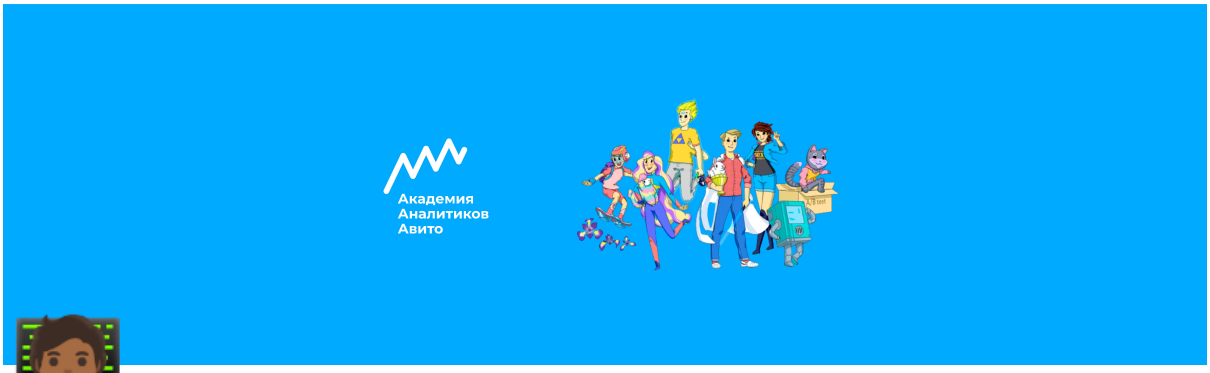


Создание таблиц

Синтаксис

Общий синтаксис запроса на создание таблицы выглядит следующим образом:

```
create table <schema>.<name> (
  <column1> <column1_type> <constraints>,
  <column2> <column2_type> <constraints>,
  ...
);
```

В зависимости от базы данных могут применяться дополнительные секции, влияющие на способ
хранения данных внутри. Об этом мы поговорим в следующий раз. Пример запроса:

```
create table learning.data (
  id int,
  address varchar,
  price int
)
```

Схема и таблица

Почти во всех базах данных есть понятие схемы (англ. schema) -- способ логического объединения таблиц вместе. Создать схему можно очень простым запросом:

```
create schema <schema_name>;
```

В нашем примере будет создана таблица data внутри схемы learning .

Внимание. Все запросы ниже можно выполнять и без указания схемы, тогда все таблицы будут браться из схемы по-умолчанию (в PostgreSQL это схема postgres).

Колонки

Внутри запроса колонки объявляются списком через запятую, сначала указывая название колонки, а потом ее тип. Мы совсем чуть чуть говорили про типы данных, поэтому приведем тут базовые типы, которые есть почти во всех СУБД.

Тип	**Значение**
INT , INTEGER	Целое число. Обычно вмещает в себя числа в диапазоне (-2^32, 2^32-1).
BOOL , BOOLEN	Логический тип. Может принимать два значения -- true , false .
FLOAT	Число с плавающей запятой. Может хранить только 24 бита значачих цифр и 8 бит экспоненты в формате <24-bit number> * 2 ^ <exponent> .
DOUBLE	Число с плавающей запятой удвоенной точности. 53 значащих бита и 11 бит экспоненты.
VARCHAR , CHARACTER VARYING	Строка переменной длины. Обычно в скобках после этого типа пишут максимальное допустимое число символов в строке. Например, тип varchar(255) может хранить 255 символов.
DATE	Тип, хранящий только дату
TIME	Тип, хранящий только определенную отметку времени в сутках.
DATETIME	Тип, хранящий дату и время вместе
TIMESTAMP	Тип, хранящий отметку времени в виде количества секунд, прошедших с 1 января 1970 года в вите 32-битного числа.

В нашей таблице будет 3 колонки. Первая колонка называется **id** , внутри мы будем хранить числа. Во второй колонке **-- address** с типом строки переменной длины. Мы не указали максимально допустимое число символов, поэтому будет использоваться значение по умолчанию -- 255. Третья колонка -- цена, принимающая значения из типа целых чисел.

Изменение структуры колонок

Чтобы поменять добавить или удалить колонку - нужно воспользоваться командой

```
-- общий вид
alter table <schema>.<name> add column <column_name> <data_type>;
alter table <schema>.<name> drop column <column_name>;

--примеры
alter table learning.data add column id integer; -- добавление
alter table learning.data add column event_date date not null; -- добавление с ограничением
alter table learning.data drop column id; -- удаление
```

Ограничения

Правилом хорошего тона является прописывание максимального количества ограничений на ваши данные. Ограничение это какое-то условие, которое должно выполняться для всех данных внутри таблицы.
Есть несколько видов ограничений. Например, проверка на то, что поле не является пустым:

```
create table learning.data (
id integer not null,
...
)
```

В таком случае как только кто-нибудь попытается вставить в таблицу строку без id, то он сразу получит ошибку.
Можно написать более хитрую и сложную проверку:

```
create table learning.data (
id integer not null,
address varchar check (address in ('moscow', 'saint-petersburg'))
)
```

Внутри скобок после ключевого слова check можно написать любое логическое условие, которое использует название текущей колонки и константы.
Можно написать проверку на несколько столбцов:

```
create table learning.data (
id integer not null,
address varchar,
price integer not null,
discount double not null,
check (price * (1 + discount) > 100)
)
```

Проверка на то, что значение поля уникально по строкам:

```
create table learning.data (
id integer not null unique
)
```

Или после списка колонок:

```
create table learning.data (
id integer not null,
unique (id)
)
```

Если колонка является первичным ключом, то ее можно указать таковой с помощью NOT NULL UNIQUE, либо с помощью PRIMARY KEY:

```
create table learning.data (
id integer primary key
)
```

Внешний ключ можно указать с помощью REFERENCES:

```
create table learning.stores (
id integer primary key,
customer_id integer references learning.customers(id)
)
```

В скобках здесь указывается колонка, на которую ссылается данная колонка, то есть колонки

stores.customer_id и customers.id являются идентифиакторами одной сущности. Есть также еще другие способы создания таблиц.

Создать таблицу с такой же структурой

Допустим у нас есть таблица learning.data , чью структуру мы хотим повторить в другой таблице. Тогда нам достаточно написать следующий запрос:

```
create table learning.another_data like learning.data;
```

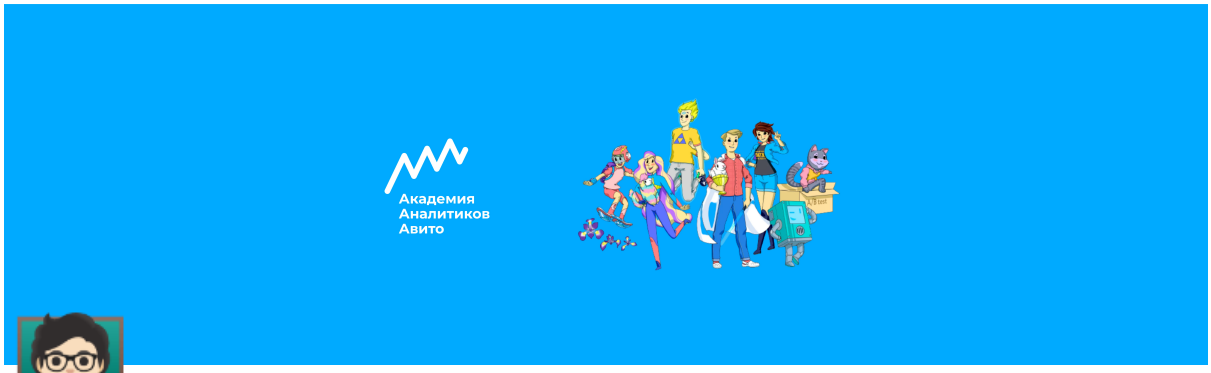
Структура таблицы learning.another_data будет полностью повторять нашу исходную таблицу вплоть до всех указанных ограничений.

Создать таблицу со структурой, получающейся из select-запроса

```
create table learning.another_data
as
select 1 as launch_id, *
from learning.data;
```

Лайфхак: если нужно только создать таблицу и не наполнять ее данными, то можно провернуть трюк с limit 0:

```
create table learning.another_data
as
select ...
from ...
limit 0
```



Вставка и обновление данных

Вставка данных

Допустим мы сделали табличку, теперь нам надо вставить в нее данные. За это отвечает запрос insert.
Вставка может происходить несколькими способами.

Вставка через VALUES

Синтаксис выглядит вот так:

```
insert into learning.data values
(1, 'moscow', 100);
```

Этот запрос вставит в таблицу learning.data одну строку. Выполнив select после вставки получим следующее:

```
=> select * from learning.data;
id | address | price
----+---------+-------
1 | moscow | 100
(1 row)
```

insert select

В таблицу также можно вставить результат запроса:

```
insert into learning.data
select 2, 'saint-petersburg', 200;
```

Изменение порядка колонок

В случае, когда порядок колонок в таблице и в запросе не совпадает и мы не хотим редактировать запрос, то мы можем в скобках после названия таблицы, куда вставлять данные, перечислить список колонок, соответствующих колонкам в запросе. Пример:

```
insert into learning.data (address, price, id)
select 'ekaterinburg', 300, 3;
```

В этом случае база будет вставлять первую колонку из результата запроса в колонку address , вторую — в price , третью — в id . Список колонок для вставки можно указать в любом запросе insert. Если список колонок не указан, то используется порядок, указанный при создании таблицы. Эта фича позволяет упростить код вставки данных. Допустим, у нас есть несколько источников, их которых мы загружаем одинаковые данные, но, из-за особенности выгрузки, порядок колонок в них не совпадает. Тогда мы можем для каждого источника перечислить порядок колонок в этом источнике и скормить это базе, а она за нас сама правильно разложит данные по колонкам.

Обновление данных

Часто возникает ситуация, когда данные уже загружены в базу, но их нужно обновить. Здесь нам на помощь придет запрос update . Синтаксис запроса следующий:

```
update <schema>.<name>
set column1 = value1,
column2 = value2
...
where <filter_expression>
```

База сначала отфильтрует из таблицы данные, соответствующие условию where, а затем во всех оставшихся строках заменит значение указанных

колонок на указанные значения. Пример:

```
update learning.data
set price = price * 2
where id = 1
```

Такой запрос удвоит значение колонки price у всех строк с id = 1.
Внутри указания значения можно написать подзапрос:

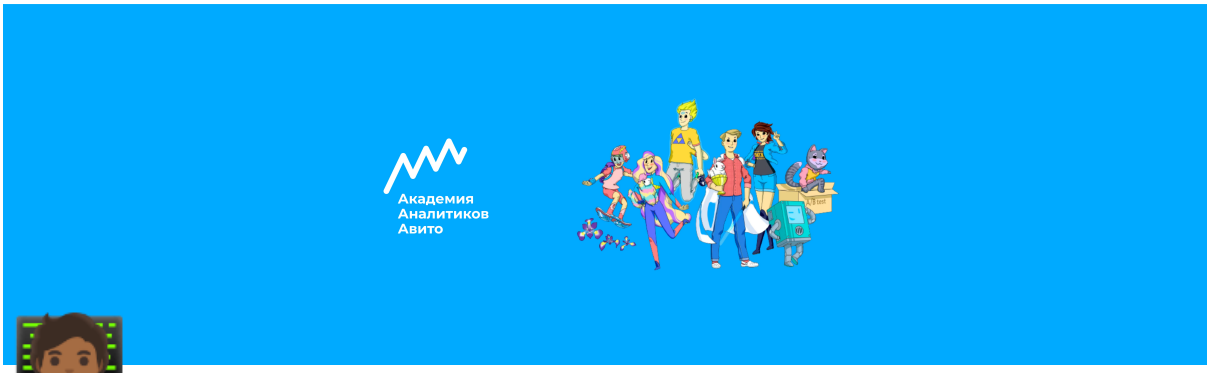
```
update learning.data
set price = (select price from learning.catalog_prices where catalog_prices.address =
learning.data.address)
```

В таком случае на каждую строку обновляемой таблицы (уже после применения where) будет сделан запрос в связанную таблицу, что дает просадку в производительности. Для того, чтобы избежать большого числа запросов, можно сделать следующее:

```
update learning.data dt
set price = catalog.price
from learning.data left join learning.catalog_prices catalog on learning.data.address =
catalog.address
```

Здесь мы присоединяем к обновляемой таблице таблицу-справочник и используем колонку из таблицы — справочника чтобы установить значение в колонке обновляемой таблицы.

Важный момент: для того, чтобы это правильно работало, надо убедиться в том, что в результате выполнения соединения не произойдет размножения строк у обновляемой таблицы. Для этого достаточно выполнять left join по уникальному ключу.



Удаление данных и транзакции

Удаление данных

Самый простой способ почистить данные в таблице -- выполнить запрос truncate :

```
truncate table learning.data;
```

Этот запрос удалит разом все строки из таблицы learning.data.
Если же мы хотим удалить только определенные строки из таблицы, мы можем использвать запрос delete :

```
delete from learning.data
where ...
```

Внутри секции where можно, помимо обычного логического выражения, написать связанный подзапрос. В таком случае удалятся только подходящие под условие строки. Чтобы проверить, что удалятся только нужные строки, можно заменить слово delete на select * .

Транзакции

При работе баз данных нормальной является ситуация, когда разные акторы (люди или приложения) работают с данными одновременно. В таком случае у нас появляется маленькая проблема -- что делать, если, например, два человека одновременно обновляют одни и те же даные?
Рассмотрим следующий пример: два одновременно выполняющихся запроса

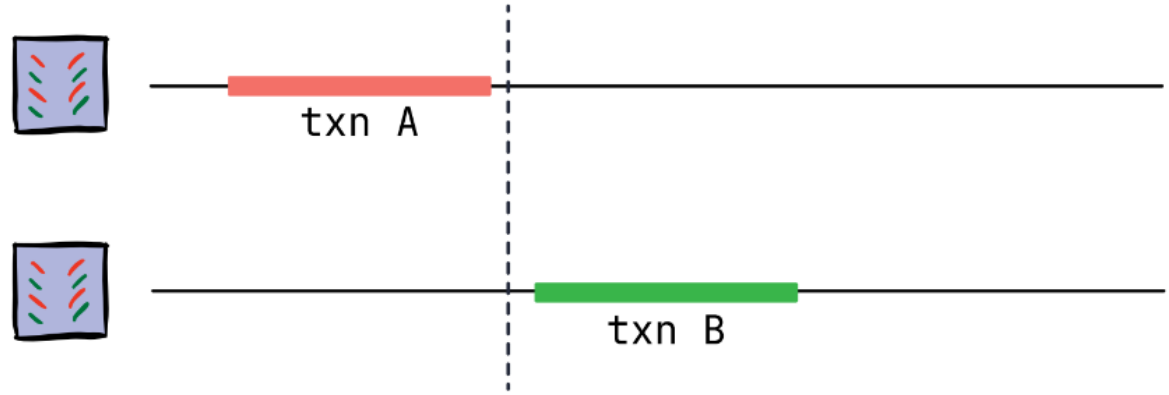
```
- Person A
update learning.data
set price = 300
where id = 1
```

```
- Person B
update learning.data
set price = 400
where id = 1
```

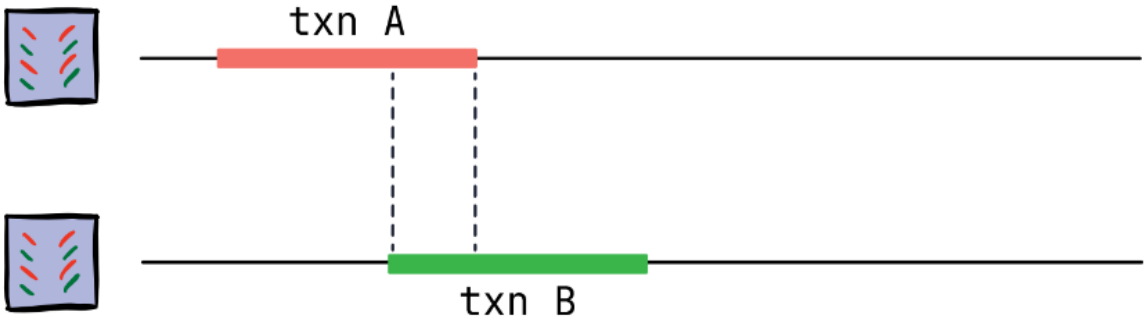
Здесь встает вопрос -- какое изменение нужно считать финальным? Нет одного правильного ответа на этот вопрос: разрешение конфликта обновлений зависит от бизнес-задачи, которую решает база данных.
Благо база данных предоставляет механизм для разрешения конфликтов -- транзакция.
Транзакцию можно определить как набор sql-выражений, которые либо применяются вместе и полностью, либо не применяются в целом. Прочтите это определение еще раз.
Концептуально происходит следующее: во время начала транзакции база запоминает текущее состояние и смотрит на выражения, которые хочет выполнить пользователь. В зависимости от выбранного уровня изоляции транзакций их поведение будет разным, об этом ниже.
Для того, чтобы сказать базе, что набор выражений является транзакцией, нужно обозначить это через специальную команду:

```
begin; -- Начало транзакции
update learning.data -- Выражение под транзакцией
set price = 300
where id = 1;
commit; -- Конец транзакции
```

Допустим, что два запроса из примера выше оба выполняются под транзакцией. Есть несколько вариантов развития событий.



Вариант первый: транзакции не пересекаются, то есть первая транзакция закончится до того, как начнется вторая. В таком случае сначала цена будет изменена на 300, а затем на 400.



Вариант второй, более интересный: вторая транзакция начнется до того, как первая закончится. В этот момент у нас возникает конфликт: какие данные должна увидеть транзакция B: исходную цену 200 или уже измененную цену в 300? В зависимости от того, что выберет программист, транзакция либо завершится с ошибкой и никакие изменения в базу не попадут; либо она увидит цену в 300. Такие варианты развития событий называются уровнем изоляции (isolation level) транзакций.

Первый вариант называется serializable уровень изоляции. Его задача — упорядочить транзакции в базе так, чтобы результат их выполнения не отличался от строго последовательного выполнения, то есть спрятать весь параллелизм. Это дает очень понятную семантику — если две транзакции конфликтуют друг с другом так, что не понятно, как их можно было бы применить последовательно, то одна из таких транзакций сваливается с ошибкой. Если очень обобщить — результаты выполнения транзакций будут такими, будто вообще транзакций нет и у нас есть только один пользователь.

Второй вариант называется read commited. Его суть состоит в следующем — транзакция видит только те изменения, чьи транзакции уже завершились. То есть, если посередине нашей транзакции соседняя транзакция завершится, то мы увидим ее изменения. Тогда, возвращаясь к нашему примеру, итоговое значение будет равно 400.

Все операции манипуляции с данными можно и нужно проводить в рамках транзакции. Если ваше приложение работает с бизнес-критичными данными, то имеет смысл поставить уровень изоляции serializable — пожертвовать производительностью, но зато иметь семантику "начав транзакцию данные внутри транзакции не поменяются". Если же вам это не нужно — смело используйте read committed. В случае использования базы как хранилища, например, аналитических событий — транзакции будут только мешать — ничего кроме дописывания в конец мы и так не делаем.