GeekBrains

Базы данных. Интерактивный курс

# Урок 2

## Язык запросов SQL



© geekbrains.ru

# Введение в SQL

Иерархические и сетевые базы данных зачастую не имели собственного языка запросов, при помощи которых можно было бы в интерактивном режиме общаться с СУБД. Для этого приходилось писать свои собственные программы, задействуя библиотеки и бинарные протоколы.

Реляционные СУБД одни из первых предоставили язык запросов SQL, которым мы, в обогащенном варианте, пользуемся по сегодняшний день. За 40 лет SQL вырос в сегмент рынка в десятки миллиардов долларов и стал стандартом де-факто для баз данных. Даже набирающее обороты новое поколение NoSQL-баз данных, зачастую старается следовать традициям и синтаксису SQL.

SQL зародился в IBM и изначально назывался SEQUEL (Structured English Query Language), однако торговая марка была занята и название языка пришлось сократить до SQL (Structured Query Language).

## Достоинства SQL

SQL — это декларативный, а не процедурный язык: с его помощью мы описываем цель, конечный результат, который хотим получить, а не инструкцию по выполнению алгоритма, как обычно поступают в императивных или объектно-ориентированных языках, таких как Python или Java.

Инструкции SQL похожи на обычные предложения английского языка, что упрощает их изучение и понимание.

В основе SQL лежит теория множеств, язык изначально спроектирован для эффективной обработки большого объема данных. По сей день SQL является одним из наиболее быстрых способов обработки большого количества записей.

Все ведущие поставщики реляционных СУБД используют SQL. Архитектура каждой базы данных может сильно отличаться, однако сам язык и его поведение остаются неизменными.

Большинство СУБД реализовано для множества самых разнообразных платформ, поэтому программы на языке SQL не зависят от операционных систем, а часто и от базы данных.

Официальный стандарт языка SQL был опубликован в 1986 году, после чего он был многократно расширен, сначала в 1989 году, а затем — в 1992, 1999, 2003 и 2006 годах.

## Недостатки SQL

SQL — это слабо структурированный язык, особенно по сравнению с Python, Ruby и Java. Инструкции SQL напоминают обычные предложения естественного языка и содержат «слова-пустышки», не влияющие на смысл инструкции, но облегчающие ее чтение.

Язык этот очень старый и формировался во времена, когда стоимость компьютеров и компьютерного времени многократно превышала заработную плату обслуживающих их программистов. Поэтому с современной точки зрения он не очень удобен и заставляет разработчика думать как машина, подстраиваться под нее. Все это выливается в большое количество времени на разработку и сопровождение программ. Это большой недостаток в современных реалиях, когда компьютеры сильно подешевели и заработная плата разработчиков стала составлять значительную часть бюджета проекта.

SQL плохо ложится на иерархическую и графовую природу объектно-ориентированного программирования, составляющего львиную долю современной разработки. В связи с этим большую

популярность стали приобретать различные ORM-системы, которые предоставляют ООП-интерфейс, скрывая за объектами и методами библиотек реляционную природу базы данных.

Язык не универсален. Это не полноценный компьютерный язык вроде Python, Ruby или Java, т.е., с его помощью нельзя создать независимую программу в бинарном коде.

Несмотря на стандарты, каждая СУБД реализует свой собственный диалект. Производители баз данных идут вперед стандарта, реализуя какую-то новую возможность, например хранимые процедуры. Конкуренты создают похожую функциональность, часто намеренно отличающуюся от оригинала, чтобы избежать судебных тяжб.

Когда особенность становится повсеместно используемой, комитет стандартизации вносит ее в стандарт, зачастую в форме, несовместимой со всеми остальными, чтобы не давать никому конкурентного преимущества. Производители реализуют стандарт, но оствляют старый вариант для обратной совместимости. Так появляются диалекты SQL.

На прошлом уроке мы пользовались операторами **SHOW**, **DESCRIBE** и информационной схемой. Информационная схема — часть стандарта SQL, вы обнаружите ее и в других реляционных базах данных. Операторы **SHOW** и **DESCRIBE** не стандартны, это особенность СУБД MySQL.

# Элементы языка

Язык состоит из множества компонентов, с которыми мы познакомимся в течение курса. Сложность заключается в том, что SQL — декларативный язык со специальной структурой. Знания других языков слабо помогают в его изучении. Поэтому придется приложить значительные усилия на его освоения.

## Комментарии

Язык SQL, помимо инструкций, поддерживает комментарии, которые позволяют задать игнорируемые участки SQL-программы. Однострочные комментарии начинаются с двух символов дефиса.

```
-- SELECT "Hello world!";
```

Можно использовать многострочные комментарии, которые начинаются с последовательности **/***, а завершаются — ***/**.

```
/* Это комментарий
SELECT "Hello world!"; */
```

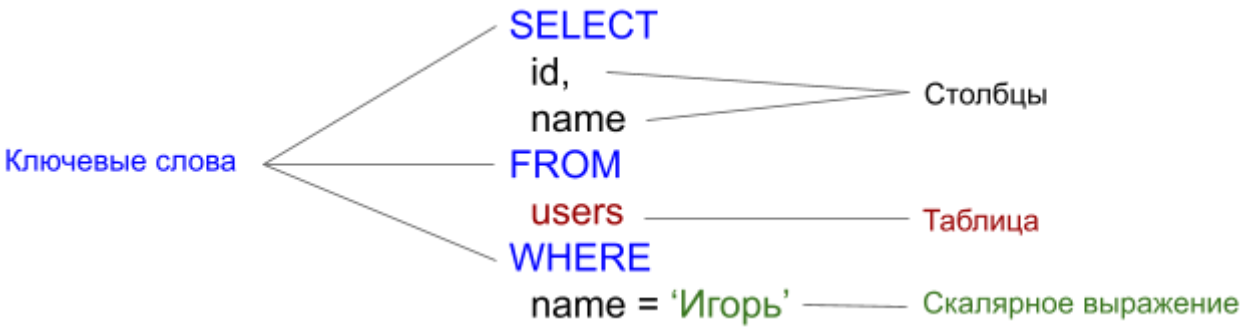
## DDL и DML

Инструкции в SQL можно разделить на две большие группы: язык описания данных (DDL) и язык управления данными (DML).

К первой группе — DDL — относятся инструкции создания, удаления и редактирования баз данных и таблиц. Ко второй — DML — запросы на создание, редактирование, удаление и извлечение данных из таблиц.

## Структура запроса

Каждая инструкция SQL начинается с ключевого слова, которое описывает выполняемое действие.




После ключевого слова идет одно или несколько предложений. Предложение может описывать данные, с которыми работает инструкция, или содержать уточняющую информацию о действии, выполняемом инструкцией.

Каждое предложение также начинается с ключевого слова, такого как **WHERE** (где), **FROM** (откуда). Одни предложения в инструкции обязательны, а другие — нет. Конкретная структура и содержимое предложения могут изменяться. Многие предложения содержат имена таблиц или столбцов; некоторые из них могут содержать дополнительные ключевые слова и скалярные выражения.

## Скалярные выражения

Скалярные выражения — это числа и строки. Фактически это константы.

```
SELECT "Hello world!";
```

В запросе выше строка Hello world — скалярное выражение. Число 2 тоже является скалярным выражением.

```
SELECT 2 + 2;
```

Попробуем вместо «Привет, мир!» написать «Мир рубистов»:

```
SELECT 'Rubist's world';
```

Мы не сможем завершить оператор — клиент **mysql** запутается. Исправить ситуацию мы можем, экранировав символ одиночной кавычки:

```
SELECT 'Rubist\'s world';
```

Есть и другой способ кодирования данной строки: мы можем воспользоваться двойными кавычками:

```
SELECT "Rubist's world";
```

Двойными кавычками не следует увлекаться, так как в некоторых СУБД, например, в PostgreSQL, у них специальное назначение и можно нарушить совместимость вашей SQL-программы.

Имена баз данных, таблиц и столбцов могут включая любые символы, за исключением '/', '\' и '.'. Если имя совпадает с ключевым словом, его необходимо заключать в обратные кавычки: `create`.

```
CREATE TABLE tbl (`create` INT);
```

В стандарте SQL определен набор зарезервированных ключевых слов, которые используются в инструкциях SQL. В соответствии со стандартом, зарезервированные ключевые слова нельзя использовать  для именования объектов базы данных, таких как таблицы, столбцы и пользователи. Во многих реализациях СУБД этот запрет ослаблен, например, в MySQL, где допускается создание таблиц и столбцов из списка зарезервированных ключевых слов. Чтобы интерпретатор не путался, такие имена необходимо заключать в обратные кавычки.

# Типы данных

MySQL поддерживает несколько типов данных, которые можно разбить на пять групп:

- числовые данные (целые, вещественные или с плавающей точкой);
- строковые данные (фиксированного и переменного размера);
- специальный тип NULL, которое обозначает неопределенное значение, отсутствие информации;
- календарные данные предназначены для сохранения даты и времени;
- коллекционные типы позволяют сохранять множество значений или даже целые документы в виде JSON-полей.

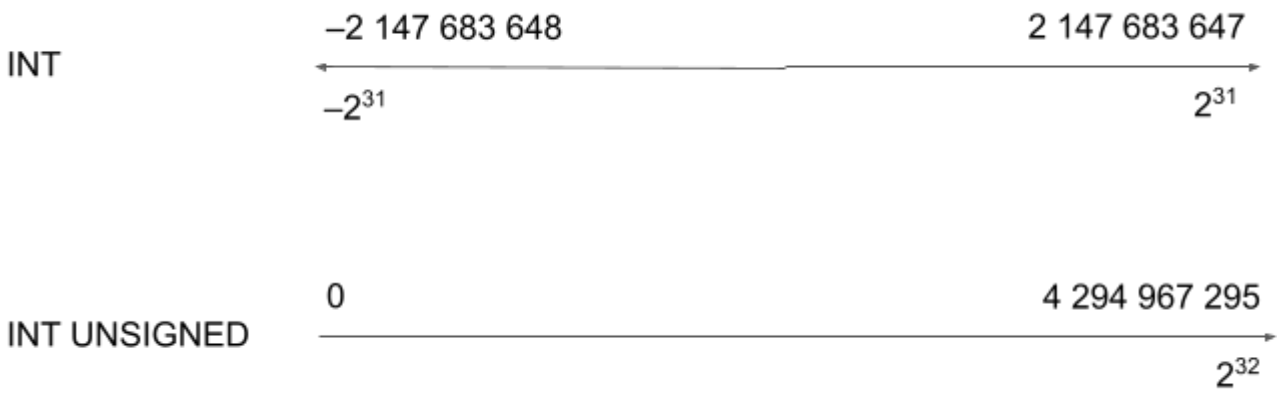
## Атрибуты

Типы определяют характеристики сохраняемых значений, а также количество памяти, которая под них отводится. Многие типы могут сопровождаться дополнительными атрибутами.

Например, атрибуты **NULL** или **NOT NULL** задают ограничение на столбец, позволяя присваивать элементам неопределенное значение или, наоборот, запрещая такое поведение.

Атрибут **DEFAULT** позволяет задать полю значение по умолчанию, которое будет присваиваться в случае, если при создании записи значение не задано.

Атрибут **UNSIGNED** относится только к числовым значениям. Поле с таким атрибутом теряет возможность хранить отрицательные значения. Для многих полей, например, первичного ключа, отрицательные значения не требуются. Под кодирование знака отводится один бит. Отказ от него позволяет его высвободить под кодирование числа и увеличить максимально допустимый диапазон.

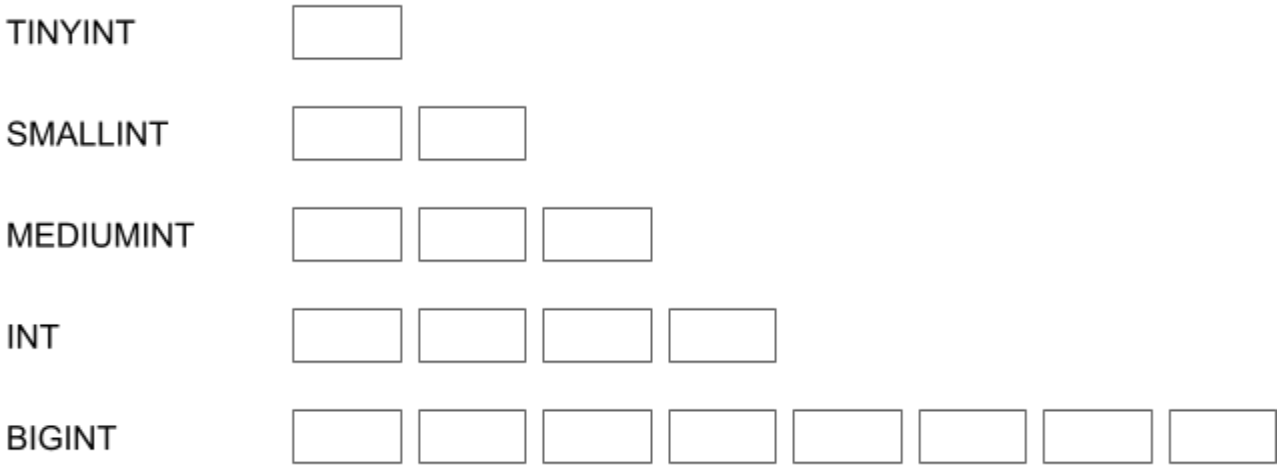



## Числовые типы

Числовые типы можно разделить на:

- целочисленные;
- вещественные, т.е., числа с плавающей точкой;
- точные — **DECIMAL**.

Целые числа обрабатываются быстрее всех, вещественные чуть медленнее, точные медленнее всех, так как это фактически строка, в которую записано число.




В MySQL предусмотрено целых 5 целых типов, прямоугольниками на рисунке показывается, сколько байт отводится под каждый из типов. Чем больше места занимает тип, тем объемнее будет конечная таблица и тем больше данные будут занимать места на жестком диске и в оперативной памяти. Кроме того, чем больше байт отводится под число, тем больший диапазон оно может обслуживать.

В **TINYINT** один байт, т. е., 8 бит, максимальное значение, которое он может обслуживать — от 0 до 2 в степени 8, т. е., 256. Большое число в поле этого типа поместить не получится. Если при этом не используется атрибут **UNSIGNED**, то эту величину следует поделить пополам и допустимый диапазон — от -128 и до 127.

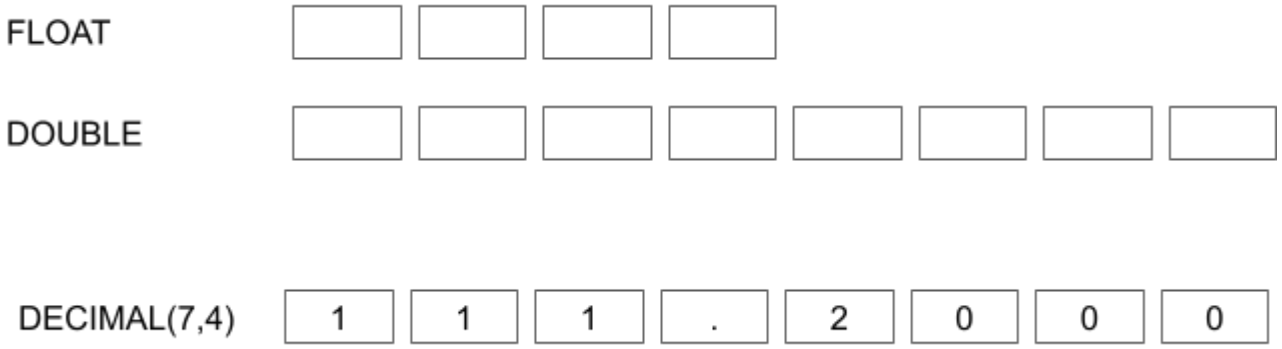
```
CREATE TABLE tbl (id INT(8));
```

При объявлении целого типа в круглых скобках можно задать количество отводимых под число символов. Это необязательное указание количества выводимых символов используется для дополнения пробелами слева. Однако ограничений ни на диапазон величин, ни на количество разрядов не налагается.

```
INSERT INTO tbl VALUES (5);
SELECT * FROM tbl;

DROP TABLE IF EXISTS tbl;
CREATE TABLE tbl (id INT(8) ZEROFILL);
INSERT INTO tbl VALUES (5);
INSERT INTO tbl VALUES (500000000);
```

Если количества символов, необходимых для вывода числа, будет недостаточно, под столбец будет выделено больше символов. Если дополнительно указан необязательный атрибут **ZEROFILL**, свободные позиции по умолчанию заполняются нулями слева.




Среди вещественных чисел различают **FLOAT**, который занимает 4 байта, и **DOUBLE**, занимающий 8 байт. Так как не вещественные числа можно закодировать при помощи двоичного кода, вычисления с участием вещественных чисел приводит к накоплению ошибок. Ряд областей, например, работа с деньгами очень чувствительна к таким ошибкам.

Поэтому в SQL предусмотрен специальный тип **DECIMAL**, в нем число хранится в виде строки, обрабатывается такой тип данных сильно медленнее, чем остальные числа, зато не теряется точность. Требуемая точность задается при объявлении столбца данных одного из этих типов.

На рисунке выше представлена схема типа **DECIMAL**, в котором под все число отводится 7 байт, а под дробную часть — 4 байта. Поместить сюда число больше 999 с четырьмя девятками после запятой уже не выйдет.

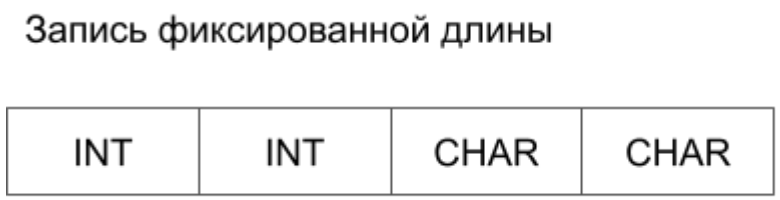
```
DROP TABLE IF EXISTS tbl;
CREATE TABLE tbl (price DECIMAL(8,4));
INSERT INTO tbl VALUES (111.2);
INSERT INTO tbl VALUES (10000.0);
ERROR 1264 (22003): Out of range value for column 'price' at row 1
```

# Строковые типы

Строковые типы можно условно разделить на:

- фиксированные строки, которые задаются типом **CHAR**, если при создании таблицы отводится 40 символов — именно столько памяти и займет запись;
- переменные строки, которые задаются типом **VARCHAR**, не имеют фиксированного размера, занимаемый объем определяется размером строки; впрочем, допускается задание максимального объема строки в круглых скобках после запятой;
- BLOB-типы, которые изначально задумывались для хранения объемных бинарных данных, однако в результате были адаптированы для хранения текстовых значений.

```
DROP TABLE IF EXISTS tbl;
CREATE TABLE tbl (
  name CHAR(10) DEFAULT 'anonymous',
  description VARCHAR(255)
);
INSERT INTO tbl VALUES(DEFAULT, 'Новый пользователь');
SELECT * FROM tbl;
INSERT INTO tbl (description) VALUES('Еще один пользователь');
INSERT INTO tbl VALUES('Очень длинное имя пользователя', 'Еще один
пользователь');
```

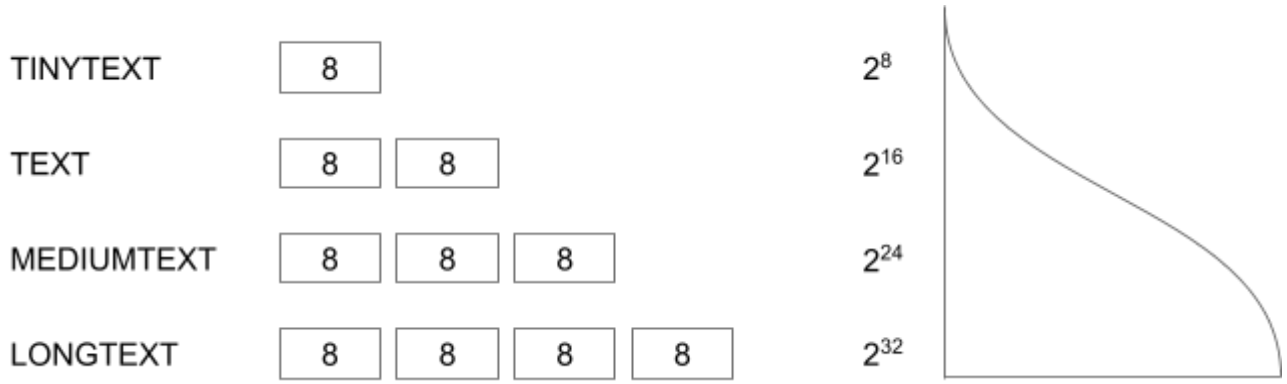




Строковые типы имеют ограничения, записи в таблице представлены в виде структуры фиксированного размера. Это позволяет быстро переходить к нужной записи, так как размер известен заранее и мы можем перемещать указатель на нужный адрес. Под столбцы переменной длины отводится специальная область длинной 65 536 байт. Таким образом, нельзя в таблице создать столбцы **VARCHAR**, совокупный размер которых больше, чем эта специальная область.

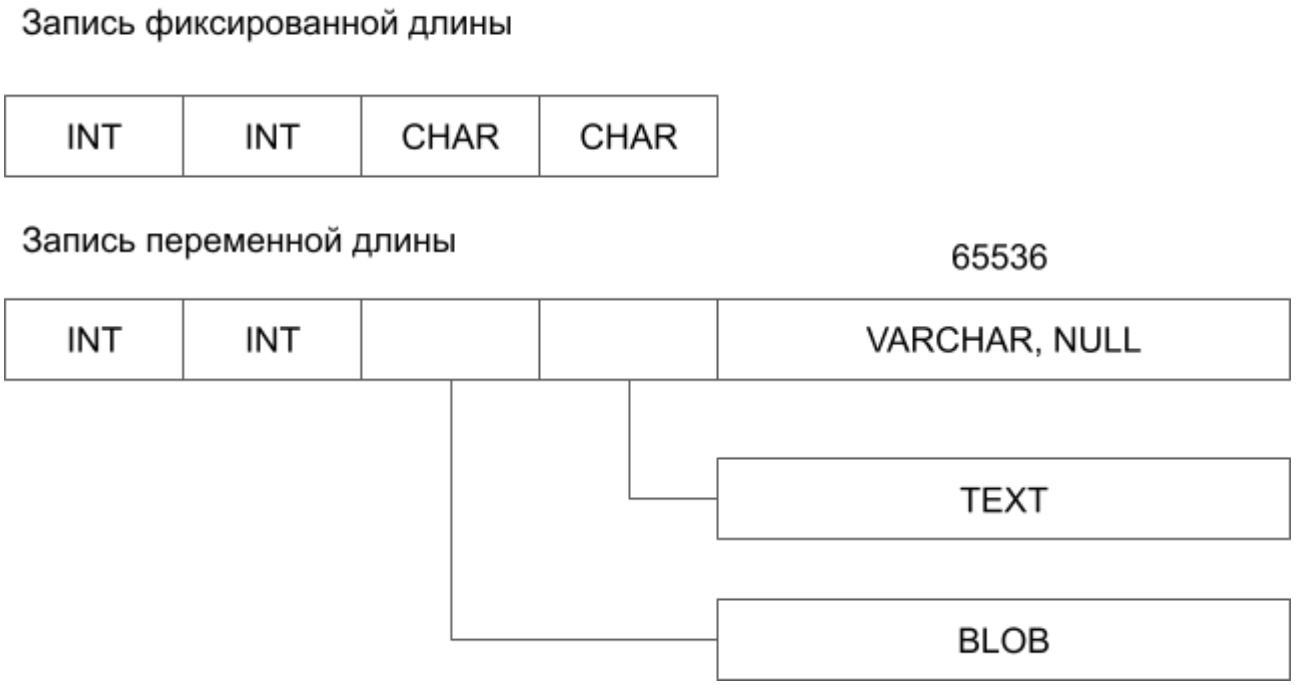
Так как для кодирования строк используется UTF-8, ситуация еще более печальная: для символов, отличных от английских, зачастую используется больше одного байта, например русский текст кодируется двумя байтами.

В MySQL нет полноценной поддержки UTF-8, поэтому под все символы этой кодировки отводится либо 3, либо 4 байта. Таким образом, в VARCHAR-значениях можно уместить лишь очень короткие строки.




Поэтому для хранения объемного текста используется тип **TEXT**, который, как и INT, имеет несколько модификаций. На рисунке прямоугольниками указывается количество байт, которые используются для адресации внутри текстовой строки. Таким образом, при помощи семейства TEXT-типов можно закодировать как короткие строки на 256 символов, так и объемный текст вплоть до 4 Гб.

Тип **TEXT** адаптирован под хранение текста из BLOB-типа, который используется для хранения бинарных данных. Впрочем, **BLOB** используется исключительно редко, гораздо чаще база данных используется для хранения текста.




Типы **TEXT** и **BLOB** еще медленнее, чем **VARCHAR**, так как они хранятся в отдельной области памяти, отделенной от данных основной таблицы. Поэтому при обращении к ним MySQL вынуждена осуществлять поиск этих данных и соединять их с основными данными записи. Поэтому прибегать к ним следует только, когда размера **VARCHAR** не хватает.

# Учебная база данных

Теперь мы готовы создавать учебную базу данных. Напомним, что мы создаем интернет-магазин в базе данных **shop**. Давайте создадим файл **shop.sql** и будем размещать в нем все наши наработки.

```
DROP TABLE IF EXISTS catalogs;
CREATE TABLE catalogs (
  id INT UNSIGNED,
  name VARCHAR(255) COMMENT 'Название раздела'
) COMMENT = 'Разделы интернет-магазина';

DROP TABLE IF EXISTS users;
CREATE TABLE users (
  id INT UNSIGNED,
  name VARCHAR(255) COMMENT 'Имя покупателя'
) COMMENT = 'Покупатели';

DROP TABLE IF EXISTS products;
CREATE TABLE products (
  id INT UNSIGNED,
  name VARCHAR(255) COMMENT 'Название',
  desription TEXT COMMENT 'Описание',
  price DECIMAL (11,2) COMMENT 'Цена',
  catalog_id INT UNSIGNED
) COMMENT = 'Товарные позиции';

DROP TABLE IF EXISTS orders;
CREATE TABLE orders (
  id INT UNSIGNED,
  user_id INT UNSIGNED
) COMMENT = 'Заказы';

DROP TABLE IF EXISTS orders_products;
CREATE TABLE orders_products (
  id INT UNSIGNED,
  order_id INT UNSIGNED,
  product_id INT UNSIGNED,
  total INT UNSIGNED DEFAULT 1 COMMENT 'Количество заказанных товарных позиций'
) COMMENT = 'Состав заказа';

DROP TABLE IF EXISTS discounts;
CREATE TABLE discounts (
  id INT UNSIGNED,
  user_id INT UNSIGNED,
  product_id INT UNSIGNED,
  discount FLOAT UNSIGNED COMMENT 'Величина скидки от 0.0 до 1.0'
) COMMENT = 'Скидки';

DROP TABLE IF EXISTS storehouses;
CREATE TABLE storehouses (
  id INT UNSIGNED,
```

```
  name VARCHAR(255) COMMENT 'Название'
) COMMENT = 'Склады';

DROP TABLE IF EXISTS storehouses_products;
CREATE TABLE storehouses_products (
  id INT UNSIGNED,
  storehouse_id INT UNSIGNED,
  product_id INT UNSIGNED,
  value INT UNSIGNED COMMENT 'Запас товарной позиции на складе'
) COMMENT = 'Запасы на складе';
```

# Тип NULL

SQL поддерживает специальные типы данных, среди них выделяется **NULL** — неизвестное значение:

```
SELECT NULL;
```

Все операции с NULL в качестве результата возвращают **NULL**.

```
SELECT NULL + 2;
```

Это довольно логично, так как любая операция с неизвестным значением, приводит к неизвестному результату. Давайте создадим простейшую таблицу, состоящую из одного целочисленного столбца **id**:

```
CREATE TABLE tbl (id INT UNSIGNED);
INSERT INTO tbl VALUES();
mysql> SELECT * FROM tbl;
+------+
| id   |
+------+
| NULL |
+------+
```

Мы можем запретить вставлять в поле **NULL** значения — для этого столбец следует снабдить атрибутом **NOT NULL**. Давайте для разнообразия не будем удалять таблицу, а попробуем ее преобразовать при помощи оператора **ALTER TABLE**.

```
ALTER TABLE tbl CHANGE id id INT UNSIGNED NOT NULL;
ERROR 1138 (22004): Invalid use of NULL value
```

Мы получаем ошибку, так как у нас уже есть запись с **NULL**, — придется очистить таблицу. Для этого можно воспользоваться оператором **TRUNCATE**.

```
TRUNCATE tbl;
```

Повторяем вызов **ALTER TABLE**:

```
ALTER TABLE tbl CHANGE id id INT UNSIGNED NOT NULL;
```

И смотрим структуру таблицы:

```
mysql> DESCRIBE tbl\G
*************************** 1. row ***************************
  Field: id
   Type: int(10) unsigned
   Null: NO
      Key:
Default: NULL
  Extra:
1 row in set (0,00 sec)
```

Обратите внимание: поле **Null** принимает значение **NO**.

```
INSERT INTO tbl VALUES();
```

Теперь вставить значение **NULL** в таблицу не получится

# Календарные типы

MySQL поддерживает пять типов календарных типов:

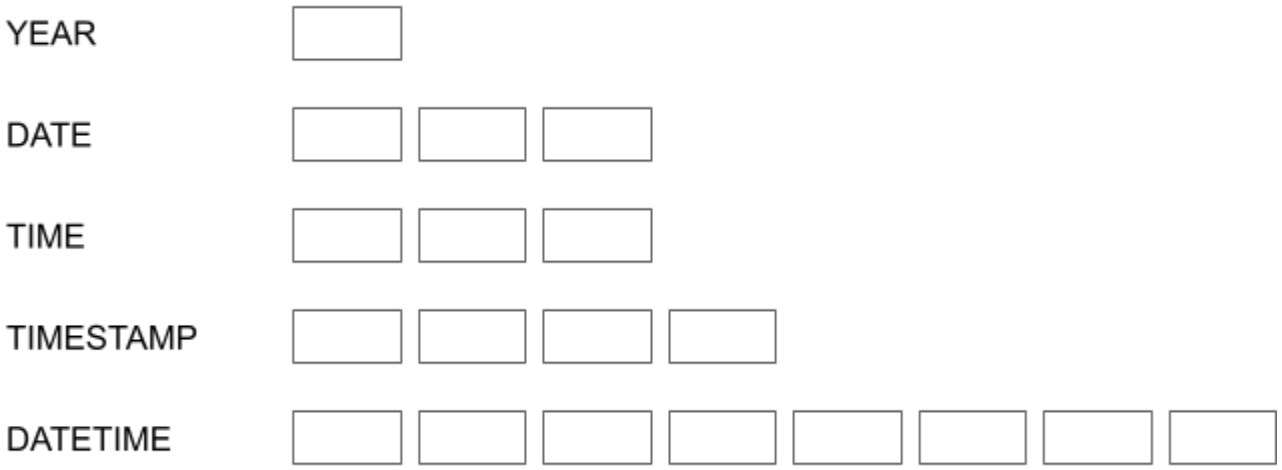
- **TIME** предназначен для хранения времени в течение суток;
- **YEAR** хранит год;
- **DATE** хранит дату с точностью до дня;
- **DATETIME** хранит дату и время;
- **TIMESTAMP** также хранит дату и время, занимает в два раза меньше места, чем **DATETIME**, но может хранить только ограниченные даты — в интервале от 1970 года до 2038;

Кроме того, первый TIMESTAMP-столбец в таблице обновляется автоматически при операциях создания и обновления. **TIMESTAMP** хранит дату в UTC-формате.

В таблице представлены календарные типы, именно в таких форматах MySQL возвращает календарные значения.

| Тип | Описание |
|---|---|
| YEAR | 0000 |
| DATE | '0000-00-00' |
| TIME | '00:00:00' |
| DATETIME | '0000-00-00 00:00:00' |
| TIMESTAMP | '0000-00-00 00:00:00' |

На рисунке прямоугольниками представлены количество байт, которые отводятся под поле каждого из календарных типов.




Если вы захотите задать дату, то сделать это можно при помощи следующего выражения:

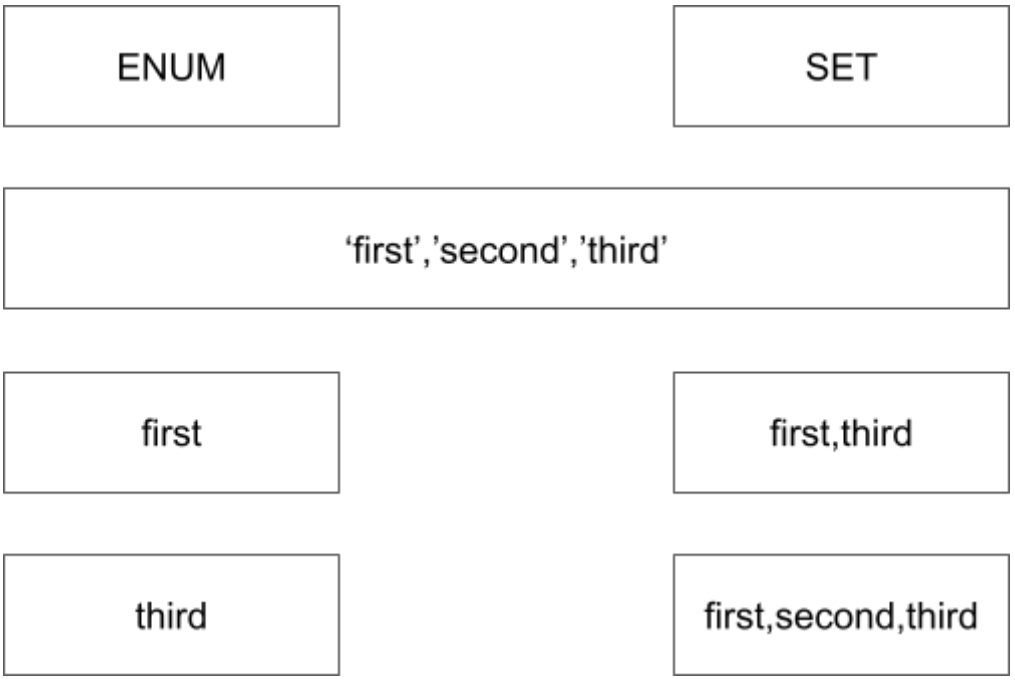
```
SELECT '2018-10-01 0:00:00';
```

С календарными типами можно проводить операции сложения и вычитания. Для этого используется специальная конструкция **INTERVAL**:

```
SELECT '2018-10-01 0:00:00' - INTERVAL 1 DAY;
SELECT '2018-10-01 0:00:00' + INTERVAL 1 WEEK;
SELECT '2018-10-01 0:00:00' + INTERVAL 1 YEAR;
SELECT '2018-10-01 0:00:00' + INTERVAL '1-1' YEAR_MONTH;
```

# Коллекционные типы

При объявлении списка допустимых значений **ENUM** и **SET** задаются списком строк, но во внутреннем представлении базы данных элементы множеств сохраняются в виде чисел. В случае **ENUM** поле может принимать лишь одно значение из списка. В случае **SET** — комбинацию заданных значений.




В последнее время большую популярность приобрел формат **JSON**, готовый объект языка JavaScript. Этот формат интенсивно используется для хранения и передачи коллекций. В MySQL предусмотрен столбец JSON-формата. Давайте добавим в таблицу **tbl** еще один столбец JSON-типа:

```
DESCRIBE tbl;
ALTER TABLE tbl ADD collect JSON;
DESCRIBE tbl;
```

Давайте вставим в это поле JSON-объект:

```
INSERT INTO tbl VALUES(1, '{"first": "Hello", "second": "World"}');
SELECT * FROM tbl;
SELECT collect->>"$.first" FROM tbl;
SELECT collect->>"$.second" FROM tbl;
```

Итак, давайте поправим базу данных нашего интернет-магазина с учетом полученных сведений:

```
DROP TABLE IF EXISTS catalogs;
CREATE TABLE catalogs (
  id INT UNSIGNED NOT NULL,
  name VARCHAR(255) COMMENT 'Название раздела'
) COMMENT = 'Разделы интернет магазина';

DROP TABLE IF EXISTS users;
CREATE TABLE users (
  id INT UNSIGNED NOT NULL,
  name VARCHAR(255) COMMENT 'Имя покупателя',
  birthday_at DATE COMMENT 'Дата рождения',
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) COMMENT = 'Покупатели';
```

```sql
DROP TABLE IF EXISTS products;
CREATE TABLE products (
  id INT UNSIGNED NOT NULL,
  name VARCHAR(255) COMMENT 'Название',
  desription TEXT COMMENT 'Описание',
  price DECIMAL (11,2) COMMENT 'Цена',
  catalog_id INT UNSIGNED,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) COMMENT = 'Товарные позиции';

DROP TABLE IF EXISTS orders;
CREATE TABLE orders (
  id INT UNSIGNED NOT NULL,
  user_id INT UNSIGNED,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) COMMENT = 'Заказы';

DROP TABLE IF EXISTS orders_products;
CREATE TABLE orders_products (
  id INT UNSIGNED NOT NULL,
  order_id INT UNSIGNED,
  product_id INT UNSIGNED,
  total INT UNSIGNED DEFAULT 1 COMMENT 'Количество заказанных товарных позиций',
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) COMMENT = 'Состав заказа';

DROP TABLE IF EXISTS discounts;
CREATE TABLE discounts (
  id INT UNSIGNED NOT NULL,
  user_id INT UNSIGNED,
  product_id INT UNSIGNED,
  discount FLOAT UNSIGNED COMMENT 'Величина скидки от 0.0 до 1.0',
  finished_at DATETIME NULL,
  started_at DATETIME NULL,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) COMMENT = 'Скидки';

DROP TABLE IF EXISTS storehouses;
CREATE TABLE storehouses (
  id INT UNSIGNED NOT NULL,
  name VARCHAR(255) COMMENT 'Название',
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) COMMENT = 'Склады';

DROP TABLE IF EXISTS storehouses_products;
CREATE TABLE storehouses_products (
```

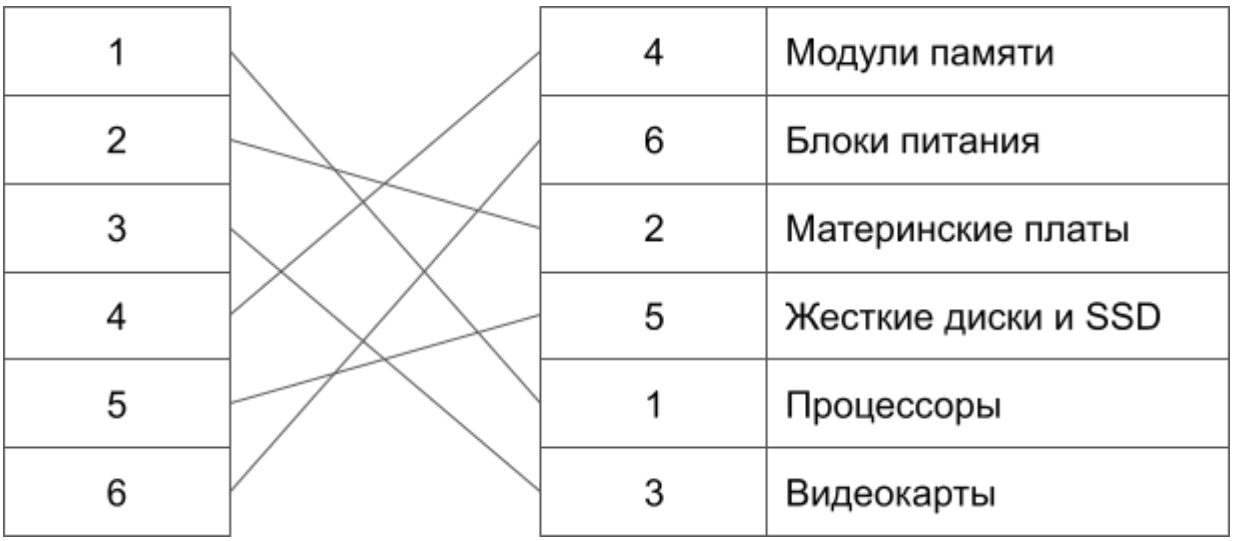
```
  id INT UNSIGNED NOT NULL,
  storehouse_id INT UNSIGNED,
  product_id INT UNSIGNED,
  value INT UNSIGNED COMMENT 'Запас товарной позиции на складе',
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) COMMENT = 'Запасы на складе';
```

# Индексы и ключи

Ключи нам с вами хорошо знакомы: это столбцы, при помощи которых мы добиваемся уникальности записей и связываем записи в разных таблицах.

Ключи очень часто снабжаются индексами, поэтому в массовом сознании разработчиков баз данных они часто сливаются в одно понятие. Тем не менее, индексировать можно не только столбцы с ключами, но и любой столбец или комбинацию столбцов таблицы.

Обычно записи в таблице располагаются в хаотическом порядке. Чтобы найти нужную запись, необходимо сканировать всю таблицу, на что уходит много времени. Идея индексов состоит в том, чтобы создать копию столбца, которая постоянно будет поддерживаться в отсортированном состоянии.




Это позволяет очень быстро осуществлять поиск по такому столбцу, т. к. заранее известно, где необходимо искать значение.

Обратная сторона медали — добавление или удаление записи требует дополнительного времени на сортировку столбца. Кроме того, создание копии увеличивает объем памяти, необходимый для размещения таблицы на жестком диске и в оперативной памяти сервера.

Именно поэтому СУБД не создает индексы для каждого столбца и их комбинаций, а отдает это на откуп разработчикам.

## Виды индексов

Существует несколько видов индексов:

- обычный индекс — таких индексов в таблице может быть несколько;
- уникальный индекс — уникальных индексов также может быть несколько, но значения в нем не должны повторяться;
- первичный ключ — уникальный индекс, предназначенный для первичного ключа таблицы. В таблице может быть только один первичный ключ;
- полнотекстовый индекс — специальный вид индекса для столбцов типа TEXT, позволяющий производить полнотекстовый поиск. Рассматривать его не будем, так как на практике задача полнотекстового поиска осуществляется специализированными базами данных, такими как ElasticSearch.

Первичный ключ в таблице помечается специальным индексом **PRIMARY KEY**. Значение первичного ключа должно быть уникально и не повторяться в пределах таблицы. Кроме того, столбцы, помеченные атрибутом **PRIMARY KEY**, не могут принимать значение **NULL**. Для пометки поля таблицы в качестве первичного ключа достаточно поместить ключевое слово **PRIMARY KEY** в определение столбца.

Давайте пометим поле **id** таблицы **catalogs** атрибутом **PRIMARY KEY**:

```
DROP TABLE IF EXISTS catalogs;
CREATE TABLE catalogs (
  id INT UNSIGNED NOT NULL PRIMARY KEY,
  name VARCHAR(255) COMMENT 'Название раздела'
) COMMENT = 'Разделы интернет-магазина';

DESCRIBE catalogs;
```

Есть альтернативный способ объявления первичного ключа — отдельной записью **PRIMARY KEY** с указанием названия столбца в круглых скобках:

```
DROP TABLE IF EXISTS catalogs;
CREATE TABLE catalogs (
  id INT UNSIGNED NOT NULL,
  name VARCHAR(255) COMMENT 'Название раздела',
  PRIMARY KEY(id)
) COMMENT = 'Разделы интернет-магазина';
```

Ключевое слово **PRIMARY KEY** может встречаться в таблице только один раз, так как в таблице разрешен только один первичный ключ. Индекс необязательно должен быть объявлен по одному столбцу, вполне допустимо объявление индекса сразу по двум или более столбцам.

```
CREATE TABLE catalogs (
  id INT UNSIGNED NOT NULL,
  name VARCHAR(255) COMMENT 'Название раздела',
  PRIMARY KEY(id, name(10))
) COMMENT = 'Разделы интернет магазина';

DESCRIBE tbl;
```

Здесь первичный ключ создается по столбцу **id** и по первым 10 символам столбца **name**. Можно индексировать по всем 255 символам текстового столбца. Однако размер индекса будет больше — как правило, для индекса достаточно первых символов строки.

В качестве первичного ключа часто выступает целочисленный столбец. Такой выбор связан с тем, что целочисленные типы данных обрабатываются быстрее всех и занимают небольшой объем.

Другая причина выбора такого типа — только данный тип столбца может быть снабжен атрибутом **AUTO_INCREMENT**, который обеспечивает автоматическое создание уникального индекса.

```
DROP TABLE IF EXISTS catalogs;
CREATE TABLE catalogs (
  id INT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(255) COMMENT 'Название раздела'
) COMMENT = 'Разделы интернет магазина';

SELECT * FROM catalogs;
```

Передача столбцу, снабженному этим атрибутом, значения **NULL** или 0, приводит к автоматическому присвоению ему максимального значения столбца, плюс 1.

```
INSERT INTO catalogs (name) VALUES ('Процессоры');
INSERT INTO catalogs VALUES (0, 'Мат.платы');
INSERT INTO catalogs VALUES (NULL, 'Видекарты');
SELECT * FROM catalogs;
```

Это удобный механизм, который позволяет не заботиться о генерации уникального значения средствами прикладной программы, работающей с СУБД MySQL.

MySQL предлагает псевдотип **SERIAL**, который является обозначением для типа **BIGINT**, снабженного дополнительными атрибутами **UNSIGNED**, **NOT NULL**, **AUTO_INCREMENT** и уникальным индексом.

## **SERIAL ==** BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE

```
DROP TABLE IF EXISTS catalogs;
CREATE TABLE catalogs (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255) COMMENT 'Название раздела'
) COMMENT = 'Разделы интернет-магазина';
```

Так гораздо компактнее, а главное — более совместимо с другими базами данных, которые зачастую тоже поддерживают псевдотип **SERIAL**.

В отличие от первичного ключа, таблица может содержать несколько обычных и уникальных индексов. Чтобы было удобно их различать, индексы могут иметь собственные имена. Часто имена индексов совпадают с именами столбцов, которые они индексируют, но для индекса можно назначить и совершенно другое имя. Объявление индекса производится при помощи ключевого слова **INDEX** или **KEY**.

Для уникальных индексов вводится дополнительное ключевое слово **UNIQUE**. Давайте в таблице **products** снабдим индексом поле **catalog_id**.

```
DROP TABLE IF EXISTS products;
CREATE TABLE products (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255) COMMENT 'Название',
  desription TEXT COMMENT 'Описание',
  price DECIMAL (11,2) COMMENT 'Цена',
  catalog_id INT UNSIGNED,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  KEY index_of_catalog_id(catalog_id)
) COMMENT = 'Товарные позиции';

DESCRIBE products;
```

Создать индекс в уже существующей таблице можно при помощи оператора **CREATE INDEX**.

```
DROP TABLE IF EXISTS products;
CREATE TABLE products (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255) COMMENT 'Название',
  desription TEXT COMMENT 'Описание',
  price DECIMAL (11,2) COMMENT 'Цена',
  catalog_id INT UNSIGNED,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) COMMENT = 'Товарные позиции';
CREATE INDEX index_of_catalog_id ON products (catalog_id);
```

Удалить индекс из таблицы можно при помощи оператора **DROP INDEX**:

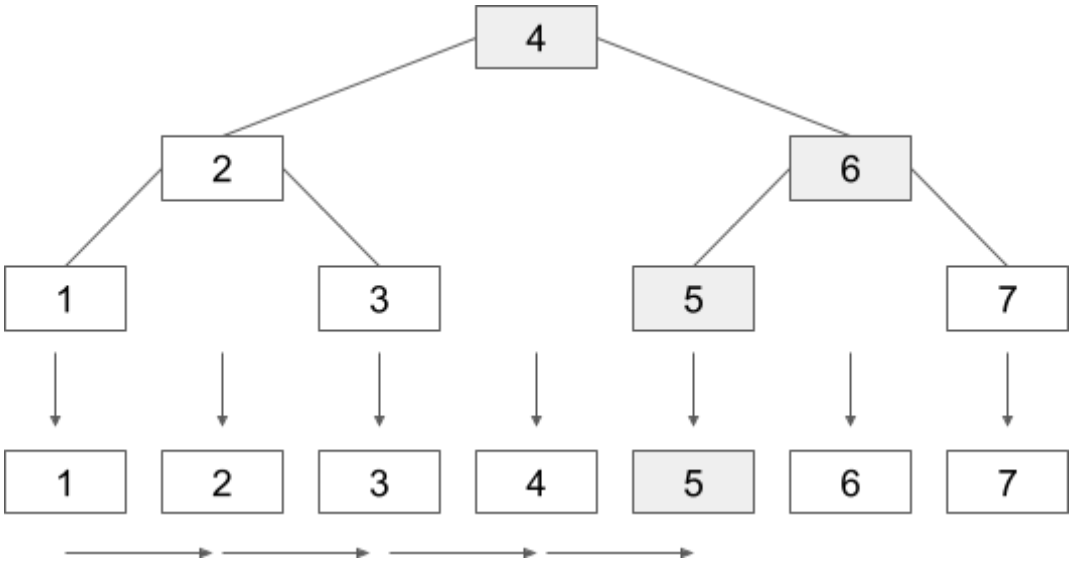
```
DROP INDEX index_of_catalog_id ON products;
```

# Устройство индекса

MySQL поддерживает два типа индекса:

- **BTREE** — бинарное дерево;
- **HASH** — хэш-таблица.

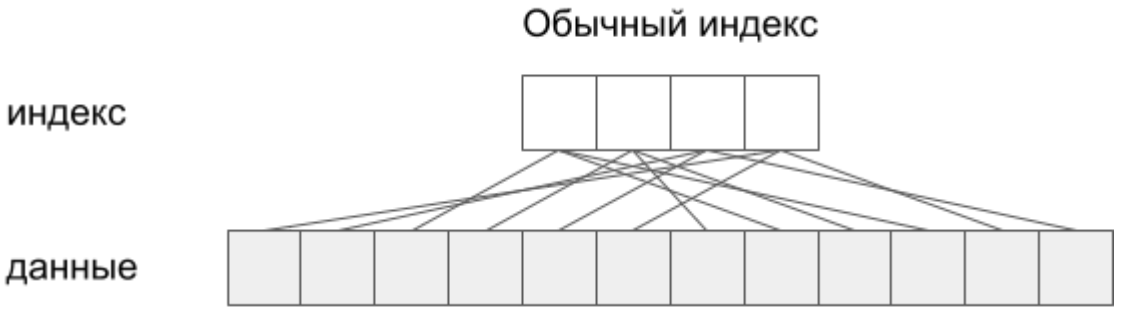
Общая идея бинарного дерева в том, что значения хранятся по порядку, и все листовые страницы находятся на одинаковом расстоянии от корня. BTREE-индекс ускоряет доступ к данным, поскольку подсистеме хранения не нужно сканировать всю таблицу для поиска нужной информации. В BTREE-индексах индексированные столбцы хранятся в упорядоченном виде, и они полезны для поиска по диапазону данных.

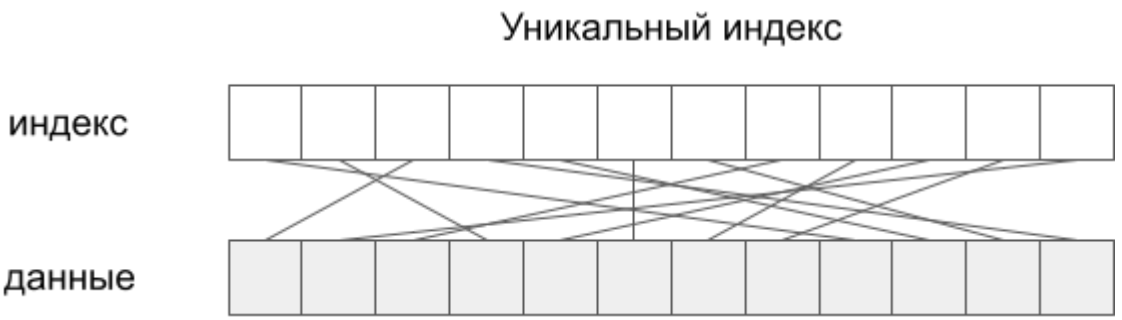



Поскольку узлы дерева отсортированы, их можно использовать как для поиска значений, так и в запросах с сортировкой при помощи **ORDER BY**.

```
CREATE INDEX index_of_catalog_id USING BTREE ON products (catalog_id);
CREATE INDEX index_of_catalog_id USING HASH ON products (catalog_id);
DESCRIBE products;
```

Хэш-индекс строится на основе хэш-таблицы и полезен только для точного поиска с указанием всех столбцов индекса. Для каждой строки подсистема хранения вычисляет хэш-код индексированных столбцов. В индексе хранятся хэш-коды и указатели на соответствующие строки.




Селективность индекса — это отношение количества проиндексированных значений к общему количеству строк в таблице. Индекс с высокой селективностью хорош тем, что позволяет MySQL при

поиске соответствий отфильтровывать больше строк. Уникальный индекс имеет селективность, равную единице.

Если селективность индекса низкая, после локализации участка или набора элементов, где находится искомое значение, его потребуется дополнительно просканировать для точного поиска значения. В случае уникального индекса мы можем сразу получить искомый результат, без сканирования даже небольшого участка таблицы.

Давайте поправим таблицы учебной базы данных:

```
CREATE TABLE products (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255) COMMENT 'Название',
  desription TEXT COMMENT 'Описание',
  price DECIMAL (11,2) COMMENT 'Цена',
  catalog_id INT UNSIGNED,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  KEY index_of_catalog_id(catalog_id)
) COMMENT = 'Товарные позиции';
DROP TABLE IF EXISTS orders;
CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  user_id INT UNSIGNED,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  KEY index_of_user_id(user_id)
) COMMENT = 'Заказы';
```

В таблице **orders_products** у нас два кандидата на индексацию: **order_id** и **product_id**. Порядок следования столбцов в индексе имеет значение:

| year | last_name | first_name |
|---|---|---|
| 1990 | Абакумов | Сергей |
| 1990 | Борисов | Игорь |
| 1990 | Сергеев | Вячеслав |
| 1991 | Антонов | Александр |
| 1991 | Ковалев | Сергей |
| 1991 | Трофимов | Антон |

```
SELECT * FROM tbl
WHERE year = 1990
```

```
SELECT * FROM tbl
WHERE
  year = 1990 AND
  last_name = Борисов
```

```
SELECT * FROM tbl
WHERE first_name = 'Сергей'
```

В каждом запросе может использоваться ровно один индекс, если в запросе участвует два поля — **order_id** и **product_id** и индекс по двум столбцам сработает. Если у нас два отдельных запроса с участием **order_id** и **product_id**, то индекс **order_id** и **product_id** может использоваться в запросе с

**order_id**, но его не получится использовать в запросе с **product_id**. В **discount**, с высокой долей вероятности, **user_id** и **product_id** будут использоваться раздельно. Давайте снабдим их индексами.

```
DROP TABLE IF EXISTS discounts;
CREATE TABLE discounts (
  id SERIAL PRIMARY KEY,
  user_id INT UNSIGNED,
  product_id INT UNSIGNED,
  discount FLOAT UNSIGNED COMMENT 'Величина скидки от 0.0 до 1.0',
  finished_at DATETIME NULL,
  started_at DATETIME NULL,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  KEY index_of_user_id(user_id),
  KEY index_of_product_id(product_id)
) COMMENT = 'Скидки';
```

# Введение в CRUD-операции

Четыре базовых операции: создание (create), чтение (read), обновление (update) и удаление (delete) записей называются по первым буквам английских слов CRUD-операциями. В языке запросов SQL им соответствует четыре оператора: **INSERT**, **SELECT**, **UPDATE** и **DELETE**.

За операцию вставки данных отвечает оператор **INSERT**. Он поддерживает несколько типов вставки, первый из которых — однострочная вставка:

```
INSERT INTO catalogs VALUES (NULL, 'Процессоры');
INSERT INTO catalogs VALUES (NULL, 'Мат.платы');
INSERT INTO catalogs VALUES (NULL, 'Видеокарты');
```

Ключевое слово **NULL** можно заменить **DEFAULT**:

```
INSERT INTO catalogs VALUES (DEFAULT, 'Процессоры');
INSERT INTO catalogs VALUES (DEFAULT, 'Мат.платы');
INSERT INTO catalogs VALUES (DEFAULT, 'Видеокарты');
```

Множество INSERT-запросов можно заменить на многострочный оператор **INSERT**:

```
INSERT INTO catalogs VALUES
  (DEFAULT, 'Процессоры'),
  (DEFAULT, 'Мат.платы'),
  (DEFAULT, 'Видеокарты');
```

Такой вариант вставки работает гораздо быстрее.

Для извлечения данных используется оператор **SELECT:**

```
SELECT id, name FROM catalogs;
```

После ключевого слова **SELECT** мы указываем список извлекаемых столбцов. У нас их всего два: **id** и **name**. После ключевого слова **FROM** мы указываем имя базы данных, в данном случае — **catalogs**. Порядок столбцов после ключевого слова **SELECT** можно менять.

```
SELECT name, id FROM catalogs;
```

Можем выводить только часть столбцов, например только **name**:

```
SELECT name FROM catalogs;
```

При помощи звездочки мы можем запросить все столбцы базы данных в том порядке, в котором они определены в таблице:

```
SELECT * FROM catalogs;
```

Мы рассмотрели лишь базовые свойства **SELECT**. Его возможности настолько велики, что ему почти полностью будут посвящены несколько следующих уроков.

Продолжим изучение вставки при помощи **INSERT**. Давайте добавим уникальный индекс для таблицы **catalogs**. Тем самым мы запретим вставку разделов, которые уже добавлены в таблицу.

```
DROP TABLE IF EXISTS catalogs;
CREATE TABLE catalogs (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255) COMMENT 'Название раздела',
  UNIQUE unique_name(name(10))
) COMMENT = 'Разделы интернет-магазина';
```

Если мы попробуем вставить повторяющиеся значение, попытка завершится сообщением об ошибке. Чтобы этого избежать, используется ключевое слово **IGNORE**:

```
INSERT IGNORE INTO catalogs VALUES
  (DEFAULT, 'Процессоры'),
  (DEFAULT, 'Мат.платы'),
  (DEFAULT, 'Видеокарты'),
  (DEFAULT, 'Видеокарты');
SELECT * FROM catalogs;
```

Запись не вставляется, но и сообщение об ошибке не выводится. Попытки вставить неправильное значение просто игнорируются.

Время от времени возникает задача удаления записей из базы данных. Решить эту задачу можно при помощи двух операторов:

- команда **DELETE** удаляет все или часть записей из таблицы;
- команда **TRUNCATE** удаляет все записи из таблицы и обнуляет счетчики **AUTO_INCREMENT**.

Давайте удалим все записи из таблицы **catalogs**. Для этого можно воспользоваться оператором **DELETE**:

```
DELETE FROM catalogs;
```

Можем удалять лишь часть данных: при помощи ключевого слова **LIMIT** мы можем удалять лишь ограниченный объем записей, например 2.

```
DELETE FROM catalogs LIMIT 2;
```

Можем задавать и более сложные условия, например:

```
DELETE FROM catalogs WHERE id > 1 LIMIT 1;
```

Оператор **TRUNCATE TABLE**, в отличие от **DELETE**, полностью очищает таблицу и не допускает условного удаления. То есть оператор **TRUNCATE TABLE** аналогичен оператору **DELETE** без условия **WHERE** и ограничения **LIMIT**.

В отличие от оператора **DELETE**, удаление происходит гораздо быстрее, т. к. идетя не перебор каждой записи, а полное очищение таблицы. Кроме того, обнуляется и счетчик **AUTO_INCREMENT**:

```
INSERT INTO catalogs VALUES
  (DEFAULT, 'Процессоры'),
  (DEFAULT, 'Мат.платы'),
  (DEFAULT, 'Видеокарты');
TRUNCATE catalogs;
INSERT INTO catalogs VALUES
  (DEFAULT, 'Процессоры'),
  (DEFAULT, 'Мат.платы'),
  (DEFAULT, 'Видеокарты');
```

Операция обновления **UPDATE** позволяет менять значения полей в уже существующих записях. Заменим название каталога «Процессоры» в таблице catalogs на «Процессоры (Intel)»:

```
UPDATE catalogs SET name = 'Процессоры (Intel)'
WHERE name = 'Процессоры';
SELECT * FROM catalogs;
```

Оператор **INSERT … SELECT** позволяет вставлять записи из одной таблицы в другую, в том числе и преобразовывая данные.

Давайте создадим таблицу **cat**, структура которой будет совпадать со структурой таблицы **catalogs**:

```
CREATE TABLE cat (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255)
);
```

Пусть требуется переместить данные из таблицы **catalogs** в таблицу **cat** таким образом, чтобы в поле **cat.name** помещалось поле **catalogs.name**. Для этого можно воспользоваться оператором **INSERT … SELECT**:

```
INSERT INTO
  cat
SELECT
  *
FROM
  catalogs;
SELECT * FROM cat;
```

Теперь содержимое таблицы **cat** совпадает с содержимым таблицы **catalogs**.

# Используемые источники


1. https://dev.mysql.com/doc/refman/5.7/en/tutorial.html
2. https://dev.mysql.com/doc/refman/5.7/en/literals.html
3. Линн Бейли. Head First. Изучаем SQL. — СПб.: Питер, 2012. — 592 с.
4. Грофф, Джеймс Р., Вайнберг, Пол Н., Оппель, Эндрю Дж. SQL: полное руководство, 3-е изд. : Пер. с англ. — М.: ООО "И.Д. Вильямс", 2015. — 960 с.
5. Дейт К. Дж. SQL и реляционная теория. Как грамотно писать код на SQL. — Пер. с англ. — СПб.: Символ-Плюс, 2010. — 480 с.
6. Кузнецов М.В., Симдянов И.В. MySQL на примерах. — СПб.: БХВ-Петербург, 2007. — 592с.
7. Кузнецов М.В., Симдянов И.В. MySQL 5. — СПб.: БХВ-Петербург, 2006. — 1024с.
8. Дейт К. Дж. Введение в системы баз данных, 8-е издание.: Пер. с англ. — М.: Издательский дом "Вильямс", 2005. — 1328 с.
9. Карвин Б. Программирование баз данных SQL. Типичные ошибки и их устранение. — Рид Групп, 2011. — 336 с.