DATABASE ACCESS OPTIMIZATION

Django mastery in Nepali

Database access optimization

Django's database layer provides various ways to help developers get the most out of their databases. This document gathers links to the relevant documentation, and adds various tips, organized under a number of headings that outline the steps to take when attempting to optimize your database usage.

Profile first

As general programming practice, this goes without saying. Find out what queries you are doing and what they are costing you. Use ***QuerySet.explain()*** to understand how specific QuerySets are executed by your database. You may also want to use an external project like django-debug-toolbar, or a tool that monitors your database directly. Remember that you may be optimizing for speed or memory or both, depending on your requirements. Sometimes optimizing for one will be detrimental to the other, but sometimes they will help each other. Also, work that is done by the database process might not have the same cost (to you) as the same amount of work done in your Python process. It is up to you to decide what your priorities are, where the balance must lie, and profile apply or these as required since this will depend on your application and server. With everything that follows, remember to profile after every change to ensure that the change is a benefit, and a big enough benefit given the decrease in readability of your code. apply or the suggestions below come with the caveat that in your circumstances the general principle might not apply or might even be reversed.

Use standard DB optimization techniques

. . .including:

- Indexes. This is a number one priority, after you have determined from profiling what indexes should be added. Use Meta.indexes or Field.db_index to add these from Django. Consider adding indexes to fields that you frequently query using filter( ), exclude( ), order_by( ), etc. as indexes may help to speed up lookups. Note that determining the best indexes is a complex database-dependent topic that will depend on your particular application. The overhead of maintaining an index may outweigh any gains in query speed.
- Appropriate use of field types.

Understand QuerySet evaluation

To avoid performance problems, it is important to understand:

- that QuerySets are lazy.
- when they are evaluated.
- how the data is held in memory.

Understand cached attributes

As well as caching of the whole QuerySet, there is caching of the result of attributes on ORM objects. In general, attributes that are not callable will be cached. For example, assuming the example blog models:

```
>>> entry = Entry.objects.get(id=1)
>>> entry.blog # Blog object is retrieved at this point
>>> entry.blog # cached version, no DB access
```

`But in general, callable attributes cause DB lookups every time:`

```
>>> entry = Entry.objects.get(id=1)
>>> entry.authors.all() # query performed
>>> entry.authors.all() # query performed again
```

Be careful with your own custom properties - it is up to you to implement caching when required, for example using the cached_property decorator.

Do database work in the database rather than in Python

For instance:

- At the most basic level, use filter and exclude to do filtering in the database.
- Use F expressions to filter based on other fields within the same model.
- Use annotate to do aggregation in the database

Retrieve individual objects using a unique, indexed column

There are two reasons to use a column with unique or db_index when using get( ) to retrieve individual objects. First, the query will be quicker because of the underlying database index. Also, the query could run much slower if multiple objects match the lookup; having a unique constraint on the column guarantees this will never happen.

So using the example blog models:

```
>>> entry = Entry.objects.get(id=10)
```

will be quicker than:

```
>>> entry = Entry.objects.get(headline="News Item Title")
```

because id is indexed by the database and is guaranteed to be unique.

Doing the following is potentially quite slow:

```
>>> entry = Entry.objects.get(headline__startswith="News")
```

First of all, headline is not indexed, which will make the underlying database fetch slower. Second, the lookup doesn’t guarantee that only one object will be returned. If the query matches more than one object, it will retrieve and transfer all of them from the database. This penalty could be substantial if hundreds or thousands of records are returned. The penalty will be compounded if the database lives on a separate server, where network overhead and latency also play a factor.

Retrieve everything at once if you know you will need it

Hitting the database multiple times for different parts of a single ‘set’ of data that you will need all parts of is, in general, less efficient than retrieving it all in one query. This is particularly important if you have a query that is executed in a loop, and could therefore end up doing many database queries, when only one was needed. So:

Use QuerySet.select_related() and prefetch_related()

Understand select_related() and prefetch_related() thoroughly, and use them:

- in managers and default managers where appropriate. Be aware when your manager is and is not used; sometimes this is tricky so don’t make assumptions.
- in view code or other layers, possibly making use of prefetch_related_objects() where needed.

Don’t retrieve things you don’t need

Use QuerySet.values() and values_list()

When you only want a dict or list of values, and don’t need ORM model objects, make appropriate usage of values(). These can be useful for replacing model objects in template code – if the dicts you supply have the same attributes as those used in the template, you are fine.

Use QuerySet.defer() and only()

Use defer() and only() if there are database columns you know that you won’t need (or won’t need in most cases) to avoid loading them. Note that if you do use them, the ORM will have to go and get them in a separate query, making this a pessimization if you use it inappropriately. Don’t be too aggressive in deferring fields without profiling as the database has to read most of the non-text, non-VARCHAR data from the disk for a single row in the results, even if it ends up only using a few columns. The defer() and only() methods are most useful when you can avoid loading a lot of text data or for fields that might take a lot of processing to convert back to Python. As always, profile first, then optimize.

Use QuerySet.contains(obj)

. . .if you only want to find out if obj is in the queryset, rather than if obj in queryset.

Use QuerySet.count()

. . .if you only want the count, rather than doing *len(queryset).*

Use QuerySet.exists()

. . .if you only want to find out if at least one result exists, rather than if queryset.

Use bulk methods

Use bulk methods to reduce the number of SQL statements.

Create in bulk

When creating objects, where possible, use the bulk_create() method to reduce the number of SQL queries.

For example:

```
Entry.objects.bulk_create(
    [
        Entry(headline="This is a test"),
        Entry(headline="This is only a test"),
    ]
)
```

Use QuerySet.update() and delete()

Rather than retrieve a load of objects, set some values, and save them individual, use a bulk SQL UPDATE statement, via QuerySet.update( ). Similarly, do bulk deletes where possible. Note, however, that these bulk update methods cannot call the save( ) or delete( ) methods of individual instances, which means that any custom behavior you have added for these methods will not be executed, including anything driven from the normal database object signals.

Don't order results if you don't care

Ordering is not free; each field to order by is an operation the database must perform. If a model has a default ordering (Meta.ordering) and you don't need it, remove it on a QuerySet by calling order_by( ) with no parameters. Adding an index to your database may help to improve ordering performance.

Update in bulk

When updating objects, where possible, use the bulk_update() method to reduce the number of SQL queries.

Given a list or queryset of objects:

```
entries = Entry.objects.bulk_create(
    [
        Entry(headline="This is a test"),
        Entry(headline="This is only a test"),
    ]
)
```

The following example:

```
entries[0].headline = "This is not a test"
entries[1].headline = "This is no longer a test"
Entry.objects.bulk_update(entries, ["headline"])
```

. . .is preferable to:

```
entries[0].headline = "This is not a test"
entries[0].save()
entries[1].headline = "This is no longer a test"
entries[1].save()
```

Insert in bulk

When inserting objects into ManyToManyFields, use add() with multiple objects to reduce the number of SQL queries. For example:

```
my_band.members.add(me, my_friend)

. . .is preferable to:
my_band.members.add(me)
my_band.members.add(my_friend)
```

. . .where Bands and Artists have a many-to-many relationship.

When inserting different pairs of objects into ManyToManyField or when the custom through table is defined, use bulk_create() method to reduce the number of SQL queries. For example:

```
PizzaToppingRelationship = Pizza.toppings.through
PizzaToppingRelationship.objects.bulk_create(
    [
        PizzaToppingRelationship(pizza=my_pizza, topping=pepperoni),
        PizzaToppingRelationship(pizza=your_pizza, topping=pepperoni),
        PizzaToppingRelationship(pizza=your_pizza, topping=mushroom),
    ],
    ignore_conflicts=True,
)
. . .is preferable to:
my_pizza.toppings.add(pepperoni)
your_pizza.toppings.add(pepperoni, mushroom)
```

. . .where Pizza and Topping have a many-to-many relationship. Note that there are a number of caveats to this method, so make sure it’s appropriate for your use case.

Remove in bulk

When removing objects from ManyToManyFields, use remove( ) with multiple objects to reduce the number of SQL queries. For example:

```
my_band.members.remove(me, my_friend)
```

. . .is preferable to:

```
my_band.members.remove(me)
my_band.members.remove(my_friend)
```

. . .where Bands and Artists have a many-to-many relationship.

When removing different pairs of objects from ManyToManyFields, use delete( ) on a Q expression with multiple through model instances to reduce the number of SQL queries. For example:

```
from django.db.models import Q
PizzaToppingRelationship = Pizza.toppings.through
PizzaToppingRelationship.objects.filter(
Q(pizza=my_pizza, topping=pepperoni)
| Q(pizza=your_pizza, topping=pepperoni)
| Q(pizza=your_pizza, topping=mushroom)
).delete()
```

. . .is preferable to:

```
my_pizza.toppings.remove(pepperoni)
your_pizza.toppings.remove(pepperoni, mushroom)
```

. . .where Pizza and Topping have a many-to-many relationship

Thanks for watching 💖