter(name__contains="man").exclude(name__contains="woman")` : Filters superheroes containing "man" but not "woman."
- `Superhero.objects.filter(enemies__name__icontains="Luthor").first().name` : Finds the name of the first superhero whose enemies include someone with "Luthor" in their name (case-insensitive).
- `Superhero.objects.filter(q_deadpool | q_woman)` : Uses Q objects for complex filtering, likely retrieving superheroes named "Deadpool" or containing "woman."

Slicing and Limiting

- `Superhero.objects.all()[:3]` : Retrieves the first three superhero objects.
- `Superhero.objects.filter(name__contains="man")[:2]` : Gets the first two superheroes whose names contain "man."

Perform various database operations using Django's ORM, including filtering, ordering, counting, aggregating, and complex queries.

Returning Objects

- `get()` : Retrieves a single, unique object based on given conditions. Raises `DoesNotExist` if no object is found, or `MultipleObjectsReturned` if multiple objects match.
- `create()` : Creates and saves a new object in the database.
- `get_or_create()` : Tries to get an object based on given conditions. If it doesn't exist, it creates and saves a new one.
- `update_or_create()` : Similar to `get_or_create()`, but updates an existing object if found, or creates a new one.
- `latest()` : Returns the latest object in the table based on a given `DateTimeField`.
- `earliest()` : Returns the earliest object in the table based on a given `DateTimeField`.
- `first()` : Returns the first object in the table.
- `last()` : Returns the last object in the table.

Returning Other Values

- `bulk_create()` : Creates multiple objects in a single query, returning `None`.
- `count()` : Returns the number of objects in the queryset (an integer).

- `in_bulk()` : Takes a list of primary keys and returns a dictionary mapping those keys to the corresponding objects.
- `iterator()` : Returns an iterator over the queryset, which can be more memory-efficient for large datasets.
- `aggregate()` : Performs aggregation functions (e.g., `Sum`, `Avg`, `Count`) on the queryset and returns a dictionary of results.
- `exists()` : Returns `True` if the queryset contains any objects, `False` otherwise.
- `update()` : Updates all objects in the queryset with the given values and returns the number of updated objects.
- `delete()` : Deletes all objects in the queryset and returns the number of deleted objects.
- `as_manager()` : Returns a `Manager` object that can be attached to a custom model manager.

Differences

- **Object Retrieval vs. Other Actions:** The first group primarily focuses on retrieving specific objects or creating new ones. The second group performs actions on the queryset itself (counting, aggregating, updating, deleting) or returns non-object values (e.g., counts, dictionaries).
- **Exceptions vs. Return Values:** Methods like `get()` raise exceptions if the expected object is not found or if multiple objects match. Other methods typically return specific values or perform actions without raising exceptions.

Choosing the Right Method

Let's explore the key differences between these two approaches:

1. Return Value

- `Superhero.objects.filter(name="Spider-Man").first()`
    - Returns the *first* `Superhero` object that matches the filter condition (name="Spider-Man").
    - If no matching object is found, it returns `None`.
    - It's possible for multiple `Superhero` objects to have the name "Spider-Man" (e.g., different versions or universes). This method will only return the first one encountered.
- `Superhero.objects.get(name="Spider-Man")`
    - Returns the *single, unique* `Superhero` object that matches the condition.
    - If no matching object is found, it raises a `DoesNotExist` exception.

    - If multiple objects match the condition, it raises a `MultipleObjectsReturned` exception.
    - This method assumes that the `name` field is unique or that you're specifically looking for one particular “Spider-Man."

2. Error Handling

- `filter().first()` is more forgiving. If no object is found, it simply returns `None`, which you can easily check in your code.
- `get()` is stricter. It forces you to handle potential exceptions (`DoesNotExist`, `MultipleObjectsReturned`) using `try-except` blocks.

3. Use Cases

- `filter().first()` is suitable when:
    - You want to retrieve *at most one* object that matches a condition.
    - You're unsure if an object with the given condition exists.
    - You want to avoid exceptions and handle the absence of an object gracefully.
- `get()` is suitable when:
    - You're certain that *exactly one* object matches the condition.
    - You want to enforce data integrity (e.g., the `name` field should be unique).
    - You want to explicitly handle cases where no object or multiple objects are found.

4. Performance

- In general, there's no significant performance difference between the two, especially if the `name` field is indexed in the database.
- `get()` might be slightly faster in some cases because it directly fetches a single object, whereas `filter().first()` might involve filtering a potentially larger set of objects first. However, the difference is usually negligible.

Recommendation

- If you expect a single, unique object and want to enforce data integrity, use `get()`.
- If you want flexibility and graceful handling of potential missing objects, use `filter().first()`.

Django ORM methods mapped to approximately their corresponding SQL equivalents. The exact SQL generated might vary slightly depending on your database backend and specific query conditions.

Returning QuerySets

- `filter()`: `SELECT ... FROM ... WHERE ...` (Conditions specified in the `filter()` translate to `WHERE` clauses)
- `exclude()`: `SELECT ... FROM ... WHERE NOT ...` (Conditions become negated `WHERE` clauses)
- `annotate()`: Often involves subqueries or joins to calculate additional fields.
- `order_by()`: `SELECT ... FROM ... ORDER BY ...`
- `reverse()`: Modifies the `ORDER BY` clause to descending order.
- `distinct()`: `SELECT DISTINCT ... FROM ...`
- `values()`: `SELECT field1, field2, ... FROM ...` (Only specified fields are selected)
- `values_list()`: Similar to `values()`, but might return a flatter structure depending on the backend.
- `dates()` / `datetimes()`: Extracts date/datetime components using SQL functions (e.g., `DATE()`, `EXTRACT()`).
- `none()`: No SQL equivalent, represents an empty result set.
- `all()`: Typically generates a simple `SELECT * FROM ...`

QuerySet Set Operations

- `union()`: `SELECT ... FROM ... UNION SELECT ... FROM ...`
- `intersection()`: `SELECT ... FROM ... INTERSECT SELECT ... FROM ...` (Some backends might require workarounds)
- `difference()`: `SELECT ... FROM ... EXCEPT SELECT ... FROM ...` (Similar to `intersection()`)

Performance Optimizations

- `select_related()`: Uses `JOIN`s to fetch related objects in a single query.
- `prefetch_related()`: Executes separate queries for related objects but optimizes fetching.

Other Methods

- `extra()`: Injects additional SQL snippets into the generated query.
- `defer()` / `only()`: Might influence the `SELECT` clause to include/exclude specific fields.
- `using()`: Can hint the database router to use a specific database connection.
- `select_for_update()`: Typically adds a `FOR UPDATE` clause to lock rows.
- `raw()`: Directly executes the provided raw SQL query.

Returning Objects

- `get()` : `SELECT ... FROM ... WHERE ... LIMIT 1` (Assumes a single result)
- `create()` : `INSERT INTO ... (...) VALUES (...)`
- `get_or_create()` / `update_or_create()` : Combines `SELECT` and `INSERT` or `UPDATE`.
- `latest()` / `earliest()` : `SELECT ... FROM ... ORDER BY ... DESC/ASC LIMIT 1`
- `first()` / `last()` : Similar to `latest()` / `earliest()`, but might use different ordering depending on the database.

Returning Other Values

- `bulk_create()` : Multiple `INSERT` statements or optimized bulk insert mechanisms.
- `count()` : `SELECT COUNT(*) FROM ...`
- `in_bulk()` : `SELECT ... FROM ... WHERE id IN (...)`
- `iterator()` : Might use cursor-based fetching to optimize memory usage.
- `aggregate()` : `SELECT AGG_FUNCTION(...) FROM ...` (e.g., `SUM`, `AVG`, `COUNT`)
- `exists()` : `SELECT EXISTS(SELECT 1 FROM ...)` (Optimized for existence check)
- `update()` : `UPDATE ... SET ... WHERE ...`
- `delete()` : `DELETE FROM ... WHERE ...`

Given a `Book` model with fields like `title`, `author`, `publication_date`, `genre`, `price`, and `is_available`:

Exact and Case-Insensitive Matches

- Find a book with the exact title "Pride and Prejudice":

```
Book.objects.get(title__exact='Pride and Prejudice')
```

- Find a book whose author's name is "jane austen" (case-insensitive):

```
Book.objects.filter(author__iexact='jane austen')
```

Contains and Case-Insensitive Contains

- Find all books whose titles contain the word "Python":

```
Book.objects.filter(title__contains='Python'
```

- Find all books whose genres contain the word "fantasy" (case-insensitive):

```
Book.objects.filter(genre__icontains='fantasy')
```

Membership

- Find all books whose genres are either "Science Fiction" or "Mystery":

```
Book.objects.filter(genre__in=['Science Fiction',
'Mystery'])
```

Comparisons

- Find all books published after the year 2000:

```
Book.objects.filter(publication_date__year__gt=2000)
```

- Find all books priced between $10 and $20 (inclusive):

```
Book.objects.filter(price__range=(10, 20))
```

Starts/Ends With

- Find all books whose titles start with "The":

```
Book.objects.filter(title__startswith='The')
```

- Find all books whose authors' names end with "Smith" (case-insensitive):

```
Book.objects.filter(author__iendswith='smith')
```

Date/Time Components

- Find all books published in the month of December:

```
Book.objects.filter(publication_date__month=12)
```

- Find all books published on a Wednesday:

```
Book.objects.filter(publication_date__weekday=3)  # 3
represents Wednesday
```

Null Checks and Regular Expressions

- Find all books that are currently unavailable:

```
Book.objects.filter(is_available__isnull=True)
```

- Find all books whose titles match the regular expression pattern '^[A-Z].*': (Starts with a capital letter)

```
Book.objects.filter(title__regex='^[A-Z].*')
```