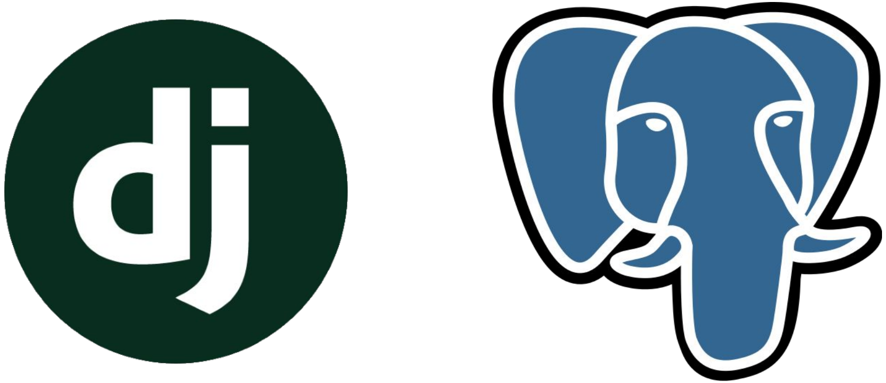
Django ORM:
a fight between MTI and STI



About

- Have been developing Python projects for the past 10 years
- Most recent projects are fintech startups
- Development Lead in QIWI Всё проще

About

- Have been developing Python projects for the past 10 years
- Most recent projects are fint
- Development Lead in QIWI Всё проще

A lot of code ahead!

Domain layer

Financial Account

Domain layer

Financial Account

- Investment Account
- Credit Card Account
- Loan Account

Python: simple class inheritance

Python: simple class inheritance

```
class FinancialAccount:
    name = ...
    member = ...
    balance = ...

class CreditCardAccount(FinancialAccount):
    due_date = ...
    available_credit = ...

class LoanAccount(FinancialAccount):
    interest_rate = ...
    recurring_payment = ...
```

Relational DB: built-in inheritance

PostgreSQL built-in inheritance is available since version 7

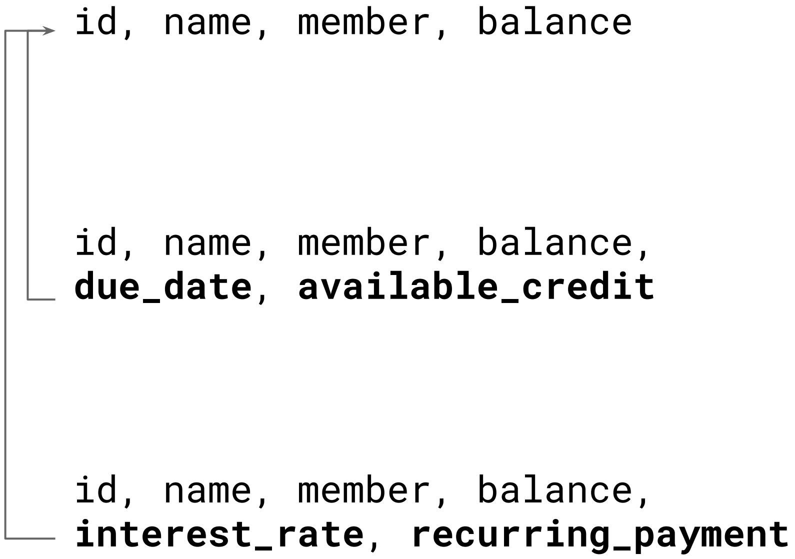
```
create table financial_account (id, name, balance, member_id);

create table credit_card_account (due_date, available_credit)
INHERITS (financial_account);

create table loan_account (interest_rate, recurring_payment)
INHERITS (financial_account);
```

One SQL query can fetch all common fields + different one

```
select * from financial_account;
```

```
select * from credit_card_account;
```

```
select * from loan_account;
```



```
id, name, member, balance
```

```
id, name, member, balance,
due_date, available_credit
```

```
id, name, member, balance,
interest_rate, recurring_payment
```

Under the hood

```
explain select * from financial_account;

                    QUERY PLAN
--------------------------------------------------
Append  (...)
  ->  Seq Scan on financial_account  (...)
  ->  Seq Scan on credit_card_account  (...)
  ->  Seq Scan on loan_account  (...)
```

Let’s select all accounts data

```
select t1.*,
      t2.interest_rate, t2.recurring_payment,
      t3.available_credit, t3.due_date
from financial_account as t1
left join loan_account as t2
   on t1.id = t2.id
left join credit_card_account as t3
   on t1.id = t3.id
where t1.member_id = X;
```

Not so easy, right?

```
select t1.*,
      t2.interest_rate, t2.recurring_payment,
      t3.available_credit, t3.due_date
from financial_account as t1
left join loan_account as t2
   on t1.id = t2.id
left join credit_card_account as t3
   on t1.id = t3.id
where t1.member_id = X;
```

Table identification is missed in the response!

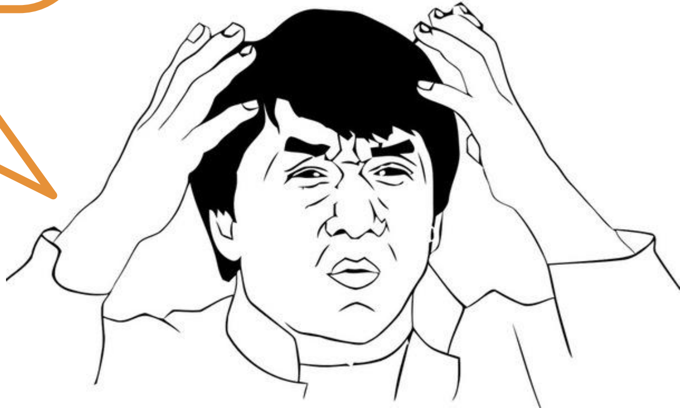
What if... UNIQUE CONSTRAINT!

```
alter table financial_account add constraint account_name_unique UNIQUE (name);


insert into loan_account (name, ...) values ('Account 1', ...); -- OK
insert into loan_account (name, ...) values ('Account 1', ...); -- OK
insert into credit_card_account (name, ...) values ('Account 1', ...); -- OK
insert into credit_card_account (name, ...) values ('Account 1', ...); -- OK
```

What if... UNIQUE CONSTRAINT!

```
alter table financial_account add constraint account_name_unique UNIQUE (name);


insert into l                                                   , ...); -- OK
insert into l                                                   , ...); -- OK
insert into c                                                   ount 1', ...); -- OK
insert into credit_card_account (name,
```

Not so obvious, but UNIQUE CONSTRAINTS are not inherited



Build-in inheritance: keep in mind

- 😕 UNIQUE CONSTRAINTS and REFERENCES are not inherited
- 😑 ALTER TABLE will surprise you for sure
- 😫 You still have to do JOIN’s to gather all accounts data
- 😵 Django team refused to add built-in inheritance support because of this mess, see https://code.djangoproject.com/ticket/24632

Build-in inheritance: when?

- You are a DBA
- You deal with partitioning (see p. 1)
- You hate ORMs (see p.1)
- You are fully aware of what are you doing (see p.1)

Django ORM: emulation of inheritance

- Abstract Base Classes
- Multi-table Inheritance
- Single Table Inheritance

Abstract Base Classes

```
class FinancialAccount(models.Model):
   name = ...
   member = ...
   balance = ...

   class Meta:
       abstract = True

class CreditCardAccount(FinancialAccount):
   due_date = ...
   available_credit = ...

class LoanAccount(FinancialAccount):
   interest_rate = ...
   recurring_payment = ...
```

Abstract Base Classes

```
class FinancialAccount(models.Model):
   name = ...
   member = ...
   balance = ...

   class Meta:
       abstract = True

class CreditCardAccount(FinancialAccount):
   due_date = ...
   available_credit = ...

class LoanAccount(FinancialAccount):
   interest_rate = ...
   recurring_payment = ...
```

Abstract Base Classes: reality

```
class CreditCardAccount(models.Model):
   name = ...
   member = ...
   balance = ...
   due_date = ...
   available_credit = ...
```

Table 1:

`id, name, member, balance,`
`due_date, available_credit`

```
class LoanAccount(models.Model):
   name = ...
   member = ...
   balance = ...
   interest_rate = ...
   recurring_payment = ...
```

Table 2:

`id, name, member, balance,`
`interest_rate, recurring_payment`

QIWI
Всё проще

Abstract Base Classes

- The inheritance does exists at code level only
- Data is stored in separate tables

Let’s try to fetch all data at once

Expectation

```
>>> FinancialAccount.objects.all()
<QuerySet [<CreditCardAccount: 1>, <LoanAccount: 2>]>
```

Let’s try to fetch all data at once

Expectation

```
>>> FinancialAccount.objects.all()
<QuerySet [<CreditCardAccount: 1>, <LoanAccount: 2>]>
```

Reality

```
AttributeError: type object 'FinancialAccount' has no
attribute 'objects'
```

Abstract Base Classes

- ❗ Tables are not connected, so you have to do **N SQL** queries to fetch all accounts for a particular member and then perform merge operation in application’s code
- ❗ There are no CONSTRAINTS for common fields (like “name”)
- 👌 Simple to add new fields and make migrations
- 👌 Parent class can be easily reused

ABC: summary

- ❗ Tables are not connected, so you have to do **N SQL** queries to fetch all accounts for a particular member and then perform merge operation in application’s code
- ❗ There are no CONSTRA lds (like “name”)
- 👌 Simple to add new fields and make migrations
- 👌 Parent class can be easily reused

You still can join tables by member id!

But ORM does not help.

Abstract Base Classes: when?

- Mixins (PermissionsMixin for example)
- Some external requirements force you to store each domain class data into a separate table: access permissions, complex replication or partitioning, specific highload profile
- You develop a framework or a package
- You consider JOINs too slow

Multi-table Inheritance

Common fields are stored in one table, different fields in child tables.

MTI: under the hood

- Simple model inheritance (technically – **OneToOneField + select_related**)
- Explicit **OneToOneField** usage
- Generic Relation / Polymorphic Associations via **ContentType** framework 😰

MTI: simple model inheritance

```
class FinancialAccount(models.Model):
    name = ...
    member = ...
    balance = ...

class CreditCardAccount(FinancialAccount):
    due_date = ...
    available_credit = ...

class LoanAccount(FinancialAccount):
    interest_rate = ...
    recurring_payment = ...
```

We have a connection between tables

```python
class FinancialAccount(models.Model):
    name = ...
    member = ...
    balance = ...

class CreditCardAccount(FinancialAccount):
    due_date = ...
    available_credit = ...

class LoanAccount(FinancialAccount):
    interest_rate = ...
    recurring_payment = ...
```

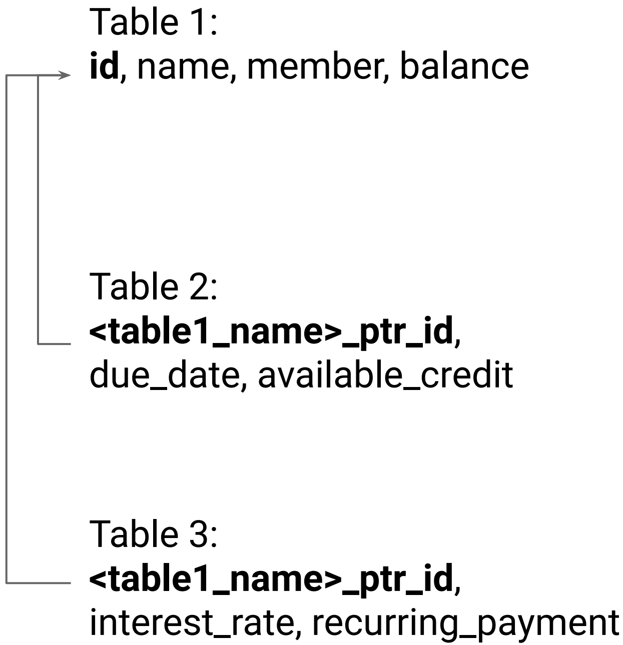


Table 1:
id, name, member, balance

Table 2:
<table1_name>_ptr_id,
due_date, available_credit

Table 3:
<table1_name>_ptr_id,
interest_rate, recurring_payment

ORM doesn’t fetch related data

Expectation

```
>>> FinancialAccount.objects.all()
<QuerySet [<CreditCardAccount: 1>, <LoanAccount: 2>]>
```

ORM doesn’t fetch related data

Expectation

```
>>> FinancialAccount.objects.all()
<QuerySet [<CreditCardAccount: 1>, <LoanAccount: 2>]>
```

Reality

```
>>> FinancialAccount.objects.all()
<QuerySet [<FinancialAccount: 1>, <FinancialAccount: 2>]>
```

Any SQL query to child table lead to INNER JOIN with parent-table

```
>>> CreditCardAccount.objects.all()
<QuerySet [<CreaditCardAccount: 1>, ...]>
```

```
SELECT *
FROM "credit_card_account"
INNER JOIN "financial_account"
ON (...)
```

Any SQL query to child table lead to INNER JOIN with parent-table

```
>>> CreditCardAccount.objects.all()
<QuerySet [<CreaditCardAccount: 1>, ...]>
```

```
SELECT *
FROM "credit_card_account"
INNER JOIN "financial_account"
ON (...)
```

You can solve this with **only**, **defer**, **values** or explicit **OneToOneField**

Django-polymorphic

```
from polymorphic.models import PolymorphicModel


class FinancialAccount(PolymorphicModel):
    …
```

Profit?

```
>>> FinancialAccount.objects.all()
<QuerySet [<CreditCardAccount: 1>, <LoanAccount: 2>, ...]>
```

Django-polymorphic

```
from polymorphic.models import PolymorphicModel


class FinancialAccount(PolymorphicModel):
    …
```

Profit?

```
>>> FinancialAccount.objects.all()
<QuerySet [<CreditCardAccount: 1>, <LoanAccount: 2>, ...
```

3 SQL queries and 2 JOINs included

Django-polymorphic

- ❌ Executes **K+1** SQL-queries with **1** INNER JOIN
- ❗ Adds new model field (ContentType)
- ❗ Requires migration for existing DB tables
- 👌 Good Django-admin integration
- 👌 Eye-candy ORM-based query syntax

django-model-utils.InheritanceManager

```python
from model_utils.managers import InheritanceManager

class FinancialAccount(Model):
    objects = InheritanceManager()
```

Profit!

```
>>> FinancialAccount.objects.select_subclasses()
<QuerySet [<CreditCardAccount: 1>, <LoanAccount: 2>, ...]>
```

django-model-utils.InheritanceManager

```
SELECT ...
FROM "financial_account"
LEFT OUTER JOIN "credit_card_account" ON (...)
LEFT OUTER JOIN "loan_account" ON (
"financial_account"."id" =
"loan_account"."financialaccount_ptr_id")
```

django-model-utils.InheritanceManager

```
>>> FinancialAccount.objects.select_subclasses().filter(
   Q(loanaccount__interest_rate__gt=1) |
   Q(creditcardaccount__available_credit__lte=100)
)
```

django-model-utils.InheritanceManager

- 👌 Plug-in-play and easy to use
- ☯️ Generic Django-ORM syntax
- ❗ Executes **ONLY ONE** SQL-query to gather all the necessary data via LEFT OUTER JOIN

MTI: summary

- 👌 Data is normalized
- ❗ Possible SQL queries overhead
- ❗ More complex coding required if you need to deal with all children in one context (e.g. sorting and merging)
- ❗ New child – new table

MTI: when?

- Few child tables
- Nested inheritance
- Supported by ORM out of the box
- Proven-by-the-time solution

Single Table Inheritance

- All data is stored in one table, data is denormalized
- Child objects logic is handled on a code level
- Django-ORM does not support STI out of the box, even via proxy-models

Classic way: django-typed-models

```
class FinancialAccount(TypedModel):
    ...
    type = models.CharField(db_index=True)

class CreditCardAccount(FinancialAccount):
    due_date = models.DateField(null=True)
    available_credit = models.DecimalField(..., null=True)

class LoanAccount(FinancialAccount):
    interest_rate = models.DecimalField(..., null=True)
    recurring_payment = models.DecimalField(..., null=True)
```

Classic way: django-typed-models

```
class FinancialAccount(TypedModel):
```

```
Single table:

id, name, member, balance,
type,
due_date (NULL), available_credit (NULL),
interest_rate (NULL), recurring_payment (NULL)
```

```
cl
                                                      ue)
cl
    interest_rate = models.DecimalField(..., null=True)
    recurring_payment = models.DecimalField(..., null=True)
```

Classic: django-typed-models

- 👌 **1 SQL** to fetch all the data
- ❗ All fields in child tables – nullable
- ❗ The more child tables, the more nullable columns in the main table
- ❗ Low cardinality index (**type** field)
- ❗ High coupling between classes (one table underhood)

Semi-structured: JSON Field

```
class AccountType(IntEnum):
    credit_card = auto()
    loan = auto()


class FinancialAccount(models.Model):
    name = ...
    member = ...
    balance = ...

    type = models.SmallIntegerField(choices=[(...) for ... in AccountType])
    data = JSONField()
```

Semi-structured: JSON Field

- 👌 Just one SQL query to perform sorting and selection
- ☯️ ORM to describe relations and DB schema, but not the same for JSON
- ❓ Support and performance?

JSON: state of support in Postgres

JSOBb → JSQuery → SQL:2016 → JSONPath (12)

SQL standard provides additional index operators and functions to effectively work with JSONb fields:

https://habr.com/ru/company/postgrespro/blog/448612/

JSONField: problems and solutions

Problem: high coupling code

```
FinancialAccount.objects.filter(
    type=AccountType.credit_card,
    member=user,
    data__balance__gt=0
).select_related('member').order_by('-created')
```

JSONField: problems and solutions

Solution: move logic to Django-managers

```
FinancialAccount.objects.filter(
    typ
    mem
    dat
).sele
```

```
FinancialAccount
 .credit_cards
 .for_member(user)
 .with_positive_balance()
```

JSONField: problems and solutions

Solution: get highly reusable code

```
FinancialAccount.objects.filter(
    typ
    mem
    dat
).sele
```

```
FinancialAccount
 .credit_cards
 .for_member(user)
 .active()
 .with_positive_balance()
```

JSONField: problems and solutions

Solution: get highly reusable code

```
FinancialAccount.objects.filter(
    typ
    mem
    dat
).sele
```

```
FinancialAccount
    .credit_cards
    .for_member(user
    .active()
    .with_positive_balance()
```

This approach can be used anywhere!

JSONField: problems and solutions

Problem: save/update method causes sending all the contents of the JSON field to the database

```
>>> account.data['interest_rate'] = 102
# UPDATE query contains all new data content
>>> account.save(update_fields=('account',))
```

JSONField: problems and solutions

Solution: django-postgres-extensions and PG function **jsonb_set**

django-postgres-extensions

```
from psycopg2.extras import Json
from django_postgres_extensions.models.functions import JSONBSet

account = FinancialAccount.objects.get(id=...)

FinancialAccount.objects.filter(id=account.id).update(
   data=JSONBSet('data', ['recurring_payment'], Json(2000))
)
```

JSONField: problems and solutions

Problem: there is no schema description and validation for JSONField – it’s really annoying and complicates development

JSONField: problems and solutions

Solution: pydantic + JSONSchemedField

pydantic + JSONSchemedField

```
class CreditCardData(pydantic.BaseModel):
   due_date: datetime.datetime
   available_credit: decimal.Decimal

class LoanData(pydantic.BaseModel):
   interest_rate: decimal.Decimal
   recurring_payment: decimal.Decimal
```

Pydantic schemes + Union

```
class CreditCardData(pydantic.BaseModel):
   due_date: datetime.datetime
   available_credit: decimal.Decimal

class LoanData(pydantic.BaseModel):
   interest_rate: decimal.Decimal
   recurring_payment: decimal.Decimal

AccountData = Union[CreditCardData, LoanData]

class FinancialAccount(models.Model):
    ...
    data: AccountData = JSONSchemedField(schema=AccountData)
```

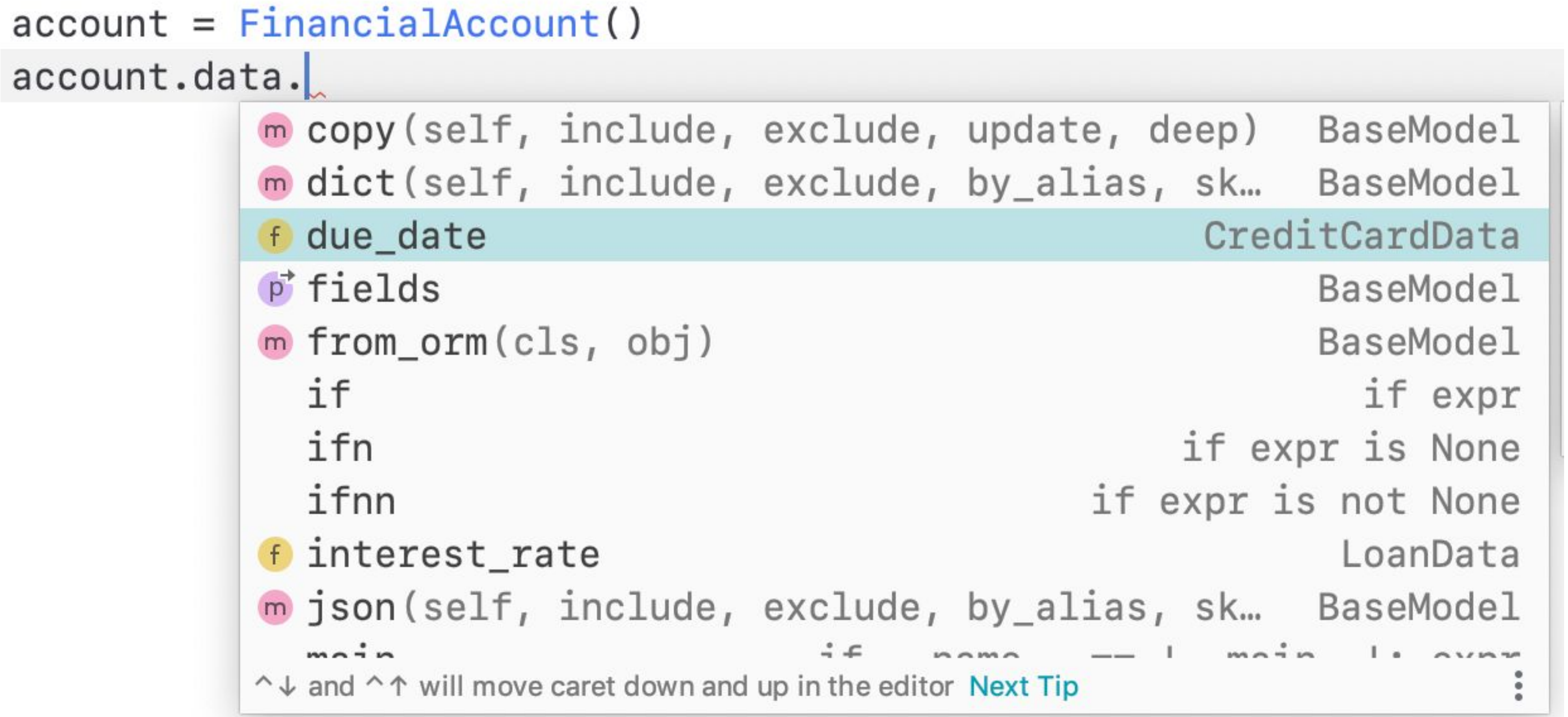
JSONSchemedField benefits

👍 Data validation on save

👍 Returns schema objects instead of a dictionary

👍 Autocomplete!

Useful autocomplete!



```
account = FinancialAccount()
account.data.
```

```
m copy(self, include, exclude, update, deep)    BaseModel
m dict(self, include, exclude, by_alias, sk…    BaseModel
f due_date                                   CreditCardData
p fields                                        BaseModel
m from_orm(cls, obj)                            BaseModel
  if                                              if expr
  ifn                                     if expr is None
  ifnn                                if expr is not None
f interest_rate                                  LoanData
m json(self, include, exclude, by_alias, sk…    BaseModel
  main                if __name__ == '__main__': expr
```

^↓ and ^↑ will move caret down and up in the editor Next Tip

JSONSchemedField benefits

Implementation: https://git.io/Je8IQ

QIWI
Всё проще

STI (JSON): when?

- If your queries use filtering by a common field
- Most of the time you need all data from JSON column (it’s pretty complex to fetch only specific keys from JSON)
- You don’t need complex CONSTRAINTS
- You are not a DBA

ABC vs MTI vs STI: summary

- One table or multiple ones on high throughput?
- Performance?
- Usability?
- Shema (db) vs semi-structured (code)?

Thank you!

Thank you!