Engineer

ORM

a library that maps database tables to objects in code

What it is

An ORM, short for object-relational mapper, is a library that maps the tables of a relational database to classes in application code, and their rows to objects. Code asks for an author and gets an `Author` object back, changes one of its attributes and saves it, and the ORM writes the `SELECT`, `INSERT` and `UPDATE` statements that do the work. Django's models and SQLAlchemy's ORM, both written in Python, are two well-known examples, and they are the two this article draws on.

The mapping is needed because the two sides organise data differently. Martin Fowler puts it plainly in his description of the Data Mapper pattern: objects and relational databases have different ways of structuring data, and parts of an object model, such as collections and inheritance, have no counterpart in a relational database. Moving data between the two becomes a problem of its own, and an ORM is a ready-made answer to it.

The parts of that answer were named in Fowler's book Patterns of Enterprise Application Architecture, published in 2003, whose pattern catalogue is online. Two patterns describe the overall shape. Active Record is an object that wraps one row of a table, handles its own database access and carries the domain logic for that data; Django's design philosophies say its models follow that pattern. Data Mapper is a separate layer that moves data between objects and the database while keeping each independent of the other, so the objects need not know that a database exists. Smaller patterns sit around them. Identity Map makes sure each object is loaded only once. Unit of Work keeps a list of the objects a piece of work has changed and writes the changes out together. Lazy Load is an object that holds only part of the data code needs but knows how to fetch the rest. SQLAlchemy's documentation names two of these directly: its `Session` keeps loaded objects in an identity map, with only one object for any primary key, and its way of collecting changes and flushing them to the database is the unit of work pattern.

Use the concept in a decision

Put “What is an ORM?” to work in a real decision. We can help relate the concept to the system you are building or buying. The part users never see: where the data lives, what the rules are, and how other systems are allowed to talk to it.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Designed and ran 6 ETL/ELT pipelines — 6 ETL pipelines across three database engines

Designed and ran 6 ETL/ELT pipelines Get in touch →

Why it matters

The first reason is the amount of code it saves. In his 2012 essay ORM Hate, Fowler calls the object-relational mapping problem hard: it means keeping two quite different representations of the same data in step, and a change on either side has to be carried over to the other. His estimate is that an ORM handles about 80 to 90% of the mapping problems, and he argues that a framework which spares a team 80% of that work is worth having even if it stops there. Every table a team does not map by hand is one less place for a column to be read into the wrong attribute.

The second reason is consistency inside a running program. With an identity map, two queries that find the same row return the same object, so a change made through one reference is visible through the other; SQLAlchemy’s documentation demonstrates it by loading the same primary key twice and getting the identical object back. With a unit of work, the ORM collects pending inserts, updates and deletes and, in the words of SQLAlchemy’s feature list, flushes them all in one batch, so application code does not have to remember which objects it touched. Django makes the opposite choice on purpose: its design philosophies require an explicit `save()` rather than saving behind the scenes, so that the framework sends as few SQL statements as possible.

The third reason is where the boundary sits. Fowler describes a good ORM as one that takes care of the routine work and “provides manholes so you can get down with the SQL” when it has to, and he is blunt that anyone building an application on a relational database should know how one works. Django’s design philosophies say the same from the framework side: the database API should know it is a shortcut rather than the whole answer, and should make custom SQL easy to write, whether a whole statement or a single `WHERE` clause. An ORM earns its place on the routine majority of the data access; the rest is where a team spends its SQL skill.

What it looks like in practice

The cost an ORM hides most often is the number of queries it sends. In SQLAlchemy 2.1, relationships load lazily by default: the related objects are fetched the first time code touches the attribute. The documentation spells out the consequence: for N objects loaded, touching their lazily loaded attributes means N+1 `SELECT` statements. The example below assumes `Author` and `Book` classes already mapped, with a one-to-many `Author.books` relationship, and walks the authors three ways.

```
from sqlalchemy import select
from sqlalchemy.orm import raiseload, selectinload

# The default: one SELECT for the authors, then one more for each
# author whose books the loop touches, N+1 statements in all.
authors = session.scalars(select(Author)).all()
for author in authors:
    print(author.name, len(author.books))

# select IN loading: a second SELECT fetches the books of every
# author at once, with the authors' primary keys in an IN clause.
stmt = select(Author).options(selectinload(Author.books))
for author in session.scalars(stmt).all():
    print(author.name, len(author.books))

# raise loading: touching author.books now raises an error
# instead of quietly sending another query.
stmt = select(Author).options(raiseload(Author.books))
```

For one-to-many and many-to-many collections, the documentation calls `selectinload()` generally the best strategy to use. `joinedload()`, which it calls the most general-purpose strategy, adds a `JOIN` to the original statement, a `LEFT OUTER JOIN` by default, so the related rows arrive in the same result set. `raiseload()` exists to guard against unwanted loads, and its wildcard form, `raiseload("*")`, extends the guard to the other relationships too, including those of the objects the statement does load.

Django 6.1 gives the same choice a name of its own: fetch modes. When code touches a field that the original query did not load, such as a foreign key, a one-to-one relation or a field left out with `defer()` or `only()`, the default mode, `FETCH_ONE`, fetches it for that one instance, and the documentation itself names the result the N+1 queries problem. `FETCH_PEERS`, new in 6.1, fetches the missing field for the instance and for every other instance from the same `QuerySet`, on

the reasoning that code which needs a field for one row usually needs it for all of them. `FETCH_RAISE` raises `FieldFetchBlocked` instead. A mode is set per `QuerySet`, as in `Book.objects.fetch_mode(models.FETCH_PEERS)`, or as a model's default through a custom manager, and Django copies it to the related objects it fetches. Where `FETCH_PEERS` is not appropriate or efficient enough, the optimisation guide points to `select_related()` and `prefetch_related()`.

The company's own pages draw the boundary further towards the database than most. Its tech stack treats SQL as a first-class language rather than something an ORM generates, because most of the performance work it is called in for lives there. In work under NextMariner, the founder took that approach to its logical conclusion with a function-first PostgreSQL data layer: every operation the app can do is one of 1,275 stored functions spread across 34 schemas, and there is no direct table access from Go at all. The role the API logs in as holds `EXECUTE` on those functions and `USAGE` on their schemas and nothing else, so it cannot touch a table directly. The page is open about the price: a new field means changing a table as well as a function, not adding a line to a query.

Where it goes wrong

A loop sends one query per row. The N+1 pattern above looks harmless on a small development database and slows a page down in step with the number of rows. Decide the loading for every path that matters, with `selectinload()` or `joinedload()` in SQLAlchemy and `FETCH_PEERS`, `select_related()` or `prefetch_related()` in Django. Then make the unplanned case fail loudly: `raiseload()` in SQLAlchemy, or `FETCH_RAISE`, which Django's documentation describes as a way to prevent unintentional queries in performance-critical code.

Nobody reads the SQL the ORM writes. An ORM writes SQL that nobody on the team chose, and that SQL stays invisible until a page turns slow. Django's optimisation guide starts here: find out which queries a page runs and what they cost, use `QuerySet.explain()` to see how the database executes a given `QuerySet`, and read `django.db.connection.queries` to see what Django is writing on the code's behalf. Make reading that SQL part of reviewing any page or job that matters, and run the slow statements through the database's own query plan.

Work that belongs in the database runs in Python. Loading rows only to filter them, compare two of their fields or add them up moves every row across the connection to do in application code what the database could do where the data already is. Django's guide lists the alternatives: `filter()` and `exclude()` to filter

in the database, `F()` expressions to compare fields within the same model, and `annotate()` to aggregate in the database. Writes follow the same rule, with `bulk_create()` and `bulk_update()` in place of one `save()` per object, after checking the caveats the documentation lists for both. For a very large result set, a `QuerySet` holds on to everything it has read and can use a lot of memory, and `iterator()` may help.

The last part is forced through the ORM. Fowler's own estimate leaves 10 to 20% of the mapping problems, and that last part, in his words, always needs careful work. Bending the ORM until it produces a particular statement hides that work inside generated SQL that nobody chose. Write those statements in SQL and review them like any other code. Django's guide names the steps down: the `RawSQL` expression adds explicit SQL to a query, less portable but more powerful, and beyond that a team can write its own SQL to retrieve data or populate models. Keep the ORM for the routine records, where it saves the most and costs the least.

Further reading

- Catalog of Patterns of Enterprise Application Architecture — Martin Fowler's summaries of Active Record, Data Mapper, Unit of Work, Identity Map and Lazy Load.
- ORM Hate — why the object-relational mapping problem is hard, and how much of it a framework can take on.
- Relationship Loading Techniques — lazy, select IN, joined and raise loading in SQLAlchemy 2.1, and which one to choose.
- Fetch modes — `FETCH_ONE`, `FETCH_PEERS` and `FETCH_RAISE`, new in Django 6.1.
- Database access optimization — profiling, `QuerySet` evaluation, work done in the database, bulk methods and raw SQL.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is an ORM?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/orm/