**Optimize SQL Requests in Django**

[[Vincent Lossel](https://vincentlossel.medium.com/?source=post_page-----5452edd61c04--------------------------------)](https://vincentlossel.medium.com/?source=post_page-----5452edd61c04--------------------------------)

[Vincent Lossel](https://vincentlossel.medium.com/?source=post_page-----5452edd61c04--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F21a8d02caca8&operation=register&redirect=https%3A%2F%2Fvincentlossel.medium.com%2Fsolving-the-n-1-problem-using-django-5452edd61c04&user=Vincent+Lossel&userId=21a8d02caca8&source=post_page-21a8d02caca8----5452edd61c04---------------------post_header-----------)

3 min read

·

1 day ago

10



Django uses an ORM (*Object-Relational Mapping*), which is an interface for managing the database by adding a layer of abstraction. Queries takes the form of operations on objects, directly in Python. Like with any ORM, using Django’s ORM has advantages and disadvantages.

Each model is accompanied by a Manager implemented through the Model class from which our models must inherits. This manager provides methods to perform various queries (Create, Retrieve, Update, Delete…).

my\_account = Account.objects.create(username="Pedro")

It’s important to understand that building an optimized query within such an automated and standardized system is a complex task. Simple queries are usually well-handled and don’t pose real performance issues. However, complexe queries often lead to a performance loss that an SQL query could have avoided.

**Measurement Tools**

Before attempting to optimize ORM queries, it’s crucial to measure existing performance and evaluate the ORM’s actions for each query. For this purpose, I recommend using django-debug-toolbar.

Keep in mind that trying to optimize a system still in development or not facing any performance issues is not a good idea. Developers often tend to optimize for future problems that may never occur.

Django Debug Toolbar is a Django application that provides extensive information about Django’s operation and its various systems. During development, django-debug-toolbar returns information about ORM queries, templates, caching, and more. In essence, it is a development tool that should be present in all your Django projects.

There are other projects, sometimes more specific, to identify or fix the N+1 query problem: nplusone, scout-apm, or django-auto-prefetch…

**The N+1 Query Problem**

Soon enough, you’ll notice that some pages perform several dozens (or hundreds) of queries. A simple .filter() that should return 200 objects turns into 200 different queries every time the page loads! It’s evident that the performance will be terrible on these pages.

In fact, Django’s ORM is lazy, meaning it only fetches from the database when necessary. This results in Django executing each query one after the other because it’s not certain it will need the next one immediately. While this choice may be beneficial for simple queries, it poses significant problems when the query is more complex. Often, it’s preferable to make a broader query in advance and process the elements afterward. This practice has severe consequences when using the ORM conventionally.

For example, if you want to retrieve information about comment authors, you first have to get information about the comments. Then, for each retrieved comment, Django will launch a new query to obtain information about the author (Foreign Key on the User table). This situation is typical and falls under an N+1 problem.

Obviously, it’s preferable to make a single SQL query with the appropriate joins to avoid bombarding the database in this manner.

**Solutions to the N+1 Query Problem**

In its current version, Django incorporates two solutions to avoid launching too many queries on the database.

select\_related()

To prevent last-minute queries by the ORM, you can specify .select\_related() to instruct it to retrieve a related item. This is a way of making a join via the ORM and thus reducing the total number of queries.

my\_account = Account.objects.get(username="Pedro").select\_related("profile")

In this case, this query will retrieve the Account model instance corresponding to username="Pedro" and perform a join on the profile field, which refers to the Profile model instance associated with this account. Instead of making two queries, only one is executed.

**prefetch\_related()**

Unlike select\_related(), this method avoids data duplication concerning the join. It operates similarly to .select\_related() as it is a join. However, Django retains information in memory to avoid any data duplication. For example, the author of multiple comments will be retrieved only once. Additionally, prefetch\_related() allows fetching models in a Many-to-Many or Many-to-One relationship.

This query is slightly more costly than .select\_related(), but it should be preferred when your queries involve many models related to each other.

[Django](https://medium.com/tag/django?source=post_page-----5452edd61c04---------------django-----------------)

[Programming](https://medium.com/tag/programming?source=post_page-----5452edd61c04---------------programming-----------------)