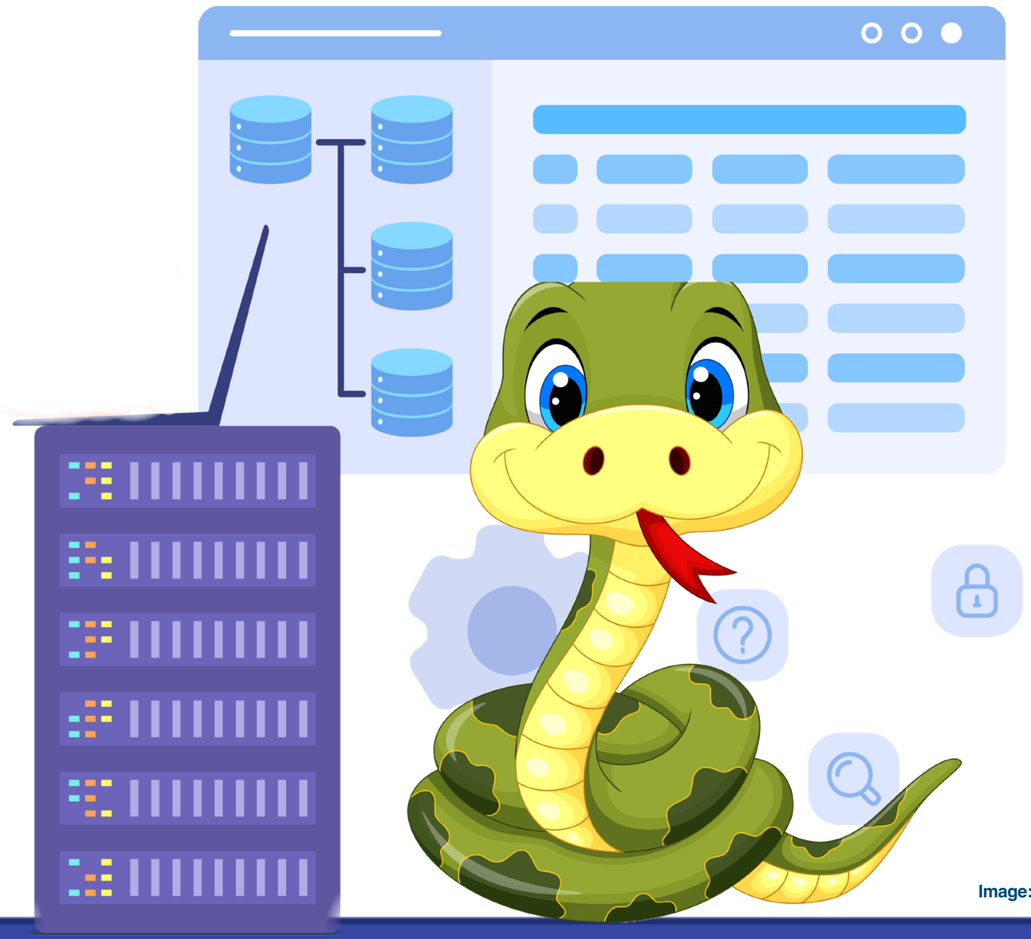
(A) SQL for Django



Stefan Baerisch, stbaer.com,
Virtual DjangoCon 2-6 June 2021

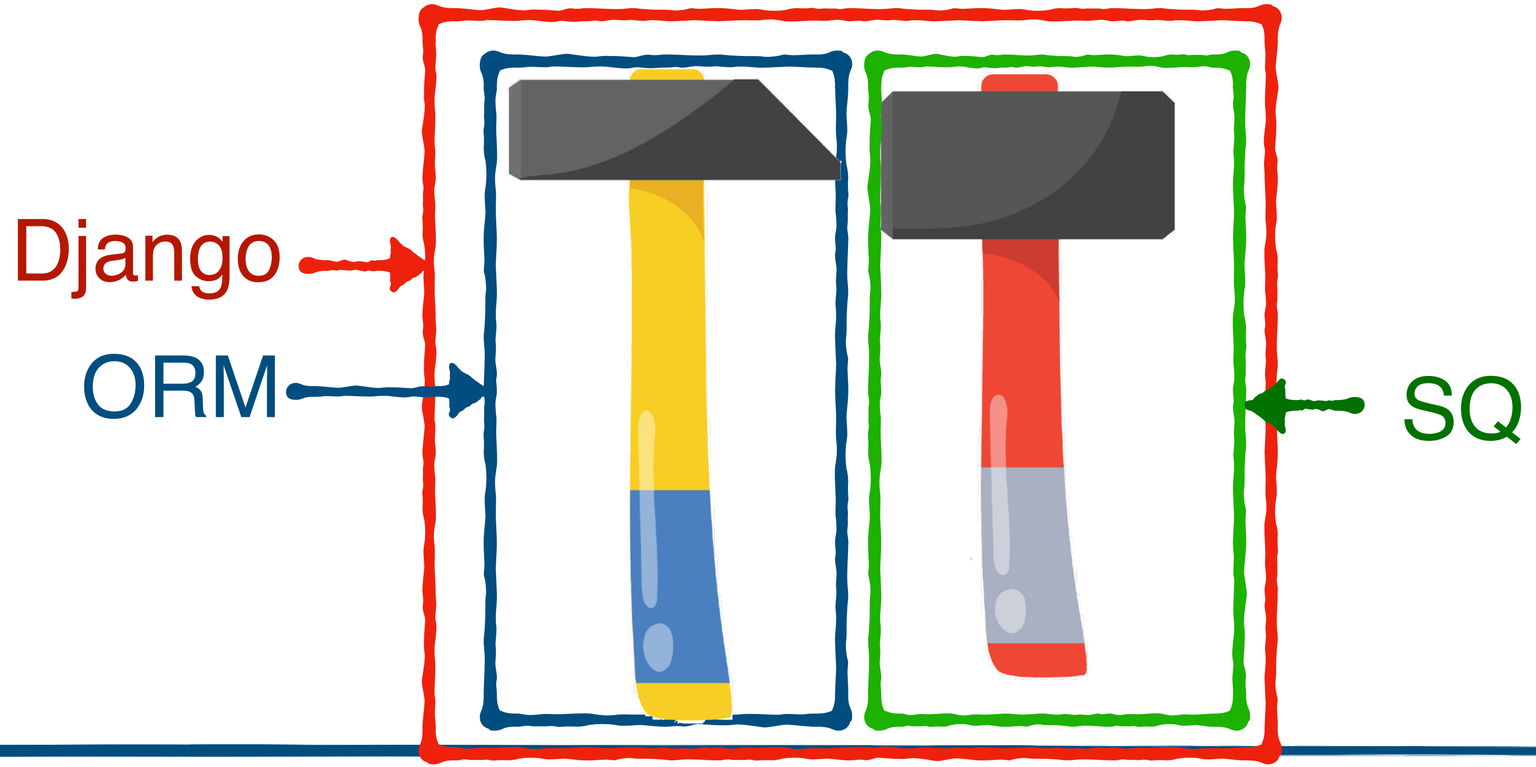
Image: Freepic Professional

Some background

- Using Python since ~ 2006
  - also Go, Rust, Java
- Django since ~ 2017
- PM / Business Analyst, Developer
  - Django is not my focus

Motivation for the talk

- You **can do** (almost) **everything** you want to do with a database in Django's ORM
- You **don't want to** do (almost) everything you can in Django's ORM
- Using **SQL with Django** is possible and has benefits

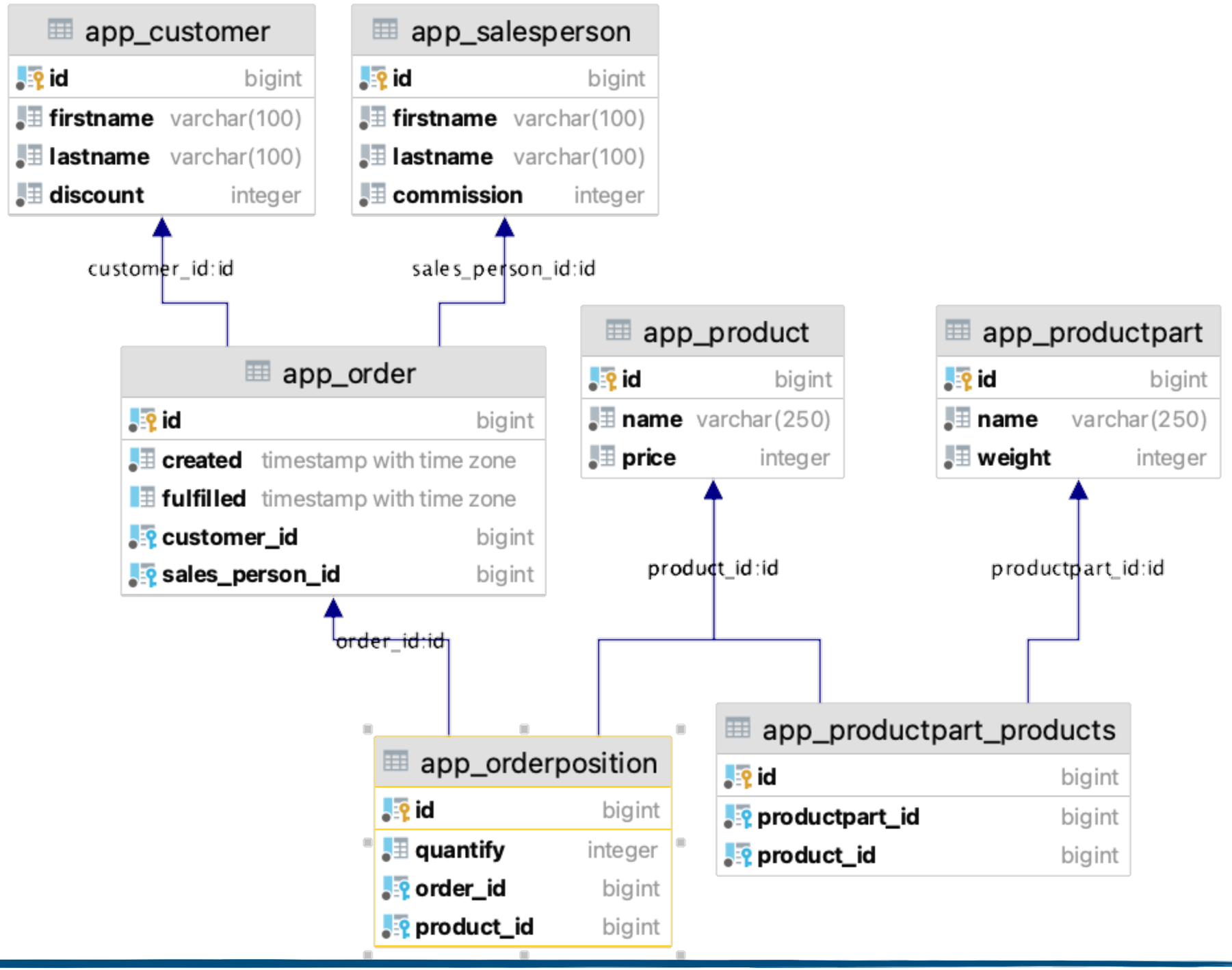


ORM and SQL, again

Scenario	ORM	SQL
CRUD	Great Code / Data Integration	Boilerplate per Object
Gathering Object Hierarchies	Some care and checks required	Still mapping efforts, may be worth it
Analytic Queries	SQL-to-ORM thinking required	SQL thinking required

Working with the Django ORM

Example Database



Working with data - Django ORM use cases

Scenario	Way to do it
CRUD	create(), get()/filter() / delete()
Gathering Object Hierarchies	Attribute Access via Forein Keys() select_related()
Analytic Queries	annotate() / aggregate() /Q / F ...

CRUD Operations

```
cust = m.Customer(firstname="Ex", lastname="Ample", discount=10)
cust.save()

cust = m.Customer.objects.filter(firstname='Ex').first()
cust.discount += 1
cust.save()

cust.delete()
```

Getting Specific Filters

```
r = m.Customer.objects. \
    filter(discount__gt=2, discount__lt=4). \
    values('lastname'). \
    order_by('discount')
q = r.query
```

```
'SELECT "app_customer"."lastname"
FROM "app_customer"
WHERE ("app_customer"."discount" > 2
AND "app_customer"."discount" < 4)
ORDER BY "app_customer"."discount" ASC'
```

The Q and F of complex (1/2)

```
r = m.Customer.objects. \
    filter(discount= F('discount') * F('discount'))
q = r.query
```

```
'SELECT "app_customer"."id",
"app_customer"."firstname",
"app_customer"."lastname",
"app_customer"."discount"
FROM "app_customer"
WHERE "app_customer"."discount" =
("app_customer"."discount" *
"app_customer"."discount")'
```

The Q and F of complex (2/2)

```
r = m.Customer.objects.filter(
    Q(discount__lt=7)  | Q(discount__gt=12)
)
```

```
'SELECT "app_customer"."id",
"app_customer"."firstname",
"app_customer"."lastname", "app_customer"."discount"
FROM "app_customer" WHERE
("app_customer"."discount" < 7 OR
"app_customer"."discount" > 12)'
```

Using Annotations

```
r = m.Customer.objects. \
    filter(id__in=[1,3,6,10,45,12]). \
    annotate(doubled=F('discount') * 2)
q = r.query
q
```

```
'SELECT "app_customer"."id", "app_customer"."firstname",
"app_customer"."lastname", "app_customer"."discount",
("app_customer"."discount" * 2) AS "doubled" FROM "app_customer" WHERE
"app_customer"."id" IN (1, 3, 6, 10, 45, 12)'
```

Using Annotations with Joins

```
r = m.Customer.objects. \
    values('firstname', 'lastname'). \
    annotate(
    Count('orders'),
    Sum('orders__positions__product__price')
)
q = r.query
q
```

```
'SELECT "app_customer"."firstname", "app_customer"."lastname", COUNT("app_order"."id") AS "orders__count",
SUM("app_product"."price") AS "orders__positions__product__price__sum" FROM "app_customer" LEFT OUTER JOIN
"app_order" ON ("app_customer"."id" = "app_order"."customer_id") LEFT OUTER JOIN "app_orderposition" ON
("app_order"."id" = "app_orderposition"."order_id") LEFT OUTER JOIN "app_product" ON
("app_orderposition"."product_id" = "app_product"."id") GROUP BY "app_customer"."firstname",
"app_customer"."lastname"'
```

Aggregations

```
reset_queries()
r = m.Customer.objects. \
    filter(id__in=[1,3,6,10,45,12]).\
    aggregate(avg= Avg('discount'), max= Max('discount'))
q = connection.queries[0]['sql']
q
```

```
'SELECT AVG("app_customer"."discount") AS "avg",
MAX("app_customer"."discount") AS "max" FROM "app_customer" WHERE
"app_customer"."id" IN (1, 3, 6, 10, 45, 12)'
```

A Complex Example

```
r = m.Customer.objects. \
    values('lastname', 'discount'). \
    annotate(
    s_lastname=F('orders__sales_person__lastname'),
    s_commission=F('orders__sales_person__commission'),
    total=F('orders__customer__discount') + F('orders__sales_person__commission')
).filter(
    (Q(orders__fulfilled__range=('2019-09-01', '2019-12-31')) & Q(total__gt=15)) |
    (Q(orders__fulfilled__range=('2018-01-01', '2018-12-31')) & Q(total__gt=10))
)
q = r.query
q
```

```
'SELECT "app_customer"."lastname", "app_customer"."discount", "app_salesperson"."lastname" AS "s_lastname",
"app_salesperson"."commission" AS "s_commission", (T4."discount" + "app_salesperson"."commission") AS "total" FROM
"app_customer" LEFT OUTER JOIN "app_order" ON ("app_customer"."id" = "app_order"."customer_id") LEFT OUTER JOIN
"app_salesperson" ON ("app_order"."sales_person_id" = "app_salesperson"."id") LEFT OUTER JOIN "app_customer" T4 ON
("app_order"."customer_id" = T4."id") INNER JOIN "app_order" T5 ON ("app_customer"."id" = T5."customer_id") WHERE
((T5."fulfilled" BETWEEN 2019-09-01 00:00:00 AND 2019-12-31 00:00:00 AND (T4."discount" + "app_salesperson"."commission")
> 15) OR (T5."fulfilled" BETWEEN 2018-01-01 00:00:00 AND 2018-12-31 00:00:00 AND (T4."discount" +
"app_salesperson"."commission") > 10))'
```

Creating the N+1 query problem

```
reset_queries()
lines = []
orders = m.Order.objects.filter(created__range=('2019-09-01', '2019-12-31'))
for order in orders:
    sp = order.sales_person
    cu = order.customer
    lines.append(f"{sp.lastname} ({sp.commission}) / {cu.lastname} {cu.discount} ")
quer = connection.queries
qs = connection.queries
```

```
len(qs):2997
```

{'sql': 'SELECT "app_salesperson"."id", "app_salesperson"."firstname", "app_salesperson"."lastname", "app_salesperson"."commission" FROM "app_salesperson" WHERE "app_salesperson"."id" = 269 LIMIT 21', 'time': '0.000'}, {'sql': 'SELECT "app_customer"."id", "app_customer"."firstname", "app_customer"."lastname", "app_customer"."discount" FROM "app_customer" WHERE "app_customer"."id" = 19 LIMIT 21', 'time': '0.000'}

Addressing the N+1 query problem

```
reset_queries()
lines = []
orders = m.Order.objects.select_related('sales_person','customer').\
    filter(created__range=('2019-09-01', '2019-12-31'))
for order in orders:
    sp = order.sales_person
    cu = order.customer
    lines.append(f"{sp.lastname}  ({sp.commission}) / {cu.lastname} {cu.discount} ")
quer = connection.queries
qs = connection.queries
```

```
[{'sql': 'SELECT "app_order"."id", "app_order"."created", "app_order"."fulfilled",
"app_order"."sales_person_id", "app_order"."customer_id", "app_salesperson"."id",
"app_salesperson"."firstname", "app_salesperson"."lastname", "app_salesperson"."commission",
"app_customer"."id", "app_customer"."firstname", "app_customer"."lastname",
"app_customer"."discount" FROM "app_order" INNER JOIN "app_salesperson" ON
("app_order"."sales_person_id" = "app_salesperson"."id") INNER JOIN "app_customer" ON
("app_order"."customer_id" = "app_customer"."id") WHERE "app_order"."created" BETWEEN
\'2019-09-01 00:00:00\' AND \'2019-12-31 00:00:00\', 'time': '0.001'}]
```

SQL Use Cases and Advantages

Everything works, so why use SQL?

- Django's ORM gives us everything we need
  - CRUD operations
  - Aggregations and Analytics
  - Optimizations (getting only some fields, specify dependent data)
- So why use SQL at all?
- Let's look at some potential advantages

Addressing the N+1 query problem with SQL

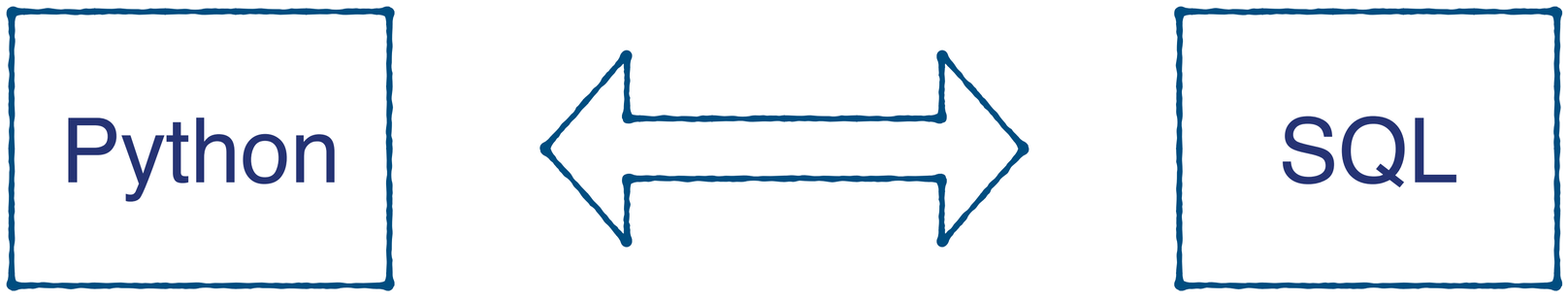
```
from django.db import connection
reset_queries()
lines = []

sql = """
select sp.lastname, sp.commission,cu.lastname,cu.discount
from app_order o
inner join app_customer cu on cu.id = o.customer_id
inner join app_salesperson sp on sp.id = o.sales_person_id
where o.created between  '2019-09-01' AND  '2019-12-31';
"""

with connection.cursor() as cursor:
    cursor.execute(sql)
    for row in cursor.fetchall():
        a = 1
        lines.append(f"{row[0]}  ({row[1]}) / {row[2]} {row[3]} ")
qs  = connection.queries
```

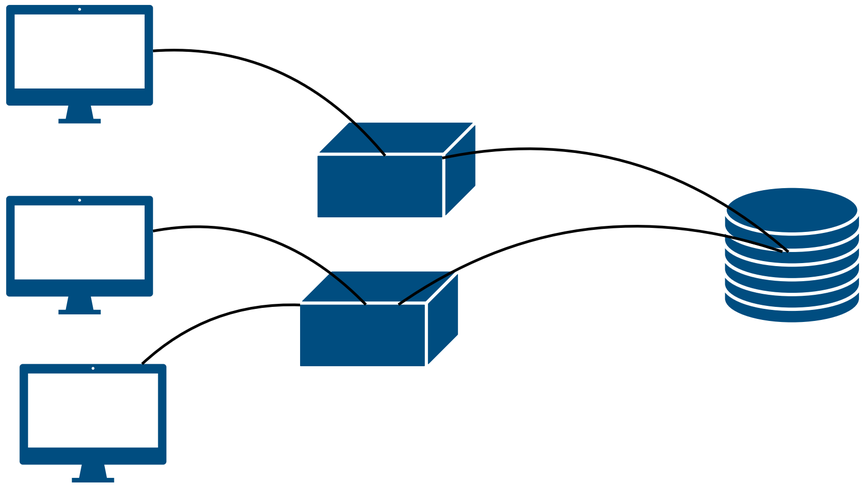
```
[{'sql': "\n        select sp.lastname, sp.commission,cu.lastname,cu.discount\n        from app_order o\n
inner join app_customer cu on cu.id = o.customer_id\n        inner join app_salesperson sp on sp.id =
o.sales_person_id\n        where o.created between  '2019-09-01' AND  '2019-12-31';\n        ", 'time': '0.000'}]
```

Separation of Concerns

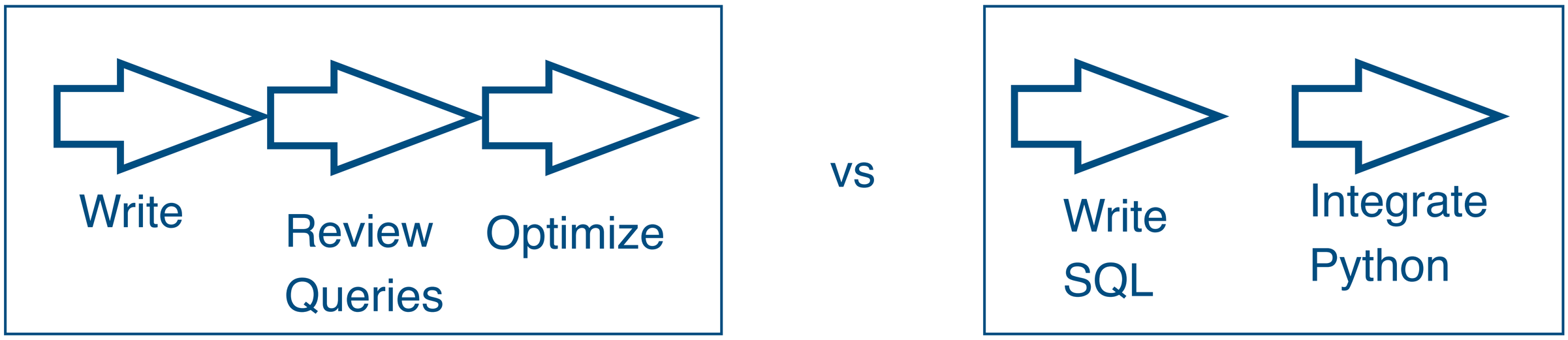


- Consider your Database an external service
- Returned objects define the interface
- A Python wrapper and SQL are the implementation

Easing Performance Analysis and Optimization



- Database scaling (still) matters



Readability

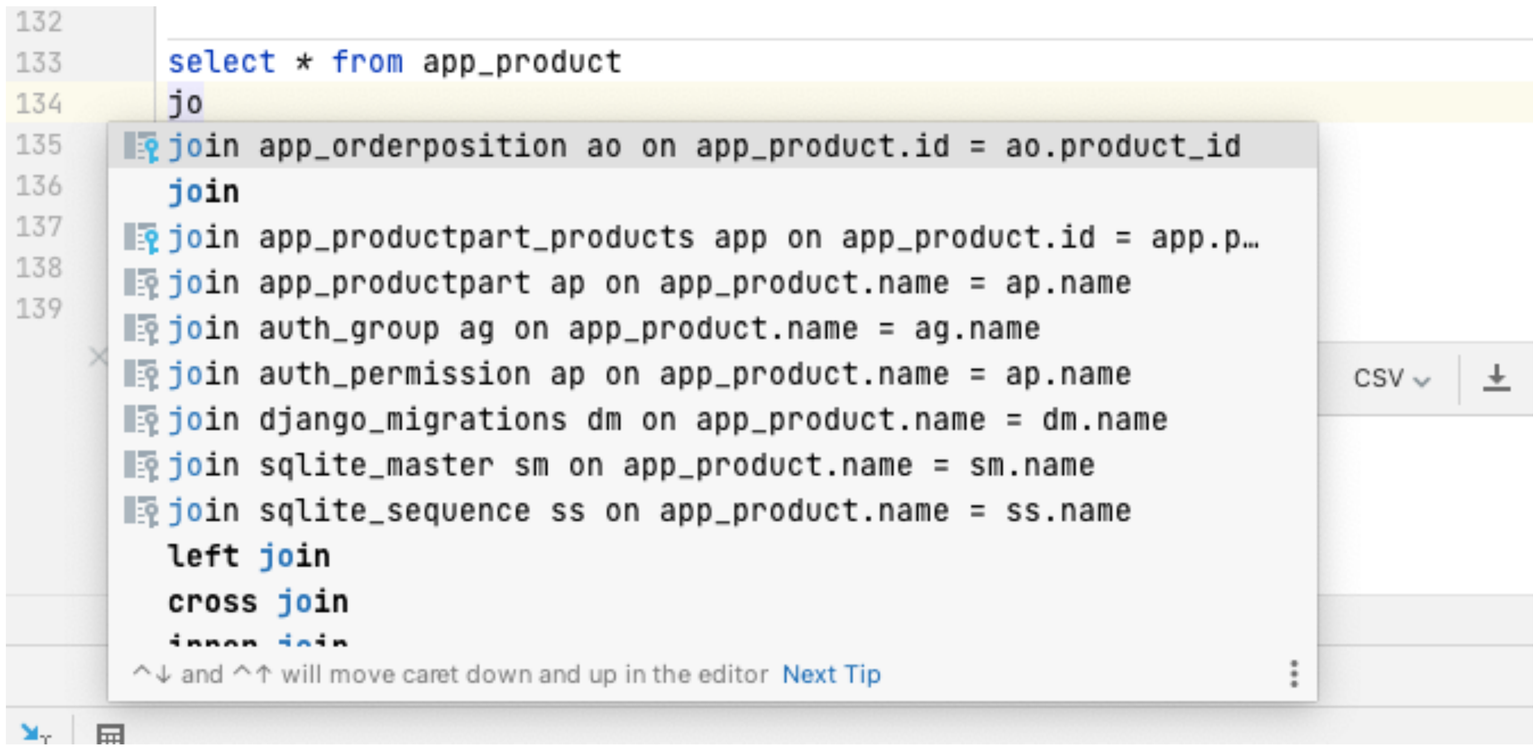
- SQL can be verbose...
- ... but as a declarative language, it is not hard to read...
- ...and by structuring your queries, you can make it even more readable

```
with priced_orders as (
    select o.id as id, sum(ap.price) as sum
    from app_order o
    join app_orderposition od on o.id = od.order_id
    join app_product ap on ap.id = od.product_id
    group by ap.id
)

select sp.lastname,sum(sp.commission * po.sum / 100) as com
from app_order o
join  priced_orders po on po.id = o.id
join app_salesperson sp on sp.id = o.sales_person_id
group by o.sales_person_id
order by 2 desc;
```

Writing Code

- Do you like IDEs? Code Completion? Supported Refactorings?
- IDEs have an easier time understanding your database then your Django model



- Also, if want to have **exactly this SQL,** writing is simpler than tuning

Commonality

- Your non-Django team members and users may understand SQL better than Django's ORM.
- Business Analyst may provide you with queries they want in their dashboard
- And the JAVA team two offices over will understand what you do

Finding Information

- Django is well documented...
- ... but it is only one of many ORMs...
- and there is still more googleable knowledge about SQL

Django

SQL

Combining SQL and Django

Best of both worlds: Getting Objects with Raw Queries

```
sql = "select * from app_customer where id = 102;"
raw_query_set = m.Customer.objects.raw(sql)
customer = raw_query_set[0]
customer.lastname
```

Getting Objects and Renaming Fields

```
sql = """
select
    1 as id,
    'hello' as firstname,
    'world' as lastname,
    10 as discount;
"""

raw_query_set = m.Customer.objects.raw(sql)
customer = raw_query_set[0]
a = customer
```

Getting Partial Objects

```
sql = """
select id, firstname from app_customer
where discount > 8;
"""
raw_query_set = m.Customer.objects.raw(sql)
customer = raw_query_set[0]
ln = customer.lastname
```

Raw SQL and Parameters

```
sql = """
select id, firstname from app_customer
where discount > %s
order by discount;
"""
raw_query_set = m.Customer.objects.raw(sql, [8])
customer = raw_query_set[0]
ln = customer.lastname
```

Some Caveat...

```
sql = """
select id, firstname from app_customer
where discount > %s
order by discount
limit 1;
"""

raw_query_set = m.Customer.objects.raw(sql, [8])
customer = raw_query_set[0]
ln = customer.lastname
```

Raw SQL in other Places

```
rsql = RawSQL(
    """
    select sum(a.quantify * ap.price) from app_customer c
    left join app_order ao on c.id = ao.customer_id
    left join app_orderposition a on ao.id = a.order_id
    left join app_product ap on ap.id = a.product_id
    where c.discount > 9
    group by c.id
    order by c.discount
    """,[]
)

r = m.Customer.objects.filter(discount__gt=9).order_by('discount')
r2 = r.annotate(tot = rsql)
```

Look, No Objects: Using Django's Database Connections

```
from django.db import connection
sql = "select * from app_customer where id = 102;"
with connection.cursor() as cursor:
    cursor.execute(sql)
    row = cursor.fetchone()
```

Bypassing Django - Why and How

```
import sqlite3
connection = sqlite3.connect(DBPATH)
cursor = connection.cursor()
cursor.executescript("""
begin;
insert into app_customer (firstname, lastname, discount)
values ('Ex', 'Ample', 10);
insert into app_customer (firstname, lastname, discount)
values ('John', 'Doe', 14);
commit;
""")
connection.close()
```

Drawback of SQL in Django

Drawback : Boilerplate Code

- An ORM may be inefficient at runtime
- No ORM may be inefficient at write time
- Without an ORM, you will have to prepare the data you pass into you views...

Drawback : Loss of Abstraction

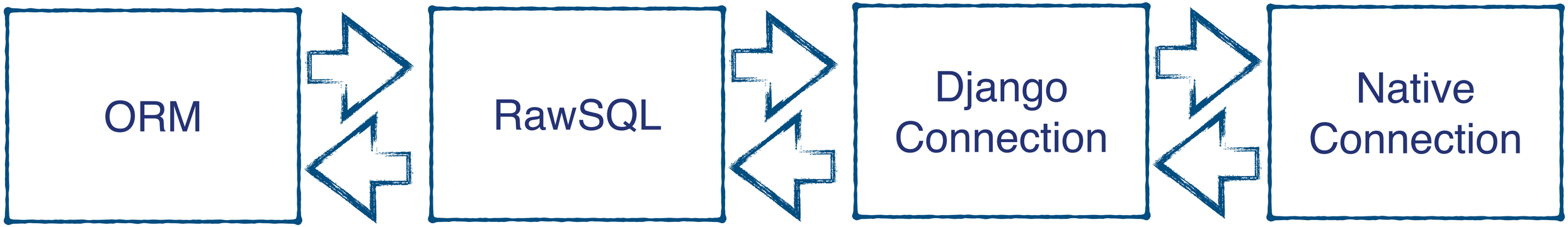
- Django helps to abstract from your database
  - If you go for SQL, you will need to think about your DBMS's SQL dialect

Loss of Features

- Saying 'No' to Django's ORM means we lose features
  - Signals
  - Migrations
  - Admin?
  - ...

Options

DDM



DDL

Review: SQL, Django - How and Why

- Use Django's **ORM for Models** and simple CRUD Operations
- If you **want objects** and filters and annotate get to **complicated**, give raw SQL a try.
- If your **organization already has the queries** you need, don't reinvent the wheel
- If you don't want objects, directly use the **Django connection**
- If you need different connection parameters, go for a native connection

Thank You