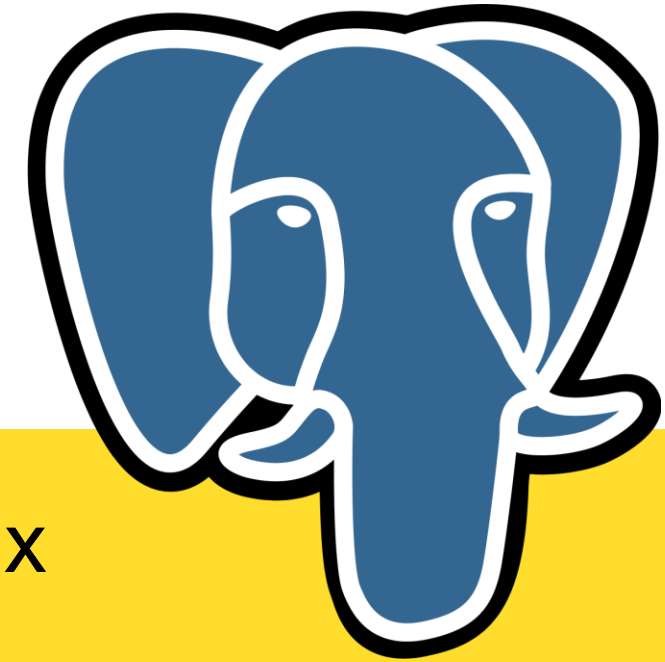


PostgeSQL и базы данных
(часть 2)

Халяпов Александр
a.khalyapov@tinkoff.ru
@khalyapov

Tinkoff.ru

Планы на вечер

— ORM: что такое, с чем едят, как пользоваться.

— ORM vs pure SQL: плюсы и минусы.

— Индексы: что ты такое и зачем насилуешь мой мозг

— Планы запросов: делаем из г*** конфетку.

— Проектирование хранения данных: как не наступить на одни и те же грабли несколько раз.

ORM

ORM (Object-Relational Mapping) — технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных». Существует множество вариантов реализации этой технологии.

ORM

Задача:

Необходимо обеспечить работу с данными в терминах классов, а не таблиц данных и напротив, преобразовать термины и данные классов в данные, пригодные для хранения в СУБД. Необходимо также обеспечить интерфейс для CRUD-операций над данными. В общем, необходимо избавиться от необходимости писать SQL-код для взаимодействия в СУБД.

ORM

Решение **проблемы хранения** существует — это реляционные СУБД. Но их использование для хранения объектно-ориентированных данных приводит к семантическому разрыву, заставляя программистов писать программное обеспечение, которое должно уметь как обрабатывать данные в объектно-ориентированном виде, так и уметь сохранить эти данные в реляционной форме.

ORM

Разработано множество пакетов, устраняющих необходимость в преобразовании объектов для хранения в реляционных базах данных.

С точки зрения программиста система должна выглядеть как постоянное хранилище объектов. Он может просто создавать объекты и работать с ними как обычно, а они автоматически будут сохраняться в реляционной базе данных.

ORM

Что же это такое? (на примере Peewee)

ORM

```
pip install peewee

# или

git clone https://github.com/coleifer/peewee.git
cd peewee
(sudo) python setup.py install
```

Подробнее про установку

ORM

```
from peewee import *

db = PostgresqlDatabase(dbname="people.db", user="postgres", password="qwerty")

class Person(Model):
    name = CharField()
    birthday = DateField()
    is_relative = BooleanField()

    class Meta:
        database = db # модель будет использовать базу данных 'people.db'
```

Подробнее про подключение
Подробнее про типы полей

ORM

Инициализирующие аргументы:

- `null=False` – возможно ли хранение null-значений;
- `index=False` – создавать ли индекс для данного столбца в базе;
- `unique=False` – создавать ли уникальный индекс для данного столбца в базе.
- `verbose_name=None` – строка для человекопонятного представления поля;
- `help_text=None` – строка с вспомогательным текстом для поля;
- `db_column=None` – строка, явно задающая название столбца в базе для данного поля, используется например при работе с legacy-базой данных;
- `default=None` – значение по умолчанию для полей класса при инстанцировании;
- `choices=None` – список или кортеж двухэлементных кортежей, где первый элемент – значение для базы, второй – отображаемое значение (аналогично джанге);
- `primary_key=False` – использовать ли данное поле, как первичный ключ;

ORM

Для каждой таблицы можно прописать единые метаданные в `class Meta:`

Опция	Описание	Наследуется ли
`database`	база данных для модели	да
`db_table`	название таблицы, в которой будут храниться данные	нет
`indexes`	список полей для индексирования	да
`order_by`	список полей для сортировки по умолчанию	да
`primary_key`	составной первичный ключ, экземпляр класса CompositeKey, пример	да
`table_alias`	алиас таблицы для использования в запросах	нет

ORM

```
class Pet(Model):
    owner = ForeignKeyField(Person, related_name='pets')
    name = CharField()
    animal_type = CharField()

    class Meta:
        database = db # модель будет использовать базу данных 'people.db'
```

ORM

```
>>> Person.create_table()
>>> Pet.create_table()
```

ORM

```
>>> from datetime import date
>>> uncle_bob = Person(name='Bob', birthday=date(1960, 1, 15),
is_relative=True)
>>> uncle_bob.save() # сохраним Боба в базе данных

>>> grandma = Person.create(name='Grandma', birthday=date(1935, 3, 1),
is_relative=True)
>>> herb = Person.create(name='Herb', birthday=date(1950, 5, 5),
is_relative=False)

>>> grandma.name = 'Grandma L.'
>>> grandma.save() # обновим запись grandma
```

ORM

```
>>> bob_kitty = Pet.create(owner=uncle_bob, name='Kitty', animal_type='cat')
>>> herb_fido = Pet.create(owner=herb, name='Fido', animal_type='dog')
>>> herb_mittens = Pet.create(owner=herb, name='Mittens', animal_type='cat')
>>> herb_mittens_jr = Pet.create(owner=herb, name='Mittens Jr',
animal_type='cat')

>>> herb_mittens.delete_instance() # удачи, Варежка
1

>>> herb_fido.owner = uncle_bob
>>> herb_fido.save()
>>> bob_fido = herb_fido # переименуем переменную для лучшего соответствия
суровой реальности
```

ORM

```
>>> grandma = Person.select().where(Person.name == 'Grandma L.').get()

>>> grandma = Person.get(Person.name == 'Grandma L.')
```

ORM

```
>>> for person in Person.select():
...     print(person.name, person.is_relative)
...
Bob True
Grandma L. True
Herb False

>>> for person in Person.select():
...     print person.name, person.pets.count(), 'pets'
...     for pet in person.pets:
...         print ' ', pet.name, pet.animal_type
...
Bob 2 pets
    Kitty cat
    Fido dog
Grandma L. 0 pets
Herb 1 pets
    Mittens Jr cat
```

ORM

```
>>> for pet in Pet.select().where(Pet.animal_type == 'cat'):
...     print pet.name, pet.owner.name
...
Kitty Bob
Mittens Jr Herb

# выберем всех животных Боба
>>> for pet in Pet.select().join(Person).where(Person.name == 'Bob'):
...     print pet.name
...
Kitty
Fido

>>> for pet in Pet.select().where(Pet.owner == uncle_bob):
...     print pet.name
```

ORM

```
>>> for pet in Pet.select().where(Pet.owner == uncle_bob).order_by(Pet.name):
...     print pet.name
...
Fido
Kitty

>>> for person in Person.select().order_by(Person.birthday.desc()):
...     print person.name
...
Bob
Herb
Grandma L.
```

ORM

```
>>> d1940 = date(1940, 1, 1)
>>> d1960 = date(1960, 1, 1)
>>> for person in Person.select().where((Person.birthday < d1940) |
(Person.birthday > d1960)):
...     print person.name
...
Bob
Grandma L.

>>> for person in Person.select().where((Person.birthday > d1940) &
(Person.birthday < d1960)):
...     print person.name
...
Herb

>>> for person in Person.select().where(fn.Lower(fn.Substr(Person.name, 1, 1))
== 'g'):
...     print person.name
...
Grandma L.
```

ORM

```
SelectQuery.group_by()

SelectQuery.having()

SelectQuery.limit()
```

Официальная документация
Набор плагинов

ORM: плюсы и минусы

ORM, по идее, должен избавить нас от написания SQL запросов и, в идеале, вообще абстрагировать от базы данных (от способа хранения данных), чтобы мы могли работать с классами, в той или иной степени выражающими объекты бизнес-логики, не задаваясь вопросом, в каких таблицах всё это по факту лежит.

Для примера возьмем две таблицы: книги и авторы книг, отношение многие-ко-многим (у книг может быть много авторов, у авторов может быть много книг). Т.е. в базе это будут books, authors и связующая таблица author_book:

ORM: плюсы и минусы

```
CREATE TABLE authors (
    id serial
    , name varchar(1000) not null
    , PRIMARY KEY (id)
);

CREATE TABLE books (
    id serial
    , name VARCHAR (1000) not null
    , text text not null
    , PRIMARY KEY (id)
);

CREATE TABLE author_book (
    author_id bigint REFERENCES authors(id)
    , book_id bigint REFERENCES books(id)
    , PRIMARY KEY (author_id, book_id)
);
```

ORM: плюсы и минусы

Кейс 1: создание записей

- Pure SQL
  - Piece of cake. Простые CREATE и INSERT
  - Много писанины
  - Нужно знать синтаксис SQL
- ORM
  - Piece of cake. Создаём нужные классы, наполняем
  - Много писанины
  - Нужно разобраться с ORM

ORM: плюсы и минусы

Кейс 2: обновление записей

- Pure SQL

  Piece of cake. Просто UPDATE

- ORM

  Piece of cake. Пользуемся нужным методом (как с именем бабули ранее)

ORM: плюсы и минусы

Кейс 3: простая выборка

```
select
    b.id as book_id
    , b.name as book_name
    , json_agg(a.name) as authors
from
    books b
inner join
    author_book ab
        on b.id = ab.book_id
inner join
    authors a
        on ab.author_id = a.id
group by b.id;
```

- Pure SQL

  Piece of cake. Запрос с агрегацией

- ORM

  Piece of cake (так ли это?). Проходим циклом и находим нужное

  Будет несколько запросов: выборка всех книг и для каждой выборка автора

  С ростом количества данных будет выполняться всё дольше

  Вытянутся все поля, а не только нужные; придётся разбираться с SQL

ORM: плюсы и минусы

Кейс 4: сложное обновление

```
UPDATE authors
SET name = ‘Олег Юрьевич'
WHERE id in (
    SELECT
        id
    FROM
        authors
    ORDER BY
        id DESC
    LIMIT 2
);
```

- Pure SQL

  Piece of cake. Простой подзапрос

- ORM

  Долгое раскуривание мануалов

  Никаких подзапросов

  Неоптимально с точки зрения БД

Индексы

Индексы в PostgreSQL — специальные объекты базы данных, предназначенные в основном для ускорения доступа к данным. Это вспомогательные структуры: любой индекс можно удалить и восстановить заново по информации в таблице. Индексы служат также для поддержки некоторых ограничений целостности.
В настоящее время в PostgreSQL 9.6 встроены шесть разных видов индексов (но мы не будем рассматривать все).

Индексы

- Все индексы — вторичные, они отделены от таблицы. Вся информация о них содержится в системном каталоге.
- Индексы связывают ключи и TID
  (tuple id - #page: offset) ← указатель на строку.
- Индексы могут быть многоколончатыми.
- Индексы не содержат информации о видимости.
- Любое обновление записи в таблице приводит к появлению новой записи в индексе.
- Обновление полей таблицы, по которым не создавались индексы, не приводит к перестроению индексов (Heap-Only Tuples).

Индексы: условия использования

- Совпадают оператор и типы аргументов (порядок важен).
- Индекс валиден.
- В многоколончатом индексе важен порядок.
- План с его использованием — оптимален (минимальная стоимость).
- Всю информацию Postgres берет из системного каталога.

CREATE INDEX

```
CREATE [UNIQUE] INDEX [CONCURRENTLY] [name] ON table_name [USING method]
          ({column_name|(expression)} [COLLATE collation] [opclass] [ASC|DESC]
          [NULLS {FIRST|LAST}] [, ...])
          [WITH (storage_parameter = value [, ...])]
          [TABLESPACE tablespace_name]
          [WHERE predicate]


ALTER INDEX [IF EXISTS] name RENAME TO new_name
ALTER INDEX [IF EXISTS] name SET TABLESPACE tablespace_name
ALTER INDEX [IF EXISTS] name SET (storage_parameter = value [, ... ])
ALTER INDEX [IF EXISTS] name RESET (storage_parameter [, ... ])

DROP INDEX [CONCURRENTLY] [IF EXISTS] name [, ...] [CASCADE|RESTRICT]
```

Индексы: способы сканирования

```
postgres=# create table t(a integer, b text, c boolean);
CREATE TABLE
postgres=# insert into t(a,b,c)
  select s.id, chr((32+random()*94)::integer), random() <
0.01
  from generate_series(1,100000) as s(id)
  order by random();
INSERT 0 100000
postgres=# create index on t(a);
CREATE INDEX
postgres=# analyze t;
ANALYZE
```

Мы создали таблицу с тремя полями.

Первое поле содержит числа от 1 до 100000, и по нему создан индекс (пока нам не важно, какой именно).

Второе поле содержит различные ASCII-символы, кроме непечатных.

Наконец, третье поле содержит логическое значение, истинное примерно для 1% строк, и ложное для остальных. Строки вставлены в таблицу в случайном порядке.

Индексы: способы сканирования

```
postgres=# explain (costs off) select * from t where a = 1;
          QUERY PLAN
------------------------------
 Index Scan using t_a_idx on t
   Index Cond: (a = 1)
(2 rows)
```

Попробуем выбрать значение по условию «a = 1».
Заметим, что условие имеет вид «*индексированное-поле оператор выражение*», где в качестве *оператора* используется «равно», а *выражением* (ключом поиска) является «1».
В большинстве случаев условие должно иметь именно такой вид, чтобы индекс мог использоваться.

В данном случае оптимизатор принял решение использовать *индексное сканирование* (Index Scan). При индексном просмотре метод доступа возвращает значения TID по одному, до тех пор, пока подходящие строки не закончатся. Механизм индексирования по очереди обращается к тем страницам таблицы, на которые указывают TID, получает версию строки, проверяет ее видимость в соответствии с правилами многоверсионности, и возвращает полученные данные.

Индексы: способы сканирования

Индексное сканирование хорошо работает, когда речь идет всего о нескольких значениях. Однако при увеличении выборки возрастают шансы, что придется возвращаться к одной и той же табличной странице несколько раз. Поэтому в таком случае оптимизатор переключается на *сканирование по битовой карте* (bitmap scan):

```
postgres=# explain (costs off) select * from t where a <= 100;
             QUERY PLAN
-----------------------------------
 Bitmap Heap Scan on t
   Recheck Cond: (a <= 100)
   ->  Bitmap Index Scan on t_a_idx
         Index Cond: (a <= 100)
(4 rows)
```

Сначала метод доступа возвращает все TID, соответствующие условию (узел Bitmap Index Scan), и по ним строится битовая карта версий строк. Затем версии строк читаются из таблицы (Bitmap Heap Scan) — при этом каждая страница будет прочитана только один раз.

Обратите внимание, что на втором шаге условие может перепроверяться (Recheck Cond). Выборка может оказаться слишком велика, чтобы битовая карта версий строк могла целиком поместиться в оперативную память (ограниченную параметром work_mem). В этом случае строится только битовая карта *страниц,* содержащих хотя бы одну подходящую версию строки. Такая «грубая» карта занимает меньше места, но при чтении страницы приходится перепроверять условия для каждой хранящейся там строки.

Индексы: способы сканирования

Если условия наложены на несколько полей таблицы, и эти поля проиндексированы, сканирование битовой карты позволяет (если оптимизатор сочтет это выгодным) использовать несколько индексов одновременно. Для каждого индекса строятся битовые карты версий строк, которые затем побитово логически умножаются (если выражения соединены условием AND), либо логически складываются (если выражения соединены условием OR):

```
postgres=# create index on t(b);
CREATE INDEX
postgres=# analyze t;
ANALYZE
postgres=# explain (costs off) select * from t where a <=
100 and b = 'a';
                     QUERY PLAN
-----------------------------------------------
 Bitmap Heap Scan on t
   Recheck Cond: ((a <= 100) AND (b = 'a'::text))
   ->  BitmapAnd
         ->  Bitmap Index Scan on t_a_idx
               Index Cond: (a <= 100)
         ->  Bitmap Index Scan on t_b_idx
               Index Cond: (b = 'a'::text)
(7 rows)
```

Здесь узел BitmapAnd объединяет две битовые карты с помощью битовой операции «и».

Индексы: способы сканирования

Что, если данные в страницах таблицы физически упорядочены точно так же, как и записи индекса? Нельзя полностью полагаться на физический порядок данных в страницах — если нужны отсортированные данные, в запросе необходимо явно указывать предложение ORDER BY. Но вполне возможны ситуации, в которых на самом деле «почти все» данные упорядочены: например, если строки добавляются в нужном порядке и не изменяются после этого, или после выполнения команды CLUSTER. Тогда построение битовой карты — лишний шаг, обычное индексное сканирование будет таким же. Поэтому при выборе метода доступа планировщик заглядывает в специальную статистику, которая показывает степень упорядоченности данных:

```
postgres=# select attname, correlation from pg_stats where
tablename = 't';

 attname | correlation
---------+-------------
 b       |    0.533512
 c       |    0.942365
 a       | -0.00768816
(3 rows)
```

Значения, близкие по модулю к единице, говорят о высокой упорядоченности (как для столбца c), а близкие к нулю — наоборот, о хаотичном распределении (столбец a).

Индексы: способы сканирования

```
postgres=# explain (costs off) select * from t
where a <= 40000;
       QUERY PLAN
------------------------
 Seq Scan on t
   Filter: (a <= 40000)
(2 rows)
```

Для полноты картины следует сказать, что при неселективном условии оптимизатор предпочтет использованию индекса *последовательное сканирование* таблицы целиком:

Индексы работают тем лучше, чем выше селективность условия, то есть чем меньше строк ему удовлетворяет. При увеличении выборки возрастают и накладные расходы на чтение страниц индекса.

Ситуация усугубляется тем, что последовательное чтение выполняется быстрее, чем чтение страниц «вразнобой». Это особенно верно для жестких дисков, где механическая операция подведения головки к дорожке занимает существенно больше времени, чем само чтение данных; в случае дисков SSD этот эффект менее выражен.

Индексы: способы сканирования

```
postgres=# vacuum t;
VACUUM
postgres=# explain (costs off) select a from t where a < 100;
             QUERY PLAN
------------------------------------
 Index Only Scan using t_a_idx on t
   Index Cond: (a < 100)
(2 rows)
```

Как правило, основная задача метода доступа — вернуть идентификаторы подходящих строк таблицы, чтобы механизм индексирования мог прочитать из них необходимые данные. Но что, если индекс уже содержит все необходимые для запроса данные?
Такой индекс называется *покрывающим* (covering), и в этом случае оптимизатор может применить *исключительно индексное сканирование* (Index Only Scan):

Название может навести на мысль, что механизм индексирования совсем не обращается к таблице, получая всю необходимую информацию исключительно от метода доступа. Но это не совсем так, потому что индексы в PostgreSQL не содержат информации, позволяющей судить о видимости строк. Поэтому метод доступа возвращает все версии строк, попадающие под условие поиска, независимо от того, видны они текущей транзакции или нет.

Индексы: способы сканирования

Неопределенные значения играют важную роль в реляционных базах данных как удобный способ представления того факта, что значение не существует или не известно.

Но особое значение требует и особого к себе отношения. Обычная булева логика превращается в трехзначную; непонятно, должно ли неопределенное значение быть меньше обычных значений или больше (отсюда специальные конструкции для сортировки NULLS FIRST и NULLS LAST); не очевидно, надо ли учитывать неопределенные значения в агрегатных функциях или нет; требуется специальная статистика для планировщика…

С точки зрения индексной поддержки с неопределенными значениями тоже имеется неясность: надо ли индексировать такие значения или нет? Если не индексировать null, то индекс может получиться компактнее. Зато если индексировать, то появляется возможность использовать индекс для условий вида «*индексированное-поле* IS [NOT] NULL», а также в качестве покрывающего индекса при полном отсутствии условий на таблицу (поскольку в этом случае индекс должен вернуть данные всех строк таблицы, в том числе и с неопределенными значениями).

Индексы: способы сканирования

```
postgres=# create index on t(a,b);
CREATE INDEX
postgres=# analyze t;
ANALYZE
```

```
postgres=# explain (costs off) select * from t where a <=
100 and b = 'a';
                    QUERY PLAN
-------------------------------------------------
 Index Scan using t_a_b_idx on t
   Index Cond: ((a <= 100) AND (b = 'a'::text))
(2 rows)
```

```
postgres=# explain (costs off) select * from t where a <=
100;
              QUERY PLAN
--------------------------------------
 Bitmap Heap Scan on t
   Recheck Cond: (a <= 100)
   ->  Bitmap Index Scan on t_a_b_idx
         Index Cond: (a <= 100)
(4 rows)
```

Условия на несколько полей могут быть поддержаны с помощью *многоколоночных индексов*. Например, мы могли бы создать индекс по двум полям нашей таблицы:

Оптимизатор скорее всего предпочтет такой индекс объединению битовых карт, поскольку здесь мы сразу получаем нужные TID без каких-либо вспомогательных действий:

Многоколоночный индекс может использоваться и для ускорения выборки по условию на часть полей — начиная с первого:

Индексы: почти конец

Что осталось за бортом:

- Функциональные индексы
- Частичные индексы
- Сортировки
- Параллельное построение
- Интерфейсы доступа

Где почитать/посмотреть:

- https://habrahabr.ru/company/postgrespro/blog/326096/
- https://vimeo.com/105424016

План запроса

Для оптимизации запросов очень важно понимать логику работы ядра PostgreSQL.

Всё не так сложно.

EXPLAIN выводит информацию, необходимую для понимания, что же делает ядро при каждом конкретном запросе.
Будем рассматривать вывод команды EXPLAIN, параллельно разбираясь, что же происходит внутри PostgreSQL.
Описанное применимо к PostgreSQL 9.2 и выше.

План запроса: тестовая таблица

```
CREATE TABLE foo (c1 integer, c2 text);
INSERT INTO foo
  SELECT
    i
    , md5(random()::text)
  FROM
    generate_series(1, 1000000) AS i;
```

План запроса: простая выборка

```
EXPLAIN SELECT * FROM foo;

QUERY PLAN
— Seq Scan on foo (cost=0.00..18334.00 rows=1000000 width=37)
(1 row)
```

Чтение данных из таблицы может выполняться несколькими способами. В нашем случае `EXPLAIN` сообщает, что используется `Seq Scan` — последовательное, блок за блоком, чтение данных таблицы `foo`.
Что такое `cost`? Это не время, а некое сферическое в вакууме понятие, призванное оценить затратность операции. Первое значение `0.00` — затраты на получение первой строки. Второе — `18334.00` — затраты на получение всех строк.
`rows` — приблизительное количество возвращаемых строк при выполнении операции `Seq Scan`. Это значение возвращает планировщик. В моём случае оно совпадает с реальным количеством строк в таблице.
`width` — средний размер одной строки в байтах.

План запроса: простая выборка

```
INSERT INTO foo
  SELECT
    i
    , md5(random()::text)
  FROM
    generate_series(1, 10) AS i;
EXPLAIN SELECT * FROM foo;

QUERY PLAN
— Seq Scan on foo (cost=0.00..18334.00 rows=1000000 width=37)
(1 row)
```

Значение `rows` не изменилось. Статистика по таблице старая. Для обновления статистики вызываем команду `ANALYZE`.

План запроса: простая выборка

```
ANALYZE foo;
EXPLAIN SELECT * FROM foo;

QUERY PLAN
— Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37)
(1 row)
```

Теперь `rows` отображает верное количество строк.

Что происходит при выполнении `ANALYZE`:

- Считывается определённое количество строк таблицы, выбранных случайным образом
- Собирается статистика значений по каждой из колонок таблицы

Всё, что мы видели выше в выводе команды `EXPLAIN` — только ожидания планировщика. Попробуем сверить их с результатами на реальных данных. Используем `EXPLAIN (ANALYZE)`.

План запроса: простая выборка

```
EXPLAIN (ANALYZE) SELECT * FROM foo;

QUERY PLAN
— Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37) (actual
time=0.012..61.524 rows=1000010 loops=1)
Total runtime: 90.944 ms
(2 rows)
```

Такой запрос будет исполняться реально. Так что если вы выполняете `EXPLAIN (ANALYZE)` для `INSERT`, `DELETE` или `UPDATE`, ваши данные изменятся. Будьте внимательны! В таких случаях используйте команду `ROLLBACK`.

В выводе команды информации добавилось:

`actual time` — реальное время в миллисекундах, затраченное для получения первой строки и всех строк соответственно.

`rows` — реальное количество строк, полученных при `Seq Scan`.

`loops` — сколько раз пришлось выполнить операцию `Seq Scan`.

`Total runtime` — общее время выполнения запроса.

План запроса: WHERE

```
EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 > 500;

QUERY PLAN
— Seq Scan on foo (cost=0.00..20834.12 rows=999519 width=37) (actual
time=0.572..848.895 rows=999500 loops=1)
  Filter: (c1 > 500)
  Rows Removed by Filter: 510
Total runtime: 1330.788 ms
(4 rows)
```

Индексов у таблицы нет. При выполнении запроса последовательно считывается каждая запись таблицы (`Seq Scan`). Каждая запись сравнивается с условием `c1 > 500`. Если условие выполняется, запись вводится в результат. Иначе — отбрасывается. `Filter` означает именно такое поведение.
Значение `cost`, что логично, увеличилось.
Ожидаемое количество строк результата — `rows` — уменьшилось.

План запроса: ORDER BY

```
EXPLAIN (ANALYZE) SELECT * FROM foo ORDER BY c1;

QUERY PLAN
— Sort (cost=117993.01..120493.04 rows=1000010 width=37) (actual
time=571.591..651.524 rows=1000010 loops=1)
  Sort Key: c1
  Sort Method: external merge Disk: 45952kB
  -> Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37) (actual
time=0.007..62.041 rows=1000010 loops=1)
Total runtime: 690.984 ms
(5 rows)
```

Сначала производится `Seq Scan` таблицы `foo`. Затем сортировка `Sort`. В выводе команды `EXPLAIN` знак `->` указывает на иерархию действий (`node`). Чем раньше выполняется действие, тем с большим отступом оно отображается.
`Sort Key` — условие сортировки.
`Sort Method: external merge Disk` — при сортировке используется временный файл на диске объёмом `45952kB`.

План запроса: LIMIT

```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM foo WHERE c2 LIKE 'ab%' LIMIT 10;

QUERY PLAN
— Limit (cost=0.00..2083.41 rows=10 width=37) (actual time=0.037..0.607 rows=10 loops=1)
  Buffers: shared hit=26
  -> Seq Scan on foo (cost=0.00..20834.12 rows=100 width=37) (actual time=0.031..0.599
rows=10 loops=1)
     Filter: (c2 ~~ 'ab%'::text)
     Rows Removed by Filter: 3053
     Buffers: shared hit=26
Total runtime: 0.628 ms
(7 rows)
```

Производится сканирование `Seq Scan` строк таблицы и сравнение `Filter` их с условием. Как только наберётся `10` записей, удовлетворяющих условию, сканирование закончится. В нашем случае для того, чтобы получить `10` строк результата пришлось прочитать не всю таблицу, а только `3063` записи, из них `3053` были отвергнуты (`Rows Removed by Filter`).
То же происходит и при `Index Scan`.
`Buffers: shared read` — количество блоков, считанное с диска.
`Buffers: shared hit` — количество блоков, считанных из кэша PostgreSQL.

План запроса: JOIN

```
CREATE TABLE bar (c1 integer, c2 boolean);
INSERT INTO bar
  SELECT
    i
    , i % 2 = 1
  FROM
    generate_series(1, 500000) AS i;
ANALYZE bar;
```

План запроса: JOIN

```
EXPLAIN (ANALYZE) SELECT * FROM foo JOIN bar ON foo.c1 = bar.c1;

QUERY PLAN
— Hash Join (cost=13463.00..49297.22 rows=500000 width=42) (actual time=87.441..907.555
rows=500010 loops=1)
  Hash Cond: (foo.c1 = bar.c1)
  -> Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37) (actual time=0.008..67.951
rows=1000010 loops=1)
  -> Hash (cost=7213.00..7213.00 rows=500000 width=5) (actual time=87.352..87.352
rows=500000 loops=1)
     Buckets: 65536 Batches: 1 Memory Usage: 18067kB
    -> Seq Scan on bar (cost=0.00..7213.00 rows=500000 width=5) (actual time=0.007..33.233
rows=500000 loops=1)
Total runtime: 920.967 ms
(7 rows)
```

Сначала просматривается (`Seq Scan`) таблица `bar`. Для каждой её строки вычисляется хэш (`Hash`).
Затем сканируется `Seq Scan` таблица `foo`, и для каждой строки этой таблицы вычисляется хэш, который сравнивается (`Hash Join`) с хэшем таблицы `bar` по условию `Hash Cond`. Если соответствие найдено, выводится результирующая строка, иначе строка будет пропущена.
Использовано `18067kB` в памяти для размещения хэшей таблицы `bar`.

План запроса: JOIN + INDEX

```
CREATE INDEX ON bar(c1);
EXPLAIN (ANALYZE) SELECT * FROM foo JOIN bar ON foo.c1=bar.c1;

QUERY PLAN
— Merge Join (cost=1.69..39879.71 rows=500000 width=42) (actual time=0.037..263.357
rows=500010 loops=1)
  Merge Cond: (foo.c1 = bar.c1)
  -> Index Scan using foo_c1_idx on foo (cost=0.42..34327.57 rows=1000010 width=37) (actual
time=0.019..58.920 rows=500011 loops=1)
  -> Index Scan using bar_c1_idx on bar (cost=0.42..15212.42 rows=500000 width=5) (actual
time=0.008..71.719 rows=500010 loops=1)
Total runtime: 283.549 ms
(5 rows)
```

Hash уже не используется. Merge Join и Index Scan по индексам обеих таблиц дают впечатляющий прирост производительности.

План запроса: LEFT JOIN

```
EXPLAIN (ANALYZE) SELECT * FROM foo LEFT JOIN bar ON foo.c1=bar.c1;

QUERY PLAN
— Hash Left Join (cost=13463.00..49297.22 rows=1000010 width=42) (actual
time=82.682..926.331 rows=1000010 loops=1)
  Hash Cond: (foo.c1 = bar.c1)
  -> Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37) (actual time=0.004..68.763
rows=1000010 loops=1)
  -> Hash (cost=7213.00..7213.00 rows=500000 width=5) (actual time=82.625..82.625
rows=500000 loops=1)
     Buckets: 65536 Batches: 1 Memory Usage: 18067kB
    -> Seq Scan on bar (cost=0.00..7213.00 rows=500000 width=5) (actual time=0.003..31.890
rows=500000 loops=1)
Total runtime: 950.625 ms
(7 rows)
```

Почему Seq Scan???
Посмотрим, что будет, если его запретить.

План запроса: LEFT JOIN

```
SET enable_seqscan TO off;
EXPLAIN (ANALYZE) SELECT * FROM foo LEFT JOIN bar ON foo.c1=bar.c1;

QUERY PLAN
— Merge Left Join (cost=0.85..58290.02 rows=1000010 width=42) (actual time=0.024..353.819
rows=1000010 loops=1)
  Merge Cond: (foo.c1 = bar.c1)
  -> Index Scan using foo_c1_idx on foo (cost=0.42..34327.57 rows=1000010 width=37) (actual
time=0.011..112.095 rows=1000010 loops=1)
  -> Index Scan using bar_c1_idx on bar (cost=0.42..15212.42 rows=500000 width=5) (actual
time=0.008..63.125 rows=500010 loops=1)
Total runtime: 378.603 ms
(5 rows)
```

По мнению планировщика, использование индексов затратнее, чем использование хэшей.
Такое возможно при достаточно большом объёме выделенной памяти.
Но, если память в дефиците, планировщик будет вести себя иначе:

План запроса: LEFT JOIN

```
SET work_mem TO '15MB’;
SET enable_seqscan TO ON;
EXPLAIN (ANALYZE) SELECT * FROM foo LEFT JOIN bar ON foo.c1=bar.c1;

QUERY PLAN
– Merge Left Join (cost=0.85..58290.02 rows=1000010 width=42) (actual time=0.014..376.395
rows=1000010 loops=1)
  Merge Cond: (foo.c1 = bar.c1)
  -> Index Scan using foo_c1_idx1 on foo (cost=0.42..34327.57 rows=1000010 width=37)
(actual time=0.005..124.698 rows=1000010 loops=1)
  -> Index Scan using bar_c1_idx on bar (cost=0.42..15212.42 rows=500000 width=5) (actual
time=0.006..66.813 rows=500010 loops=1)
Total runtime: 401.990 ms
(5 rows)
```

А как будет выглядеть вывод EXPLAIN при запрещённом Index Scan?

План запроса: LEFT JOIN

```
SET work_mem TO '15MB’;
SET enable_indexscan TO off;
EXPLAIN (ANALYZE) SELECT * FROM foo LEFT JOIN bar ON foo.c1=bar.c1;

QUERY PLAN
— Hash Left Join (cost=15417.00..63831.18 rows=1000010 width=42) (actual
time=93.440..712.056 rows=1000010 loops=1)
  Hash Cond: (foo.c1 = bar.c1)
  -> Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37) (actual time=0.008..65.901
rows=1000010 loops=1)
  -> Hash (cost=7213.00..7213.00 rows=500000 width=5) (actual time=93.308..93.308
rows=500000 loops=1)
     Buckets: 65536 Batches: 2 Memory Usage: 9045kB
    -> Seq Scan on bar (cost=0.00..7213.00 rows=500000 width=5) (actual time=0.007..33.718
rows=500000 loops=1)
Total runtime: 736.726 ms
(7 rows)
```

cost явно увеличился. Причина в Batches: 2. Весь хэш не поместился в память, его пришлось разбить на 2 пакета по 9045kB.

План запроса: почти-почти конец

Что осталось за бортом:

- А ничего, дальше можно только смотреть и пробовать самостоятельно.

Где почитать:

- http://langtoday.com/?p=229
- http://langtoday.com/?p=270
- https://habrahabr.ru/post/203320/

Проектирование базы данных

Процесс создания детализированной модели данных* БД, а также необходимых ограничений целостности

Модель данных – абстрактная модель, которая:

- Организует элементы данных
- Описывает, как они взаимодействуют друг с другом
- Описывает, как они взаимодействуют с объектами внешнего мира

Основные задачи проектирования БД

- Обеспечение хранения в БД всей необходимой информации
- Обеспечение возможности получения данных по всем необходимым запросам
- Сокращение избыточности и дублирования данных
- Обеспечение целостности базы данных

Основные шаги проектирования

- Определение данных, которые будут храниться в БД
- Определение взаимосвязей между различными элементами данных
- Наложение логической структуры на данные на основе определенных соотношений
- Создание спроектированной БД с использованием СУБД

Этапы проектирования БД

- Концептуальное (инфологическое) проектирование
- Логическое (даталогическое) проектирование
- Физическое проектирование

Шаг 1. Определение данных

- Ожидание:
  - Найти эксперта по проектированию БД
  - Найти эксперта в предметной области БД
  - Делегировать задачу этим людям

- Реальность:
  - Разобраться во всем самостоятельно

Шаг 1. Определение данных

- Какую предметную область описываем?
  - Полная база товаров магазинов «Пятерочка»
  - Полная клиентская база банка «Сбербанк»
  - Учебные процессы «МФТИ»
  - И т.д.

Шаг 1. Определение данных

- Как будем описывать? Как будем детализировать?
  - Полная база товаров магазина «Пятерочка»:
    - Каталог товаров
    - Каталог складов
    - Актуальные характеристики наполнения склада
    - Планы поставок
    - И т.д.

Шаг 1. Определение данных

- Как будем описывать? Как будем детализировать?
  - Полная клиентская база банка «Сбербанк»:
    - Клиент
    - Дебетовые продукты клиента
    - Кредитные продукты клиента
    - Транзакции
    - И т.д.

Шаг 1. Определение данных

- Как будем описывать? Как будем детализировать?
  - Учебные процессы «МФТИ»
    - Студент
    - Группа
    - Преподаватель
    - Расписание пар
    - Расписание экзаменов
    - Распределение аудиторий
    - И т.д.

Концептуальное проектирование

Концептуальная модель данных – описание основных объектов и отношений между ними

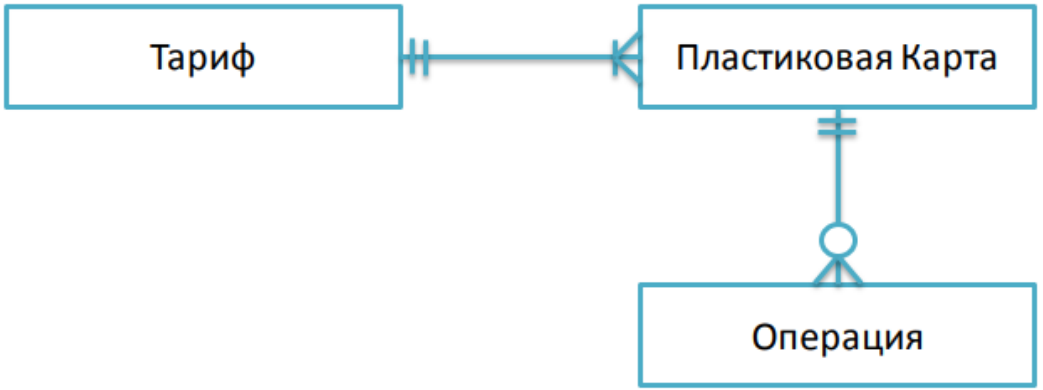
Про описание объектов поговорили. Что насчет связей?

ER-модель

- ER-модель (entity-relationship model) – модель данных, позволяющая описывать концептуальные и логические схемы.

- Задаются:
  - Сущности
  - Связи

ER-модель

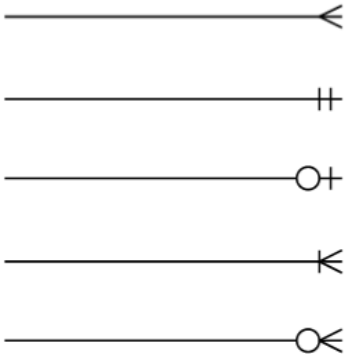


Шаг 2. Определение взаимосвязей

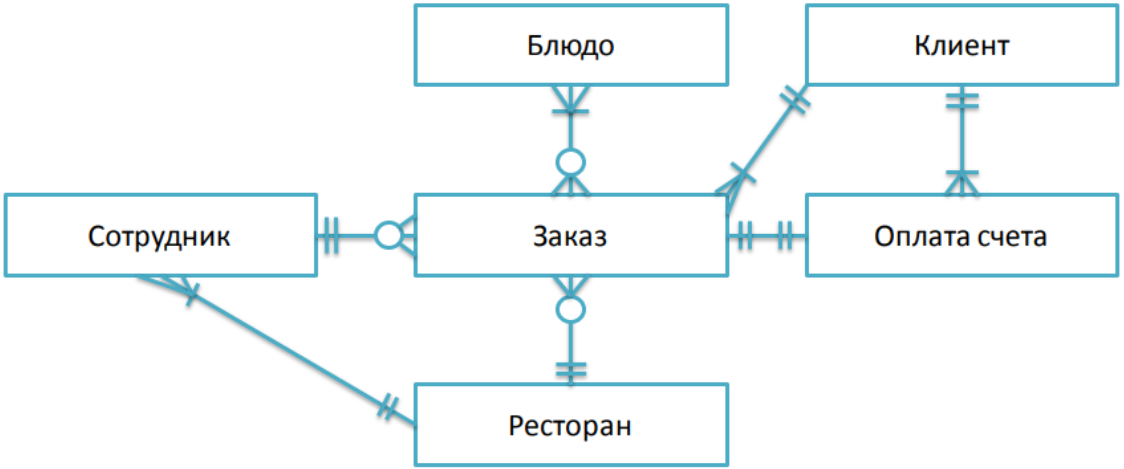
- Определили предметную область и детализировали разбиение на сущности
- Необходимо установить взаимосвязи

Нотация «Воронья лапка»

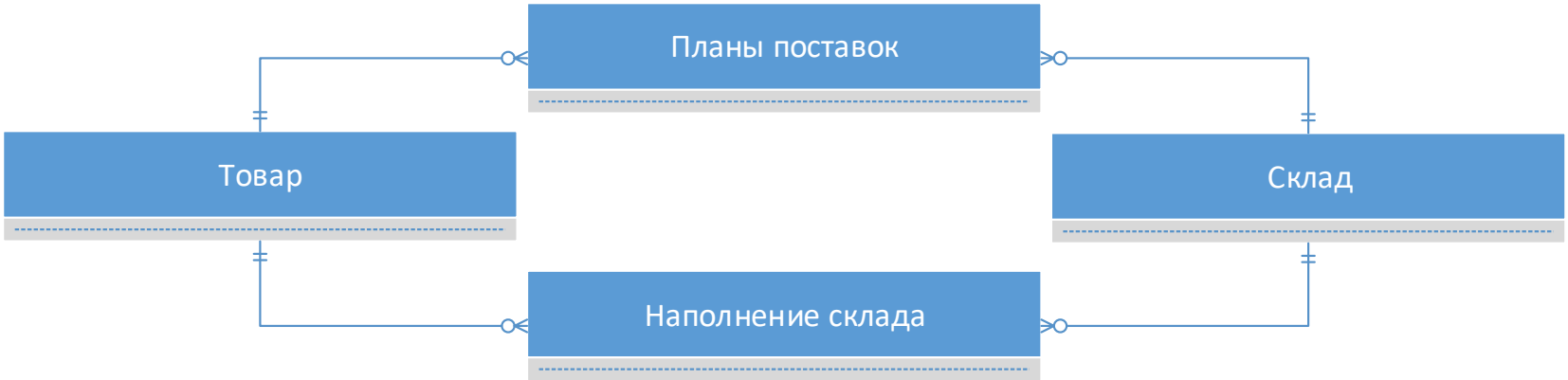
- В оригинале Crow’s Foot notation
  - Многие (строго больше одного)
  - Один и только один
  - Один или ноль
  - Многие или один
  - Многие или один или ноль



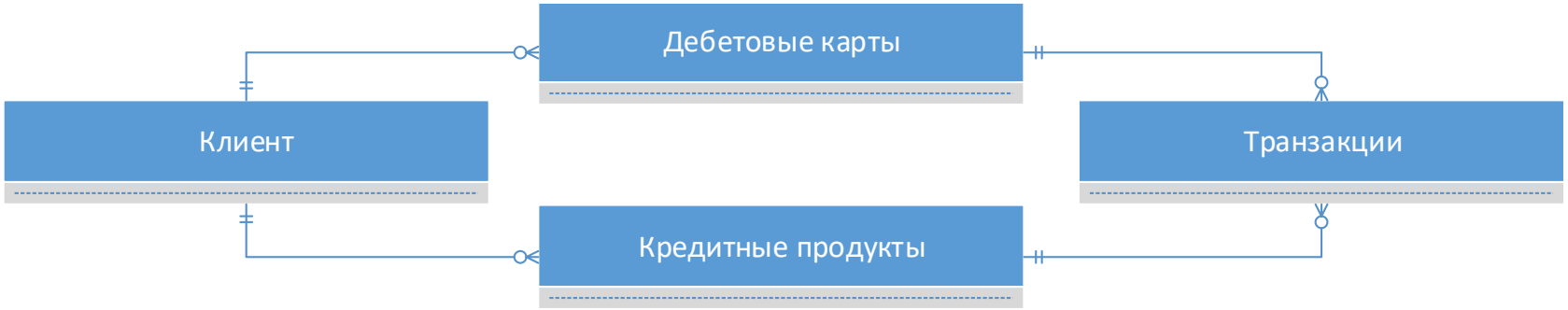
Концептуальная модель БД сети ресторанов



Примеры ER-диаграмм: «Пятерочка»



Примеры ER-диаграмм: «Сбербанк»



Потенциальный ключ

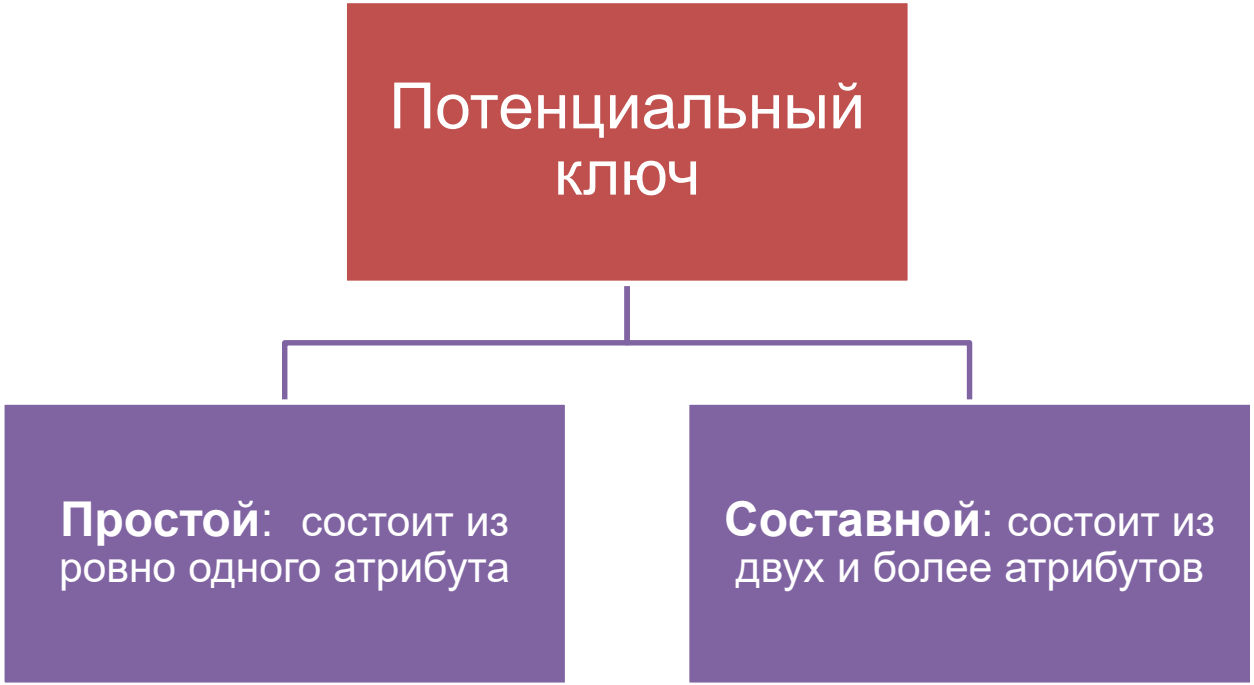
… – это подмножество атрибутов отношения, удовлетворяющее требованиям уникальности и минимальности.

- *Уникальность*: нет и не может быть двух кортежей данного отношения, в которых значения этого подмножества атрибутов совпадают.
- *Минимальность*: в составе потенциального ключа отсутствует меньшее подмножество атрибутов, удовлетворяющее условию уникальности.

Потенциальный ключ

- Из свойства отношения: потенциальный ключ существует **всегда**, даже если он включает **все** атрибуты отношения
- Допустимо наличие нескольких потенциальных ключей в отношении

Классификация по признаку общности



Первичный ключ

… – это один из потенциальных ключей отношения, выбранный в качестве основного. (Primary key, PK)

- Если в отношении имеется лишь один потенциальный ключ, он и будет первичным ключом.
- Если потенциальных ключей несколько, то:
  - Один из них выбирается в качестве первичного
  - Остальные ключи называются ***альтернативными***
- Недопустимо отсутствие значения

Первичный ключ

- Теоретически, все потенциальные ключи одинаково пригодны для использования в качестве первичного ключа
- На практике, используют тот потенциальный ключ, который:
  - Занимает меньше места при хранении
  - Не утратит свою уникальность со временем

Суррогатный ключ

… – это дополнительное служебное поле, которое добавляется к уже имеющимся информационным полям таблицы, единственное предназначение которое – служить первичным ключом.

- Значение такого поля генерируется искусственно
- Не стоит искать в нем какой-то глубинный смысл

- Ключ, который основан на уже существующем поле, называется ***естественным***.
- Ключ, который основан на естественном ключе путем добавления дополнительного поля, называется ***интеллектуальным***.

Суррогатный ключ

- Обычно суррогатный ключ – числовое поле
- Значения суррогатного ключа генерируется арифметической прогрессией с шагом 1
- В ряде СУБД существует специальный тип данных, автоматически генерирующий такую последовательность:
  - PostgreSQL – SERIAL
  - MySQL – AUTO_INCREMENT

Достоинства суррогатных ключей

- **Неизменность**: заполнили одним значением раз и навсегда (кроме экстраординарных ситуаций)
- **Гарантированная уникальность**: т.к. значение генерируется автоинкрементом, повторение значений исключено
- **Гибкость**: т.к. такой ключ не несет никакой информативной нагрузки, его можно свободно заменить
- **Проще программировать**: позволяет не завязывать на структуре конкретной БД. Особенно удобно для языков со статической типизацией
- **Эффективность**: удобнее при создании ссылок на другие таблицы

Недостатки суррогатных ключей

- **Уязвимость генераторов**: по номерам ключей возможно узнать число новых записей за определенный период времение
- **Неинформативность**: усложняется ручная проверка БД
- **Склоняет администратора пропустить нормализацию**: вместо того, чтобы разбить отношение на несколько отношений и аккуратно учесть все связи, велик соблазн просто создать суррогатный ключ
- **Вопросы оптимизации**: необходимость поддержания и суррогатного, и естественного ключей (об этом поговорим на лекции 6)
- **Невольная привязка разработчика к поведению генератора ключей в конкретной СУБД**

Внешний ключ

Пусть R_1 и R_2 – две переменные отношения, не обязательно различные.
Внешним ключом FK в R_2 является подмножество атрибутов переменной R_2 такое, что выполняются следующие требования:

- В переменной отношения R_1 имеется потенциальный ключ PK такой, что PK и FK совпадают с точностью до переименования атрибутов
- В любой момент времени каждое значение FK в текущем значении R_2 идентично значению PK в некотором кортеже в текущем значении R_1. Иными словами, в любой момент времени множество всех значений FK в R_2 является подмножеством значений PK в R_1.

Внешний ключ

- Отношение R_1, содержащее потенциальный ключ, называется ***главным***, ***целевым*** или ***родительским***
- Отношение R_2, содержащее внешний ключ, называется ***подчиненным*** или ***дочерним***

Внешний ключ

ID (PK)	CITY
1	Москва
2	Санкт-Петербург
3	Владивосток

ID (PK)	STREET	CITY_ID (FK)
181	Малая Бронная	1
182	Тверской бульвар	1
183	Невский проспект	2
184	Пушкинская	2
185	Светланская	3
186	Пушкинская	3

Ссылочная целостность

… – это необходимое качество реляционной базы данных, заключающееся в отсутствии в любом её отношении внешних ключей, ссылающихся на несуществующие кортежи.

- База данных обладает свойством ссылочной целостности, когда для любой пары связанных внешним ключом отношений в ней условие ссылочной целостности выполняется

Поддержание ссылочной целостности

- **CASCADE**
  - При удалении / изменении строки главной таблицы соответствующая запись дочерней таблицы также будет удалена / изменена
- **RESTRICT**
  - Строка не может быть удалена / изменена, если на нее имеется ссылка
  - Значение не может быть удалено / изменено, если на него есть ссылка
- **NO ACTION**
  - Похож на RESTRICT, только проверка происходит при операции ALTER TABLE, а не при UPDATE или DELETE
- **SET NULL**
  - При удалении записи главной таблицы, соответствующее значение дочерней таблицы становится NULL
- **SET DEFAULT**
  - Аналогично SET NULL, только вместо значения NULL устанавливается некоторое значение по умолчанию

Нормальная форма

- **Нормальная форма** — свойство отношения в реляционной модели данных, характеризующее его с точки зрения избыточности, потенциально приводящей к логически ошибочным результатам выборки или изменения данных.

- Нормальная форма определяется как совокупность требований, которым должно удовлетворять отношение.

- Приведение БД к нормальной форме – нормализация.
  - Каждая следующая форма включает в себя ограничения предыдущих.

Нормализация БД

- Предназначена:
  - Минимизация логической избыточности
  - Уменьшение потенциальной противоречивости
- Не предназначена:
  - Уменьшение / увеличение производительности БД
  - Уменьшение / увеличение физического объема БД

Нормализация БД

- Исключение некоторых типов избыточности
- Устранение некоторых аномалий* обновления
- Разработка проекта БД, который является:
  - Качественным представление реального мира
  - Интуитивно понятен
  - Легко расширяем в дальнейшем
- Упрощение процедуры применения необходимых ограничений целостности

Аномалии

- Ситуация в таблице БД такая, что:
  - Существенно осложнена работа с БД
  - В БД присутствует противоречия

- Причина:
  - Излишнее дублирование данных в таблице

Аномалии модификации

- Изменение данных одной записи влечет за собой необходимость изменения аналогичных данных еще некоторых записей

Номер поставки (PK)	Название товара (PK)	Цена товара	Количество	Дата поставки	Название поставщика	Адрес поставщика
1	Карандаш	15	10000	12.10.2017	Поставщик_1	Адрес_1
2	Клей	30	1500	03.03.2018	Поставщик_1	Адрес_1
2	Тетрадь	5	10000	03.03.2018	Поставщик_2	Адрес_2
3	Ручка	5	13000	05.03.2018	Поставщик_1	Адрес_1
3	Блокнот	50	20000	05.03.2018	Поставщик_1	Адрес_1
3	Альбом	100	25000	05.03.2018	Поставщик_2	Адрес_2

- Хотим изменить адрес поставщика 1:
  - Придется менять адрес во всех строках

Аномалии модификации

- Удаление определенных записей несет потерю информации, которую удалять не хотели

Номер поставки (PK)	Название товара (PK)	Цена товара	Количество	Дата поставки	Название поставщика	Адрес поставщика
1	Карандаш	15	10000	12.10.2017	Поставщик_1	Адрес_1
2	Клей	30	1500	03.03.2018	Поставщик_1	Адрес_1
2	Тетрадь	5	10000	03.03.2018	Поставщик_2	Адрес_2
3	Ручка	5	13000	05.03.2018	Поставщик_1	Адрес_1
3	Блокнот	50	20000	05.03.2018	Поставщик_1	Адрес_1
3	Альбом	100	25000	05.03.2018	Поставщик_2	Адрес_2

- Хотим удалить записи о поставках от поставщика 2:
  - Теряем всю информации о поставщике 2, включая его адрес

Аномалии модификации

- Не можем добавить новую запись, если неизвестны значения первичных ключей

Номер поставки (PK)	Название товара (PK)	Цена товара	Количество	Дата поставки	Название поставщика	Адрес поставщика
1	Карандаш	15	10000	12.10.2017	Поставщик_1	Адрес_1
2	Клей	30	1500	03.03.2018	Поставщик_1	Адрес_1
2	Тетрадь	5	10000	03.03.2018	Поставщик_2	Адрес_2
3	Ручка	5	13000	05.03.2018	Поставщик_1	Адрес_1
3	Блокнот	50	20000	05.03.2018	Поставщик_1	Адрес_1
3	Альбом	100	25000	05.03.2018	Поставщик_2	Адрес_2

- Заключили контракт с поставщиком 3:
  - Не можем добавить информацию о нем в таблицу, т.к. еще не было поставок

Нормализация БД

- Нормализация БД производится за счет декомпозиции отношения

- Декомпозиция – разложение исходной переменной отношения на несколько эквивалентных
- Декомпозиция обратна соединению

- Декомпозиция называется ***декомпозицией без потерь*** или ***правильной***, если она обратима

Нормальные формы

- Первая нормальная форма (1NF)
- Вторая нормальная форма (2NF)
- Третья нормальная форма (3NF)
- Нормальная форма Бойса-Кодда (BCNF)
- Четвертая нормальная форма (4NF)
- Пятая нормальная форма / Нормальная форма проекции-соединения (5NF / PJNF)
- Доменно-ключевая нормальная форма (DKNF)
- Шестая нормальная форма (6NF)

Нормальные формы

- **Первая нормальная форма (1NF)**
- **Вторая нормальная форма (2NF)**
- **Третья нормальная форма (3NF)**
- Нормальная форма Бойса-Кодда (BCNF)
- Четвертая нормальная форма (4NF)
- Пятая нормальная форма / Нормальная форма проекции-соединения (5NF / PJNF)
- Доменно-ключевая нормальная форма (DKNF)
- Шестая нормальная форма (6NF)

Первая нормальная форма

- Переменная отношения находится в ***первой нормальной форме*** тогда и только тогда, когда значения всех атрибутов отношения атомарны.

Первая нормальная форма

- Переменная отношения находится в ***первой нормальной форме*** тогда и только тогда, когда значения всех атрибутов отношения атомарны.

- Отношение находится в 1NF, если все его атрибуты являются простыми. Все используемые домены содержат только скалярные значения.

Первая нормальная форма

Группа	Студент
291	Шехтер
293	Гусев, Медведева, Меркурьева, Шапошников
298	Мавлютов

Таблица не находится в 1NF, т.к. студенты группы 293 записаны единым списком в одной строке

Первая нормальная форма

Приведем таблицу к 1NF:

Группа	Студент
291	Шехтер
293	Гусев
293	Медведева
293	Меркурьева
293	Шапошников
298	Мавлютов

Вторая нормальная форма

- Переменная отношения находится во ***второй нормальной форме*** тогда и только тогда, когда она находится в первой нормальной форме, и каждый неключевой атрибут минимально функционально зависит от потенциального ключа.
- ***Функциональная зависимость*** между множествами атрибутов X и Y означает, что для любого допустимого набора кортежей в данном отношении верно следующее: если два кортежа совпадают по значению X, то они совпадают по значению Y.
- ***Минимальная функциональная зависимость*** означает, что в составе первичного ключа отсутствует меньшее подмножество атрибутов, от которого можно также вывести данную функциональную зависимость.

Вторая нормальная форма

- Любая переменная отношения, находящаяся в 1NF, но не находящаяся в 2NF, может быть приведена к набору переменных отношений, находящихся в 2NF.

- В результате декомпозиции получим набор проекций исходной переменной отношения, причем обратимый.

Вторая нормальная форма

Название группы	Название CD-диска	Название песни	Автор слов	Композитор
Scorpions	World Wide Live	Countdown	Klaus Meine	Matthias Jabs
Scorpions	World Wide Live	Coming Home	Rudolf Schenker	Klaus Meine
Scorpions	World Wide Live	Blackout	Rudolf Schenker	Klaus Meine
Scorpions	Blackout	Blackout	Rudolf Schenker	Klaus Meine
The Big City	Blackout	Blackout	Rudolf Schenker	Klaus Meine

Вторая нормальная форма

Название группы	Название CD-диска	Название песни	Автор слов	Композитор
Scorpions	World Wide Live	Countdown	Klaus Meine	Matthias Jabs
Scorpions	World Wide Live	Coming Home	Rudolf Schenker	Klaus Meine
Scorpions	World Wide Live	Blackout	Rudolf Schenker	Klaus Meine
Scorpions	Blackout	Blackout	Rudolf Schenker	Klaus Meine
The Big City	Blackout	Blackout	Rudolf Schenker	Klaus Meine

Данная таблица находится в 1NF, но не во 2NF, т.к. автор слов и композитор зависят только от полей «Название группы» и «Название песни», т.е. от того, что песня включена на другой CD-диск, значения этих полей никак не изменятся.

Вторая нормальная форма

Название группы	Название CD-диска	Название песни
Scorpions	World Wide Live	Countdown
Scorpions	World Wide Live	Coming Home
Scorpions	World Wide Live	Blackout
Scorpions	Blackout	Blackout
The Big City	Blackout	Blackout

Название группы	Название песни	Автор слов	Композитор
Scorpions	Countdown	Klaus Meine	Matthias Jabs
Scorpions	Coming Home	Rudolf Schenker	Klaus Meine
Scorpions	Blackout	Rudolf Schenker	Klaus Meine
Scorpions	Blackout	Rudolf Schenker	Klaus Meine
The Big City	Blackout	Rudolf Schenker	Klaus Meine

Вторая нормальная форма

Название группы	Название CD-диска	Название песни
Scorpions	World Wide Live	Countdown
Scorpions	World Wide Live	Coming Home
Scorpions	World Wide Live	Blackout
Scorpions	Blackout	Blackout
The Big City	Blackout	Blackout

Название группы	Название песни	Автор слов	Композитор
Scorpions	Countdown	Klaus Meine	Matthias Jabs
Scorpions	Coming Home	Rudolf Schenker	Klaus Meine
Scorpions	Blackout	Rudolf Schenker	Klaus Meine
Scorpions	Blackout	Rudolf Schenker	Klaus Meine
The Big City	Blackout	Rudolf Schenker	Klaus Meine

Третья нормальная форма

- Переменная отношения находится в ***третьей нормальной форме*** (3NF) в том и только в том случае, когда она находится во второй нормальной форме, и каждый неключевой атрибут нетранзитивно функционально зависит от первичного ключа.

- Иными словами, каждое неключевое поле должно содержать информацию о ключе, полном ключе и ни о чём, кроме ключа.

Третья нормальная форма

Сотрудник	Отдел	Телефон
Иванов	Бухгалтерия	11-22-334
Петров	Бухгалтерия	11-22-334
Сидоров	Снабжение	22-33-445

Данная таблица находится в 2NF, но не во 3NF, т.к. поле «Телефон» зависит от поля «Отдел».

Функциональные зависимости:

- Сотрудник → Отдел
- Отдел → Телефон
- Сотрудник → Телефон

Третья нормальная форма

Отдел	Телефон
Бухгалтерия	11-22-334
Снабжение	22-33-445

Сотрудник	Отдел
Иванов	Бухгалтерия
Петров	Бухгалтерия
Сидоров	Снабжение

Зачем вообще это нужно?

- На первых порах позволяет проектировать неплохие БД
- Помогает избегать типичных ошибок при проектировании БД людям без опыта
- Формирует привычку делать _нормально_ сразу, не оставляя на потом
- Постепенно вырабатывает навык проектирования, без прочной завязки на нормальных формах

Шаг 3. Создание логической структуры

- Уже имеем концептуальную схему

- Хотим получить больший уровень детализации:
  - Уточнение атрибутивного состава
  - Уточнение ограничений, накладываемых на атрибутивный состав
  - В некоторых случаях допускается уточнение типа атрибута

Логическая модель данных

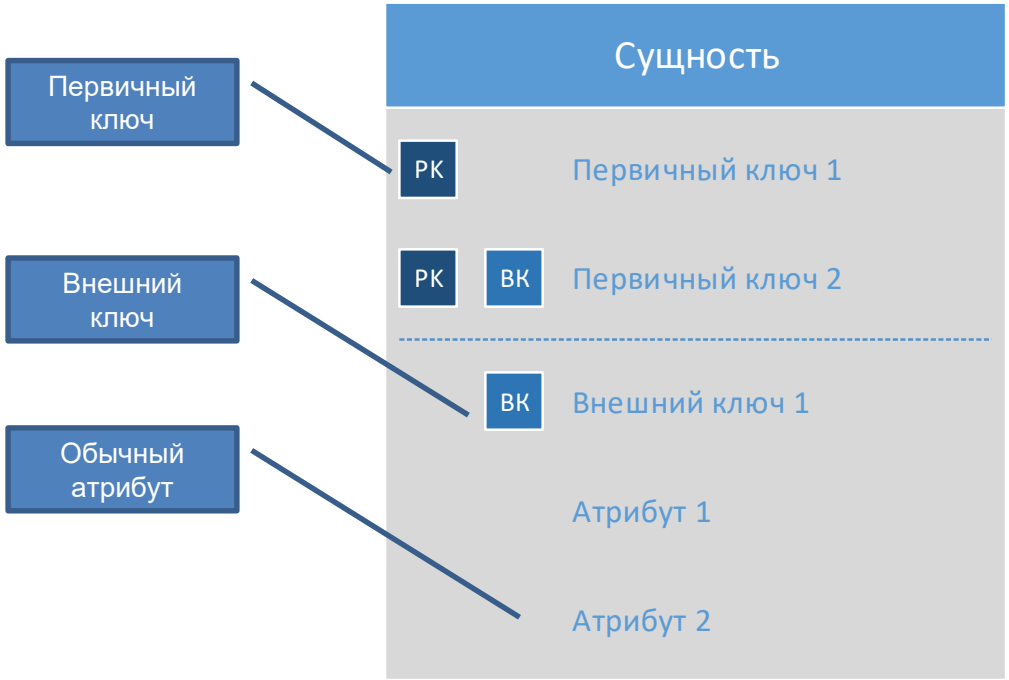
- **Логическая модель данных** – расширение концептуальной модели данных путем определения для сущностей их атрибутов, описаний и ограничений, частично уточняет состав сущностей и взаимосвязи между ними

Логическая модель данных

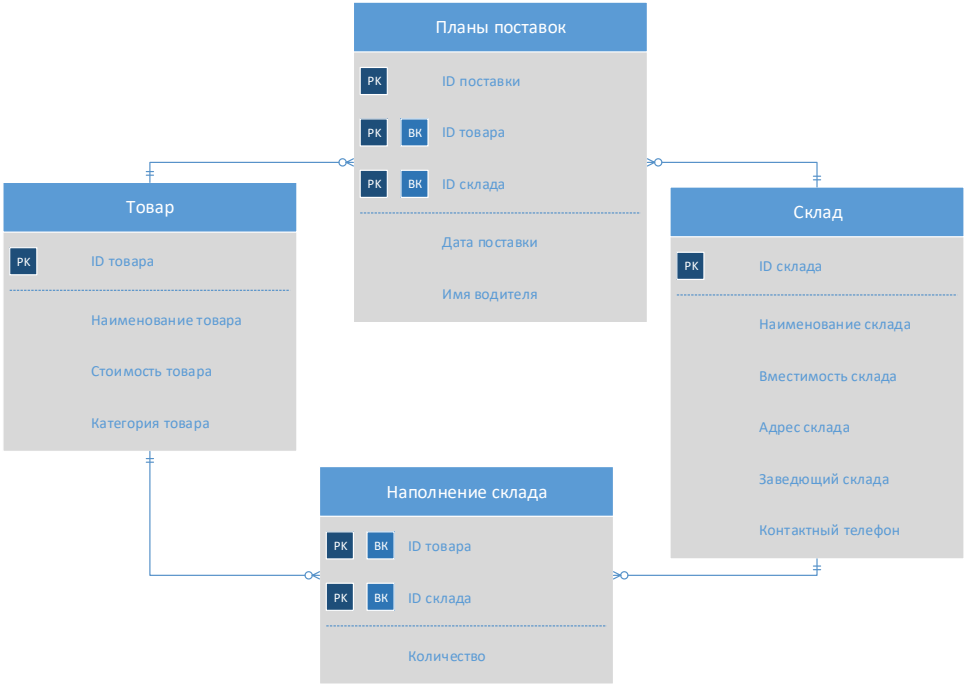
- Разрешает выход за рамки концептуальной модели
- Является прототипом будущей физической модели
- Не учитывает специфику какой-либо конкретной СУБД

- Для иллюстрации также используется ER-нотация

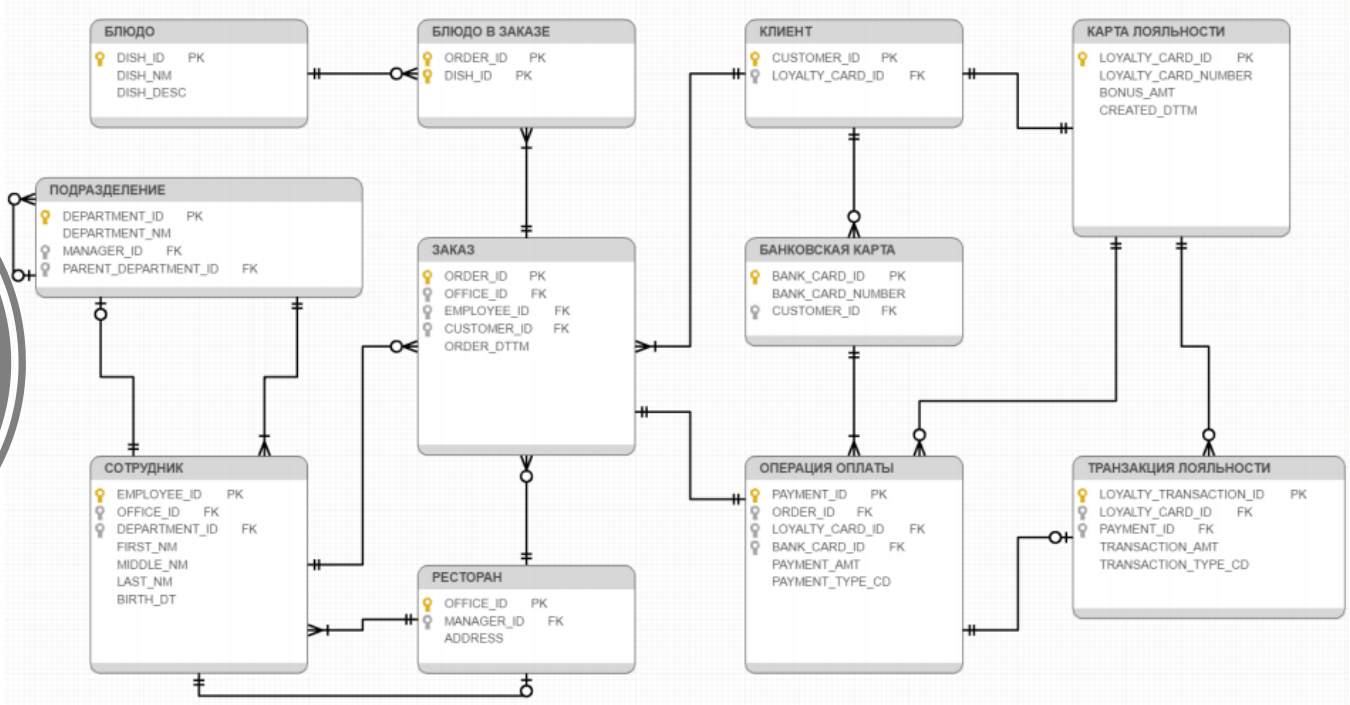
ER-нотация «Сущность»



Примеры ER-диаграмм: «Пятерочка»



Пример логической модели данных



Домашнее задание

1. Поставить и поиграться с PeeWee
2. Выполнить задание из лекций по БД (часть 1) с помощью ORM
3. Создать тестовые таблицы из раздела Индексы, попробовать добиться похожих рез-тов
4. Выбрать один из 6 типов реализации индексов в PostgreSQL, написать краткую выжимку о механизме работы (не больше листа А4)
5. Создать тестовые таблицы из раздела План запроса, попробовать добиться похожих рез-тов
6. Придумать логическую схему хранения данных для простого приложения (4-5 сущностей). Кратко описать, что это за приложение; описать, почему схема спроектирована именно так.

СПАСИБО!