



Курс “Базы данных и SQL”

Составитель конспектов: @can_u_feel_my_heart

Требуемая подготовка студента:

- понимание основных терминов реляционных баз — таблица, ключ.

Цели курса:

- Научиться писать SQL запросы
- Научиться работать с СУБД
- Закрепить навыки проектирования реляционных баз данных

Структура:

Вступительный материал:

- ↗ Основы реляционных баз данных (вступительная лекция)

Основной материал:

Лекция №1

Курс базы данных и SQL. Лекция 2

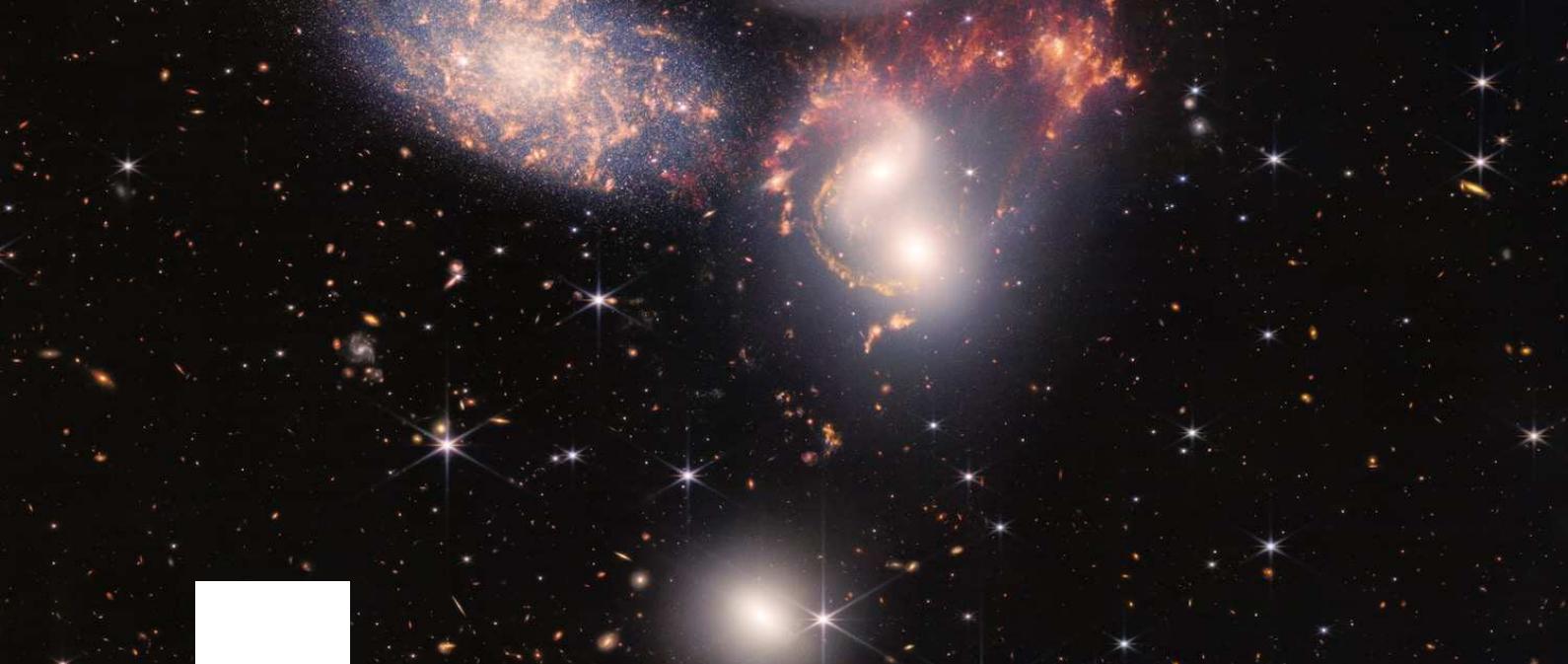
Курс базы данных и SQL. Лекции 3 и 4

Курс базы данных и SQL. Лекция 5

Курс базы данных и SQL. Лекция 6

Дополнительные материалы к курсу:

Индексы и ключи

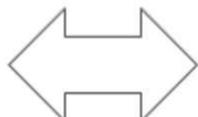
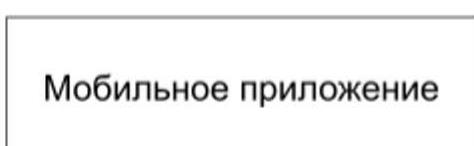
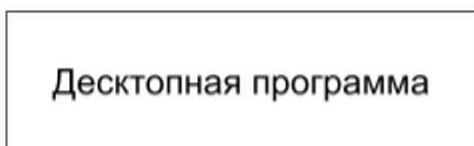


Основы реляционных баз данных (вступительная лекция)

Доброго времени суток, уважаемые студенты! В этом небольшом конспекте мы поговорим о истории развития СУБД, о реляционных и не реляционных БД

Данные и программы

Данные живут дольше, чем программы. Наши программы пока слишком недолговечны и часто меняются. Во время жизненного цикла данных в разное время их может обслуживать несколько программ. Иногда одни и те же данные обслуживает несколько программ одновременно. Поэтому в программировании принято отделять данные от кода и держать их в специализированном хранилище — **базе данных**.



База данных — это совокупность информационных материалов, организованных таким образом, чтобы их можно было найти и обработать при помощи компьютера.

Обычный текстовый файл тоже база данных, пусть и очень примитивная.

Почему недостаточно обычных файлов?

Дело в том, что файлы довольно ограничены по возможностям. При работе с большим объемом данных необходимо обеспечить их компактность. Их лучше записывать в бинарном, а не текстовом формате, возможно, применяя механизмы сжатия. Это не очень наглядно, и для восприятия в любом случае потребуется конвертация данных в формат, доступный человеку. Когда множество параллельных процессов или клиентов обращаются к файлу с целью записать или извлечь информацию из него, очень трудно обеспечить конкурентный доступ. Необходимо либо прибегать к блокировкам файла, либо создавать очередь для запросов. В файлы очень легко записывать информацию, если мы вносим ее в самый конец, однако очень не просто отредактировать запись в середине. Кроме того, чтобы что-то найти в файле, приходится его сканировать от начала до конца. Чтобы кэшировать часто используемые данные в оперативной памяти, вам придется писать собственную программу. Ситуация еще больше усложняется, если наш файл гигантского объема и просто не помещается на одном компьютере. И вот у вас уже несколько файлов, которые хранятся на нескольких компьютерах. Как искать среди них информацию? Придется писать дополнительное программное обеспечение. А что будем делать, если одновременно два клиента захотят исправить один и тот же документ, внося в него совершенно разную информацию?

Поэтому практически сразу после появления операционных систем и файлов над базами данных стали появляться программные надстройки. Они позволяли управлять, искать, пополнять и редактировать данные внутри базы данных. Решать те все проблемы, которые мы с вами обозначили. Такая надстройка стала называться системой управления базами данных или сокращенно СУБД. В нашем курсе для краткости мы будем называть базами данных совокупность как самого хранилища, так и этой программной надстройки.

История развития СУБД

Современные базы данных за последние 60 лет прошли длительный путь развития. Давайте кратко побежимся по истории СУБД, от первых иерархических баз данных до современных NoSQL-решений.

Иерархические базы данных

Первые базы данных были иерархическими. Это, наверное, вообще первое, что приходит в голову программистам.

Иерархия — это дерево, состоящее из узлов, у которых может быть несколько потомков. При помощи такой структуры хорошо описываются иерархические структуры организаций и производств. Примеров иерархий очень много и они постоянно находятся у нас перед глазами. На экране вы можете видеть иерархию транспортной системы. Есть вершина — транспорт, от которого расходятся узлы с видами транспорта и последующей детализацией специализации ТС.

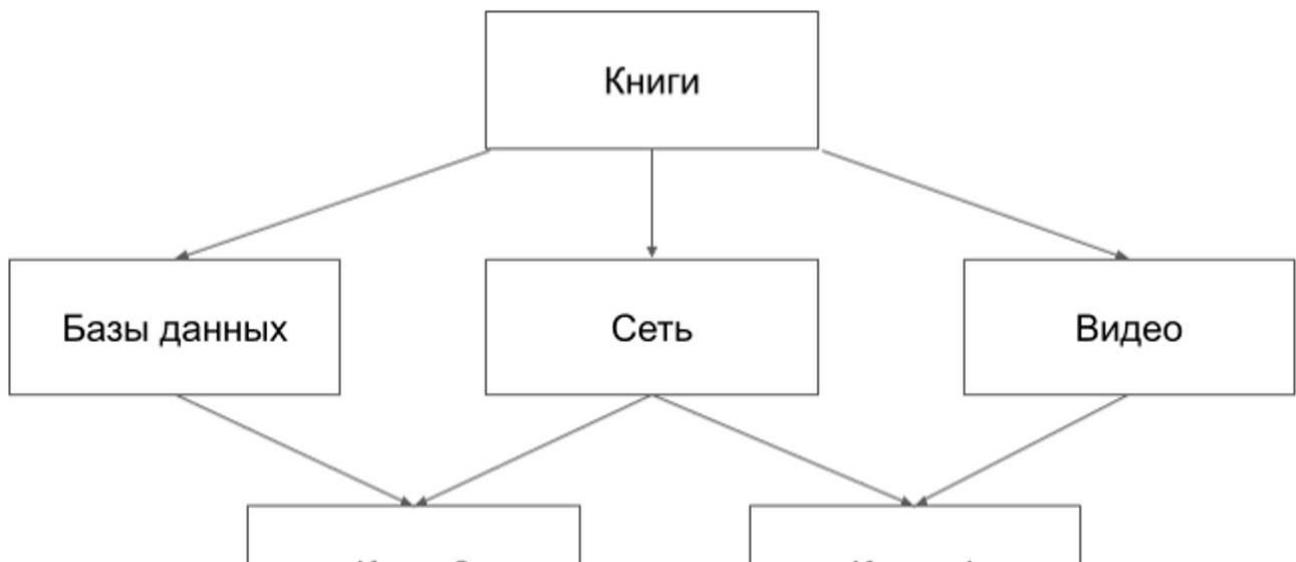
Иерархии очень наглядны и хорошо описываются деревьями, у которых прекрасно изученный мат. аппарат.



Главное их достоинство — высокая скорость обработки операций. Первые компьютеры не отличались высокой производительностью: чем проще организована база данных, тем быстрее она работает.

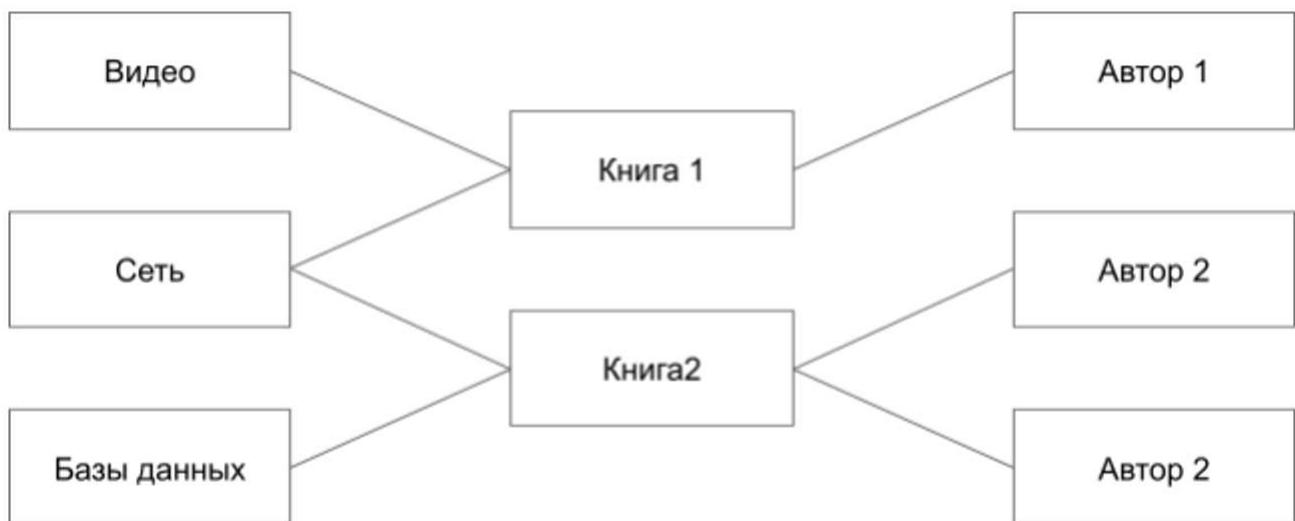
Основной недостаток иерархической структуры базы данных — невозможность реализовать

отношения «многие ко многим». Например, если мы создаём каталог книг, одна книга может относиться сразу к нескольким разделам.



Сетевые базы данных

Была разработана новая модель данных — сетевая. Она расширила иерархическую модель, позволяя одной записи участвовать в нескольких отношениях «предок-потомок».



В связи с развитием социальных сетей эти базы данных получили второе дыхание в виде графовых СУБД, которые относятся к современному NoSQL-течению. Только в современной интерпретации ценность приобретают не сами данные, а связи между узлами. Тем не менее, что примечательно, многие идеи, которые появляются в новых популярных NoSQL-базах, были придуманы или опробованы в прошлом.

Просто на тот момент это было либо экономически нецелесообразно, либо мода и влияние больших компаний толкало рынок в сторону других моделей. Конечно, у сетевых баз данных имелись недостатки: подобно своим иерархическим предкам, сетевые базы данных были очень жесткими. Наборы отношений и структура записей должны были быть заданы наперед. Изменение структуры базы данных обычно означало ее полную перестройку.

Реляционные базы данных

Следующим шагом стало развитие реляционных баз данных. Реляционная модель данных была попыткой упростить структуру базы данных. В ней отсутствовала явная структура «предок-потомок», а все данные были представлены в виде простых таблиц, разбитых на строки и столбцы.

name	id	author_id	book_id	id	name
Автор 1	1	1	1	1	Книга 1
Автор 2	2	2	2	2	Книга 2
Автор 3	3	3	2		

Теоретические основы новой реляционной модели данных впервые были описаны доктором Коддом в 1970 году. Поначалу его работа предоставляла лишь академический интерес. Однако, спустя 10 лет, основываясь на его работе, реляционные базы данных создали сначала Oracle, затем IBM, а потом и множество других компаний.

Реляционные СУБД прочно вошли в компьютерный мир и актуальны до сих пор. Это самый распространенный вид баз данных. Львиная доля курса будет посвящена именно ему.

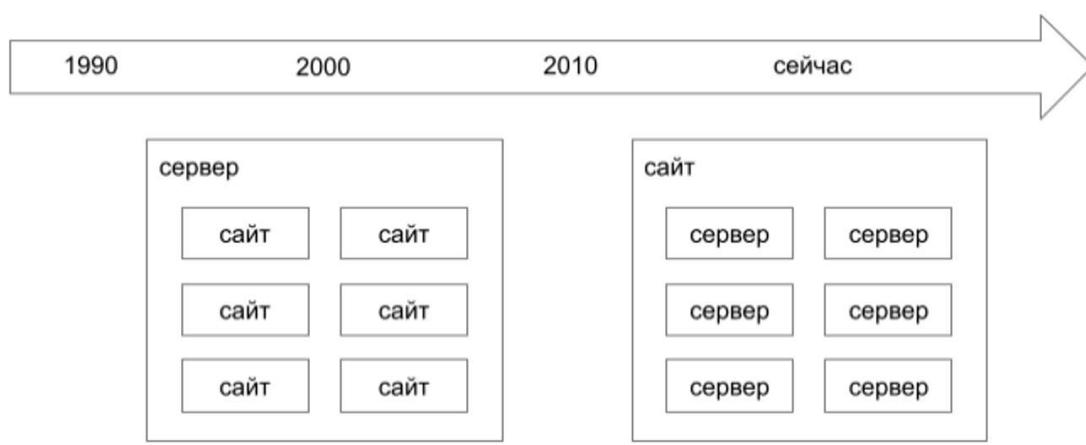
За 40 лет было множество попыток заменить реляционные баз данных чем-то более новым и прогрессивным. Долгое время на смену СУБД пророчили приход XML и объектно-ориентированных баз данных. Однако эти технологии так и не стали массовыми, хотя продукты существуют и по сей день. XML вышел из моды из-за своей избыточности. ООП-базы данных, которые решают проблему несовместимости реляционных баз данных, основанных на множествах, и ООП-программ, основанных на деревьях, тоже не стали слишком популярны. Наиболее популярные СУБД на сегодняшний день — Oracle, MS SQL и DB2 среди коммерческих, MySQL, PostgreSQL и Firebird среди свободных.

Индекс популярности баз данных

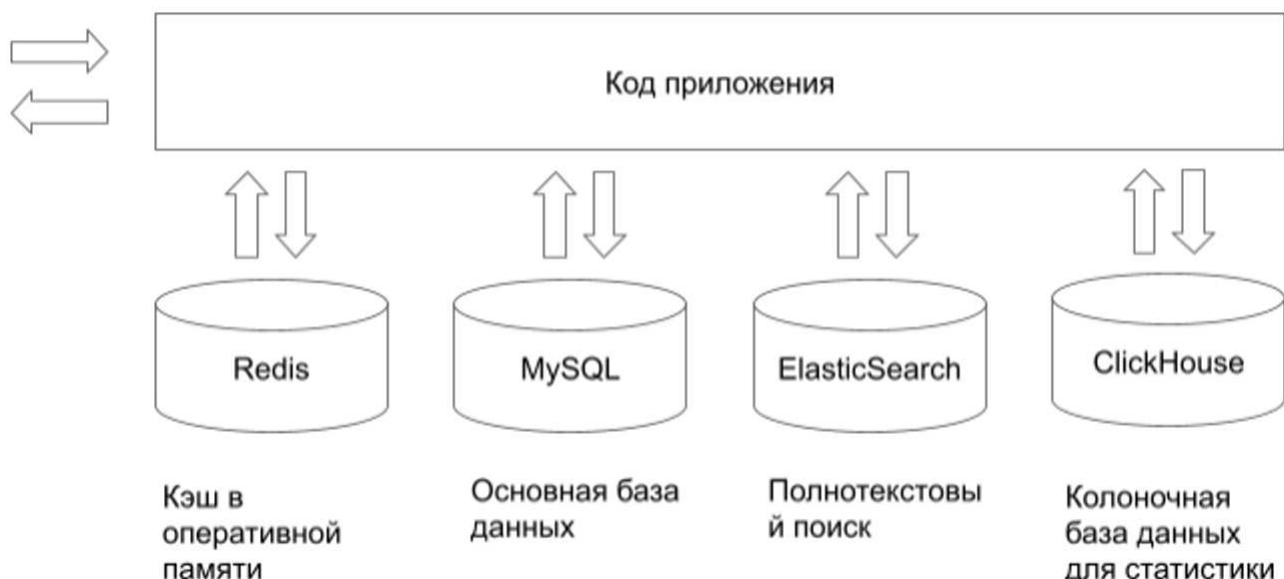
Следить за индексом популярности баз данных можно по рейтингу на сайте db-engines.com. Здесь представлены не только реляционные базы данных, но и NoSQL-решения. Тем не менее, по этому ресурсу вы можете отслеживать динамику интереса к базам данных на протяжении длительного времени.

NoSQL базы данных

После стремительного развития интернета в принципах построения баз данных произошел переломный момент. Первые веб-сайты и сервисы были очень невелики: зачастую на одном сервере убирались сотни сайтов. Однако по мере вовлечения все новых и новых пользователей проекты начали укрупняться и очень скоро им стало не хватать не то что одного, а десятков, сотен, а затем и тысяч серверов.



Не стали помещаться на одном сервере и базы данных. Поэтому очень скоро сначала реляционные СУБД, а потом и новые игроки стали отказываться от традиционного подхода хранения данных. Стали строиться распределенные хранилища и интенсивно использоваться гораздо большие объемы оперативной памяти. Новые подходы и распределенная структура баз данных привели к ситуации, когда реализовать стандартный SQL-язык стало либо очень сложно, либо почти невозможно. В результате на рынке стали появляться специализированные СУБД, ориентированные под решение тех или иных задач, зачастую полностью расположенные в оперативной памяти, представляющие свой язык запросов, иногда вообще не следующий многолетней традиции SQL.



На экране представлены типичные представители NoSQL-базы данных. На самом деле их гораздо больше.

Redis — это очень быстрое хранилище построенное по принципу «ключ-значение». Оно полностью расположено в оперативной памяти, сервер реализован в виде однопоточного EventLoop-цикла, когда один поток опрашивает по кругу соединения в неблокирующем режиме. За счет того, что не происходит переключение процессора на другие процессы, достигается гигантская производительность порядка 100 000 RPS (это зачастую в сотни раз выше, чем в лучших реляционных базах данных).

Если на сервере не хватает оперативной памяти, чтобы разместить индекс, можно разбить данные на части — шарды и хранить несколько копий такого шарда на разных компьютерах. В результате образуется кластер, который ведет себя как единый компьютер с огромным количеством оперативной памяти. Так действуют grid-решения в **Oracle**, **MySQL**, так могут поступать и **NoSQL**-базы данных вроде **ElasticSearch** и **MongoDB**.

Собирать JSON-документ из нескольких таблиц может быть долго и накладно. В этом случае можно хранить не отдельные значения документа, а готовый, собранный заранее, документ. Для этого используются документо-ориентированные СУБД, примером может служить та же **MongoDB**. Часто в реляционных базах данных штатный механизм полнотекстового поиска не предусмотрен или реализован неэффективно. Поэтому можно прибегать к базам данных, специально предназначенным для полнотекстового поиска.

базам данных, специально предназначенных для полнотекстового поиска, позволяющих регулировать любые параметры поискового механизма. Яркий представитель таких баз данных — **ElasticSearch**.

Традиционные СУБД плохо предназначены для операций в реальном времени, например для обсчета статистики. Гораздо лучше для этих целей подходит колоночная база данных. В ней запрещены операции редактирования и удаления, данные сжаты, что позволяет обеспечивать исключительно быстрый механизм агрегации. Один из представителей таких баз данных — **ClickHouse**. Большой проблемой для баз данных, разработанных в прошлом, стала распределенная природа современных приложений. Они не только работают на нескольких серверах, но и зачастую разбросаны по нескольким данным центрам в разных точках мира. Это грозит тем, что разные части распределенной базы данных могут терять связность и необходимы меры по поддержанию работоспособности и доступности в таких условиях. Как раз для этого предназначена база данных **Cassandra**.

Это не значит, что у новых NoSQL-баз данных вообще нет недостатков, их много и они настолько существенны, что не позволяют вам отказаться от традиционных реляционных баз данных. Просто эксплуатируя NoSQL-базу данных для решения задач, под которые она заточена, можно добиться удивительных успехов. При этом важно не использовать такое решение там, где проявляются слабые стороны того или иного хранилища.

Поэтому современный сайт или приложение может использовать совокупность нескольких хранилищ. Например, мы можем запоминать результат ресурсоемкой операции для ускорения чтения, т. е., использовать кеш. Можем использовать основную базу данных для долговременного хранения. Можем предоставлять пользователям возможность искать данные по ключевому слову или фильтровать их различными способами. Чтобы время от времени перемалывать большие объемы накопленных данных, идеально подходят колоночные базы данных. На протяжении всего курса мы будем рассматривать реляционные базы данных на примере **MySQL**.

Основы реляционных баз данных

Большая часть курса будет посвящена реляционным базам данных. В них информация организована в виде прямоугольных таблиц, разделенных на строки и столбцы, на пересечении которых содержатся значения.

Таблицы

База данных состоит из нескольких таблиц. Каждая таблица имеет уникальное имя, описывающее ее содержимое.

catalogs

users

products

Начнем формировать базу данных, с которой мы будем работать в течение курса. Пусть это будет база данных интернет-магазина компьютерных комплектующих. Ниже представлена таблица **catalogs**.

id	name	total
1	Процессоры	15
2	Видеокарты	10
3	Материнские платы	24
4	Оперативная память	12

Строки

Каждая горизонтальная строка этой таблицы представляет отдельную физическую сущность — один каталог. Четыре строки таблицы вместе представляют все четыре каталога интернет-магазина. Все данные, содержащиеся в конкретной строке таблицы, относятся к каталогу, который описывается этой строкой.

Столбцы

Каждый вертикальный столбец таблицы **catalogs** представляет один элемент данных для каждого из каталогов. На пересечении строки и столбца таблицы содержится только одно значение. Например, в строке, представляющей видеокарты, в столбце **name** содержится название раздела. В столбце **total** этой же строки находится значение 10, сообщающее количество доступных для покупки товаров. Все значения, содержащиеся в одном и том же столбце — данные одного типа. Например, в столбце **name** содержатся только строки, в столбце **total** — только числовые значения. У каждого столбца в таблице есть свое имя, которое обычно служит его заголовком. Все столбцы в одной таблице должны

иметь уникальные имена, однако разрешается присваивать одинаковые имена столбцам, расположенным в различных таблицах. На практике такие имена столбцов, как `name` (имя), `id` (идентификатор), `description` (описание) и тому подобные, часто встречаются в различных таблицах одной базы данных. Столбцы таблицы упорядочены слева направо, и их порядок определяется при создании таблицы. В любой таблице всегда есть как минимум один столбец. В отличие от столбцов, строки таблицы не имеют определенного порядка. Это значит, что если последовательно выполнить два одинаковых запроса для отображения содержимого таблицы, нет гарантии, что оба раза строки будут перечислены в одном и том же порядке. Конечно, можно попросить SQL-запрос отсортировать строки перед выводом, однако порядок сортировки не имеет ничего общего с фактическим расположением строк в таблице.

Пустая таблица

В таблице может содержаться любое количество строк. В том числе и ноль строк, в этом случае таблица называется пустой. Пустая таблица сохраняет структуру, определенную ее столбцами, просто в ней не содержатся данные.

Первичный ключ

Поскольку строки в реляционной таблице не упорядочены, нельзя выбрать строку по ее номеру в таблице. В таблице нет первой, последней или тринадцатой строки.

В правильно построенной реляционной базе данных в каждой таблице есть столбец (или комбинация столбцов), для которого значения во всех строках различны. Этот столбец (или столбцы) называется первичным ключом (*primary key*) таблицы.

Первичный ключ у каждой строки уникальный. В таблице с первичным ключом нет двух совершенно одинаковых строк. Таблица, в которой все строки отличаются друг от друга, в математических терминах называется отношением (*relation*). Именно этому термину реляционные базы данных и обязаны своим названием, поскольку в их основе лежат отношения, т. е., таблицы с отличающимися друг от друга строками.

<code>id</code>	<code>name</code>	<code>total</code>
1	Процессоры	15
2	Видеокарты	10
3	Материнские платы	24
4	Оперативная память	12

Связи между таблицами

В иерархических базах данных довольно легко выстраивать отношения «предок-потомок». В реляционной базе данных происходит отказ от явных связей, однако, отношение «предок-потомок» между категориями и товарными позициями не утеряно.

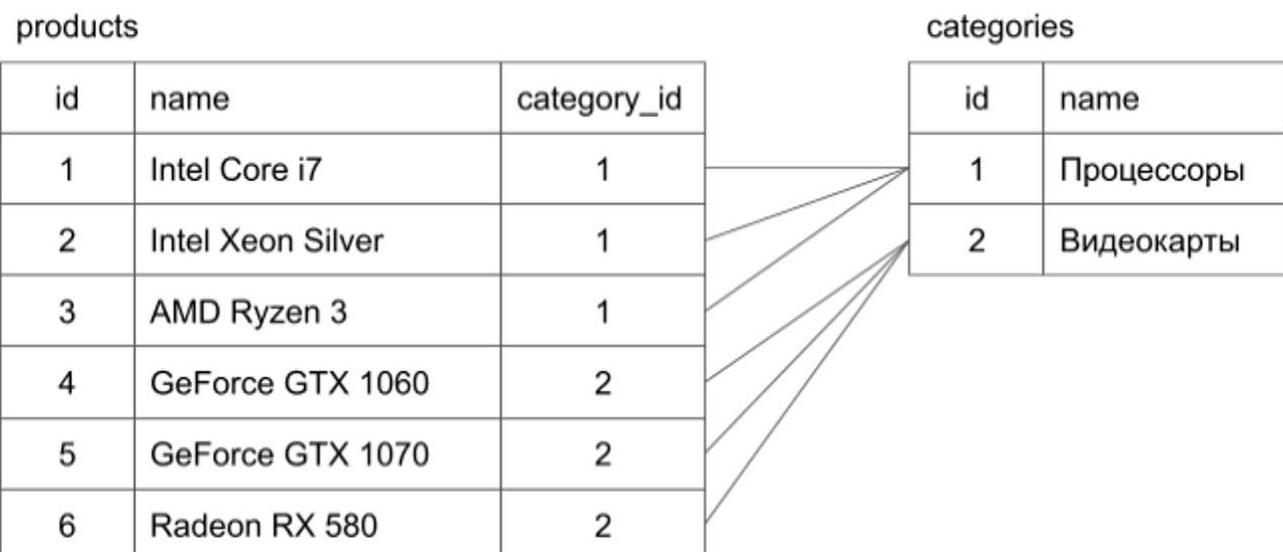
Оно реализовано в виде одинаковых значений, хранящихся в двух таблицах, а не в виде явного указателя. Таким способом реализуются все отношения, существующие между таблицами реляционной базы данных.

Столбец одной таблицы, значения в котором совпадают со значениями столбца, являющегося

первичным ключом другой таблицы, называется внешним ключом (foreign key).

Например, здесь на экране внешним ключом выступает столбец `category_id`.

Одним из главных преимуществ реляционных баз данных — возможность извлекать связанные между собой данные, используя эти отношения.

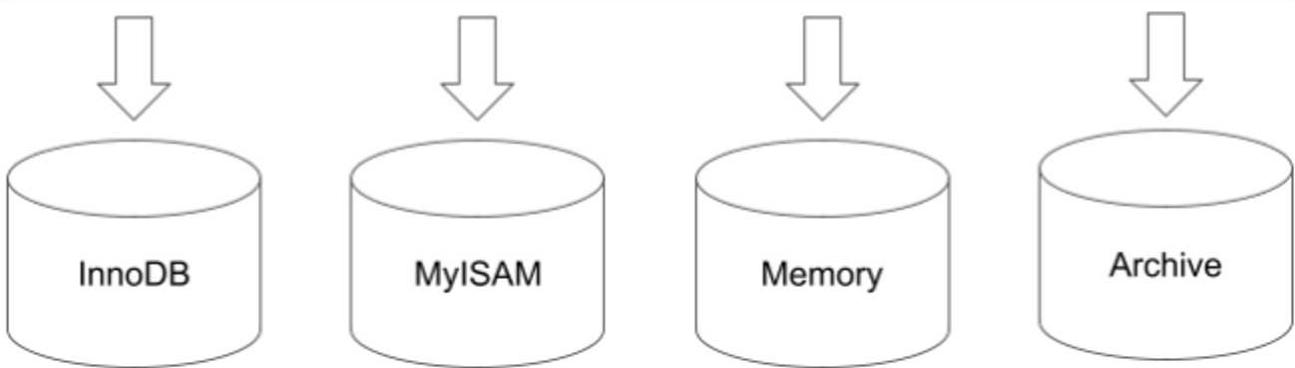


Для нас наличие первичного ключа сейчас является чем-то само собой разумеющимся, хотя первые реляционные СУБД его просто не поддерживали.

Архитектура MySQL

MySQL на сегодняшний день самая популярная база данных с открытым кодом. По популярности ее превосходит только коммерческая **СУБД Oracle**. **MySQL** используется во множестве проектов, наверное, самый известный из них — электронная энциклопедия **Wikipedia**. Существует множество форков этой базы данных: **Percona**, **MariaDB**. Мы будем знакомиться с оригинальной версией **MySQL**.





Архитектуру **MySQL** можно условно разбить на две части: это ядро, сама база данных и движки, которые реализуют тот или иной механизм баз данных. На экране представлено несколько движков, которые могут использоваться **MySQL**. По умолчанию используется **InnoDB**. Более подробно каждый из движков будет рассмотрен на следующих уроках.

Такая архитектура позволяет разрабатывать базу данных усилиями нескольких команд. Одна команда может сосредоточиться на ядре, другая — на каком-либо отдельном движке. Например, движок **InnoDB** долго разрабатывался отдельной компанией.

MySQL построена по клиент-серверной технологии: и клиент, и сервер являются программами, которые могут быть расположены на разных компьютерах или на том же самом. Сервер хранит и обслуживает базы данных, клиенты шлют ему запросы на языке **SQL** и получают в ответ результирующие таблицы с данными.

Информационная схема

Операторы **SHOW** и **DESCRIBE** являются нестандартными, другие базы данных, отличные от **MySQL**, их могут не предоставлять. Более того, возможности этих операторов довольно ограничены. Каждая СУБД имеет системную базу данных, в которой хранятся учетные записи, таблицы привилегий и другая информация, необходимая для управления СУБД. В **MySQL** такая системная база данных носит название **mysql**. Структура системной базы данных для разных СУБД отличается: для того, чтобы унифицировать процесс обращения к системной базе данных, вводится специальный набор представлений, оформленных в виде базы данных **INFORMATION_SCHEMA**, которая доступна каждому клиенту **MySQL**.

База данных **INFORMATION_SCHEMA** является виртуальной и располагается в оперативной памяти — для неё нет физического соответствия на жёстком диске, как для других баз данных. Это означает, что невозможно выбрать базу данных **INFORMATION_SCHEMA** при помощи оператора **USE**, как и выполнить по отношению к таблицам этой базы данных запросы с участием операторов **INSERT**, **UPDATE** и **DELETE**.

Допускается использование только оператора **SELECT**:

```
SELECT * FROM INFORMATION_SCHEMA.SCHEMATA
```

Операторы SHOW и DESCRIBE короче, но не закреплены в стандарте SQL, они работают только в MySQL. Информационную схему обязаны реализовывать все реляционные СУБД, поддерживающие язык запросов SQL. С таблицами информационной схемы можно работать как с таблицами любой базы данных. Например, чтобы извлечь список таблиц базы данных shop, можно воспользоваться следующим запросом:

```
SELECT * FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_SCHEMA =  
'shop';
```

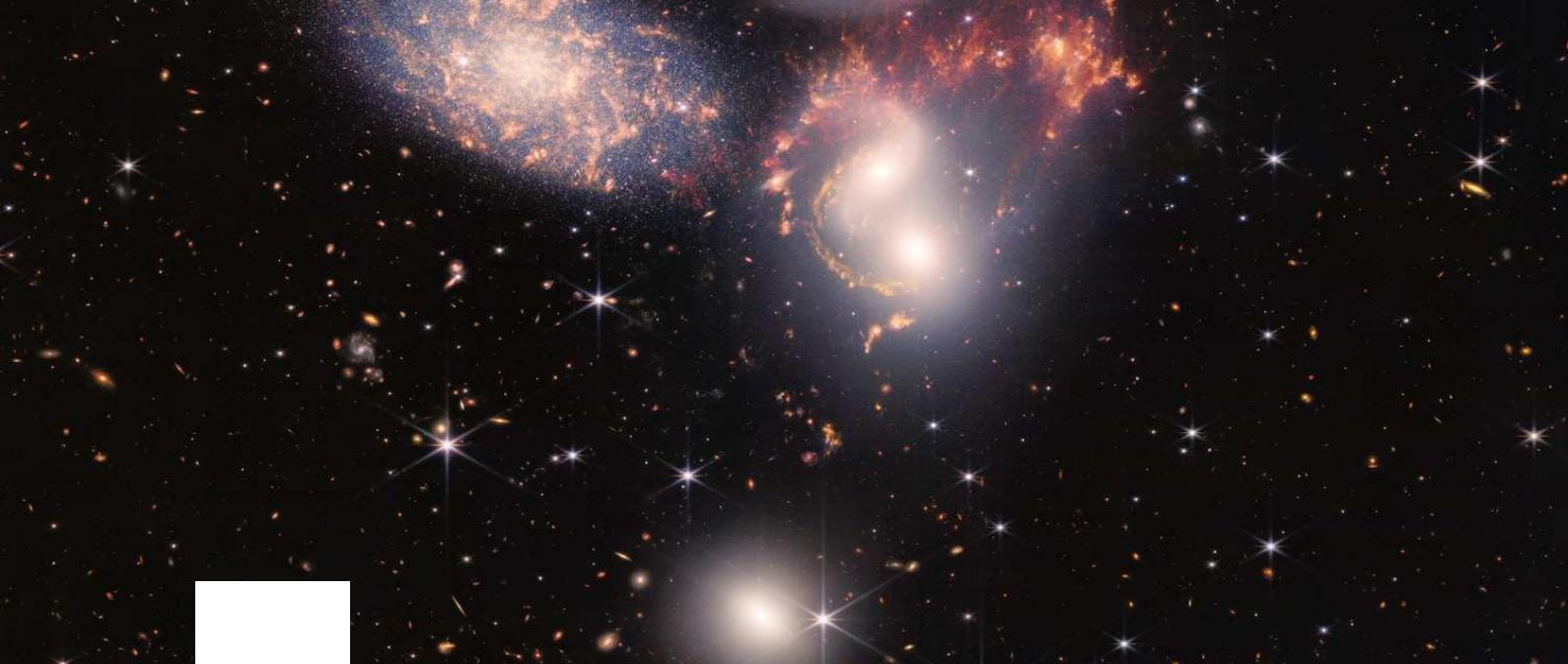
Дополнительные материалы

Базам данных вообще и MySQL в частности посвящено большое количество литературы разной степени сложности. Помимо книг, посвященных СУБД MySQL, следует обращать внимание на книги, которые специализируются на языке запросов SQL. Это стандартный специализированный язык для общения с реляционными базами данных. Больше половины курса будет посвящена именно ему, почти все его элементы одинаковы для всех баз данных. Существуют исключения, вроде команд SHOW и DESCRIBE, которые могут быть реализованы в SQL-диалекте одной базы данных и отсутствовать в другой.

1. Шварц Б., Зайцев П., Ткаченко В., Заводны Дж., Ленц А., Бэллинг Д. MySQL. Оптимизация производительности, 2-е издание. — Пер. с англ. — СПб.: Символ-Плюс, 2010. — 832 с.
2. Чарльз Белл, Мэтс Киндал и Ларс Талманн. Обеспечение высокой доступности систем на основе MySQL / Пер. с англ. — М. : Издательство "Русская редакция"; СПб. : БХВ-Петербург,
3. — 624 с.
4. Чаллавала Ш., Лакхатария Дж., Мехта Ч., Патель К. MySQL 8 для больших данных. — М.: ДМК Пресс, 2018. — 226с.
5. Поль Дюбуа. MySQL. — Пер. с англ. — М.: ООО "И.Д. Вильямс", 2007. — 1168 с.
6. Поль Дюбуа. MySQL. Сборник рецептов. — Пер. с англ. — М.: Символ-Плюс, 2004. — 1056 с.
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.: Издательство

М.: Издательский
дом "Вильямс", 2005. — 1328 с.

9. Кляйн К., Кляйн Д., Хант Б. SQL. Справочник, 3-е издание. — Пер. с англ. — СПб:
Символ-Плюс, 2010. — 656с.
10. Линн Бейли. Head First. Изучаем SQL. — СПб.: Питер, 2012. — 592 с.
11. 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.
12. Дейт К. Дж. SQL и реляционная теория. Как грамотно писать код на SQL. —
Пер. с англ. —
СПб.: Символ-Плюс, 2010. — 480 с.
13. Карвин Б. Программирование баз данных SQL. Типичные ошибки и их
устранение. — Рид
Групп, 2011. — 336 с.
14. Клеппман М. Высоконагруженные приложения. Программирование,
масштабирование,
поддержка. — СПб.: Питер, 2018. — 640 с.: ил.
15. Редмонд Эрик , Уилсон Джим Р. Семь баз данных за семь недель.
Введение в современные
базы данных и идеологию NoSQL. — М: ДМК Пресс — 384с.
16. Фаулер, Мартин, Садаладж, Прамодкумар Дж. NoSQL: новая методология
разработки
нереляционных баз данных. — Пер. с англ. — М.: ООО "И.Д. Вильямс",
2013. — 192 с.
17. Робинсон Ян, Вебер Джим, Эифрем Эмиль. Графовые базы данных: новые
возможности для
работы со связанными данными. — 2-е изд. — М.: ДМК Пресс, 2016. — 256
с.
18. Карпентер Д., Хьюитт Э. Cassandra. Полное руководство. 2-е изд. — М.:
ДМК Пресс, 2017. —
400 с.
19. Бэнкер Кайл. MongoDB в действии. — М.: ДМК Пресс, 2017. — 394с.
20. <https://redis.io/documentation>.



Лекция №1

На первой лекции будем рассматривать структуру базу данных. Вспомним реляционные базы данных, разберем понятие “СУБД”, напишем первые запросы с помощью MySQL.

Что мы узнаем:

- Историю языка SQL
- Модель данных
- Узнаем про императивное и декларативное программирование
- Понятие внешнего и первичного ключа
- Основные операторы SQL
- Понятие СУБД
- Работа с UI MySQL
- Оператор SELECT

Здравствуйте, уважаемые студенты! На нашем курсе мы будем работать на языке SQL. Перед началом работы хотелось бы немного углубиться в историю.

Истоки SQL возвращают нас в 1970-е годы, когда в лабораториях IBM было создано новое программное обеспечение баз данных System R, а для управления язык - SEQUEL. Сейчас это название применяется в качестве альтернативы произношения SQL.

В 1979 году в компании Oracle (на тот момент компания называлась Relational Software), распознали коммерческий потенциал языка SQL и выпустила собственную модифицированную версию.

- Аббревиатура SEQUEL расшифровывалась как Structured English

- Аббревиатура SEQUEL расшифровывалась как Structured English QUEry Language — «структурированный английский язык запросов». Позже по юридическим соображениям («„sequel“ был торговой маркой британской авиастроительной группы компаний Hawker Siddeley») язык SEQUEL был переименован в SQL (Structured Query Language — «язык структурированных запросов»).

Сегодня SQL стал стандартом языка запросов к базам данных. Он удовлетворяет как отраслевые, так и академические потребности и используется как на индивидуальных компьютерах, так и на корпоративных серверах. С развитием технологий баз данных приложения на основе SQL становятся все более доступными для обычного пользователя. В современной корпоративной культуре, знание SQL это одно из основных требований к аналитику. Давайте разбираться.

Поговорим о реляционных базах данных(БД)

Данные - информация, окружающая нас повсюду. Для визуализации данных чаще всего используют **таблицы**. Таблицы состоят из строк и столбцов. Они являются объектами, которые содержат все данные в базах данных. **Базы данных (БД)** — это структурная совокупность взаимосвязанных данных определённой предметной области: реальных объектов, процессов, явлений. БД решает следующие задачи:

- Хранение данных
- Получение данных
- Обработка данных

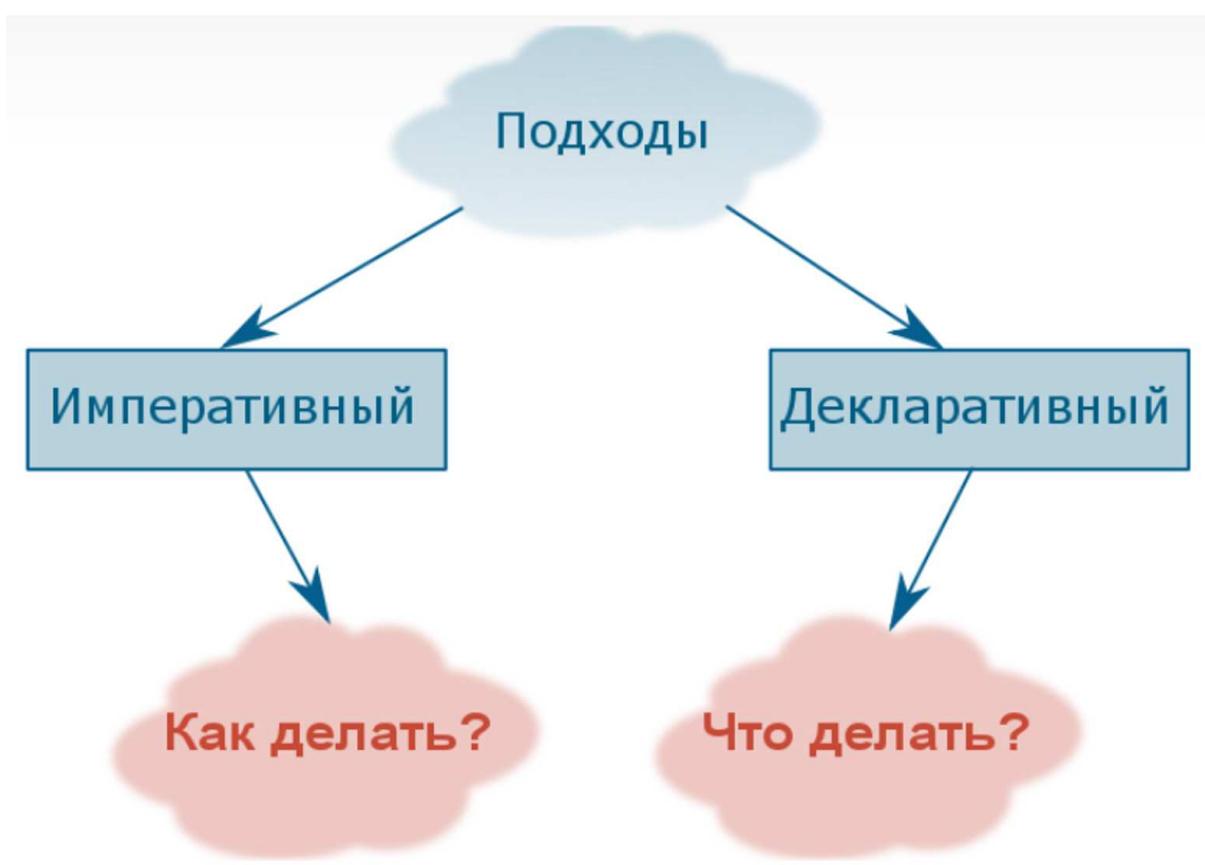
Технологии БД были отнюдь не всегда. Изначально данные хранились в формате плоских файлов (flat files). Эти данные имели простую структуру: данные представляли из себя записи, разделенные на поля фиксированной длины. В реальной жизни происходит необходимость организовывать сложные записи, которые нужно перенести в БД. Основным понятием в теории баз данных является **модель данных**. Она характеризует способ организации данных и методы доступа к ним. Изначально были предложены иерархическая и сетевая модель данных. В ходе эволюций теории была разработана реляционная модель данных. Их особенность заключается в хранении данных в формате таблицы.

Id	Name	Group
1	Михаил	IT - Специалист 7777 выходные - утро
2	Антон	IT - Специалист 8888 выходные - утро
3	Тимофей	IT - Специалист 9999 будни - вечер

Таблицы, которые связаны между собой, называются **реляционными**.

Реляционная база данных — база, где данные хранятся в формате таблиц.

они строго структурированы и связаны друг с другом. Данный термин введен в 1969 году ученым из IBM, Эдгаром Коддом. Девять лет спустя технологию стали использовать в коммерческих целях. Преимуществом реляционных баз данных является способность осуществлять связи между элементами, тем самым, облегчая жизнь программисту. Ранее многие задачи решались с помощью процедурных языков программирования реализовывать операции каскадного удаления и обновления данных. Приведем пример: пусть информация о пользователях и заказах, которые они сделали, хранятся в двух файлах: "Пользователь", "Заказы". Для удаления заказа конкретного пользователя нужно организовать проход по каждому из файлов и удаление данных по конкретному условию. Работа с БД во многом организована иначе: SQL является декларативным языком программирования. Для результата мы указываем, **что** хотим получить без указания способа получения. До этого мы работали с императивными языками, где для получения решения мы указывали, как получился результат.



Чтобы получить сумму 2 чисел, мы использовали 3 переменные: в первых 2 находились числа, в 3 - результат:

```
int firstNumber = 1; // первое число int secondNumber = 2; //  
второе число int result = firstNumber + secondNumber;
```

Для решения описанной выше задачи можно и применять БД. Информация хранится в двух измерениях: **строчки и столбцы**. Строчки в таблице часто именуют как "записи", также их называют "кортежами". Столбцы называют "полями" или же "атрибутами". Для удобного запоминания составим небольшую табличку:

Каждая запись разбита на несколько полей, представляющие из себя конкретный элемент. В нашем примере база данных, скорее всего, из нескольких таблиц (преподаватели, кураторы и т.д.). Понимание взаимосвязи таблиц - ключ к пониманию основной архитектуры баз данных.

Чтобы получить представление о работе реляционных БД, необходимо понимать, как организовывается связь между ними. Реляционная БД включает в себя множество таблиц, соединенных друг с другом ключевым полем.

Каждая таблица содержит в себе **первичный ключ**. Первичный ключ (Primary key) – поле(или набор полей) позволяющее однозначно идентифицировать запись в БД. Если ключ состоит из нескольких полей его называют составным. Связь происходит по **внешнему ключу**. Внешний ключ (foreign key) - поле в таблице, значение которого соответствует первичному ключу в другой таблице.

Не всегда в таблице по существующим данным можно однозначно идентифицировать запись, даже используя несколько полей, также значения этих полей могут изменится (изменение первичного ключа может повлечь изменение во многих связанных таблицах). В таких случаях часто используются автоматически генерируемые атрибуты добавляемые к таблице, например числовые последовательности увеличивающиеся при каждой вставке в таблицу.

Суррогатный ключ - автоматически сгенерированное уникальное поле, никак не связанное с информационным содержанием записи.

Естественный ключ — ключ, состоящий из информационных полей таблицы. У таблицы может быть несколько ключей или не быть вообще, если ключей несколько то самый короткий из ключей выбирают в качестве первичного (это может быть как естественный так и суррогатный ключ).

Давайте рассмотрим эти понятия на примере задачи по созданию базы обучающихся, пример списка сведений о студентах, подумайте какие ключи тут можно выделить подумайте 1-2 минуты и разберем варианты:

Студенты				
Фамилия	Имя	Год рождения	Паспорт серия	Паспорт номер
Иванов	Петр	1992	0111	121245
Пупкин	Федор	1995	1102	457879
Смирнов	Иван	1986	0013	787952
Смирнов	Степан	1997	0013	784593
Петрова	Ирина	1996	1802	596485

Серия и номер паспорта — хорошие кандидаты для первичного ключа (потенциальный ключ), набор полей “Фамилия”, “Имя” и “год рождения” — скорее, не будет уникальным с увеличением данных. Вернемся к набору из полей “Серия” и “Номер паспорта” посмотрим на недостатки этого естественного первичного ключа:

- может изменяться
- может отсутствовать
- возможны технические ошибки при вводе, опечатки приводящие к дубликатам

- достаточно широкий (два текстовых поля) — почему это важная характеристика? Если в базе много таблиц будут ссылаться на эту таблицу, то они должны будут использовать ключ таблицы, т. е. хранить эти поля. Пусть наша таблица “Студенты” связана с таблицей “Телефоны”. Таблица телефонов в данном случае будет выглядеть примерно так:

Телефоны студентов					
Студент	Паспорт_серия	Студент	Паспорт_номер	Телефон	Использование
	0111		121245	89211285696	основной
	1102		457879	89651238956	основной
	0013		787952	65428967	основной
	0013		787952	49626548596	дополнительный

Задача выделения первичного ключа не такая простая как могло показаться в начале, в большинстве случаев в реляционных базах используются суррогатные ключи, особенно если таблицы связаны между собой. К таблице добавляется одно числовое поле - id.

Студенты						
Студент	id	Фамилия	Имя	Год рождения	Паспорт серия	Паспорт номер
1	1	Иванов	Петр	1992	0111	121245
2	2	Пупкин	Федор	1995	1102	457879
3	3	Смирнов	Иван	1986	0013	787952
4	4	Смирнов	Степан	1997	0013	784593
5	5	Петрова	Ирина	1996	1802	596485

Но где же можно вести БД?

Базу данных можно вести и на листе бумаги, бумажные ежедневники и блокноты также являются базами данных, на некоторых кстати нанесены по краям цветовые или буквенные метки, что по сути является индексом для упрощения поиска данных. Т.е., под базой данных понимаются непосредственно сами данные. Когда мы говорим про данные на жестком диске компьютера, например, то нам требуются специализированные программы для работы с ними, в русскоязычной терминологии класс таких программ получил название — система управления базами данных, сокращенно СУБД. В англоязычной терминологии DBMS - Database Management System. На сегодняшний день существует огромное множество различных СУБД от коммерческих до открытых разрабатываемых open-source сообществом, использующих разные модели хранения данных, различные технологии поиска и хранения данных. На сайте db-engines.com ежемесячно составляется рейтинг СУБД, сейчас в нем более 350 СУБД (<https://db-engines.com/en/ranking>). Давайте посмотрим на первые 20 из них:

Rank			DBMS	Database Model	Score		
	Jun 2022	May 2022			Jun 2022	May 2022	Jun 2021
1.	1.	1.	Oracle	Relational, Multi-model	1287.74	+24.92	+16.80
2.	2.	2.	MySQL	Relational, Multi-model	1189.21	-12.89	-38.65
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	933.83	-7.37	-57.25
4.	4.	4.	PostgreSQL	Relational, Multi-model	620.84	+5.55	+52.32
5.	5.	5.	MongoDB	Document, Multi-model	480.73	+2.49	-7.49
6.	6.	7.	Redis	Key-value, Multi-model	175.31	-3.71	+10.06
7.	7.	6.	IBM Db2	Relational, Multi-model	159.19	-1.14	-7.85
8.	8.	8.	Elasticsearch	Search engine, Multi-model	156.00	-1.70	+1.29
9.	9.	↑ 10.	Microsoft Access	Relational	141.82	-1.62	+26.88
10.	10.	↓ 9.	SQLite	Relational	135.44	+0.70	+4.90

11.	11.	11.	Cassandra	+	Wide column	115.45	-2.56	+1.34
12.	12.	12.	MariaDB	+	Relational, Multi-model	111.58	+0.45	+14.79
13.	↑ 14.	↑ 26.	Snowflake	+	Relational	96.42	+2.91	+61.67
14.	↓ 13.	↓ 13.	Splunk		Search engine	95.56	-0.79	+5.30
15.	15.	15.	Microsoft Azure SQL Database		Relational, Multi-model	86.01	+0.68	+11.22
16.	16.	16.	Amazon DynamoDB	+	Multi-model	83.88	-0.58	+10.12
17.	17.	↓ 14.	Hive	+	Relational	81.58	-0.03	+1.89
18.	18.	↓ 17.	Teradata	+	Relational, Multi-model	70.41	+2.02	+1.07
19.	19.	↓ 18.	Neo4j	+	Graph	59.53	-0.61	+3.78
20.	20.	20.	Solr		Search engine, Multi-model	56.61	-0.64	+4.52

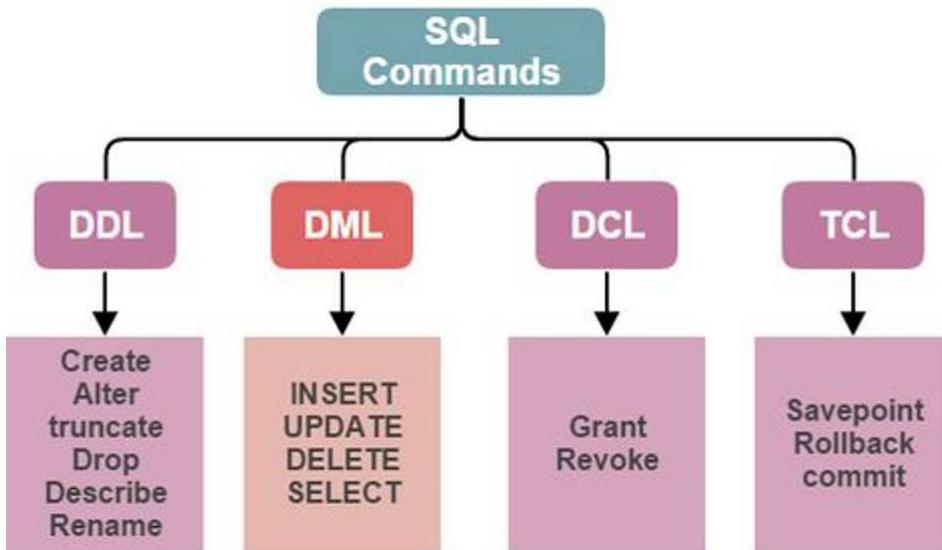
Итак, мы выяснили, что существует множество реляционных СУБД, есть стандартизованный язык SQL, но вместе с тем практически в каждой СУБД реализован свой язык запросов где-то расширяющий стандарт, где-то изменяющий, какие-то конструкции стандарта могут быть не реализованы, такие вариации получили названия **диалектов SQL** у некоторых наиболее популярных даже есть названия:

СУБД	Диалект
Oracle	PL/SQL (Procedural Language SQL)
MSSQL	T/SQL (Transact SQL)
Postgresql	PL/pgSQL (Procedural Language PostGres SQL)

Основные операторы SQL

Оператор (statement) — это наименьшая автономная часть языка программирования, команда или набор команд. (примеры из языков программирования: оператор присваивания `:=`, оператор `IF`, `While`, `For` и другие). Программа представляет собой последовательность операторов.

В языке SQL можно выделить несколько групп операторов по типу выполняемых ими задач:



DDL (Data Definition Language)

Эта группа операторов для определения данных. Они необходимы, когда нужно произвести манипуляции с таблицами. Эти операторы SQL используются в тех случаях, когда нужно создать в базе новую таблицу или, напротив, удалить старую. Они включают в себя следующие командные слова:

- **CREATE** — создание нового объекта в существующей базе.
- **ALTER** — изменение существующего объекта.
- **DROP** — удаление объекта из базы.

DML (Data Manipulation Language)

Эти операторы языка SQL предназначены для манипуляции данными. С помощью операторов мы можем изменять содержимое таблиц. Они помогают добавлять и удалять информацию, изменять значение строк, столбцов и прочих атрибутов. Эти операторы помогают удалить заказ, который отменили клиенты популярного маркетплейса. Это операторы:

- **SELECT** — позволяет выбрать данные в соответствии с необходимым условием.
- **INSERT** — осуществляют добавление новых данных.
- **UPDATE** — производит замену существующих данных.
- **DELETE** — удаление информации.

DCL (Data Control Language)

Операторы, позволяющие предоставлять право доступа к файлам. Делают файлы либо приватными, либо открытыми. Операторы необходимы, чтобы ограничить кого-либо из сотрудников в доступе к информации или, наоборот, позволить работать с базой новому специалисту.

- **GRANT**— предоставляет доступ к объекту.
- **REVOKE**— аннулирует выданное ранее разрешение на доступ.
- **DENY**— запрет, который прекращает действие разрешения.

TCL (Transaction Control Language)

Данный блок операторов предназначен для управления транзакциями. Слово “транзакция” знакомо нам из банковской сферы. На самом деле, транзакция - набор инструкции, которые выполняются как единое целое. У транзакции всего два исхода: проведена успешно, если все необходимые команды выполнены или откат, если в какой-либо инструкции произошёл сбой, то вся операция, включая предыдущие команды, отменяется. Пример - банковские платежи.

Представьте, что вы хотите заказать доставку пиццы, оплатив заказ онлайн. После ввода суммы зачастую банк высылает код подтверждения. При вводе кода мы неожиданно вижуем “Подтверждено”, но если код не был введен

кода мы наслаждаемся вкусной «Пепперони», но если код не был введен, платеж отменяется автоматически

Детально о транзакциях и о требованиях к ним мы поговорим в самом конце нашего курса, на 6 лекции.

- **BEGIN TRANSACTION** — начало транзакции.
- **COMMIT TRANSACTION** — изменение команд транзакции.
- **ROLLBACK TRANSACTION** — отказ в транзакции.
- **SAVE TRANSACTION** — формирование промежуточной точки сохранения внутри операции.

Основные компоненты СУБД

Попробуем построить обобщенную архитектуру СУБД, несмотря на большое количество различных вендоров и СУБД можно выделить общие функциональные блоки:

- подсистема постоянного хранения данных (Storage Engine). В большинстве случаев СУБД использует файлы и каталоги операционной системы, некоторые, например Oracle могут работать напрямую с дисками минуя слой операционной системы. Может использоваться сжатие данных.
- парсер и транслятор запросов (Query parser). СУБД получает от пользователя SQL запрос (это текст) его необходимо проверить на синтаксис, перевести (транслировать) во внутренний формат, определить какие объекты, таблицы например используются
- оптимизатор запросов (Query optimizer) Запрос не определяет четкого алгоритма действий над объектами (в каком порядке соединять таблице, какие индексы использовать и многие другие технические детали), поэтому СУБД пытается построить наиболее оптимальный план выполнения — алгоритм выполнения запроса. Дальше в СУБД будет выполняться выбранный план запроса.
- подсистема выполнения (Query executor). Получает готовый план и шаг за шагом выполняет инструкции.
- системы кэширования данных. Обращение в постоянное хранилище, к жесткому диску довольно дорогостоящий и медленный процесс, поэтому часто используемые данные СУБД пытаются кэшировать — хранить например в оперативной памяти или ssd дисках.

Большинство СУБД используют клиент-серверную архитектуру, на выделенном сервере или кластере устанавливается СУБД, с клиентских компьютеров выполняется подключение и передача запросов, вся вычислительная нагрузка выполняется сервером. Для подключения к СУБД используются компоненты доступа — подпрограммы, иногда их называют драйверами, обычно они предоставляются разработчиком СУБД, для использования их в языках программирования разработаны модули и фреймворки которые скрывают рутинную работу от программиста.

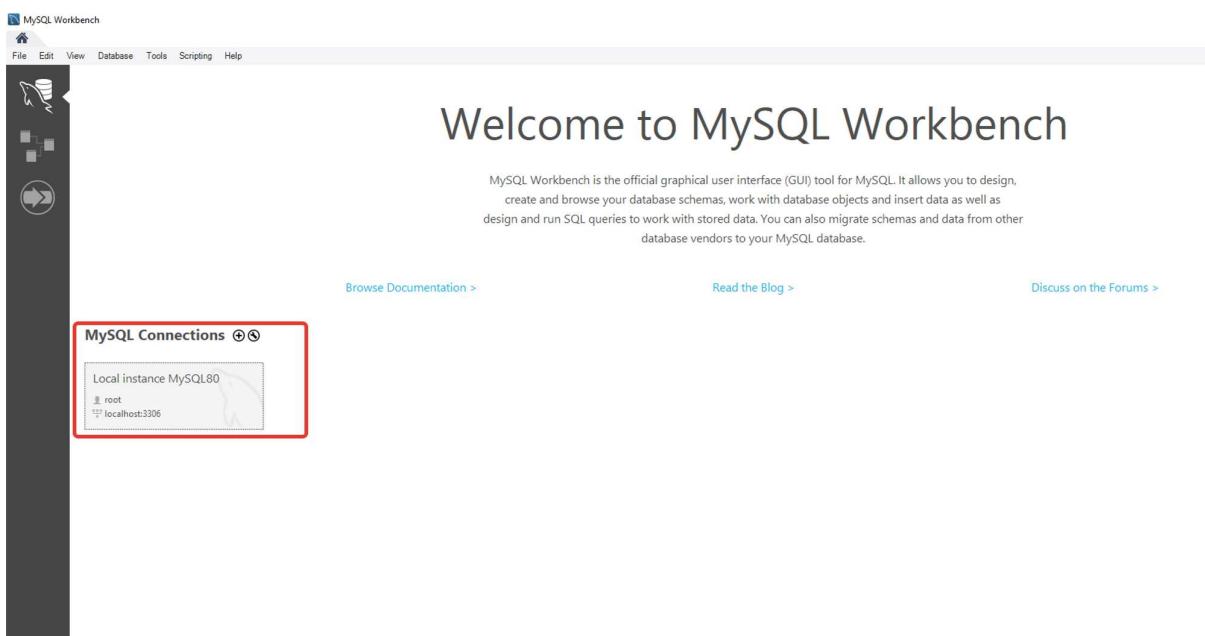
В нашей программе мы будем использовать MySQL.

MySQL появилась в 1995 году. Она изначально была легка, доступна и интуитивно понятна. В итоге ее стали использовать компании со всего мира. В настоящее время система MySQL является негласным стандартом для баз данных.

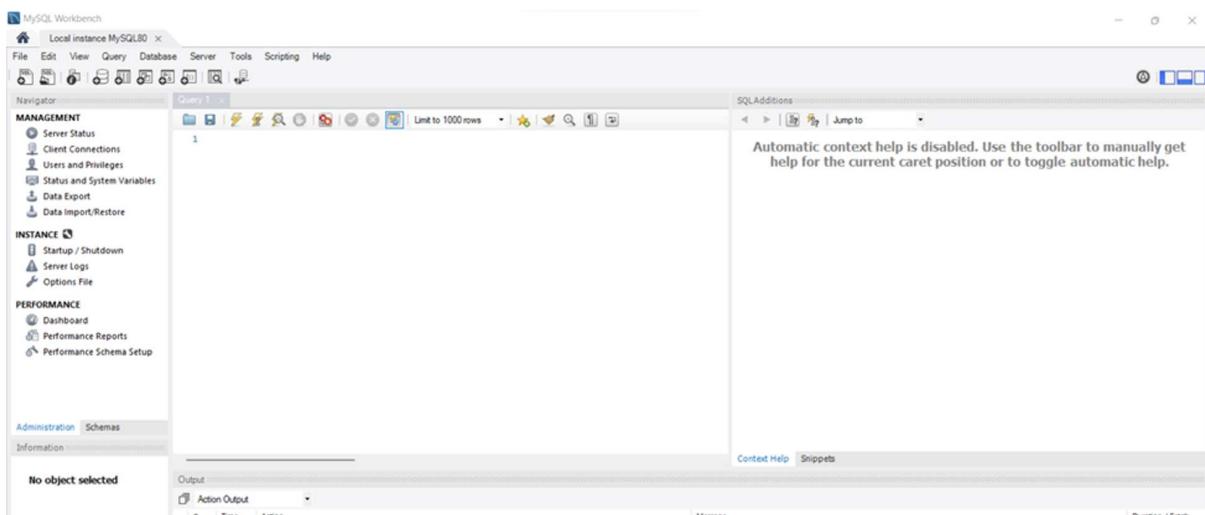
Программа гибкая и простая в использовании. Она даже позволяет пользователям поменять исходный код, чтобы настроить сервер баз данных MySQL конкретно под себя. Доплачивать за это не придется даже в расширенных коммерческих версиях. В установке этой СУБД также нет ничего сложного — процесс займет не больше получаса.

Работа с БД ,используя графический интерфейс, создание и просмотр объектов

Рассмотрим интерфейс MySQL.

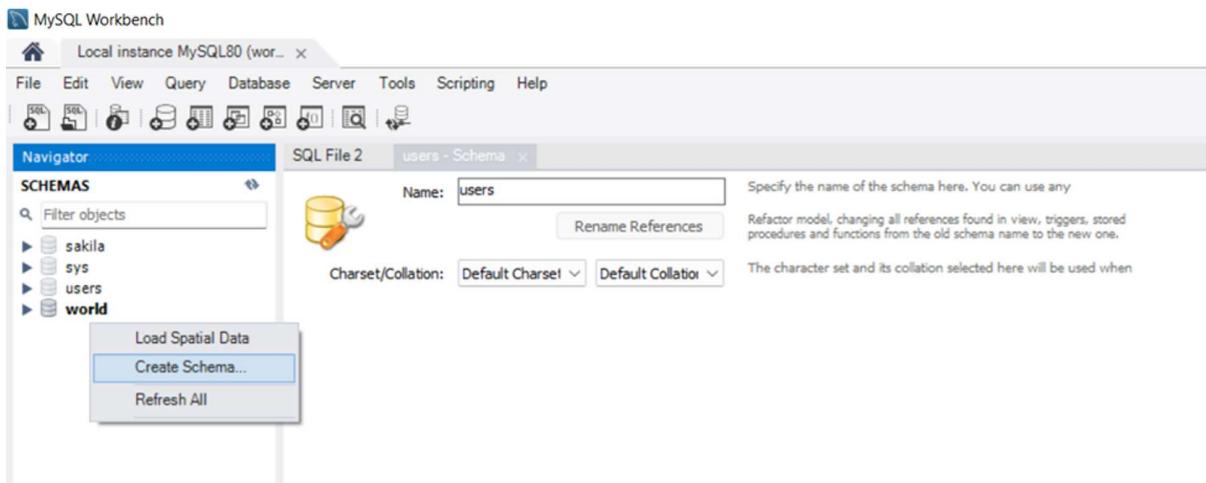


Необходимо войти в нашу БД: для этого нажимаем на имя нашего подключения и вводим данные для входа (их вы прописывали при установке MySQL). Если все выполнится корректно, нас ожидает вот такое окно:

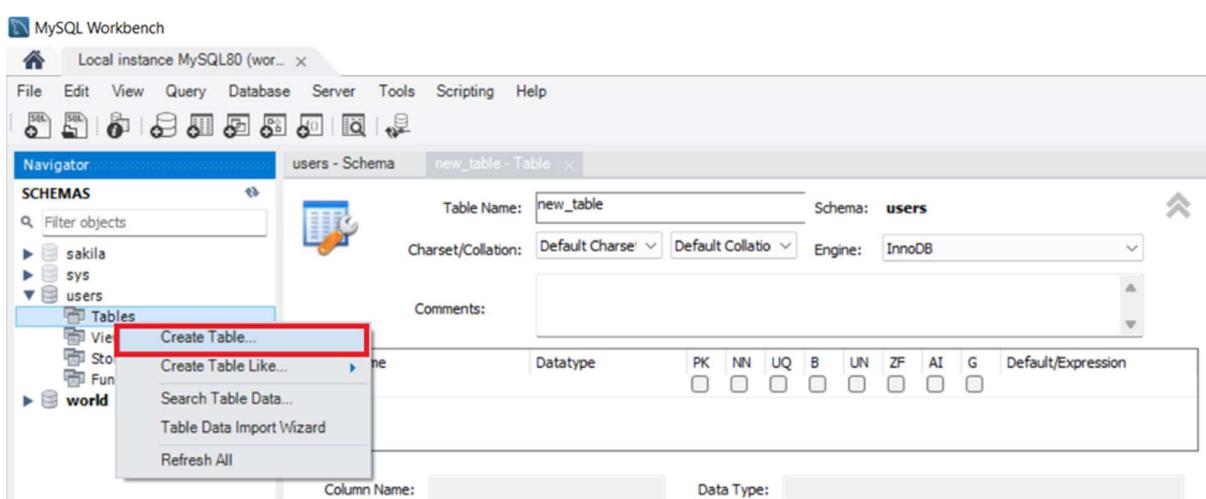




Попробуем создать нашу первую схему: переходим во вкладку “Schemas” в правой нижней части экрана и нажимаем на клавишу создания схемы: UI мы будем использовать только на сегодняшнем уроке. Команды в терминале будем изучать на следующем уроке.



В результате получим следующее окно:



Имя нашей базы данных можно изменить со стандартного на “**users**”.

Обратите внимание, что имя должно стоять из 1-2 слов, без нижних подчеркиваний. Кодировка utf-8 поддерживает и русские символы.

Вернемся к базе данных. Первую табличку назовем **students**. Она будет хранить данные о пользователях информационной системы, в поле “table Name” впишем имя таблицы, в разделе формы “Columns” создадим поля таблицы:

- **Первое поле id** будет содержать уникальный номер пользователя, зададим ему свойства: Auto Increment (AI), Not Null (NN), Primary key(PK) и Unique (UQ), в разделе Data type выберем целочисленный тип *integer*.
- **Второе поле fio**, где будет хранится Ф.И.О. пользователя, установим полю свойства: Not Null, в разделе Data type выберем строковый тип VARCHAR и зададим количество символов в 255.
- **Третье поле login**, будет содержать логин пользователя, оно должно быть уникальным, как и поле id, поэтому установим ему свойство Unique и зададим количество символов в 255.

— Следующие поля: **password** содержащее пароль, **e_email** содержащее адрес электронной почты и поле **type**, содержащее тип пользователя будут без особых свойств, со строковым типом VARCHAR длинной в 255 символов.

После проделанных манипуляций форма с именем таблицы users будет выглядеть так:

Table Name: student Schema: users

Charset/Collation: utf8mb4 Engine: InnoDB

Comments:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
fio	VARCHAR(255)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
login	VARCHAR(255)	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
password	VARCHAR(255)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
email	VARCHAR(255)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL

Column Name: Data Type: Default: Storage:

Charset/Collation: Default Charset Default Collation

Comments:

Storage: Virtual Stored
Primary Key Not Null Unique
Binary Unsigned Zero Fill
Auto Increment Generated

Columns Indexes Foreign Keys Triggers Partitioning Options

Apply Revert

Для сохранения нашей таблицы используем кнопку “Apply”.

Добавим данные в нашу табличку, заполнив первыми значениями:

Tables

- student
- Views
- Stored Procedures
- Functions

Запрос выборки данных с простыми условиями

Давайте разберем синтаксис простого SQL запроса выборки данных и научимся читать его определение. Многие операторы вам уже знакомы

1. Вывод всех данных из таблицы

SELECT * FROM student;

Этим запросом будут выведены все строки из таблицы student и все столбцы (* - означает все доступные столбцы)

2. Вывод ограниченного числа столбцов — нужно явно перечислить столбцы

SELECT fio, login FROM student;

Этим запросом будут выведены все строки из таблицы student, но только

столбцы fio и login

3. Применение фильтров, отбор данных по условиям

SELECT * FROM student WHERE login='test2';

4. Применение оператора “LIKE”:

Оператор имеет ряд символов подстановки:

“%:” - любая подстрока, которая имеет любое количество символов, либо является пустой.

Пример: "WHERE Car LIKE 'Audi%' соответствует таким значениям как: "Audi A3" / "Audi Q3", "Audi TT"

- "Audi A3"
- "Audi Q3"
- "Audi TT"
- "Audi RS4"

“_” - любой одиночный символ

Пример: "WHERE Car LIKE 'Audi A_'" соответствует таким значениям как:

- "Audi A3"
- "Audi A4"
- "Audi A6"
- "Audi A8"

После буквы “A” идет 1 одиночный символ.

Полезные ссылки и рекомендации:

- <https://habr.com/ru/company/oleg-bunin/blog/348172/> - естественные и суррогатные ключи
- Руководство по стилю написания SQL: <https://www.sqlstyle.guide/ru/>
- Страйтесь не использовать русские буквы при работе с СУБД

Книги:

- “Изучаем SQL”, книга Бейли Л.
- Алан Бьюли “Изучаем SQL” (2007)
- Энтони Молинаро “SQL. Сборник рецептов” (2009)



Курс базы данных и SQL.

Лекция 2

На второй лекции будем рассматривать создание объектов: БД, таблиц. Разберемся с типами данных. Узнаем, как с помощью MySQL заполнить таблицу данными; пользоваться логическими операторами и операторами CASE, IF

Что мы узнаем:

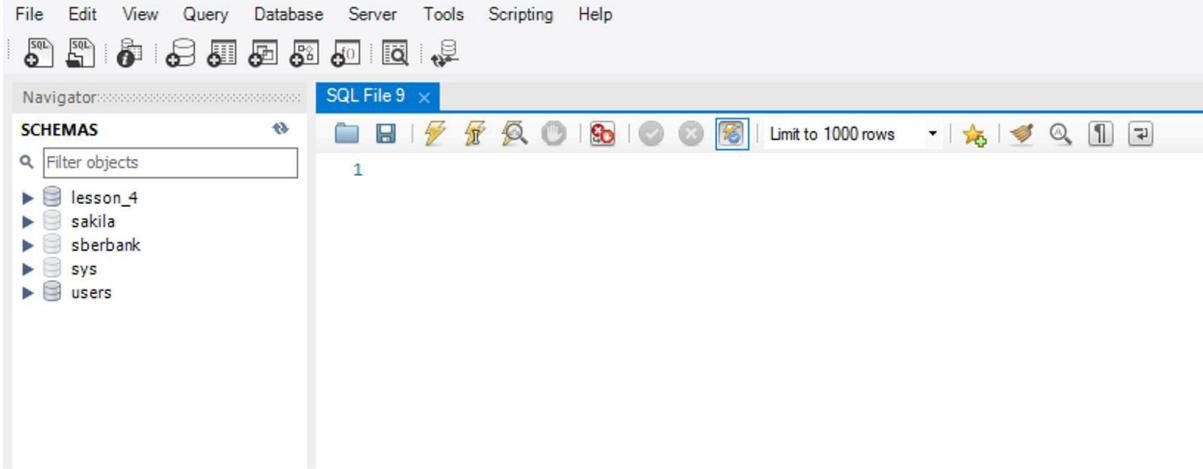
- Группа операторов: DDL (create table, PK, FK)
- Типы данных
- Комментарии
- Группа операторов: DML (insert, update, delete, select)
- Арифметические операции
- Логические операторы (and, or, between, not, in)
- Оператор CASE, IF

Доброго времени суток, уважаемые студенты! Сегодня мы продолжим изучать SQL, затронув следующие операторы DDL, DML.

Начнем с написания SQL - скриптов: создадим тестовую БД и парочку таблиц. Итак, создадим первый файл для скрипта: кнопка находится под кнопкой “File” в левой верхней части экрана.



После нажатия в в верхнем поле



Комментарии

Прежде чем написать наши SQL - запросы, рассмотрим, как же написать комментарии. Комментарии позволяют лучше понимать логику SQL - запроса. Служебные строчки бывают двух видов:

- Однострочные комментарии, обозначаемые двумя дефисами и знаком “решетки”
- Многострочные комментарии начинаются с сочетания символов /* и заканчиваются символами */.

Пример однострочного комментария представлен на первой и второй строкке.

```
-- Однострочный комментарий. После двух дефисов обязательно ставится пробел
# И это тоже комментарий
```

Однострочный комментарий

This screenshot shows a SQL file with the following content:

```
-- Однострочный комментарий. После двух дефисов обязательно ставится пробел
# И это тоже комментарий
```

The first two lines are highlighted with a red box and labeled "Однострочный комментарий" with an arrow pointing to it.

После двух дефисов обязательно нужно ставить “пробел”.

```
-- Однострочный комментарий. После двух дефисов обязательно ставится пробел
#
# И это тоже комментарий
/*
  Многострочный
  комментарий
*/
```

Многострочный комментарий

This screenshot shows a SQL file with the following content:

```
-- Однострочный комментарий. После двух дефисов обязательно ставится пробел
#
# И это тоже комментарий
/*
  Многострочный
  комментарий
*/
```

The fifth line and its block are highlighted with a red box and labeled "Многострочный комментарий" with an arrow pointing to it.

Создание базы данных

Для создания базы используется SQL-запрос CREATE DATABASE.

Рассмотрим подробнее его использование. Новая база данных создается с помощью оператора SQL CREATE DATABASE, который используется для

помощью оператора SQL: **CREATE DATABASE**, за которым следует имя создаваемой базы данных. Для этой цели также используется оператор **CREATE SCHEMA**. Например, для создания новой базы данных под названием **lesson_2** в командной строке mysql нужно ввести следующий запрос:

```
CREATE DATABASE lesson_2; -- В конце ставится точка с запятой -  
завершение оператора
```

Если все прошло нормально, команда сгенерирует следующий вывод:

Query OK, 1 row affected (0.00 sec)

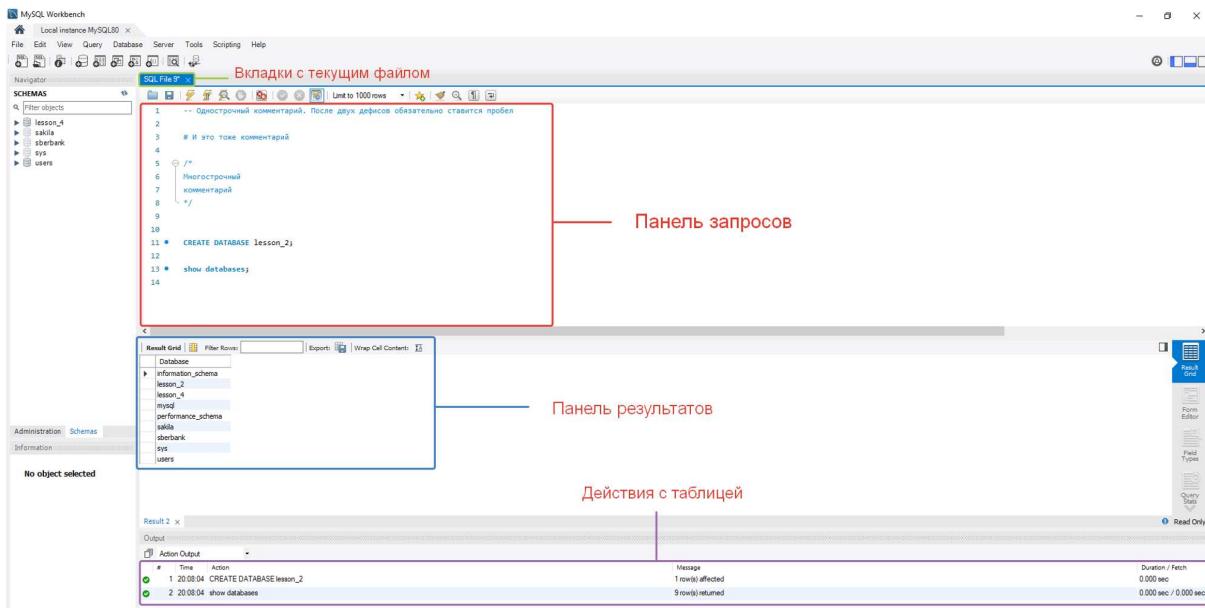
Если указанное имя базы данных конфликтует с существующей базой данных MySQL, будет выведено сообщение об ошибке:

ERROR 1007 (HY000): Can't create database 'lesson_2'; database exists

Проверить, что база появилась можно командой:

```
SHOW DATABASES;
```

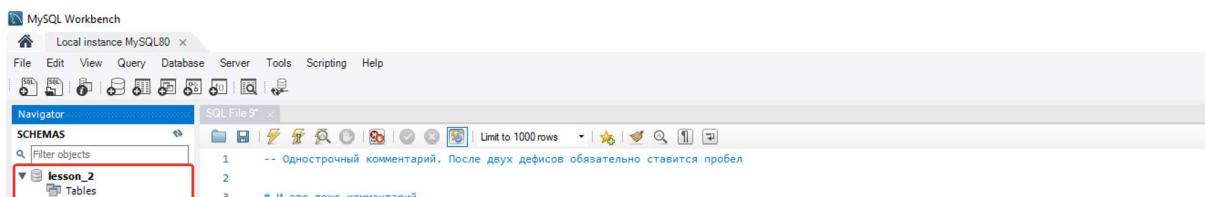
- данная команда выводит в консоль список баз, созданных в СУБД.



Подключиться к базе можно командой:

```
USE lesson_2;
```

Данная команда подключается к базе данных с именем **lesson_2** из списка созданных баз. Если база данных была успешно подключена, то в левой части экрана черным цветом подсвечивается активная БД.



```
Views  
Stored Procedures  
Functions  
lesson_4  
sakila  
sberbank  
sys  
users  
1 /*  
2 Многострочный  
3 комментарий  
4 */  
5  
6  
7  
8  
9  
10  
11 • CREATE DATABASE lesson_2;  
12  
13 • SHOW DATABASES;  
14  
15 • USE lesson_2;  
16
```

Создание таблиц:

Реляционные базы данных хранят данные в таблицах, внутри которой содержатся некоторые столбцы. Каждый столбец имеет имя и тип данных. У столбца есть название и тип данных. Для создания таблиц используется команда **CREATE TABLE**:

```
CREATE TABLE table_name ( column_name_1 column_type_1,  
column_name_2 column_type_2, column_name_N column_type_N, );
```

Имя столбца и таблицы придумать мы можем, но какие же типы данных бывают? Давайте разбираться, какие возможные типы данных можно применять для создания таблиц. После типов данных закрепим наши знания, заполнив БД, как минимум, 2 таблицами. Пример будет немного ниже.

Типы данных

Более детально получить информацию о типах данных можно узнать по ссылке:

MySQL :: MySQL 8.0 Reference Manual :: 11 Data Types

MySQL supports SQL data types in several categories: numeric types, date and time types, string (character and byte) types, spatial types, and the data type. This chapter provides an overview and more



<https://dev.mysql.com/doc/refman/8.0/en/data-types.html>

Для каждого столбца таблицы будет определен тип данных. Неправильное использование типов данных увеличивает как объем занимаемой памяти, так и время выполнения запросов к таблице. Это может быть незаметно на таблицах в несколько строк, но очень существенно, если количество строк будет измеряться десятками и сотнями тысяч, и это далеко не предел для рабочей базы данных. Проведем краткий обзор наиболее часто используемых типов. В MySQL все типы данных делятся на несколько классов: числовые типы, символьные, дата/время и так далее. В каждом классе есть несколько типов данных, которые внешне могут быть похожи, но их поведение или принципы хранения отличаются. Важно выбрать правильный тип сразу при создании таблицы, потому что потом готовую структуру и приложения будет сложней переделать.

Числовые типы

INT — целочисленные значения от –2147483648 до 2147483647, 4 байта.

DECIMAL — хранит числа с заданной точностью. Использует два параметра — максимальное количество цифр всего числа (precision) и количество цифр дробной части (scale). Рекомендуемый тип данных для работы с валютами и координатами. Можно использовать синонимы NUMERIC, DEC, FIXED.

TINYINT — целые числа от –127 до 128, занимает 1 байт хранимой памяти.

BOOL — 0 или 1. Однозначный ответ на однозначный вопрос — false или true. Название столбцов типа boolean часто начинается с is, has, can, allow. По факту это даже не отдельный тип данных, а псевдоним для типа TINYINT (1). Тип настолько востребован на практике, что для него в MySQL создали встроенные константы FALSE (0) или TRUE (1). Можно использовать синоним BOOLEAN.

FLOAT — дробные числа с плавающей запятой (точкой).

Символьные

VARCHAR(N) — N определяет максимально возможную длину строки. Создан для хранения текстовых данных переменной длины, поэтому память хранения зависит от длины строки. Наиболее часто используемый тип строковых данных.

TEXT — подходит для хранения большого объема текста до 65 КБ, например, целой статьи.

CHAR - строка фиксированной длины. Длина хранимой строки указывается в скобках, например, CHAR(10) - строка из десяти символов. И если в таблицу в данный столбец сохраняется строка из 6 символов (то есть меньше установленной длины в 10 символов), то строка дополняется 4 пробелами и в итоге все равно будет занимать 10 символов

Дата и время

DATE — только дата. Диапазон от 1000-01-01 по 9999-12-31. Подходит для хранения дат рождения, исторических дат, начиная с 11 века. Память хранения — 3 байта.

TIME — только время — часы, минуты, секунды — «hh:mm:ss». Память хранения — 3 байта.

DATETIME — соединяет оба предыдущих типа — дату и время. Использует 8 байтов памяти.

TIMESTAMP — хранит дату и время начиная с 1970 года. Подходит для большинства бизнес-задач. Потребляет 4 байта памяти, что в два раза меньше, чем DATETIME, поскольку использует более скромный диапазон дат.

Бинарные

Используются для хранения файлов, фото, документов, аудио и видеоконтента. Все это хранится в бинарном виде.

BLOB — до 65 КБ бинарных данных

LARGELOB — до 4 ГБ.

Отдельно используется NULL. NULL соответствует понятию «пустое поле» null, то есть «поле, не содержащее никакого значения». Введено для того, чтобы различать в полях БД пустые (визуально не отображаемые) значения (например, строку нулевой длины) и отсутствующие значения (когда в поле не записано вообще никакого значения, даже пустого). NULL означает отсутствие, неизвестность информации.

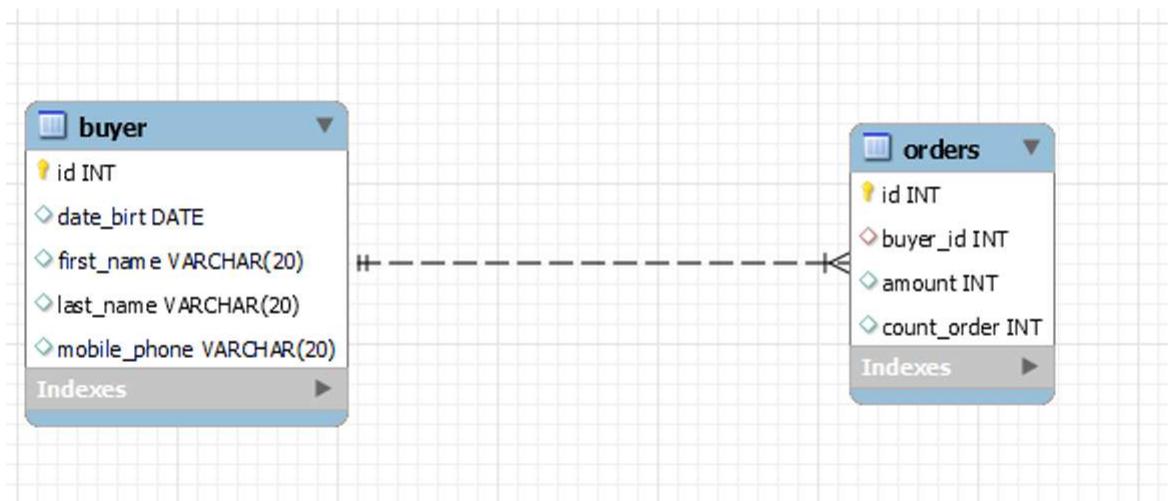
Типы данных изучены и можем приступать к созданию таблиц. Таблицы будем соединять с помощью внешнего ключа.

Создание таблицы: первичные и внешние ключи

Первичные ключи PRIMARY KEY

Атрибут PRIMARY KEY задает первичный ключ таблицы. Первичный ключ уникально идентифицирует строку в таблице. Первичный ключ может представлять любой тип данных. Создадим тестовую БД, заполнив ее двумя таблицами: представим, что у нас есть покупатели, оформившие заказ в Интернет-магазине. Создадим две сущности: “Покупатель”, “Заказы”.

ER - диаграмма и связи: <https://habr.com/ru/post/440556/>



Создадим таблицу “Покупатель”. Он обладает следующими атрибутами (столбцами): фамилия, имя, дата рождения, мобильный телефон.

```
CREATE TABLE Buyer ( id INT PRIMARY KEY AUTO_INCREMENT, date_birt DATE, first_name VARCHAR(20), last_name VARCHAR(20), mobile_phone VARCHAR(20) );
```

Атрибут AUTO_INCREMENT позволяет указать, что значение столбца будет автоматически увеличиваться при добавлении новой строки. Данный атрибут работает для столбцов, которые представляют целочисленный тип или числа с плавающей точкой.

Для второй таблицы мы будем использовать связь путем добавления

внешнего ключа.

внешнего ключа.

Внешние ключи FOREIGN KEY

Внешние ключи позволяют установить связи между таблицами. Внешний ключ устанавливается для столбцов из зависимой, подчиненной таблицы, и указывает на один из столбцов из главной таблицы. Как правило, внешний ключ указывает на первичный ключ из связанной главной таблицы.

Общий синтаксис установки внешнего ключа на уровне таблицы:

```
FOREIGN KEY (столбец) REFERENCES главная_таблица  
(столбец_главной_таблицы)
```

Для создания ограничения внешнего ключа после FOREIGN KEY указывается столбец таблицы, который будет представлять внешний ключ. А после ключевого слова REFERENCES указывается имя связанной таблицы, а затем в скобках имя связанного столбца, на который будет указывать внешний ключ.

Пример схемы базы данных (DDL):

```
CREATE TABLE Buyer ( id INT PRIMARY KEY AUTO_INCREMENT, date_birt  
DATE, first_name VARCHAR(20), last_name VARCHAR(20), mobile_phone  
VARCHAR(20) ); CREATE TABLE Orders ( id INT PRIMARY KEY  
AUTO_INCREMENT, buyer_id INT, amount INT, count_order INT,  
manufacter VARCHAR(45), FOREIGN KEY (buyer_id) REFERENCES Buyer(id)  
);
```

В нашем случае созданы две таблицы: “Buyer” и “Orders”. Таблица “Заказы” связана с таблицей “Покупатели” через внешний ключ: “buyer_id” ссылается на таблицу “Buyer” поле “id”.

Если вдруг наша БД не актуальна, то можно ее удалить с помощью команды:

```
DROP объект имя_объекта;
```

```
DROP DATABASE Test; -- Удалить БД с именем Test
```

Удаление таблицы:

```
DROP TABLE Test; -- Удалить таблицу с именем Test
```

Переименовать таблицу можно с помощью команды RENAME:

```
RENAME TABLE old name TO new name;
```

```
RENAME TABLE buyer TO customer; -- Изменить имя таблицы "buyer" на  
"customer"
```

Небольшая база данных создана, давайте наполним ее первыми данными - следующая часть группы операторов - DML (insert, update, delete).

DML (insert, update, delete, select)

Первая команда, которую мы изучим - заполнение таблицы данными.

- **INSERT – вставка новых данных**

Данный оператор имеет 2 основные формы:

```
-- Пусть имеются 2 столбца в таблице Table: column1, column2 -- 1.  
Заполняется только 1 столбец в таблице Table INSERT Table (column1)  
VALUES (value1); -- 2. Заполняются все столбцы в таблице Table.  
INSERT Table VALUES (value1, value2);
```

Заполним нашу таблицу “Покупатели” данными:

```
-- DATE - format YYYY-MM-DD -- DATETIME - format: YYYY-MM-DD  
HH:MI:SS -- TIMESTAMP - format: YYYY-MM-DD HH:MI:SS -- YEAR -  
format YYYY or YY -- Способ №1 INSERT Buyer (date_birt,  
first_name,last_name,mobile_phone) VALUES ("2023-01-01", "Михаил",  
"Меркушов", "+7-999-888-77-66"), -- id = 1 ("2022-12-31", "Сергей",  
"Сергеев", "60-70-80"), -- id = 2 ("2022-12-30", "Том", "Круз",  
"80-70-80"), -- id = 3 ("2022-01-02", "Филл", "Поляков",  
"+7-999-888-77-55"); -- id = 4
```

```
-- DATE - format YYYY-MM-DD -- DATETIME - format: YYYY-MM-DD  
HH:MI:SS -- TIMESTAMP - format: YYYY-MM-DD HH:MI:SS -- YEAR -  
format YYYY or YY -- Способ №2 INSERT Buyer VALUES (1,  
"2023-01-01", "Михаил", "Меркушов", "+7-999-888-77-66"), (2,  
"2022-12-31", "Сергей", "Сергеев", "60-70-80"), (3, "2022-12-30",  
"Том", "Круз", "80-70-80"), (4, "2022-01-02", "Филл", "Поляков",  
"+7-999-888-77-55");
```

Покупатели сделали заказы в нашем магазине. Чтобы увидеть их, создадим таблицу с заказами клиентов:

```
INSERT Orders (buyer_id, amount,count_order, manufacter) VALUES (1,
```

`1000, 3, "Ягодки"),-- Первый заказ из "Покупатели" по id = 1
 (Меркушов Михал) (1, 400, 2, "Амазон"),-- Второй заказ из
 "Покупатели" по id = 2 (Меркушов Михал) (2, 1200, 5, "Амазон"),
 (3, 2000, 1, "Ягодки"), (4, 5000, 4, "Ягодки"); -- Получается,
 что Меркушов Михаил сделал 2 заказа в 1 день`

Попробуем увидеть эту связь среди таблиц:

```
SELECT Buyer.first_name, Buyer.id, Orders.buyer_id, Orders.amount
FROM Orders, Buyer WHERE Orders.buyer_id = Buyer.id;
```

	first_name	id	buyer_id	amount
▶	Михаил	1	1	1000
	Михаил	1	1	400
	Сергей	2	2	1200
	Том	3	3	2000
	Филл	4	4	5000

Получается, что в операторе SELECT мы указали, какие данные и из какой таблицы мы получаем. Запись "Buyer.id" позволяет получить столбец "id" из таблицы "Buyer", "Orders.buyer_id" позволяет получить столбец "buyer_id" из таблицы "Orders". Чтобы соединить две таблички, мы использовали равенство ключей: если внешний ключ равняется первичному, значит, наш клиент имеет как минимум 1 заказ. В SQL для оптимизации больших запросов применяются "псевдонимы". Они упрощают читаемость кода и немного сокращают его:

- Псевдонимы - временное имя.
- Псевдонимы делает имена столбцов более удобочитаемыми.
- Псевдоним существует только на время выполнения запроса.

Как же добавить псевдонимы? Рассмотрим простые примеры:

1. Псевдоним столбца

1.1. Задается с помощью команды "AS":

```
-- Посчитаем чек по заказу. Для этого умножаю количество на цену:
SELECT amount * count_order AS result -- Псевдоним - result FROM
Orders;
```

	id	buyer_id	amount	count_order		result
▶	1	1	1000	3		3000
	2	1	400	2		800
	3	2	1200	5		6000
	4	3	2000	1		2000
	5	4	5000	4		20000
*	NULL	NULL	NULL	NULL		

Если псевдонима не будет, то столбец с результатом назывался бы "amount * count_order"

	amount * count_order
▶	3000
	800
	6000
	2000
	20000

1.2. Задается с помощью пробела после имени столбца:

```
SELECT amount * count_order result -- Псевдоним - result FROM Orders;
```

После задания столбца можно указать его псевдоним.

2. Псевдоним таблицы - задается аналогично: с помощью пробела и слова AS:

```
-- 1 SELECT B.first_name, B.id, O.buyer_id, O.amount FROM Orders O,
Buyer B WHERE O.buyer_id = B.id;
```

```
-- 2 SELECT B.first_name, B.id, O.buyer_id, O.amount FROM Orders AS O,
Buyer AS B WHERE O.buyer_id = B.id;
```

- **UPDATE** - обновление данных

Команда применяется для обновления уже имеющихся строк:

```
UPDATE имя_таблицы SET столбец1 = значение1, столбец2 = значение2
[WHERE условие_обновления];
```

Представим, что в нашем магазине произошла акция, благодаря которой мы снижаем цены на 25%. Чтобы это сделать, нужно разобраться, какие операторы бывают в SQL. Краткую справку представлю вам прямо сейчас:

Сложение “+”:

```
mysql> SELECT 3+5;
-> 8
```

Вычитание “-”:

```
mysql> SELECT 3-5;
-> -2
```

Умножение “*”:

```
mysql> SELECT 3*5;
-> 15
```

```
mysql> SELECT 18014309500481084*18014309500481084;
```

```
mysql> SELECT 16014398509481984 / 16014398509481984;
```

-> 0

В последнем выражении мы получим неверный результат, так как произведение умножения целых чисел выходит за границы 64-битового диапазона для вычислений с точностью BIGINT.

Деление “/”:

```
mysql> SELECT 3/5;
```

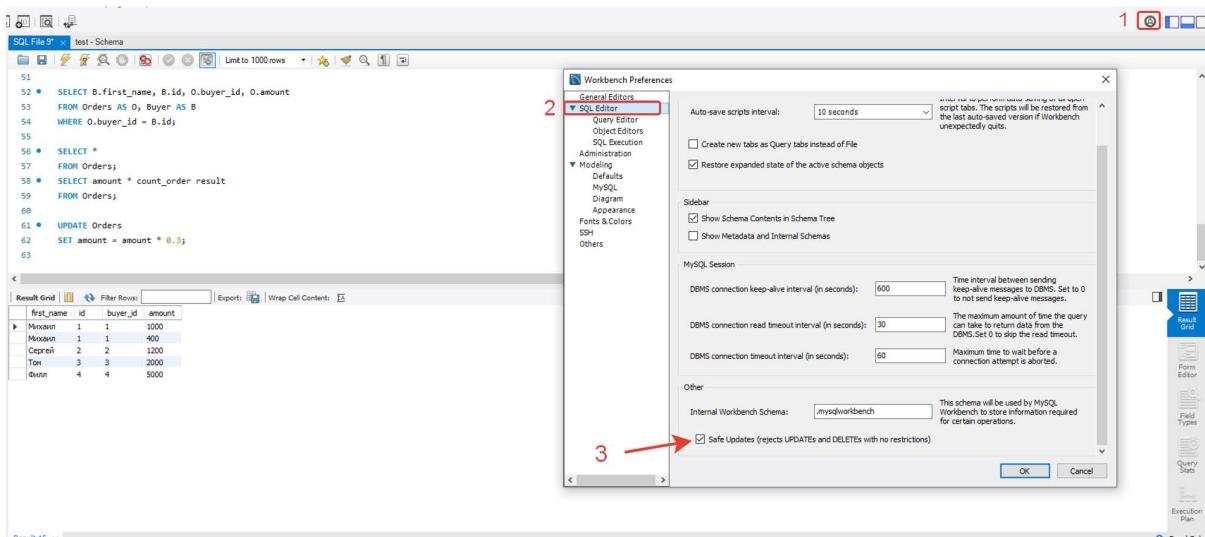
-> 0.60

Деление на ноль приводит к результату NULL:

```
mysql> SELECT 102/0;
```

-> NULL

Приступим к обновлению. Для начала отключим мод, который сохраняет ваши данные в исходном виде: **галочку, отеченную в 3 пункте, необходимо убрать и перезапустить MySQL.**



Готово! Пора скидок активирована:

```
UPDATE Orders SET amount = amount * 0.75; -- 100% цены = 1, 25%  
скидка = 0.25 -- Товар после уценки: 1.00 - 0.25 = 0.75  
SELECT amount new_amount FROM Orders;
```

	id	buyer_id	amount	count_order
▶	1	1	1000	3
	2	1	400	2
	3	2	1200	5
	4	3	2000	1
*	5	4	5000	4
	NULL	NULL	NULL	NULL

	new_amount
▶	750
	300
	900
	1500
	3750

Новая цена стала ниже на 25% :) Проверим, так ли это: 1000 рублей - 100%.

$1000 : 100 = 10$ рублей (1 %), $25\% = 10$ рублей * $25\% = 250$ рублей (25 %).

$1000 - 250 = 750$ рублей.

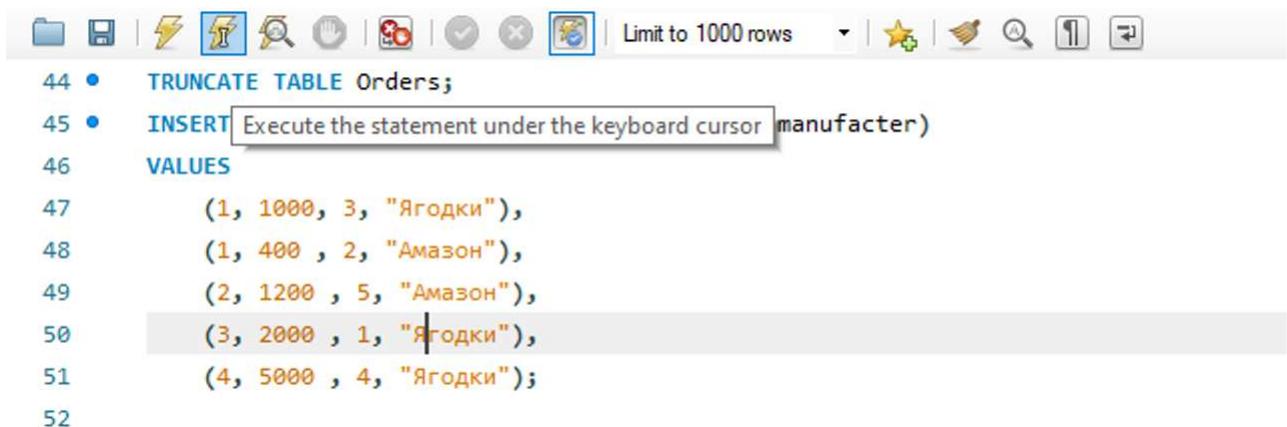
Мы изменили данные в исходной таблице и хотим вернуть исходный результат. Для этого можно полностью очистить таблицу от данных с помощью

результат. Для этого можно полностью очистить таблицу от данных с помощью команды TRUNCATE:

```
TRUNCATE Orders; -- Удаляет все записи из таблицы Orders
SELECT * FROM Orders;
```

	id	buyer_id	amount	count_order	status	manufacter
*	NULL	NULL	NULL	NULL	NULL	NULL

Чтобы заполнить данными нашу табличку, можно заново выполнить операцию заполнения таблицы:



```
44 • TRUNCATE TABLE Orders;
45 • INSERT Execute the statement under the keyboard cursor manufacter)
46 VALUES
47     (1, 1000, 3, "Ягодки"),
48     (1, 400, 2, "Амазон"),
49     (2, 1200, 5, "Амазон"),
50     (3, 2000, 1, "Ягодки"),
51     (4, 5000, 4, "Ягодки");
52
```

Запускаем конкретный запрос. Теперь таблица заполнена.

```
SELECT * FROM Orders;
```

	id	buyer_id	amount	count_order
▶	1	1	1000	3
	2	1	400	2
	3	2	1200	5
	4	3	2000	1
	5	4	5000	4
*	NULL	NULL	NULL	NULL

Небольшое задание:

В нашем магазине действует акция: скидка 50% на заказы, в которых есть минимум 4 товара. Скидка распространяется на 2 покупателей: id = 2, amount = 1200 , count = 5 и id = 4, amount = 5000 , count = 4

Решение:

```
UPDATE Orders SET amount = amount * 0.50 WHERE count_order >= 4;
-- ИЛИ WHERE count_order > 3 SELECT amount new_amount, id FROM Orders;
```

	id	buyer_id	amount	count_order
▶	1	1	1000	3
	2	1	400	2
	3	2	1200	5
	4	3	2000	1
	5	4	5000	4
*	NULL	NULL	NULL	NULL

	id	buyer_id	amount	count_order	manufacter
▶	1	1	1000	3	Ягодки
	2	1	400	2	Амазон
	3	2	600	5	Амазон
	4	3	2000	1	Ягодки
	5	4	2500	4	Ягодки
*	NULL	NULL	NULL	NULL	NULL

И последняя команда, которую мы с вами не затронули - удаление.

- **DELETE** - удаление данных

Синтаксис:

```
DELETE FROM имя_таблицы [WHERE условие_удаления]
```

Пусть в нашей базе хранятся тестовые данные, используемые нами только для тестирования. Необходимость в тестировании пропала и мы можем их удалить:

```
INSERT Buyer (date_birt, first_name, last_name, mobile_phone) VALUES  
("2023-01-01", "Тестовый", "Пользователь", "+7-999-888-77-66"); --  
Добавили клиента DELETE FROM Buyer WHERE first_name='Тестовый'; --  
Удалили строчку со значением
```

Если имеется необходимость объединять несколько условий, то нам потребуются логические операторы.

Логические операторы

Логические операторы позволяют объединить несколько условий. В MySQL можно использовать следующие логические операторы:

AND - операция логического И.

Она объединяет два выражения

выражение1 AND выражение2

Только если оба этих выражения одновременно истинны, то и общее условие оператора AND также будет истинно. Должны быть истинны и первое условие , и второе.

```
-- Получим заказы от 1500 рублей из магазина "Ягодки" SELECT  
amount, count_order FROM Orders WHERE amount > 1500 AND manufacter  
= "Ягодки";
```

	id	buyer_id	amount	count_order	manufacter
▶	1	1	1000	3	Ягодки
	2	1	400	2	Амазон
	3	2	1200	5	Амазон
	4	3	2000	1	Ягодки
	5	4	5000	4	Ягодки

	amount	count_order
▶	2000	1
	5000	4

OR: операция логического ИЛИ.

Она также объединяет два выражения:

выражение1 OR выражение2

Если хотя бы одно из этих выражений истинно, то общее условие оператора OR также будет истинно. Должно быть истинно хотя бы 1 условие: или первое условие , или второе.

```
-- Хотим получить товары или из "Амазона", или товары из диапазона  
(3;5) SELECT amount, count_order, manufacter FROM Orders WHERE  
manufacter = "Амазон" OR count_order > 2 AND count_order < 5; --  
Оператор AND имеет более высокий приоритет, чем OR
```

	id	buyer_id	amount	count_order	manufacter		amount	count_order	manufacter
▶	1	1	1000	3	Ягодки	▶	1000	3	Ягодки
	2	1	400	2	Амазон		400	2	Амазон
	3	2	1200	5	Амазон		1200	5	Амазон
	4	3	2000	1	Ягодки		5000	4	Ягодки
	5	4	5000	4	Ягодки				

NOT: операция логического отрицания. Если выражение в этой операции должно, то общее условие истинно.

```
-- Исключим товары марки "Ягодки" SELECT amount, count_order,  
manufacter FROM Orders WHERE manufacter != "Ягодки"; -- ИЛИ через  
"!=" SELECT amount, count_order, manufacter FROM Orders WHERE NOT  
manufacter = "Ягодки";
```

	id	buyer_id	amount	count_order	manufacter
▶	1	1	1000	3	Ягодки
	2	1	400	2	Амазон
	3	2	1200	5	Амазон
	4	3	2000	1	Ягодки
	5	4	5000	4	Ягодки

	amount	count_order	manufacter
▶	400	2	Амазон
	1200	5	Амазон

Отлично, почти все. Остались операторы, проверяющие истинность значения:

- CASE
- IF

Операторы CASE, IF

1. CASE

Проверяет истинность набора условий и возвращает результат в зависимости от проверки.

```
CASE WHEN condition1 THEN result1 WHEN condition2 THEN result2 WHEN  
conditionN THEN resultN ELSE result END;
```

Чтобы продемонстрировать пример на нашей таблице, немного обновим нашу таблицу. Для изменения таблицы используется оператор “ALTER TABLE”

```
-- Добавить столбец "new_column" в таблицу "Table_name" ALTER TABLE  
Table_name ADD new_column VARCHAR(50); -- Удалить столбец  
"new_column" из таблицы "Table_name" ALTER TABLE Table_name DROP  
COLUMN new_column;
```

Давайте добавим в исходную таблицу столбец “статус”, в котором будет два значения:

- 0 - заказ не оплачен
- 1 - заказ оплачен

Заполнение произведем с помощью функции для получения рандомного числа = **RAND()**.

```
ALTER TABLE Orders ADD COLUMN status INT AFTER count_order; --  
RAND(): https://dev.mysql.com/doc/refman/8.0/en/mathematical-functions.html#function\_rand -- Возвращает числа от 0 до 1 UPDATE  
Orders SET status = RAND();
```

Задачка: в зависимости от значения поля “status” вывести сообщение о факте оплаты: “заказ оплачен”, “оплатите заказ”.

```
SELECT * FROM Orders; SELECT status, -- Перед "CASE" ставится запятая, после перечисления столбцов CASE WHEN status IS TRUE THEN 'заказ оплачен' ELSE 'оплатите заказ' END AS message FROM Orders;  
-- ИЛИ SELECT status, -- Перед "CASE" ставится запятая, после перечисления столбцов CASE WHEN status = 1 THEN 'заказ оплачен'  
ELSE 'оплатите заказ' END AS message FROM Orders;
```

	id	buyer_id	amount	count_order	status	manufacter
▶	1	1	1000	3	1	Ягодки
	2	1	400	2	1	Амазон
	3	2	1200	5	1	Амазон
	4	3	2000	1	1	Ягодки
	5	4	5000	4	0	Ягодки

	status	message
▶	1	заказ оплачен
	0	оплатите заказ

2. Функция IF

Функция IF в зависимости от результата условного выражения возвращает одно из двух значений.

```
IF(условие, значение_для_истины, значение_для_лжи);
```

```
-- Представьте, что мы страхуем заказы со средним чеком от 3000
включительно. -- Сообщим клиентам о наличии или отсутствии
страховки
SELECT status, amount, count_order, manufacter,-- Перед
"IF" тоже ставится запятая
IF(amount * count_order >= 3000,
'Страховка включена в стоимость', 'Страховка оплачивается
отдельно') AS info_message FROM Orders;
```

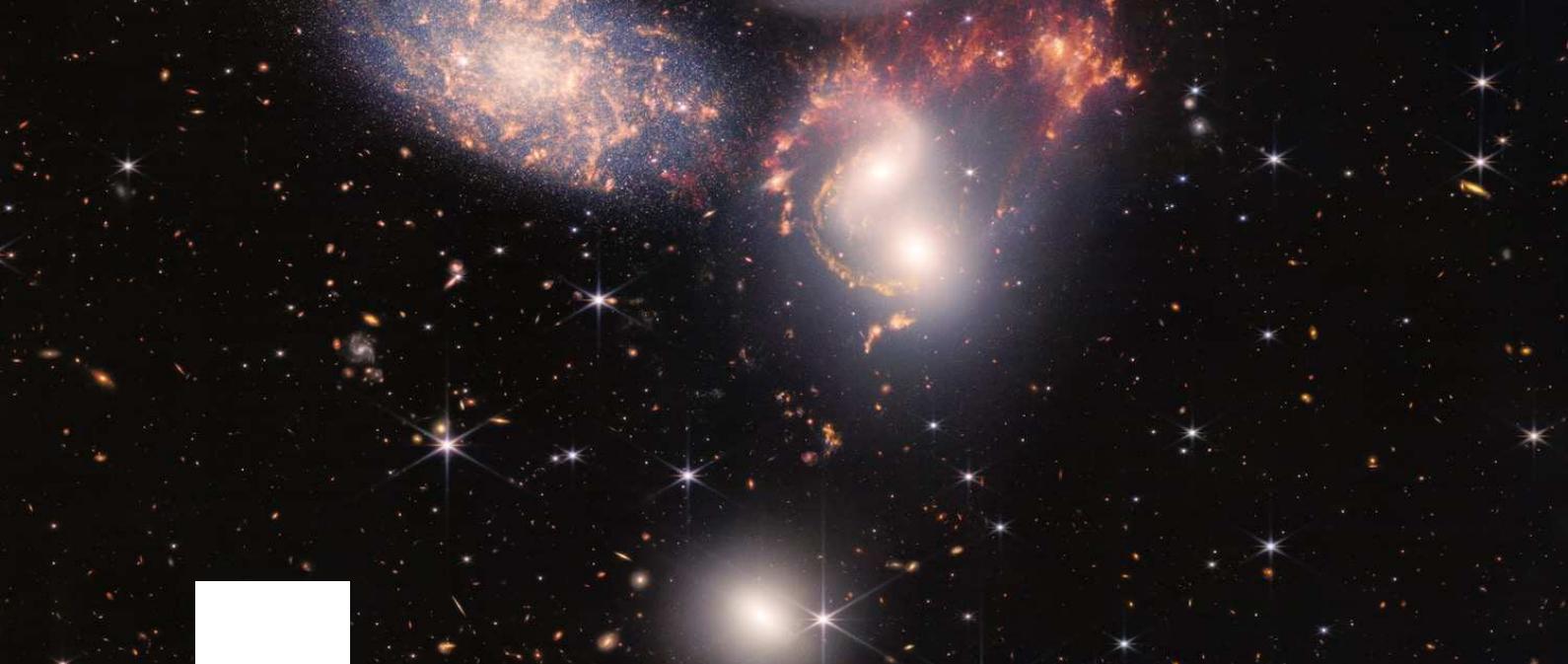
▶	1	1	1000	3	1	Яго	▶	1	1000	3	Ягодки	Страховка включе	
	2	1	400	2	1	Ама		1	400	2	Амазон	Страховка оплачива	
	3	2	1200	5	1	Ама		1	1200	5	Амазон	Страховка включе	
	4	3	2000	1	1	Яго		1	2000	1	Ягодки	Страховка оплачива	
	5	4	5000	4	0	Яго		0	5000	4	Ягодки	Страховка включе	

Полезные ссылки и рекомендации:

- <https://habr.com/ru/company/oleg-bunin/blog/348172/> - DDL, DCL на **MS SQL Server (1 часть)**.
- <https://habr.com/post/255523/> - DDL, DCL на **MS SQL Server (2 часть)**.
- Руководство по стилю написания SQL: <https://www.sqlstyle.guide/ru/>
- ER - диаграмма и связи: <https://habr.com/post/440556/> (связи были изучены, конкретнее изучить ER - диаграммы можно по ссылке)
- Страйтесь не использовать русские буквы при работе с СУБД

Книги:

- "Изучаем SQL", книга Бейли Л.
- Алан Бьюли "Изучаем SQL" (2007)
- Энтони Молинаро "SQL. Сборник рецептов" (2009)



Курс базы данных и SQL.

Лекции 3 и 4

На 3 и 4 лекции будем более детально рассматривать типы данных, рассмотрим структуру запроса, поговорим о понятии "сортировка" и "группировка". Мы научимся ограничивать вывод строк с помощью оператора LIMIT, применим агрегатные функции. Будем получать только уникальные данные из вашей таблицы, применим оператор соединения таблиц(он не такой сложный, но очень интересный), рассмотрим общую структуру подзапросов. Поймем, как соединять два запроса в 1 с помощью UNION и UNION ALL.

Что мы узнаем:

- Немного вспомним типы данных
- Структура запроса
- Сортировка
- Ограничения выборки. LIMIT
- Агрегатные функции
- Уникальные значения
- Подзапросы, подзапросы и еще раз - подзапросы
- UNION ALL и UNION
- JOIN - соединения таблиц

Доброго времени суток, уважаемые студенты!

Постарался приложить побольше картинок для вашего запоминания и углубить наши знания в рамках общего курса по SQL

Терминология

NULL соответствует понятию «пустое поле»null, то есть «поле, не содержащее никакого значения».

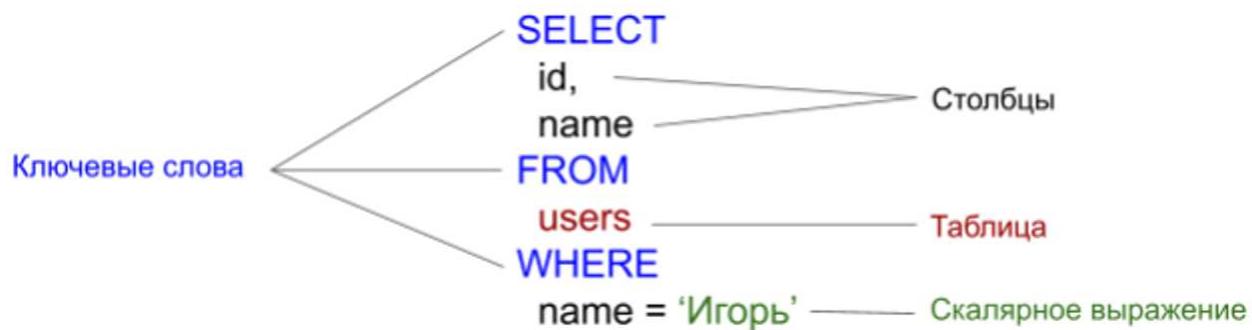
JOIN - оператор языка SQL, который является реализацией операции соединения реляционной алгебры.

Группировка — операция, которая создает из записей таблицы независимые группы записей, по которым проводится анализ.

Агрегатные функции (агрегации) — это функции, которые вычисляются от группы значений и объединяют их в одно результирующее.

Структура любого 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, из какой таблицы).

Скалярное выражение — это сочетание символов и операторов, в результате вычисления которых возвращается одно значение. Фактически, это константы(строчки и числа).

```
SELECT "Привет, мир!"; -- В запросе выше строка "Привет, мир!" –  
скалярное выражение SELECT 2 + 2; --Числа - тоже скалярные  
выражения
```

И... немного обобщим типы данных

Более детально получить информацию о типах данных можно узнать по ссылке:

MySQL supports SQL data types in several categories: numeric types, date and time types, string (character and byte) types, spatial types, and the data type. This chapter provides an overview and more



<https://dev.mysql.com/doc/refman/8.0/en/data-types.html>

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.

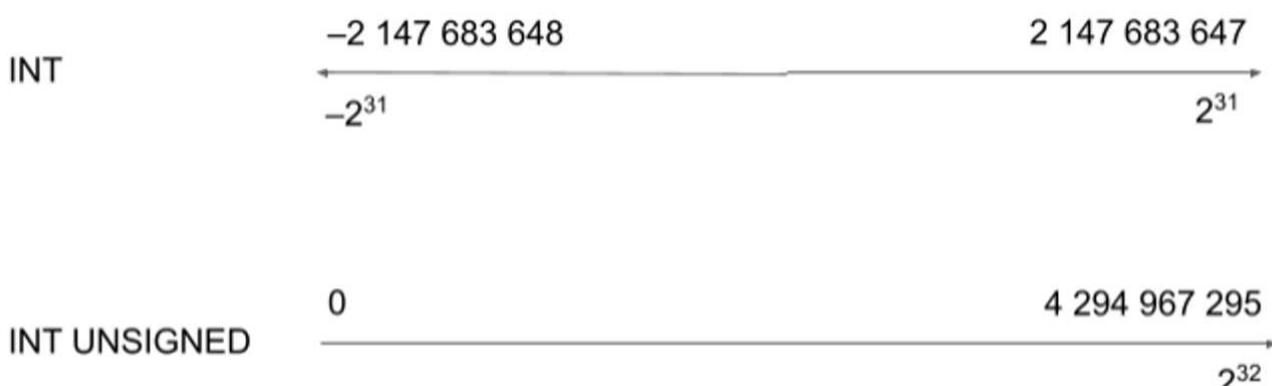
Типы определяют характеристики сохраняемых значений, а также количество памяти, которая под них отводится. Многие типы могут сопровождаться дополнительными атрибутами.

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

- 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.

TINYINT	<input type="text"/>
SMALLINT	<input type="text"/> <input type="text"/>
MEDIUMINT	<input type="text"/> <input type="text"/> <input type="text"/>
INT	<input type="text"/> <input type="text"/> <input type="text"/> <input type="text"/>
BIGINT	<input type="text"/>

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.

```
-- IF EXISTS - проверка на наличие таблицы или любого другого
объекта. -- Таким образом можно запускать скрипт без ошибки о том,
что объект уже создан. -- IF NOT EXISTS - проверка на отсутствие
объекта CREATE TABLE IF NOT EXISTS tb1 (id INT(8)); -- Создадим
таблицу, если ранее такой же не существовало INSERT INTO tb1 VALUES
(5); SELECT * FROM tb1; DROP TABLE IF EXISTS tb1; -- Удалить
таблицу tb1, если она существует CREATE TABLE tb1 (id INT(8)
ZEROFILL); INSERT INTO tb1 VALUES (5); INSERT INTO tb1 VALUES
(5000000000);
```

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. 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 приводят 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

FLOAT	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>				
DOUBLE	<input type="text"/>							
DECIMAL(7,4)	<input type="text"/> 1	<input type="text"/> 1	<input type="text"/> 1	<input type="text"/> .	<input type="text"/> 2	<input type="text"/> 0	<input type="text"/> 0	<input type="text"/> 0

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.

Запись фиксированной длины

INT	INT	CHAR	CHAR
-----	-----	------	------

Запись переменной длины

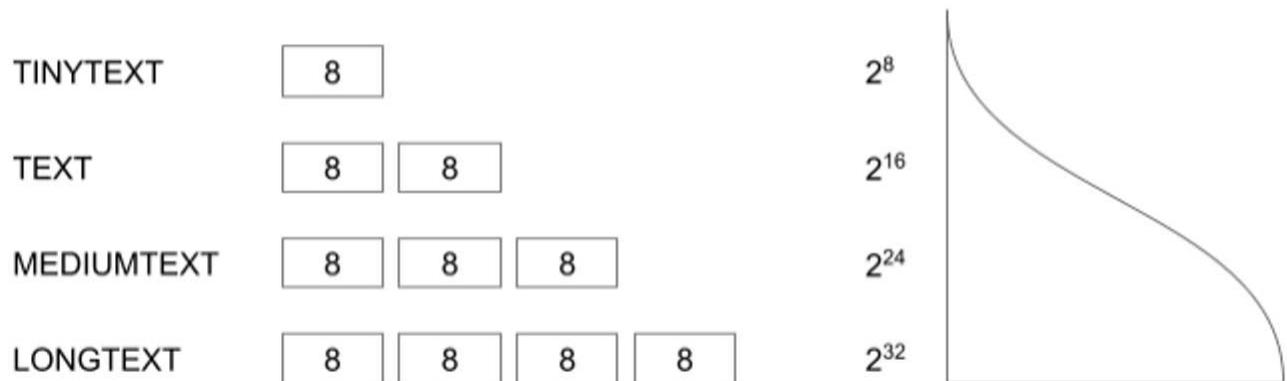
INT	INT	VARCHAR, NULL
-----	-----	---------------

65536

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 длиной **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 аж двумя байтами.



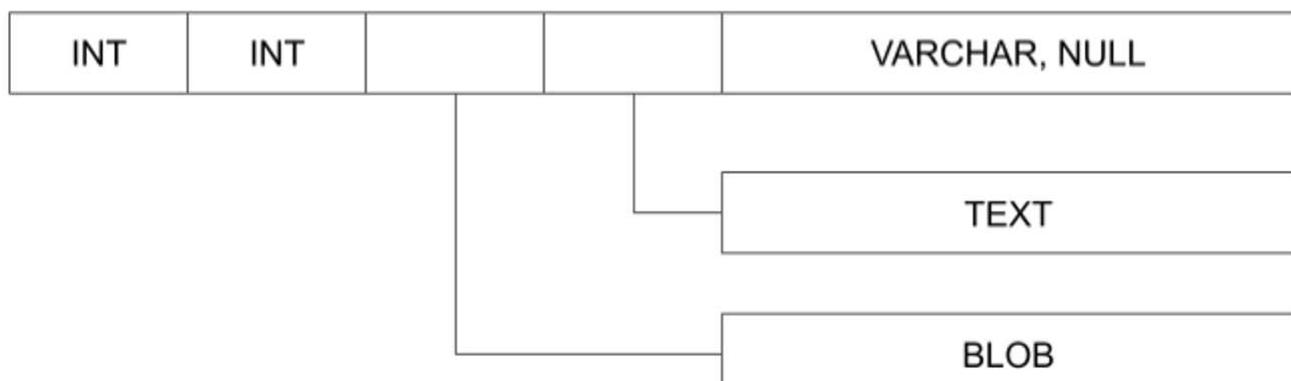
Поэтому для хранения объемного текста используется тип **TEXT**, который, как и **INT**, имеет несколько модификаций. На рисунке прямоугольниками указывается количество байт, которые используются для адресации внутри текстовой строки. Таким образом, при помощи семейства **TEXT**-типов можно закодировать 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.

Запись фиксированной длины

INT	INT	CHAR	CHAR
-----	-----	------	------

Запись переменной длины

65536



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.

Тип NULL

SQL поддерживает специальные типы данных, среди них выделяется NULL — неизвестное значение. Все операции с NULL в качестве результата возвращают NULL. SELECT NULL + 2;

```
SELECT NULL + 2; -- Любая операция с неизвестным значением,
приводит к неизвестному результату
```

Имя столбца и таблицы придумать мы можем, но какие же типы данных бывают? Давайте разбираться, какие возможные типы данных можно применять для создания таблиц. После типов данных закрепим наши знания, заполнив БД, как минимум, 2 таблицами. Пример будет немного ниже.

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
<code>YEAR</code>	0000
<code>DATE</code>	'0000-00-00'
<code>TIME</code>	'00:00:00'
<code>DATETIME</code>	0000-00-00 00:00:00'
<code>TIMESTAMP</code>	'0000-00-00 00:00:00'

`YEAR`



DATE	<input type="text"/>	<input type="text"/>	<input type="text"/>				
TIME	<input type="text"/>	<input type="text"/>	<input type="text"/>				
TIMESTAMP	<input type="text"/>	<input type="text"/>	<input type="text"/>				
DATETIME	<input type="text"/>						

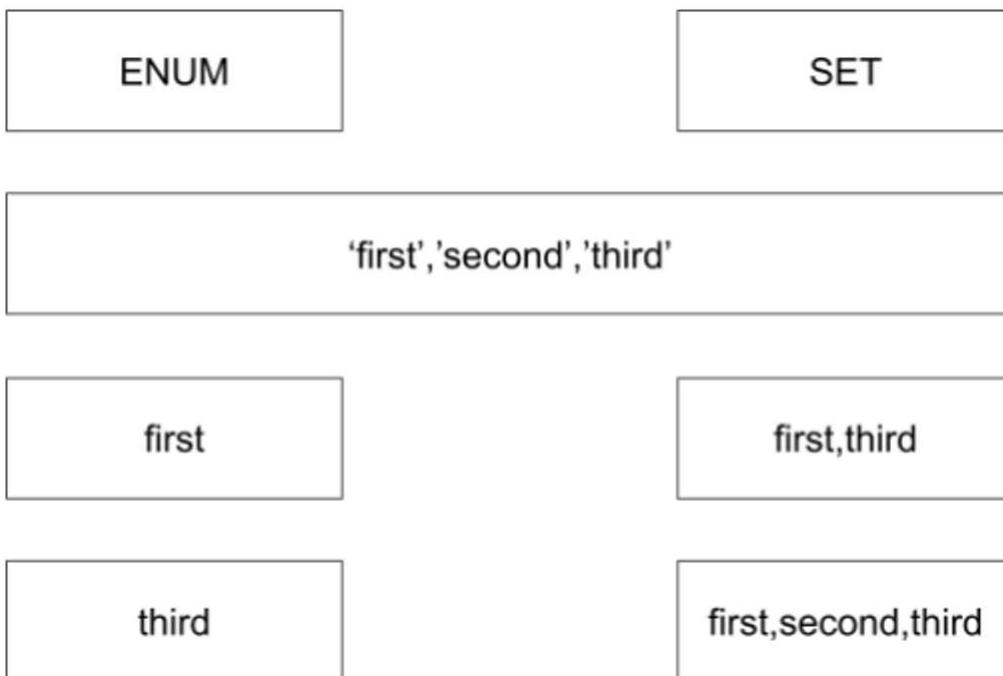
С календарными типами можно проводить операции сложения и вычитания.

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 множества 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:

JSON

JSON — текстовый формат обмена данными, основанный на JavaScript. Как и многие другие текстовые форматы, JSON

☞ <https://ru.wikipedia.org/wiki/JSON>



```
DESCRIBE tbl; -- Получим информацию о столбцах
ALTER TABLE tbl ADD
collect JSON; -- Добавили столбец "collect" с типом JSON
DESCRIBE
tbl; -- Получим информацию о столбцах с новым столбцом "collect"
```

- 1 • DESCRIBE tbl; -- Получим информацию о столбцах
- 2 ALTER TABLE tbl ADD collect JSON; -- Добавили столбец "collect" с типом JSON
- 3 • DESCRIBE tbl; -- Получим информацию о столбцах с новым столбцом "collect"
- 4

MySQL :: MySQL 8.0 Reference Manual :: 11.5 The JSON Data Type

☞ <https://dev.mysql.com/doc/refman/8.0/en/json.html>

	Field	Type	Null	Key	Default	Extra
▶	id	int	YES		NULL	
	collect	json	YES		NULL	

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

Декодирование типа данных JSON MySQL

В этом посте мы собираемся исследовать тип данных JSON в MySQL 5.7 и во время погружения будем использовать

☞ <https://habr.com/ru/post/279155/>

Хабр

Декодирование типа данных JSON MySQL

Разработка

БД для работы. Интернет-магазин

CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP :

MySQL :: MySQL 8.0 Reference Manual :: 11.2.5 Automatic Initialization and Updating for...

☞ <https://dev.mysql.com/doc/refman/8.0/en/timestamp-initialization.html>

CURRENT_TIMESTAMP :

MySQL :: MySQL 8.0 Reference Manual :: 11.2.5 Automatic Initialization and Updating for...

☞ <https://dev.mysql.com/doc/refman/8.0/en/timestamp-initialization.html>

```
-- CURRENT_TIMESTAMP возвращает текущую дату и время -- COMMENT -  
комментарий к объекту DROP TABLE IF EXISTS products; CREATE TABLE  
products ( id INT PRIMARY KEY AUTO_INCREMENT, name VARCHAR(255)  
COMMENT 'Название', description 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'; INSERT INTO products (name, description, price,  
catalog_id) VALUES ('Intel Core i3-8100', 'Процессор для настольных  
персональных компьютеров, основанных на платформе Intel.', 7890.00,  
1), ('Intel Core i5-7400', 'Процессор для настольных персональных  
компьютеров, основанных на платформе Intel.', 12700.00, 1), ('AMD  
FX-8320E', 'Процессор для настольных персональных компьютеров,  
основанных на платформе AMD.', 4780.00, 1), ('AMD FX-8320',  
'Процессор для настольных персональных компьютеров, основанных на  
платформе AMD.', 7120.00, 1), ('ASUS ROG MAXIMUS X HERO',  
'Материнская плата ASUS ROG MAXIMUS X HERO, Z370, Socket 1151-V2,  
DDR4, ATX', 19310.00, 2), ('Gigabyte H310M S2H', 'Материнская плата  
Gigabyte H310M S2H, H310, Socket 1151-V2, DDR4, mATX', 4790.00, 2),  
('MSI B250M GAMING PRO', 'Материнская плата MSI B250M GAMING PRO,  
B250, Socket 1151, DDR4, mATX', 5060.00, 2); CREATE TABLE users (  
id INT PRIMARY KEY AUTO_INCREMENT, 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'; INSERT INTO users (name, birthday_at) VALUES  
( 'Геннадий', '1990-10-05' ), ( 'Наталья', '1984-11-12' ),  
( 'Александр', '1985-05-20' ), ( 'Сергей', '1988-02-14' ), ( 'Иван',  
'2001-01-12' ), ( 'Мария', '2002-08-29' );
```

Сортировка

Запрос выдает результаты в том порядке, в котором они хранятся в базе данных. Однако часто требуется отсортировать значения по одному из столбцов. Это делается при помощи конструкции **ORDER BY**. После конструкции **ORDER BY** указывается столбец (или столбцы), по которому следует сортировать данные. По умолчанию сортировка производится в прямом порядке **ASC**, однако, добавив после имени столбца ключевое слово **DESC**, можно добиться сортировки в обратном порядке.

Синтаксис:

```
SELECT expressions FROM tables [WHERE conditions] ORDER BY  
expression [ ASC | DESC ];
```

Пример:

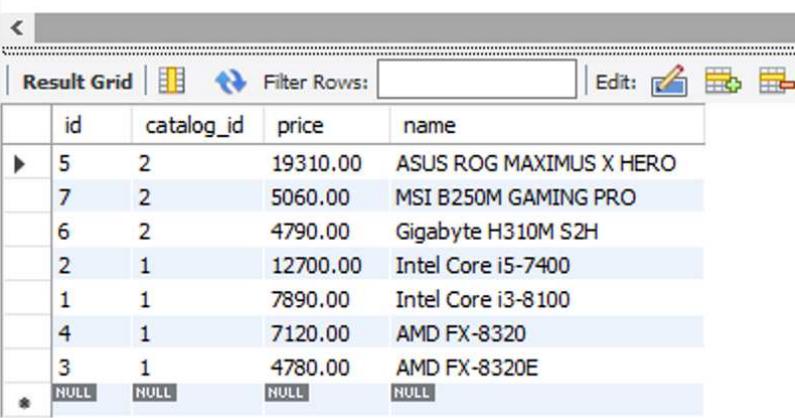
```
SELECT * FROM products ORDER BY id; -- Запрос сортирует результат выборки по полю id -- Чтобы отсортировать таблицу по каталогам, -- в рамках каждого каталога, по цене, мы можем указать -- после ключевого слова ORDER BY сначала поле catalog_id, а затем поле price: SELECT id, catalog_id, price, name FROM products ORDER BY catalog_id, price;
```

Ключевое слово `DESC` относится только к полю `price`, и чтобы отсортировать оба столбца в обратном порядке, потребуется снабдить `DESC` как `id_catalog`, так и `price`:

```
SELECT id, catalog_id, price, name FROM products ORDER BY catalog_id DESC, price DESC;
```

Если в `ORDER BY` указаны 2 столбца, то сначала сортировка производится по первому столбцу, затем - по второму. Рассмотрим пример для запроса выше:

```
39 •  SELECT id, catalog_id, price, name
40      FROM products
41      ORDER BY catalog_id DESC, price DESC
42
```



	id	catalog_id	price	name
▶	5	2	19310.00	ASUS ROG MAXIMUS X HERO
	7	2	5060.00	MSI B250M GAMING PRO
	6	2	4790.00	Gigabyte H310M S2H
	2	1	12700.00	Intel Core i5-7400
	1	1	7890.00	Intel Core i3-8100
	4	1	7120.00	AMD FX-8320
*	3	1	4780.00	AMD FX-8320E
	NULL	NULL	NULL	NULL

- Сортировка по убыванию (от большего к меньшему) по столбцу `id_catalog`: от 2 до 1.
- Для данных, которые отсортированы по убыванию, происходит еще 1 сортировка: по убыванию цены.

Ограничения выборки. Оператор LIMIT

Результат выборки может содержать сотни и тысячи записей, их вывод и обработка занимают

значительное время и серьезно нагружают сервер базы данных. Поэтому информацию часто

разбивают на страницы и предоставляют ее пользователю порциями.

Извлечение только части запроса требует меньше времени и вычислений, кроме того, пользователю часто бывает достаточно посмотреть первые несколько записей. Постраничная навигация используется при помощи

ключевого слова **LIMIT**, за которым следует число выводимых записей.

```
SELECT * FROM products ORDER BY name LIMIT 2;
```

Здесь извлекаются первые две записи таблицы `products`, при этом записи сортируются по полю `name`. Чтобы извлечь следующие две записи, используется ключевое слово **LIMIT** с двумя числами. Первое указывает позицию, начиная с которой необходимо вернуть результат, а второе — количество извлекаемых записей.

```
SELECT * FROM products ORDER BY name LIMIT 2, 2;
```

Существует и альтернативная форма записи такого оператора, с использованием ключевого слова **OFFSET**:

```
SELECT * FROM products ORDER BY name LIMIT 2 OFFSET 2;
```

Уникальные значения

Очень часто возникает задача вывода уникальных значений из таблицы. Для этого перед именем столбца можно использовать ключевое слово **DISTINCT**:

```
SELECT DISTINCT catalog_id FROM products ORDER BY catalog_id;
```

Данный запрос возвращает уникальные номера каталогов, без повторений

Группировка данных

Мы уже знаем механизм получения уникальных значений. В языке SQL для работы с такими группами предназначено специальное ключевое слово **GROUP BY**.

Синтаксис: (в “[]” указаны необязательные части)

```
SELECT столбцы FROM таблица [WHERE условие_фильтрации_строк] [GROUP BY столбцы_для_группировки] [HAVING условие_фильтрации_групп] [ORDER BY столбцы_для_сортировки]
```

Пример:

```
-- Уникальные значения SELECT DISTINCT catalog_id FROM products
ORDER BY catalog_id; -- Группировка данных через GROUP BY SELECT
catalog_id FROM products GROUP BY catalog_id;
```

В качестве значений для создания групп могут выступать не только столбцы таблицы, но и

вычисляемые значения. Например, давайте разделим пользователей в таблице на три группы:

родившихся в 80-х, 90-х и 2000-х годах. Для этого из даты рождения можно получить только первые 3 цифры. К примеру:

- 2001 год - первые 3 цифры - это "200" - обозначает, что человек родился в 2000 годах
- 1991 год - первые 3 цифры - это "199" - обозначает, что человек родился в 90 - х годах

```
SELECT id, name, SUBSTRING(birthday_at, 1, 3) FROM users;
```

The screenshot shows the MySQL Workbench interface with a query editor containing the following SQL code:

```
19 • SELECT id, name, SUBSTRING(birthday_at, 1, 3) FROM users;
```

Below the query editor is a results grid. The grid has four columns: id, name, and two unnamed columns for the result of the SUBSTRING function. The data is as follows:

	id	name	SUBSTRING(birthday_at, 1, 3)
▶	1	Геннадий	199
	2	Наталья	198
	3	Александр	198
	4	Сергей	198
	5	Иван	200
	6	Мария	200

Здесь мы преобразуем календарный тип `DATETIME` поля `birthday_at` к строковому значению и при помощи функции `SUBSTRING` `SUBSTRING` извлекаем первые три цифры года рождения. Давайте назначим вычисляемому значению псевдоним при помощи ключевого слова `AS` и отсортируем значения при помощи `ORDER BY`

```
SELECT id, name, SUBSTRING(birthday_at, 1, 3) AS decade FROM users
ORDER BY decade;
```

The screenshot shows the MySQL Workbench interface with a query editor containing the following SQL code:

```
20 • SELECT
21     id, name,
22     SUBSTRING(birthday_at, 1, 3) AS decade
23     FROM users
24     ORDER BY decade;
```

Below the query editor is a results grid. The grid has four columns: id, name, and two unnamed columns for the result of the SUBSTRING function and its alias. The data is as follows:

	id	name	decade
◀			

▶	2	Наталья	198
	3	Александр	198
	4	Сергей	198
	1	Геннадий	199
	5	Иван	200
	6	Мария	200

Обратите внимание, что мы можем использовать псевдоним decade в конструкции `ORDER BY`.

При помощи конструкции `GROUP BY` мы можем сгруппировать поля по декадам.

Каждая из групп содержит в себе несколько пользователей и непонятно, какого из них следует выводить. Ранее MySQL выводила случайного пользователя, однако сейчас такое поведение отменено. Такой режим по-прежнему можно включить, СУБД даже подсказывает в сообщении об ошибке, как это можно сделать. Однако лучше этого не делать, чтобы ваш SQL-код оставался совместимым с другими СУБД. Какую пользу можно извлечь из сгруппированных значений? MySQL предоставляет несколько функций, которые называются агрегатными. Они позволяют работать с содержимым групп, полученных `GROUP BY`. Например, мы можем подсчитать количество записей внутри каждой из

групп:

```
SELECT COUNT(*), SUBSTRING(birthday_at, 1, 3) AS decade FROM users
GROUP BY decade;
```

```
26 •   SELECT COUNT(*),
27       SUBSTRING(birthday_at, 1, 3) AS decade
28   FROM users
29   GROUP BY decade;
30
```

The screenshot shows the MySQL Workbench interface with the query editor containing the code above. Below it, the results are displayed in a grid:

	COUNT(*)	decade
▶	1	199
	3	198
	2	200

Таким образом, у нас 3 пользователя родились в 80-х, два в 90-х и один в 2000-х. Полученные значения мы по-прежнему можем сортировать при помощи конструкции `ORDER BY`.

```
SELECT COUNT(*), SUBSTRING(birthday_at, 1, 3) AS decade FROM users
GROUP BY decade ORDER BY decade DESC;
```

Причем сортировать можно не группируемому значению, но и по любому другому полю. Например, давайте назначим функции `COUNT()` псевдоним `total` и отсортируем результаты по этому значению:

```
SELECT COUNT(*) AS total, SUBSTRING(birthday_at, 1, 3) AS decade
FROM users GROUP BY decade ORDER BY total DESC;
```

Посмотреть содержимое группы мы можем при помощи специальной функции **GROUP_CONCAT** :

```
SELECT GROUP_CONCAT(name), SUBSTRING(birthday_at, 1, 3) AS decade
FROM users GROUP BY decade;
```

```
26 •   SELECT
27      GROUP_CONCAT(name),
28      SUBSTRING(birthday_at, 1, 3) AS decade
29  FROM users
30  GROUP BY decade;
31
```

	GROUP_CONCAT(name)	decade
▶	Наталья,Александр,Сергей	198
	Геннадий	199
	Иван,Мария	200

Функция **GROUP_CONCAT** допускает задание разделителя, для этого внутри функции используется ключевое слово **SEPARATOR**. Давайте зададим в качестве разделителя пробел:

```
SELECT GROUP_CONCAT(name SEPARATOR ' '), SUBSTRING(birthday_at, 1,
3) AS decade FROM users GROUP BY decade;
```

```
26 •   SELECT
27      GROUP_CONCAT(name SEPARATOR ' '),
28      SUBSTRING(birthday_at, 1, 3) AS decade
29  FROM users
30  GROUP BY decade;
31
```

	GROUP_CONCAT(name SEPARATOR '')	decade
▶	Наталья Александр Сергей	198
	Геннадий	199
	Иван Мария	200

Ключевое слово **ORDER BY** позволяет отсортировать значения в рамках возвращаемой строки. Давайте отсортируем имена пользователей в обратном порядке:

```
SELECT GROUP_CONCAT(name ORDER BY name DESC SEPARATOR ' '),
SUBSTRING(birthday_at, 1, 3) AS decade FROM users GROUP BY decade;
```

```
26 •  SELECT
27      GROUP_CONCAT(name ORDER BY name DESC SEPARATOR ' '),
28      SUBSTRING(birthday_at, 1, 3) AS decade
29  FROM users
30  GROUP BY decade;
31
32
```

	GROUP_CONCAT(name ORDER BY name DESC SEPARATOR ' ')	decade
▶	Сергей Наталья Александр	198
	Геннадий	199
	Мария Иван	200

Агрегационные функции позволяют получать результаты для каждой из групп в отдельности. Чаще при составлении условий требуется ограничить выборку по результату функции, например выбрать группы, где количество записей больше или равно двум.

Использование для этих целей конструкции `WHERE` приводит к ошибке. Для решения этой проблемы вместо ключевого слова `WHERE` используется ключевое слово `HAVING`, которое располагается вслед за конструкцией `GROUP BY`. Выберем группы людей (рожденных в 80, 90, 2000х), где количество людей больше, чем 2 человека:

```
SELECT COUNT(*) AS total, SUBSTRING(birthday_at, 1, 3) AS decade
FROM users GROUP BY decade HAVING total >= 2;
```

```
26 •  SELECT
27      COUNT(*) AS total,
28      SUBSTRING(birthday_at, 1, 3) AS decade
29  FROM users
30  GROUP BY decade
31  HAVING total >= 2;
32
33
```

	total	decade
▶	3	198
	2	200

Агрегатные функции

Агрегатные функции (агрегации) — это функции, которые вычисляются от группы значений и объединяют их в одно результирующее.

Количество записей в таблице можно узнать при помощи функции `COUNT()`, которая принимает в качестве аргумента имя столбца. Функция возвращает число строк в таблице, значения столбца для которых отличны от `NULL`.

В качестве параметра функции наряду с именами столбцов может выступать символ звездочки (*). При использовании символа * будет возвращено число строк таблицы независимо от того, принимают какие-то из них значение `NULL` или нет.

```
SELECT COUNT(*) FROM catalogs; -- Количество строк в таблице
SELECT COUNT(id) FROM catalogs; -- Количество id-шников в таблице
```

Конструкция `GROUP BY` разбивает таблицу на отдельные группы. Функция `COUNT()` возвращает результат для каждой из этих групп.

```
SELECT catalog_id, COUNT(*) AS total FROM products GROUP BY
catalog_id;
```

	catalog_id	total
▶	1	4
	2	3

Функции `MIN()` и `MAX()` возвращают минимальное и максимальное значения столбца.

Получим максимальную и минимальную цену:

```
SELECT MIN(price) AS min, MAX(price) AS max FROM products;
```

Сверху - таблица искомая, снизу - результат запроса:

	id	name	description	price	catalog_id	created_at	updated_at
▶	1	Intel Core i3-8100	Процессор для настольных персональных ко...	7890.00	1	2023-03-17 13:33:44	2023-03-17 13:33:44
	2	Intel Core i5-7400	Процессор для настольных персональных ко...	12700.00	1	2023-03-17 13:33:44	2023-03-17 13:33:44
	3	AMD FX-8320E	Процессор для настольных персональных ко...	4780.00	1	2023-03-17 13:33:44	2023-03-17 13:33:44
	4	AMD FX-8320	Процессор для настольных персональных ко...	7120.00	1	2023-03-17 13:33:44	2023-03-17 13:33:44
	5	ASUS ROG MAXIMUS X HERO	Материнская плата ASUS ROG MAXIMUS X HE...	19310.00	2	2023-03-17 13:33:44	2023-03-17 13:33:44
	6	Gigabyte H310M S2H	Материнская плата Gigabyte H310M S2H, H31...	4790.00	2	2023-03-17 13:33:44	2023-03-17 13:33:44
	7	MSI B250M GAMING PRO	Материнская плата MSI B250M GAMING PRO, ...	5060.00	2	2023-03-17 13:33:44	2023-03-17 13:33:44
*	HULL	HULL	HULL	HULL	HULL	HULL	HULL

	min	max
▶	4780.00	19310.00

Функция `AVG()` возвращает среднее значение аргумента. Давайте подсчитаем среднюю цену товара в нашем магазине. Чтобы проверить поиск среднего значения, вспомним формулу: сумма всех значений / количество. Если решать задачку поиска среднего, то можно решить эту же задачку математически:

`SUM(price) / COUNT(price)`

Для поиска суммы используем `SUM()` - поиск суммы внутри столбца.

```
SELECT AVG(price) AS "Средняя цена через AVG", SUM(price) AS "Сумма товаров через SUM", COUNT(price) AS "Количество всех товаров через COUNT", SUM(price) / COUNT(price) AS "Проверка ср. арифм" FROM products;
```

	Средняя цена через AVG	Сумма товаров через SUM	Количество всех товаров через COUNT	Проверка ср. арифм
▶	8807.142857	61650.00	7	8807.142857

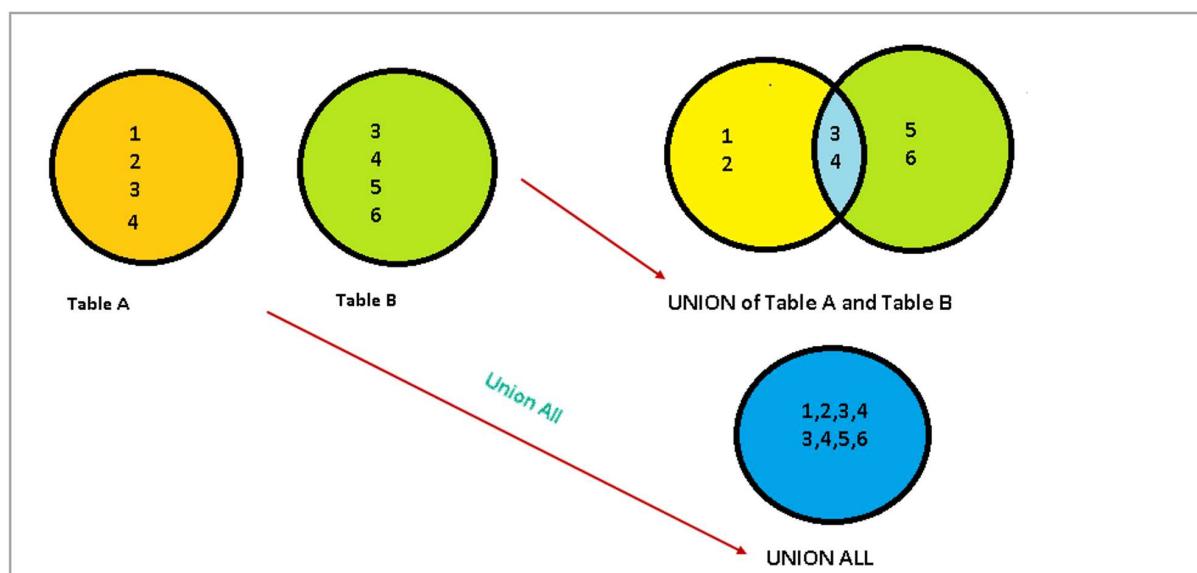
Многотабличные запросы

До этого мы обращались только к одной таблице, но настало время попробовать многотабличные запросы, результат в которых можно формировать из двух и более таблиц.

Многотабличные запросы условно можно поделить на три большие группы:

- объединение UNION,
- вложенные запросы
- JOIN-соединения.

Сильная сторона SQL — то, что в его основе лежит теория множества. В отличие от других языков программирования, мы оперируем не отдельными значениями, а их наборами. В теории множеств описывается, как можно складывать и вычитать такие наборы или получать их пересечения.



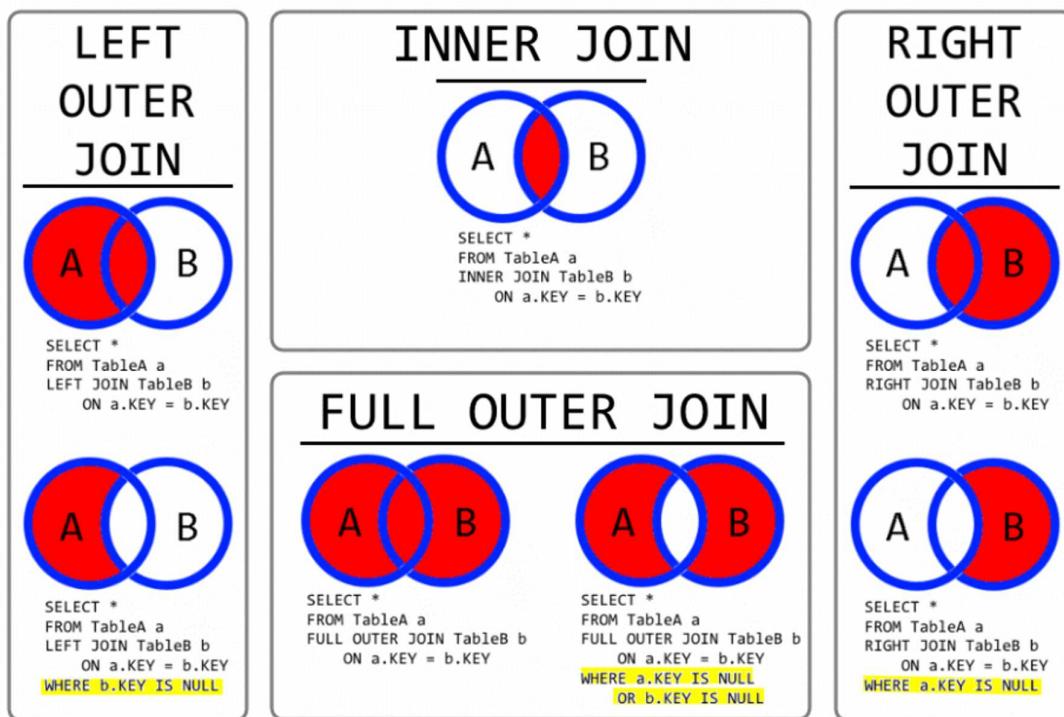
Вложенный запрос позволяет использовать результат, возвращаемый одним запросом, в другом. Здесь синим цветом представлены точки в запросе, где мы можем использовать вложенные запросы.

```
SELECT
    id,
    <SUBQUERY>
FROM
    <SUBQUERY>
WHERE
    <SUBQUERY>
GROUP BY
```

```
GROUP BY  
    id  
HAVING  
<SUBQUERY>
```

И, наконец, третий тип запросов — это JOIN-соединения. Они очень похожи на UNION-запросы, однако вместо объединения однотипных результатов, допускают соединения совершенно разноплановых таблиц, задействуя связь «первичный-внешний ключ».

ANSI SQL JOIN



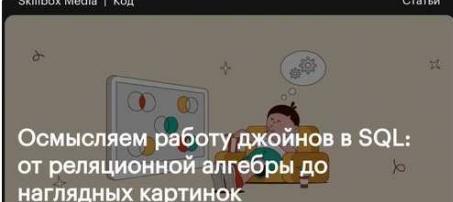
Очень неплохое графическое представление JOIN-ов. При первом знакомстве, еще на курсе из 3 семинаров от Ильнара, показываю этот пример:

Осмыслиаем работу джойнов в SQL: от реляционн...

Выбираем, какие фильмы посмотреть, с помощью соединения данных в SQL.

[S https://skillbox.ru/media/code/osmyslyam_rabotu_dzhoyn...](https://skillbox.ru/media/code/osmyslyam_rabotu_dzhoyn...)

Skillbox media | код



Осмыслиаем работу джойнов в SQL:
от реляционной алгебры до
наглядных картинок

1. Объединение UNION

Если формат результирующих таблиц совпадает, возможно объединение результатов выполнения двух операторов **SELECT** в одну результирующую таблицу. Для этого используется оператор **UNION**. Важное условие — совпадение всех параметров результирующих запросов. Количество, порядок следования и тип столбцов должны совпадать. Для демонстрации работы **UNION**-запроса создадим таблицу `rubrics`, структура которой полностью совпадает с таблицей `catalogs`:

```

DROP TABLE IF EXISTS catalogs; CREATE TABLE catalogs ( id INT
UNSIGNED, name VARCHAR(255) COMMENT 'Название раздела' ) COMMENT =
'Разделы интернет-магазина'; INSERT INTO catalogs (name) VALUES
('Процессоры'); INSERT INTO catalogs VALUES (0, 'Мат.платы');
INSERT INTO catalogs VALUES (NULL, 'Видеокарты'); DROP TABLE IF
EXISTS rubrics; CREATE TABLE rubrics ( id SERIAL PRIMARY KEY, name
VARCHAR(255) COMMENT 'Название раздела' ) COMMENT = 'Разделы
интернет-магазина'; INSERT INTO rubrics VALUES (NULL,
'Видеокарты'), (NULL, 'Память');

```

Давайте попробуем получить уникальные данные из 2 таблиц. Для этого используем **UNION**:

```
SELECT name FROM catalogs UNION SELECT name FROM rubrics;
```

catalogs

name
Процессоры
Мат.платы
Видеокарты

rubrics

name
Видеокарты
Память

```
SELECT
  name
FROM
  catalogs
```

UNION

```
SELECT
  name
FROM
  rubrics
```

```
ORDER BY
  name;
```

name
Видеокарты
Мат.платы
Память
Процессоры

Обратите внимание, что в результирующий запрос попадают только не повторяющиеся результаты. Несмотря на то, что раздел «Видеокарты» присутствует и в первой, и во второй таблицах, в результирующий запрос этот раздел попал в единственном экземпляре.

Если необходимо вывести все данные без повторения, используется оператор **UNION ALL**:

```
SELECT name FROM catalogs UNION ALL SELECT name FROM rubrics;
```

catalogs

name
Процессоры
Мат.платы

```
SELECT
  name
FROM
  catalogs
```

name

Мат.платы

Видеокарты

rubrics

name

Видеокарты

Память

UNION ALL

SELECT

name

FROM

rubrics

ORDER BY

name;

Видеокарты

Видеокарты

Мат.платы

Память

Процессоры

2. Вложенные запросы

Вложенный запрос позволяет использовать результат, возвращаемый одним запросом, в другом. Синтаксис основного запроса остается неизменным, однако в местах помеченным синим цветом, можно использовать подзапрос или, как еще говорят, вложенный запрос:

```
SELECT
    id,
    <SUBQUERY>
FROM
    <SUBQUERY>
WHERE
    <SUBQUERY>
GROUP BY
    id
HAVING
    <SUBQUERY>
```

Чтобы СУБД могла отличать основной запрос и подзапрос, последний заключают в круглые скобки. Попробуем вернуть максимальную цену из всех товаров, но без агрегатной функции:

```
SELECT *, (SELECT MAX(price) FROM products ) AS 'max_price' FROM
products;
```

Здесь мы видим максимальную цену у товара “Материнская плата ASUS ROG MAXIMUS X HERO, Z370, Socket 1151-V2, DDR4, ATX” стоимостью 19310.00.

Большинство подзапрос мы будем писать на семинарах

3. JOIN - соединения таблиц

1. **INNER JOIN**: возвращает записи с совпадающими значениями в обеих таблицах.

2. **LEFT JOIN**: возвращает все записи из левой таблицы и соответствующие записи из правой таблицы.
3. **RIGHT JOIN**: возвращает все записи из правой таблицы и соответствующие записи из левой таблицы.
4. **CROSS JOIN**: возвращает все записи из обеих таблиц.

Рассмотрим каждый из типов:

1. CROSS JOIN

Первый тип соединения - декартово произведение или CROSS JOIN.

В MySQL CROSS JOIN генерирует результирующий набор, который является произведением строк двух связанных таблиц. Каждая строка одной таблицы соединяется с каждой строкой второй таблицы, давая тем самым в результате все возможные сочетания строк двух таблиц.

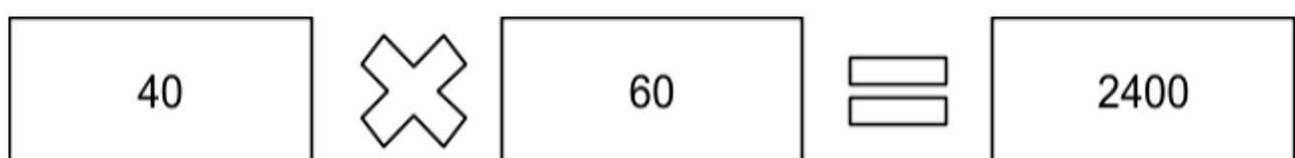
```
CREATE TABLE fst( value VARCHAR(255) ); INSERT INTO fst VALUES ('fst1'), ('fst2'), ('fst3'); CREATE TABLE snd( value VARCHAR(255) ); INSERT INTO snd VALUES ('snd1'), ('snd2'), ('snd3');
```

```
SELECT * FROM snd CROSS JOIN fst;
```

fst		SELECT		value	value
value		*		fst1	snd1
fst1		FROM		fst2	snd1
fst2		fst, snd;		fst3	snd1
fst3				fst1	snd2
				fst2	snd2
				fst3	snd2
				fst1	snd3
				fst2	snd3
				fst3	snd3

snd		SELECT		value	value
value		*		fst1	snd1
snd1		FROM		fst2	snd1
snd2		fst		fst3	snd1
snd3		JOIN		fst1	snd2
		snd;		fst2	snd2
				fst3	snd2
				fst1	snd3
				fst2	snd3
				fst3	snd3

Размер в таком случае можно определить по формуле: $M * N$, где M - количество строк в первой таблице, N - количество строк во второй таблице.
Если $M = 40$, $N = 60$, то CROSS JOIN вернет нам 2400 строк.



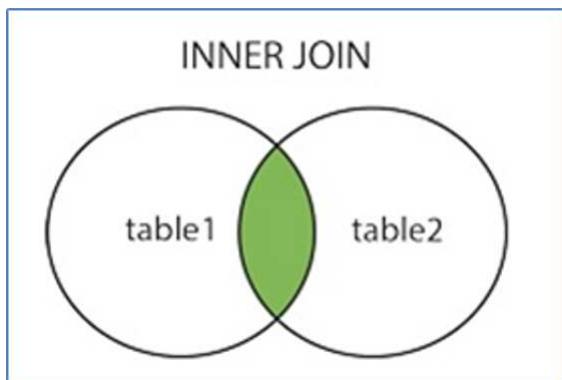
2. INNER JOIN

Оператору передаются две таблицы, и он возвращает их внутреннее пересечение по какому-либо критерию. Результатом будут записи, которые

пересечение по какому-либо критерию. Результатом будут записи, которые соответствуют обеим таблицам, — их перед отправкой объединят.

Пример(для понимания):

Например, если в одной таблице будут перечислены черные животные, а в другой — собаки, то **INNER JOIN** вернет одну таблицу с перечислением черных собак. Столбцы будут «склеены» друг с другом, несмотря на то что в базе данные хранятся в разных таблицах. Это похоже на бинарное «и» из алгебры логики.



```
SELECT столбцы FROM таблица1 [INNER] JOIN таблица2 ON условие1  
[[INNER] JOIN таблица3 ON условие2]
```

Пример(кодом):

Возвращает записи с совпадающими значениями в обеих таблицах.

Модифицируем наши таблицы:

```
DROP TABLE IF EXISTS catalog; CREATE TABLE catalog ( id INT PRIMARY  
KEY AUTO_INCREMENT, name VARCHAR(255) COMMENT 'Название раздела',  
UNIQUE unique_name(name(10)) ) COMMENT = 'Разделы интернет-  
магазина'; INSERT INTO catalog (name) VALUES ('Процессоры'), -- id  
= 1 ('Материнские платы'), -- id = 2 ('Видеокарты'), -- id = 3  
('Жесткие диски'), -- id = 4 ('Оперативная память'); -- id = 5 DROP  
TABLE IF EXISTS product; CREATE TABLE product ( id INT PRIMARY KEY  
AUTO_INCREMENT, name VARCHAR(255) COMMENT 'Название', description  
TEXT COMMENT 'Описание', price DECIMAL (11,2) COMMENT 'Цена',  
catalog_id INT, created_at DATETIME DEFAULT CURRENT_TIMESTAMP,  
updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE  
CURRENT_TIMESTAMP, FOREIGN KEY (catalog_id) REFERENCES catalog(id)  
) COMMENT = 'Товарные позиции'; SELECT * FROM catalog; INSERT  
product(name, description, price, catalog_id) VALUES ('Intel Core  
i3-8100', 'Процессор для настольных персональных компьютеров,  
основанных на платформе Intel.', 7890.00, 1), ('Intel Core  
i5-7400', 'Процессор для настольных персональных компьютеров,  
основанных на платформе Intel.', 12700.00, 1), ('AMD FX-8320E',  
'Процессор для настольных персональных компьютеров, основанных на  
платформе AMD.', 4780.00, 1), ('AMD FX-8320', 'Процессор для  
настольных персональных компьютеров, основанных на платформе AMD.',  
7120.00, 1), ('ASUS ROG MAXIMUS X HERO', 'Материнская плата ASUS  
ROG MAXIMUS X HERO, Z370, Socket 1151-V2, DDR4, ATX', 19310.00, 2),  
( 'Gigabyte H310M S2H', 'Материнская плата Gigabyte H310M S2H, H310,  
Socket 1151-V2, DDR4, mATX', 4790.00, 2), ('MSI B250M GAMING PRO',
```

'Материнская плата MSI B250M GAMING PRO, B250, Socket 1151, DDR4, mATX', 5060.00, 2);

Выведем товар и его группу. Для этого выберем столбцы, которые нас интересуют из каждой таблицы. Мы видим, что в табличке products имеется ссылка на табличку catalogs . Если эти ссылки равны, то мы узнаем товар и принадлежность к группе.

Допустим, имеется

('Intel Core i3-8100', 'Процессор для настольных персональных компьютеров, основанных на платформе Intel.', 7890.00, 1);

В конце указана ссылка catalog_id(внешний ключ) - группа №1 из таблицы catalog.

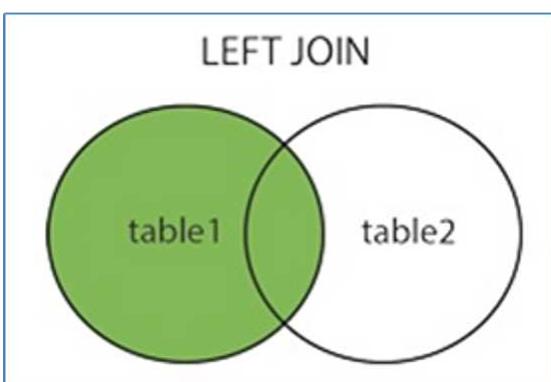
Это ссылка на первичный ключ из таблицы catalog (FOREIGN KEY (catalog_id) REFERENCES catalog(id)). Указав равенство ключей, мы можем сделать вывод, что 'Intel Core i3-8100' принадлежит к группе с номером №1: 'Процессоры'.

```
SELECT p.name, p.price, c.name FROM catalog AS c JOIN product AS p  
ON c.id = p.catalog_id;
```

	name	price	name
▶	Intel Core i3-8100	7890.00	Процессоры
	Intel Core i5-7400	12700.00	Процессоры
	AMD FX-8320E	4780.00	Процессоры
	AMD FX-8320	7120.00	Процессоры
	ASUS ROG MAXIMUS X HERO	19310.00	Материнские платы
	Gigabyte H310M S2H	4790.00	Материнские платы
	MSI B250M GAMING PRO	5060.00	Материнские платы

3. LEFT JOIN.

LEFT JOIN и **RIGHT JOIN** осуществляют левое и правое соединение, в результирующей таблице присутствуют все записи левой или правой таблицы, даже если им нет подходящего сопоставления.



Пример(для понимания):

Возвращает пересечение множеств и все элементы из левой таблицы.

Например, человек хочет посмотреть кино, но на русский фильм согласен, только если это боевик. Фильтр вернет ему все фильмы из множества

«боевики», фильмы из подмножества «русские боевики», но других фильмов из множества «русские» там не будет.

Пример(кодом):

Допустим, попробуем получить полную информацию о каталоге товаров. Даже если товар не продается, мы будем его выводить. Для этого нам нужен LEFT JOIN. В запросе первая таблица всегда является **левой**, вторая - **правой**. Для запроса выше левая таблица - catalog, правая - product.

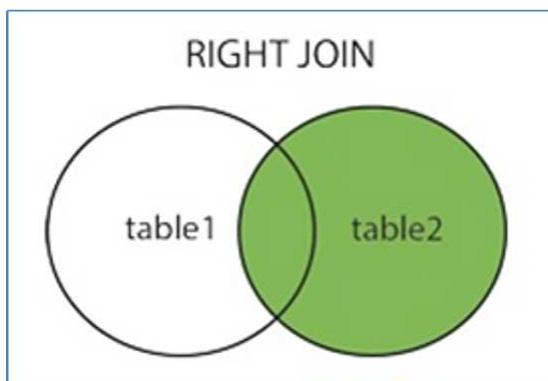
В таблице catalogs у нас три записи, для раздела «Видеокарты» сопоставления в таблице products нет, поэтому эта запись не попадает в результирующую таблицу JOIN-соединения. Обратите внимание, что в catalog есть группы видеокарт, жестких дисков и процессоров, а в таблице product таких товаров нет. Если совпадений не найдено, возвращается значение NULL.

```
SELECT p.name, p.price, c.name FROM catalog AS c -- Берется левая таблица и совпадения из правой LEFT JOIN product AS p -- Правая таблица ON c.id = p.catalog_id; -- Условие для совпадения (область пересечения 2 множеств)
```

```
43 •   SELECT
44       p.name,
45       p.price,
46       c.name
47   FROM catalog AS c -- Берется левая таблица и совпадения из правой
48   LEFT JOIN product AS p -- Правая таблица
49   ON c.id = p.catalog_id; -- Условие для совпадения (область пересечения 2 множеств)
```

	name	price	name
	Intel Core i3-8100	7890.00	Процессоры
	Intel Core i5-7400	12700.00	Процессоры
	AMD FX-8320E	4780.00	Процессоры
	AMD FX-8320	7120.00	Процессоры
	ASUS ROG MAXIMUS X HERO	19310.00	Материнские платы
	Gigabyte H310M S2H	4790.00	Материнские платы
	MSI B250M GAMING PRO	5060.00	Материнские платы
	NULL	NULL	Видеокарты
	NULL	NULL	Жесткие диски
	NULL	NULL	Оперативная память

4. RIGHT JOIN



Пример(для понимания):

Работает по тому же принципу, но вместо левой таблицы — правая. То есть человек получит в результатах боевики, только если они русские.

Пример(кодом):

Аналогично, можем посмотреть товары и группы, к которым они принадлежат.

```
SELECT * FROM catalogs; SELECT p.name, p.price, c.name FROM catalog
AS c -- Левая таблица RIGHT JOIN product AS p -- Правая таблица ON
c.id = p.catalog_id; -- Условие для совпадения (область пересечения
2 множеств)
```

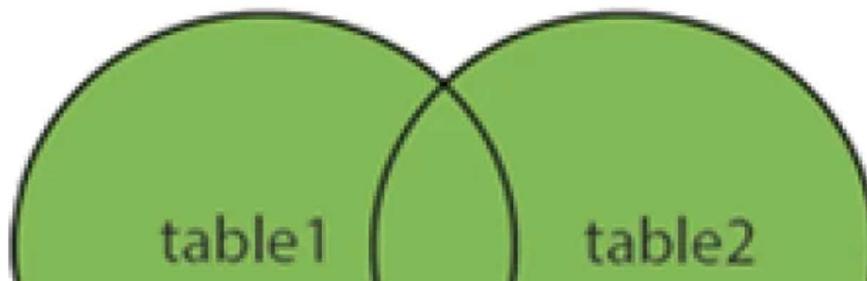
```
43 •   SELECT
44     p.name,
45     p.price,
46     c.name
47     FROM catalog AS c -- Левая таблица
48     RIGHT JOIN product AS p -- Правая таблица
49     ON c.id = p.catalog_id; -- Условие для совпадения (область пересечения 2 множеств)
```

Result Grid | Filter Rows: [] | Export: | Wrap Cell Content:

	name	price	name
▶	Intel Core i3-8100	7890.00	Процессоры
	Intel Core i5-7400	12700.00	Процессоры
	AMD FX-8320E	4780.00	Процессоры
	AMD FX-8320	7120.00	Процессоры
	ASUS ROG MAXIMUS X HERO	19310.00	Материнские платы
	Gigabyte H310M S2H	4790.00	Материнские платы
	MSI B250M GAMING PRO	5060.00	Материнские платы

Запрос очень напоминает INNER JOIN, но почему? Дело в том, что продукт имеет ссылку на каталог. Нет такого значения, которое бы не ссылалось на каталог (имеется внешний ключ, который ссылается ТОЛЬКО на реальные объекты в вашей БД).

Так же имеет смысл рассмотреть **FULL JOIN** в MySQL. **FULL JOIN** возвращает обе таблицы, объединенные в одну. Данный тип соединения не поддерживается, так как операция считается избыточной. Мы можем реализовать **FULL JOIN**, объединив два запроса при помощи **UNION**, оставив только уникальные значения и **UNION ALL**, который не удаляет дубликаты. В первом запросе будем использовать **LEFT JOIN**, а во втором - **RIGHT JOIN**.



Пример(для понимания):

Например, человек хочет увидеть список из всех боевиков и всех русских фильмов, без исключений.

Пример(кодом):

Попробуем объединить табличку **product** с таблицей **catalog**, **удалив дубликаты**.

```
SELECT p.name, p.price, c.name FROM catalog AS c LEFT JOIN product
AS p ON c.id = p.catalog_id UNION -- Удалили дубликаты
SELECT p.name, p.price, c.name FROM catalog AS c RIGHT JOIN product AS p
ON c.id = p.catalog_id;
```

Иногда имеет смысл не удалять дубликаты. В таком случае используется оператор **UNION ALL**.

Парочка вопросов из реальных собеседований

1. Представьте, что запрос с оператором **UNION** и **UNION ALL** вернул одинаковое количество строчек (пусть вернулось 10 строчек). Какой из операторов сработал быстрее и почему?

Ответ: сработает быстрее UNION ALL, оператор не делает проверку на уникальность

2. Чем отличается **WHERE** от **HAVING**?

Ответ: HAVING очень похож на WHERE - это фильтр. Мы можем написать в HAVING name = 'AMD FX-8320', как и в WHERE. **Ошибки не будет.**

В **HAVING** и только в нём можно писать условия по агрегатным функциям (SUM, COUNT, MAX, MIN, AVG). Если мы хотим сделать что-то вроде **SUM(salary) > 1000000**, то это возможно сделать только в **HAVING**. Но мы могли оставить только **HAVING**, зачем сложности с оператором **WHERE**?

Загвоздка в выполнении запроса, работа с данными и порядок. **WHERE** выполняется на раннем этапе, до **GROUP BY**, чтобы экономить время выполнения запроса и ресурсы сервера.

Далее мы применяем GROUP BY, который делит данные на группы. На группы можно накладывать условия на результаты агрегатных функций.

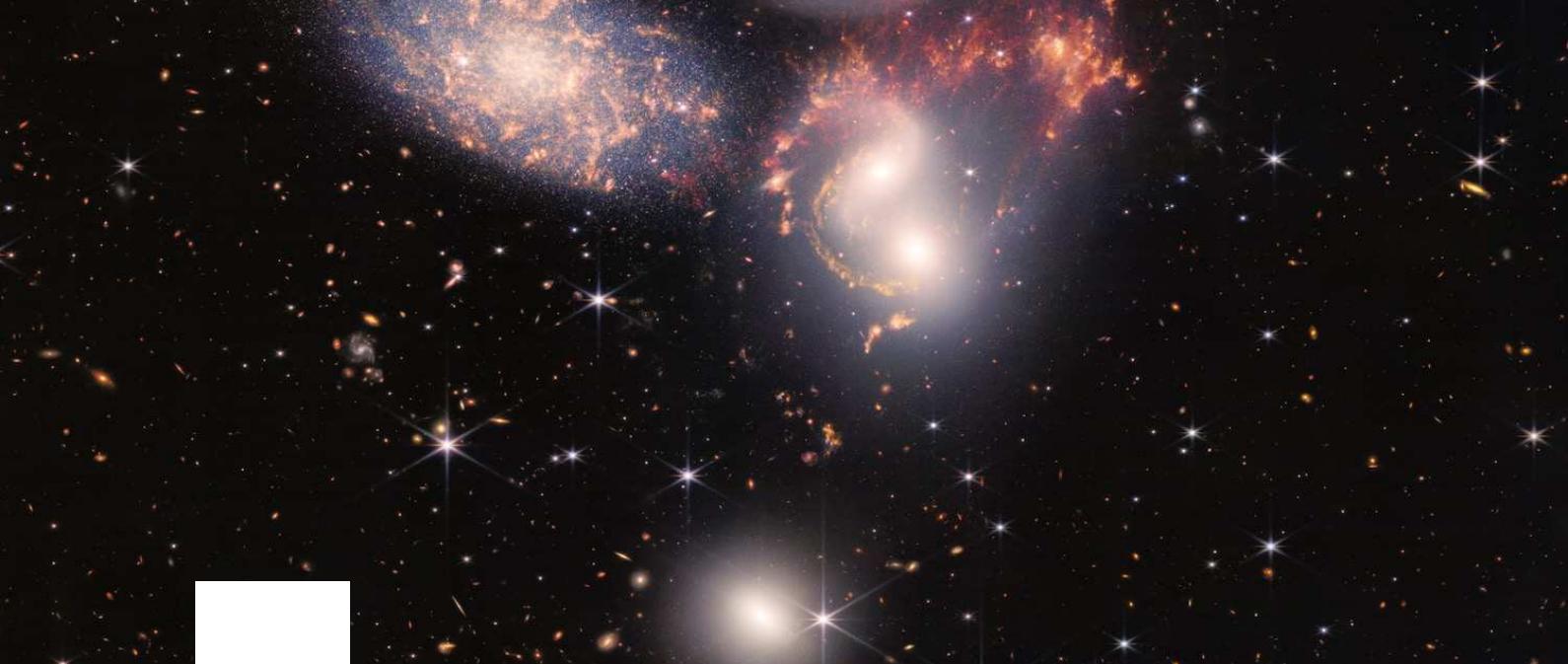
Главное отличие HAVING от WHERE в том, что в HAVING можно наложить условия на результаты группировки, потому что порядок исполнения запроса устроен таким образом, что на этапе, когда выполняется WHERE, ещё нет групп, а HAVING выполняется уже после формирования групп. Оператор HAVING работает так же с агрегатными функциями, а WHERE - нет.

Используемые источники

1. <https://dev.mysql.com/doc/refman/5.7/en/group-by-functions-and-modifiers.html>
2. Линн Бейли. Head First. Изучаем SQL. — СПб.: Питер, 2012. — 592 с.
3. 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.
4. Дейт К. Дж. SQL и реляционная теория. Как грамотно писать код на SQL. — Пер. с англ. — СПб.: Символ-Плюс, 2010. — 480 с.
5. Кузнецов М.В., Симдянов И.В. MySQL на примерах. — СПб.: БХВ-Петербург, 2007. — 592с.
6. Кузнецов М.В., Симдянов И.В. MySQL 5. — СПб.: БХВ-Петербург, 2006. — 1024с.
7. Дейт, К. Дж. Введение в системы баз данных, 8-е издание.: Пер. с англ. — М.: Издательский дом "Вильямс", 2005. — 1328 с.
8. Карвин Б. Программирование баз данных SQL. Типичные ошибки и их устранение. — Рид Групп, 2011. — 336 с.
9. <https://dev.mysql.com/doc/refman/5.7/en/union.html>
10. <https://dev.mysql.com/doc/refman/5.7/en/subqueries.html>
11. <https://dev.mysql.com/doc/refman/5.7/en/join.html>

Книги:

- “Изучаем SQL”, книга Бейли Л.
- Алан Бьюоли "Изучаем SQL" (2007)
- Энтони Молинаро "SQL. Сборник рецептов" (2009)



Курс базы данных и SQL.

Лекция 5

На пятой лекции будем рассматривать: оконные функции(агрегирующие, ранжирующие, функции смещения, аналитические функции) и узнаем про представления

Терминология

Оконная функция в SQL - функция, которая работает с выделенным набором строк (окном, партицией) и выполняет вычисление для этого набора строк в отдельном столбце.

Партиции (окна из набора строк) - это набор строк, указанный для оконной функции по одному из столбцов или группе столбцов таблицы. Партиции для каждой оконной функции в запросе могут быть разделены по различным колонкам таблицы.

Агрегатные функции (агрегации) — это функции, которые вычисляются от группы значений и объединяют их в одно результирующее.

Функции смещения – это функции, которые позволяют перемещаться и обращаться к разным строкам в окне, относительно текущей строки, а также обращаться к значениям в начале или в конце окна.

Аналитические функции — это функции которые возвращают информацию о распределении данных и используются для статистического анализа.

Предикат в SQL это:

в широком смысле, — любое выражение, результатом которого являются значения булевого типа — *TRUE*, *FALSE*, а так же *UNKNOWN*

в узком смысле, — некий уточняющий фильтр. Самым явным примером предиката служит оператор WHERE

Что мы узнаем:

- Оконные функции
- Ранжирующие оконные функции
- Агрегирующие оконные функции
- Функции смещения в оконных функциях
- Арифметические операции
- Логические операторы (and, or, between, not, in)
- Оператор CASE, IF

Доброго времени суток, уважаемые студенты!

На сегодняшней лекции особое внимание хотелось бы уделить мощному инструменту аналитика - оконным функциям. “Окошки” помогают делать различные аналитические отчеты без участия стороннего ПО (“экселя”). Давайте попробуем рассмотреть основную массу задач, которые могут решать оконные функции, затем - разберемся, как же воплотить наши ожидания с помощью кода:

- Ранжирование. Составление всевозможных ТОПов: топ-100 популярных треков на какой-то радиостанции
- Просмотр разницы между близкими элементами (2 и 3 человек в топ-5 самых богатых людей мира)
- “Скользящие” агрегаты
- Агрегация (топ самых высоких зарплат)

Классы оконных функций

Множество оконных функций можно поделить на 3 группы:

- Агрегирующие (Aggregate)
- Ранжирующие (Ranking)
- Функции смещения (Value)
- Аналитические функции.

Составление рейтинга по ЗП

```
-- Создание таблицы CREATE TABLE IF NOT EXISTS staff ( id INT  
PRIMARY KEY, first_name VARCHAR(30), post VARCHAR(30), discipline
```

```

    VARCHAR(30), salary INT ); -- Заполнение таблицы данными INSERT
staff (id, first_name, post, discipline, salary) VALUES
(100, 'Антон', 'Преподаватель', 'Программирование', 50),
(101, 'Василий', 'Преподаватель', 'Программирование', 60),
(103, 'Александр', 'Ассистент', 'Программирование', 25),
(104, 'Владимир', 'Профессор', 'Математика', 120), (105, 'Иван',
'Профессор', 'Математика', 120), (106, 'Михаил', 'Доцент', 'Физика',
70), (107, 'Анна', 'Доцент', 'Физика', 70), (108, 'Вероника',
'Доцент', 'ИКТ', 30), (109, 'Григорий', 'Преподаватель', 'ИКТ', 25),
(110, 'Георгий', 'Ассистент', 'Программирование', 30);

```

	id	first_name	post	discipline	salary
▶	100	Антон	Преподаватель	Программирование	50
	101	Василий	Преподаватель	Программирование	60
	103	Александр	Ассистент	Программирование	25
	104	Владимир	Профессор	Математика	120
	105	Иван	Профессор	Математика	120
	106	Михаил	Доцент	Физика	70
	107	Анна	Доцент	Физика	70
	108	Вероника	Доцент	ИКТ	30
	109	Григорий	Преподаватель	ИКТ	25
	110	Георгий	Ассистент	Программирование	30

Как мы видим, одинаковый ранг будет у сотрудников с одинаковой ЗП (допустим, Владимир и Иван в правой таблице, Иван и Михаил, Вероника и Георгий).

	rank	first_name	discipline	salary
▶	6	Владимир	Математика	120
	6	Иван	Математика	120
	5	Михаил	Физика	70
	5	Анна	Физика	70
	4	Василий	Программирование	60
	3	Антон	Программирование	50
	2	Вероника	ИКТ	30
	2	Георгий	Программирование	30
	1	Александр	Программирование	25
	1	Григорий	ИКТ	25

Давайте попробуем пройтись по всем строчкам. Начнем с 1 и будем проставлять ранг. Для простановки ранга необходимо учитывать только значение конкретного столбца - `salary`. В контексте оконной функции - окно. При обычном запросе все множество строк обрабатывается единым потоком, для которого считаются агрегаты. При работе с оконными функциями наш запрос делится на небольшие части (окна) и уже для каждой из отдельных частей считаются свои агрегаты.

Обычный запрос

Запрос с оконной функцией

Пример для абстрактной таблицы:

Исходная таблица				Пример оконных функций			Окна
ID	PRODUCT	TYPE	PRICE	func_sum	func_count	func_row_number	
1	Монитор LG	Монитор	2,0	2,0	1	1	окно 1 (partition)
2	Ноутбук HP	Ноутбук	4,0	8,0	2	1	окно 2 (partition)
3	Ноутбук SONY	Ноутбук	4,0	8,0	2	2	
4	Принтер Xerox	Принтер	3,0	9,0	3	1	окно 3 (partition)
5	Принтер Canon	Принтер	3,0	9,0	3	2	
6	Принтер Brother	Принтер	3,0	9,0	3	3	
7	Смартфон Huawei	Смартфон	2,0	8,0	4	1	окно 4 (partition)
8	Смартфон Samsung	Смартфон	2,0	8,0	4	2	
9	Смартфон Apple	Смартфон	2,0	8,0	4	3	
10	Смартфон Xiaomi	Смартфон	2,0	8,0	4	4	

Наше окно - содержимое столбца salary, которое мы отсортировали по убыванию.

Как же записать это в контексте SQL?

MySQL :: MySQL 8.0 Reference Manual :: 12.21.2 Window Function Concepts and Syntax

 <https://dev.mysql.com/doc/refman/8.0/en/window-functions-usage.html>

Для открытия окна используется обязательная инструкция OVER:

`SELECT Название_функции (столбец для вычислений) OVER (PARTITION
BY столбец для группировки ORDER BY столбец для сортировки ROWS или
RANGE выражение для ограничения строк в пределах группы)`

Нашему окну можно так же присвоить имя, которое можно будет использовать в запросе. Наше окно должно проставлять ранг по убыванию столбца salary :

`WINDOW w AS (ORDER BY salary desc)`

- `window` — определение окна;
- `w` — название окна, указываем через пробел, как и псевдоним без AS;
- `order by salary desc` — сортировка столбца salary.

Внутри конкретного окна нужно посчитать ранг (ранг по окну `w`).

	shopping_day	department	count
▶	2022-12-21	Бытовая техника	3
	2022-12-21	Свежая выпечка	4
	2022-12-21	Напитки	4
	2022-12-22	Бытовая техника	1
	2022-12-22	Свежая выпечка	3
	2022-12-22	Напитки	3
	2022-12-22	Бытовая техника	1

2022-12-22	Замороженные продукты	1
2022-12-23	Бытовая техника	2
2022-12-23	Свежая выпечка	4
2022-12-23	Замороженные продукты	4

Начнем с примера: откроем окно **over** и найдем сумму и количество покупок:

```
SELECT shopping_day, department, count, SUM(count) OVER() AS 'Sum',
COUNT(count) OVER() AS 'Count' FROM shop;
```

	shopping_day	department	count	Sum	Count
▶	2022-12-21	Бытовая техника	3	29	10
	2022-12-21	Свежая выпечка	4	29	10
	2022-12-21	Напитки	4	29	10
	2022-12-22	Бытовая техника	1	29	10
	2022-12-22	Свежая выпечка	3	29	10
	2022-12-22	Напитки	3	29	10
	2022-12-22	Замороженные продукты	1	29	10
	2022-12-23	Бытовая техника	2	29	10
	2022-12-23	Свежая выпечка	4	29	10
	2022-12-23	Замороженные продукты	4	29	10

В нашем примере сумма, равная 29 - это общее количество покупок(суммирование всего столбца **count** и подсчет общего количества строк). Если бы мы использовали агрегатную функцию в чистом виде, то получим сужение выборки до 1 строки.

В оконной функции исходное количество строк не уменьшается по сравнению с исходной таблицей. При использовании агрегирующих функций предложение **GROUP BY** сокращает количество строк в запросе с помощью их группировки.

```
SELECT SUM(count) AS 'Sum', COUNT(count) AS 'Count' FROM shop;
```



Внутри окна так же возможно применить сортировку и группировку.

- **Группировка**

Для группировки используется **PARTITION BY**, которая определяет столбец для группировки в окне:

	shopping_day	department	count
▶	2022-12-21	Бытовая техника	3
	2022-12-21	Свежая выпечка	4

2022-12-21	Напитки	4
2022-12-22	Бытовая техника	1
2022-12-22	Свежая выпечка	3
2022-12-22	Напитки	3
2022-12-22	Замороженные продукты	1
2022-12-23	Бытовая техника	2
2022-12-23	Свежая выпечка	4
2022-12-23	Замороженные продукты	4

```
SELECT shopping_day, department, count, SUM(count) OVER(PARTITION
BY shopping_day) AS 'Sum' FROM shop;
```

	shopping_day	department	count	Sum
▶	2022-12-21	Бытовая техника	3	11
	2022-12-21	Свежая выпечка	4	11
	2022-12-21	Напитки	4	11
	2022-12-22	Бытовая техника	1	8
	2022-12-22	Свежая выпечка	3	8
	2022-12-22	Напитки	3	8
	2022-12-22	Замороженные продукты	1	8
	2022-12-23	Бытовая техника	2	10
	2022-12-23	Свежая выпечка	4	10
	2022-12-23	Замороженные продукты	4	10

Инструкция `PARTITION BY` сгруппировала строки по полю «`shopping_day`» и каждой группы рассчитывается сумма значений по столбцу «`count`».

- Сортировка

	shopping_day	department	count
▶	2022-12-21	Бытовая техника	3
	2022-12-21	Свежая выпечка	4
	2022-12-21	Напитки	4
	2022-12-22	Бытовая техника	1
	2022-12-22	Свежая выпечка	3
	2022-12-22	Напитки	3
	2022-12-22	Замороженные продукты	1
	2022-12-23	Бытовая техника	2
	2022-12-23	Свежая выпечка	4
	2022-12-23	Замороженные продукты	4

Добавим к `PARTITION BY` сортировку по столбцу «`department`» (`ORDER BY department`). В данной ситуации мы будем считать нарастающий итог: для каждого значения «`count`» мы ищем сумму всех значений с предыдущими.

```
SELECT shopping_day, department, count, SUM(count) OVER(PARTITION
BY shopping_day ORDER BY department) AS 'Sum' FROM shop;
```

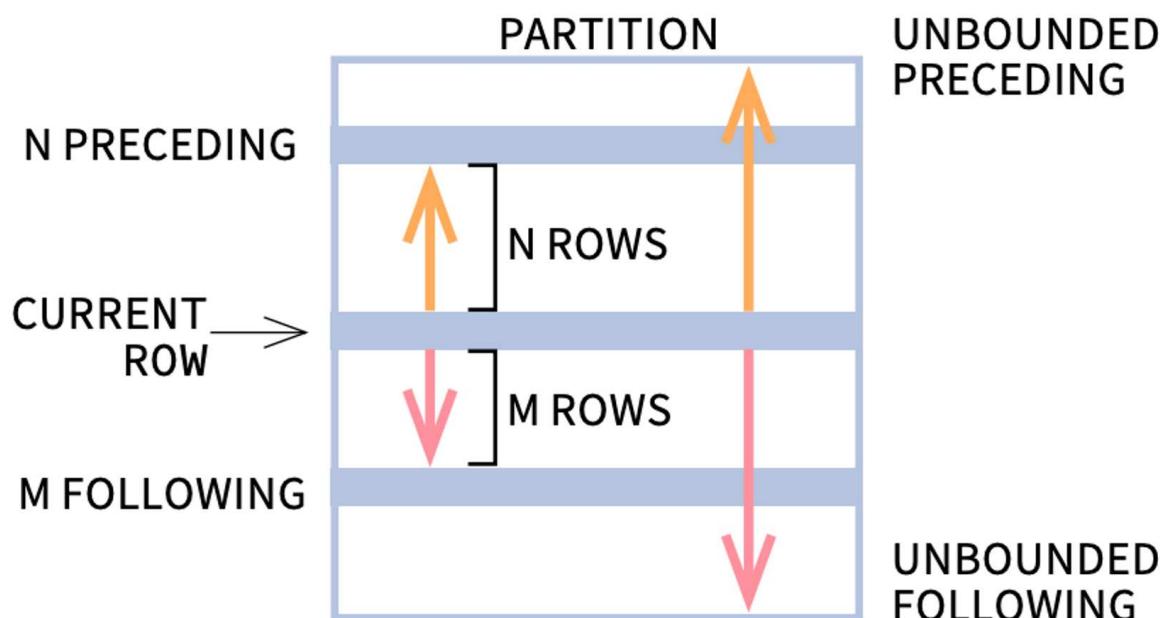
	shopping_day	department	count	Sum
▶	2022-12-21	Бытовая техника	3	3
	2022-12-21	Напитки	4	7
	2022-12-21	Свежая выпечка	4	11
	2022-12-22	Бытовая техника	1	1
	2022-12-22	Замороженные продукты	1	2
	2022-12-22	Напитки	3	5
	2022-12-22	Свежая выпечка	3	8
	2022-12-23	Бытовая техника	2	2
	2022-12-23	Замороженные продукты	4	6
	2022-12-23	Свежая выпечка	4	10

И вишенка на тортике - ограничение строк в окне :)

Инструкция `rows` ограничивает строки в окне, указывается определенное количество строк, предшествующих или следующих за текущей.

Инструкция `range`, в отличие от `rows`, работает не со строками, а с диапазоном строк в инструкции ORDER BY.

Обе инструкции `rows` и `range` всегда используются вместе с ORDER BY.



MySQL :: MySQL 8.0 Reference Manual :: 12.21.3 Window Function Frame Specification

 <https://dev.mysql.com/doc/refman/8.0/en/window-functions-frames.html>

- **UNBOUNDED PRECEDING** – указывает, что окно начинается с первой строки группы;
- **n PRECEDING** – n строк перед текущей строкой.
- **CURRENT ROW** – окно начинается или заканчивается на текущей строке;
- **n FOLLOWING** – определяет число строк после текущей строки (не допускается в предложении RANGE). n строк после текущей строки.
- **UNBOUNDED FOLLOWING** – С помощью данной инструкции можно указать, что окно заканчивается на последней строке группы

Сумма рассчитывается по текущей и следующей ячейке в окне. А последняя строка в окне имеет то же значение, что и столбец «count»: больше не с чем складывать.

```
SELECT shopping_day, department, count, SUM(count) OVER(PARTITION
BY shopping_day ORDER BY count ROWS BETWEEN CURRENT ROW AND 1
FOLLOWING) AS 'Sum' FROM shop;
```

	shopping_day	department	count	Sum
▶	2022-12-21	Бытовая техника	3	=3+4
	2022-12-21	Свежая выпечка	4	=4+4
	2022-12-21	Напитки	4	=4

2022-12-22	Бытовая техника	1	2
2022-12-22	Замороженные продукты	1	4
2022-12-22	Свежая выпечка	3	6
2022-12-22	Напитки	3	3
2022-12-23	Бытовая техника	2	6
2022-12-23	Свежая выпечка	4	8
2022-12-23	Замороженные продукты	4	4

Ранжирующие функции

Эти функции позволяют использовать для указания ранга или порядкового номера, который у нас на рукаве

- `row_number` – задаем номер строки, используется для нумерации;
- `rank` — функция возвращает ранг каждой строки, в случае нахождения одинаковых, возвращается одинаковый ранг с пропуском следующего значения;
- `dense_rank` — функция возвращает так же ранг каждой строки. Но в отличие от функции `rank`, она для одинаковых значений возвращает ранг, не пропуская следующий;
- `ntile` – это функция, которая позволяет поделить вашу выборку на группы, количество групп задается в скобках.

Пример:

	shopping_day	department	count
▶	2022-12-21	Бытовая техника	3
	2022-12-21	Свежая выпечка	4
	2022-12-21	Напитки	4
	2022-12-22	Бытовая техника	1
	2022-12-22	Свежая выпечка	3
	2022-12-22	Напитки	3
	2022-12-22	Замороженные продукты	1
	2022-12-23	Бытовая техника	2
	2022-12-23	Свежая выпечка	4
	2022-12-23	Замороженные продукты	4

```
SELECT shopping_day, department, count, ROW_NUMBER() OVER(PARTITION
BY shopping_day ORDER BY count) AS 'Row_number', RANK(),
OVER(PARTITION BY shopping_day ORDER BY count) AS 'Rank' ,
DENSE_RANK() OVER(PARTITION BY shopping_day ORDER BY count) AS
'Dense_Rank' , NTILE(3) OVER(PARTITION BY shopping_day ORDER BY
count) AS 'Ntile' FROM shop;
```

	shopping_day	department	count	Row_number	Rank	Dense_Rank	Ntile
▶	2022-12-21	Бытовая техника	3	1	1	1	1
	2022-12-21	Свежая выпечка	4	2	2	2	2
	2022-12-21	Напитки	4	3	2	2	3
	2022-12-22	Бытовая техника	1	1	1	1	1
	2022-12-22	Замороженные продукты	1	2	1	1	1
	2022-12-22	Свежая выпечка	3	3	3	2	2
	2022-12-22	Напитки	3	4	3	2	3
	2022-12-23	Бытовая техника	2	1	1	1	1
	2022-12-23	Свежая выпечка	4	2	2	2	2
	2022-12-23	Замороженные продукты	4	3	2	2	3

Для подсчета ранга в нашем случае используем функцию `dense_rank()`. В нашем случае пропуск ранга не нужен: начинаем с 1 и увеличиваем ранг при отличии от предыдущего. Слово `rank` выделяем в экранированные кавычки: название столбца совпадает с функцией ранжирования, `rank` будет считаться как имя столбца:

```
SELECT DENSE_RANK() OVER W AS `rank`, first_name, discipline,
salary FROM staff WINDOW w AS (ORDER BY salary DESC) ORDER BY
`rank`, id;
```

	rank	first_name	discipline	salary
▶	6	Владимир	Математика	120
	6	Иван	Математика	120
	5	Михаил	Физика	70
	5	Анна	Физика	70
	4	Василий	Программирование	60
	3	Антон	Программирование	50
	2	Вероника	ИКТ	30
	2	Георгий	Программирование	30
	1	Александр	Программирование	25
	1	Григорий	ИКТ	25

Обратите внимание, что `order by` в окне задает сортировку окна, а `order by` в основном запросе — сортировку результатов всего запроса после отработки окна. Пусть целью моего запроса будет проставить ранг по возрастанию зарплаты, а сортировка - по убыванию:

```
SELECT DENSE_RANK() OVER W AS `rank`, first_name, discipline,
salary FROM staff WINDOW w AS (ORDER BY salary ASC) ORDER BY salary
DESC;
```

	rank	first_name	discipline	salary
▶	6	Владимир	Математика	120
	6	Иван	Математика	120
	5	Михаил	Физика	70
	5	Анна	Физика	70
	4	Василий	Программирование	60
	3	Антон	Программирование	50
	2	Вероника	ИКТ	30
	2	Георгий	Программирование	30
	1	Александр	Программирование	25
	1	Григорий	ИКТ	25

Агрегирующие оконные функции

Следующий вид оконных функций - агрегирующие. Они используются для учета средней зарплаты, подсчета количества сотрудников, максимальную и минимальную ЗП и сумму по чеку. Эти функции выполняют манипуляции с набором данных и возвращают итоговое значение.

`sum()`

сумма значений в столбце;

count()	количество значений в столбце (NULL не учитываются);
avg()	среднее значение в столбце
max()	определяет максимальное значение в столбце;
min()	определяет минимальное значение в столбце.

	shopping_day	department	count
▶	2022-12-21	Бытовая техника	3
	2022-12-21	Свежая выпечка	4
	2022-12-21	Напитки	4
	2022-12-22	Бытовая техника	1
	2022-12-22	Свежая выпечка	3
	2022-12-22	Напитки	3
	2022-12-22	Замороженные продукты	1
	2022-12-23	Бытовая техника	2
	2022-12-23	Свежая выпечка	4
	2022-12-23	Замороженные продукты	4

```
SELECT shopping_day, department, count, SUM(count) OVER(PARTITION
BY shopping_day) AS 'Sum' , COUNT(count) OVER(PARTITION BY
shopping_day) AS 'Count' , AVG(count) OVER(PARTITION BY
shopping_day) AS 'Avg' , MAX(count) OVER(PARTITION BY shopping_day)
AS 'Max' , MIN(count) OVER(PARTITION BY shopping_day) AS 'Min' FROM
shop;
```

	shopping_day	department	count	Sum	Count	Avg	Max	Min
▶	2022-12-21	Бытовая техника	3	11	3	3.6667	4	3
	2022-12-21	Свежая выпечка	4	11	3	3.6667	4	3
	2022-12-21	Напитки	4	11	3	3.6667	4	3
	2022-12-22	Бытовая техника	1	8	4	2.0000	3	1
	2022-12-22	Свежая выпечка	3	8	4	2.0000	3	1
	2022-12-22	Напитки	3	8	4	2.0000	3	1
	2022-12-22	Замороженные продукты	1	8	4	2.0000	3	1
	2022-12-23	Бытовая техника	2	10	3	3.3333	4	2
	2022-12-23	Свежая выпечка	4	10	3	3.3333	4	2
	2022-12-23	Замороженные продукты	4	10	3	3.3333	4	2

Давайте попробуем применить знания на практике. Каждая группа преподавателей по конкретному предмету получают зарплату. Давайте узнаем, какой процент от суммарной ЗП по отделу получает каждый педагог.

	id	first_name	post	discipline	salary
▶	100	Антон	Преподаватель	Программирование	50
	101	Василий	Преподаватель	Программирование	60
	103	Александр	Ассистент	Программирование	25
	104	Владимир	Профессор	Математика	120
	105	Иван	Профессор	Математика	120
	106	Михаил	Доцент	Физика	70
	107	Анна	Доцент	Физика	70
	108	Вероника	Доцент	ИКТ	30
	109	Григорий	Преподаватель	ИКТ	25
	110	Георгий	Ассистент	Программирование	30

```

SELECT first_name, discipline, salary, SUM(salary) OVER w AS
payment_fund, ROUND(salary * 100.0 / SUM(salary) OVER w) AS
percentage FROM staff WINDOW w AS (PARTITION BY discipline) ORDER
BY discipline, salary, id;

```

Пройдемся по всей таблице. Считаем:

- `payment_fund` — показывает суммарную ЗП отдела (она одинакова для всех сотрудников департамента);
- `percentage` — процент зарплаты от этой суммы.

Окно состоит из секций по дисциплинам:

`WINDOW w AS (PARTITION BY discipline)`

Чтобы найти суммарную зарплату, мы используем функцию агрегатную функцию `SUM(salary)` по столбцу `salary`. Для расчета процента ЗП будем использовать формулу: `salary/sum(salary)`

	first_name	discipline	salary	payment_fund	percentage
►	Григорий	ИКТ	25	55	45
	Вероника	ИКТ	30	55	55
	Владимир	Математика	120	240	50
	Иван	Математика	120	240	50
	Александр	Программирование	25	165	15
	Георгий	Программирование	30	165	18
	Антон	Программирование	50	165	30
	Василий	Программирование	60	165	36
	Михаил	Физика	70	140	50
	Анна	Физика	70	140	50

Функции смещения

Это функции, которые позволяют перемещаться и обращаться к разным строкам в окне, относительно текущей строки, а также обращаться к значениям в начале или в конце окна.

<code>lag(value)</code>	функция <code>lag()</code> обращается к данным из предыдущей строки окна	
<code>lead(value)</code>	функция <code>lead()</code> обращается к данным из следующей строки	
<code>first_value()</code>	с помощью функции <code>first_value()</code> можно получить первое значение в окне, параметр - столбец, который нужно вернуть	

`last_value()`

с помощью функции

`last_value()` можно получить последнее значение в окне, параметр - столбец, который нужно вернуть

```
SELECT shopping_day, department, count, LAG(count) OVER(PARTITION
BY shopping_day ORDER BY shopping_day) AS 'Lag' , LEAD(count)
OVER(PARTITION BY shopping_day ORDER BY shopping_day) AS 'Lead' ,
FIRST_VALUE(count) OVER(PARTITION BY shopping_day ORDER BY
shopping_day) AS 'First_Value' , LAST_VALUE(count) OVER(PARTITION
BY shopping_day ORDER BY shopping_day) AS 'Last_Value' FROM shop;
```

	shopping_day	department	count	Lag	Lead	First_Value	Last_Value
▶	2022-12-21	Бытовая техника	3	NULL	4	3	4
	2022-12-21	Свежая выпечка	4	3	4	3	4
	2022-12-21	Напитки	4	4	NULL	3	4
	2022-12-22	Бытовая техника	1	NULL	3	1	1
	2022-12-22	Свежая выпечка	3	1	3	1	1
	2022-12-22	Напитки	3	3	1	1	1
	2022-12-22	Замороженные продукты	1	3	NULL	1	1
	2022-12-23	Бытовая техника	2	NULL	4	2	4
	2022-12-23	Свежая выпечка	4	2	4	2	4
	2022-12-23	Замороженные продукты	4	4	NULL	2	4

Аналитические функции

Аналитические функции — это функции которые возвращают информацию о распределении данных и используются для статистического анализа.

Показывать математику без уточнения деталей будет не очень, уместно:

MySQL :: MySQL 8.0 Reference Manual :: 12.21.1 Window Function Descriptions

 <https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html?ff=nopfpls>

Представления (VIEW)

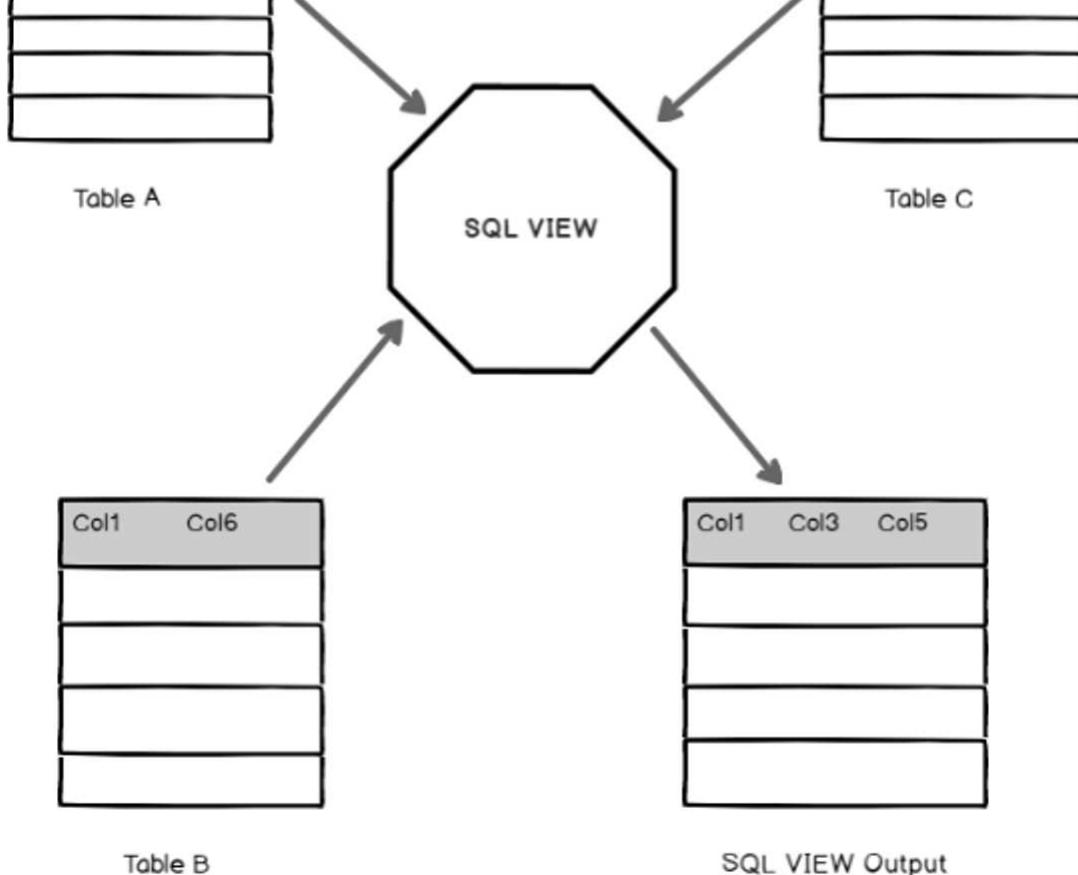
MySQL :: MySQL 8.0 Reference Manual :: 25.5 Using Views

 <https://dev.mysql.com/doc/refman/8.0/en/views.html>

При работе с БД требуется многократно запускать запросы, которые могут быть сложными и требовать обращения к нескольким таблицам. Чтобы не использовать многократно запросы, имеет смысл обратиться к так называемым представлениям (view). Представление (VIEW) — объект базы данных, являющийся результатом выполнения запроса к базе данных, определенного с помощью оператора SELECT, в момент обращения к представлению. Представления иногда называют «виртуальными таблицами».

Col1	Col2

Col1	Col3	Col4



Синтаксис представления:

```
CREATE VIEW name_view AS ( ... query ... );
```

В этой команде обязательно имя представления и сам запрос к БД. Попробуем реализовать простое представление. Попробуем реализовать запрос, который выводит дисциплины и количество преподавателей, которые эти предметы ведут:

```
SELECT discipline, count(first_name) FROM staff GROUP BY discipline
ORDER BY count(first_name) DESC;
```

Сделаем этот запрос представлением:

```
CREATE OR REPLACE VIEW count_teacher AS SELECT discipline,
count(first_name) AS res FROM staff GROUP BY discipline ORDER BY
count(first_name) DESC; -- "OR REPLACE" заменяет представление,
если оно существует
```

Теперь вместо указания какого - либо сложного запроса можно вызывать представление:

```
SELECT * FROM count_teacher;
```

	discipline	res
▶	Программирование	4
	Математика	2
	Физика	2
	ИКТ	2

Если данные изменены в базовой таблице, то пользователь получит актуальные данные при обращении к представлению, использующему данную таблицу; кэширования результатов выборки из таблицы при работе представлений не производится. При этом, механизм кэширования запросов (query cache) работает на уровне запросов пользователя относительно к тому, обращается ли пользователь к таблицам или представлениям. Представления могут основываться как на таблицах, так и на других представлениях, т.е. могут быть вложенными (до 32 уровней вложенности).

Операции с представлениями

- **DROP** : представление/виртуальную таблицу можно удалить с помощью команды **DROP VIEW**
- Объединение: мы также можем создать представление, объединив несколько таблиц. Это соединение будет извлекать совпадающие записи из обеих таблиц. (
- Создание рассмотрено выше (**CREATE VIEW**)
- Изменение существующего представления - **ALTER VIEW**

```
-- Исключим математиков
ALTER VIEW count_teacher AS SELECT
discipline, count(first_name) AS res FROM staff WHERE
discipline!="Математика" GROUP BY discipline ORDER BY
count(first_name) DESC; -- Показ SELECT * FROM count_teacher;
```

	discipline	res
▶	Программирование	4
	Физика	2
	ИКТ	2

Преимущества использования представлений:

1. Дает возможность гибкой настройки прав доступа к данным за счет того, что права даются не на таблицу, а на представление. Это очень удобно в случае если пользователю нужно дать права на отдельные строки таблицы или возможность получения не самих данных, а результата каких-то действий над ними.
2. Позволяет разделить логику хранения данных и программного обеспечения. Можно менять структуру данных, не затрагивая программный код, нужно лишь создать представления, аналогичные таблицам, к которым раньше обращались приложения. Это очень удобно когда нет возможности изменить программный код или к одной базе данных обращаются несколько приложений с различными требованиями к

сформирована несколько приложений с различными требованиями к структуре данных.

3. Удобство в использовании за счет автоматического выполнения таких действий как доступ к определенной части строк и/или столбцов, получение данных из нескольких таблиц и их преобразование с помощью различных функций.

Полезные ссылки и рекомендации:

- <https://habr.com/ru/company/oleg-bunin/blog/348172/> - DDL, DCL на **MS SQL Server (1 часть)**.
- <https://habr.com/post/255523/> - DDL, DCL на **MS SQL Server (2 часть)**.
- Руководство по стилю написания SQL: <https://www.sqlstyle.guide/ru/>
- <https://tproger.ru/translations/sql-window-functions/> - оконные функции
- Страйтесь не использовать русские буквы при работе с СУБД

Книги:

- “Изучаем SQL”, книга Бейли Л.
 - Алан Бьюли "Изучаем SQL" (2007)
 - Энтони Молинаро "SQL. Сборник рецептов" (2009)
1. Виктор Гольцман “MySQL 5.0. Библиотека программиста”
 2. “Изучаем SQL”, книга Бейли Л.
 3. <https://www.webmasterwiki.ru/MySQL>
 4. Поль Дюбуа “MySQL. Сборник рецептов”



Курс базы данных и SQL.

Лекция 6

Терминология

Транзакция — это набор последовательных операций с базой данных, соединенных в одну логическую единицу.

Изоляция — это свойство транзакции, которое позволяет скрывать изменения, внесенные одной операцией транзакции при возникновении явления race condition

Процедура - это подпрограмма (например, подпрограмма) на обычном языке сценариев, хранящаяся в базе данных.

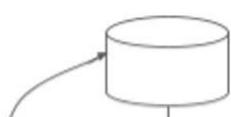
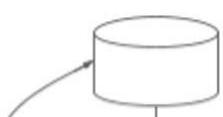
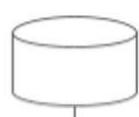
ACID - это набор из четырех требований к транзакционной системе, обеспечивающих максимально надежную и предсказуемую работу.

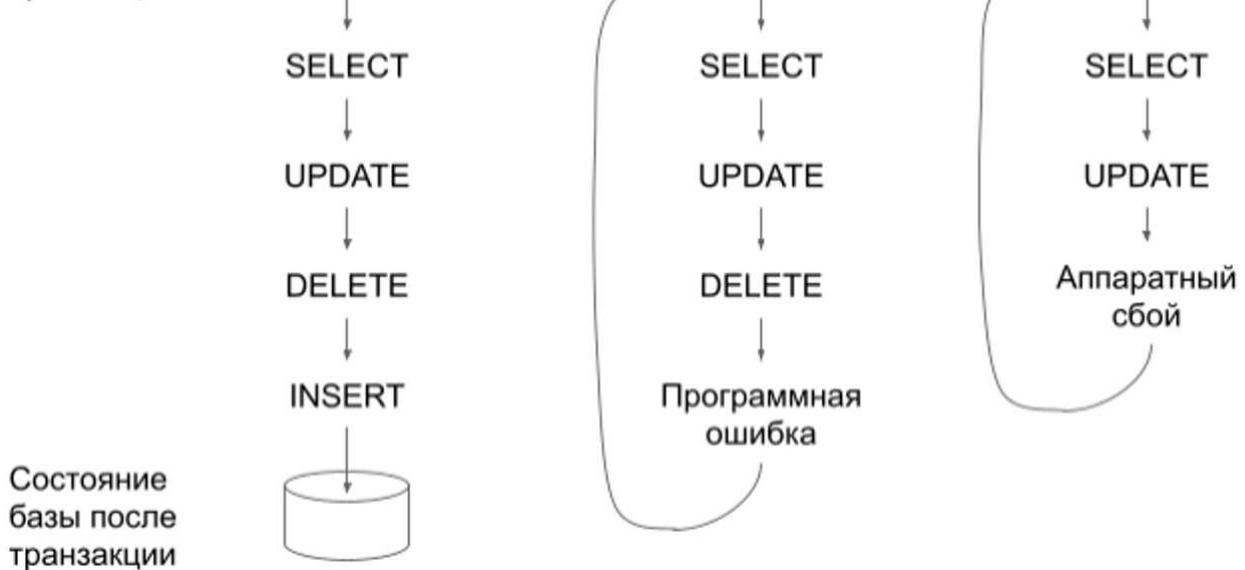
Доброго времени суток, уважаемые студенты!

Понятие транзакции, свойства ACID.

Транзакцией называется атомарная группа запросов SQL, т. е. запросы, которые рассматриваются как единое целое. Если база данных может выполнить всю группу запросов, она делает это, но если любой из них не может быть выполнен в результате сбоя или по какой-то другой причине, не будет выполнен ни один запрос группы. **Все или ничего.**

Состояние
базы до
транзакции





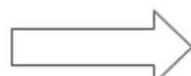
Операции с денежными средствами — классический пример, показывающий, почему необходимы транзакции. Если при оплате покупки происходит перевод от клиента электронному магазину, то счет клиента должен уменьшиться на эту сумму, а счет электронного магазина — увеличиться на нее же.

Пусть у нас есть таблица **account** со счетами пользователей. В этой же таблице есть счет

интернет-магазина. Он отличается тем, что внешний ключ `user_id` у него принимает значение `NULL`. Для осуществления покупки нам необходимо переместить 2000 рублей со счета клиента на счет магазина.

1. Убедиться, что остаток на счете клиента больше 2000 рублей.
2. Вычесть 2000 рублей со счета клиента.
3. Добавить 2000 к счету интернет-магазина.

<code>id</code>	<code>user_id</code>	<code>total</code>
1	4	5000
2	3	0
3	2	200
4	NULL	25000



<code>id</code>	<code>user_id</code>	<code>total</code>
1	4	3000
2	9	0
3	2	200
4	4	27000

Вся операция должна быть организована как транзакция, чтобы в случае неудачи на любом из этих трех этапов все выполненные ранее шаги были отменены.

Давайте смоделируем ситуацию. Для начала создадим таблицу **accounts**:

```
DROP TABLE IF EXISTS accounts;
CREATE TABLE accounts (
    id INT PRIMARY KEY AUTO_INCREMENT,
    user_id INT,
    total DECIMAL (11, 2)
    COMMENT 'Счет',
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
    ) COMMENT = 'Счета пользователей и интернет магазина';
INSERT INTO accounts (user_id, total) VALUES (4, 5000.00), (3, 0.00), (2, 200.00), (NULL, 25000.00);
```

Начинаем транзакцию командой START TRANSACTION:

```
START TRANSACTION; -- Далее выполняем команды, входящие в  
транзакцию: SELECT total FROM accounts WHERE user_id = 4; --  
Убеждаемся, что на счету пользователя достаточно средств: UPDATE  
accounts SET total = total - 2000 WHERE user_id = 4; -- Снимаем  
средства со счета пользователя: UPDATE accounts SET total = total +  
2000 WHERE user_id IS NULL; -- Чтобы изменения вступили в -- силу,  
мы должны выполнить команду COMMIT COMMIT; -- скрипт выполнять  
полностью: начиная от первой и до самой последней строчки
```

Если команда проходит без ошибок, изменения фиксируются базой данных и другие пользователи тоже начинают их видеть:

```
SELECT * FROM accounts;
```

Если мы выясняем, что не можем завершить транзакцию, например, пользователь ее отменяет или происходит еще что-то. Чтобы ее отметить мы можем воспользоваться командой ROLLBACK:

```
START TRANSACTION; SELECT total FROM accounts WHERE user_id = 4;  
UPDATE accounts SET total = total - 2000 WHERE user_id = 4; UPDATE  
accounts SET total = total + 2000 WHERE user_id IS NULL; ROLLBACK;  
-- Откат до исходного состояния
```

Для некоторых операторов нельзя выполнить откат при помощи оператора ROLLBACK. К их числу

относят следующие команды:

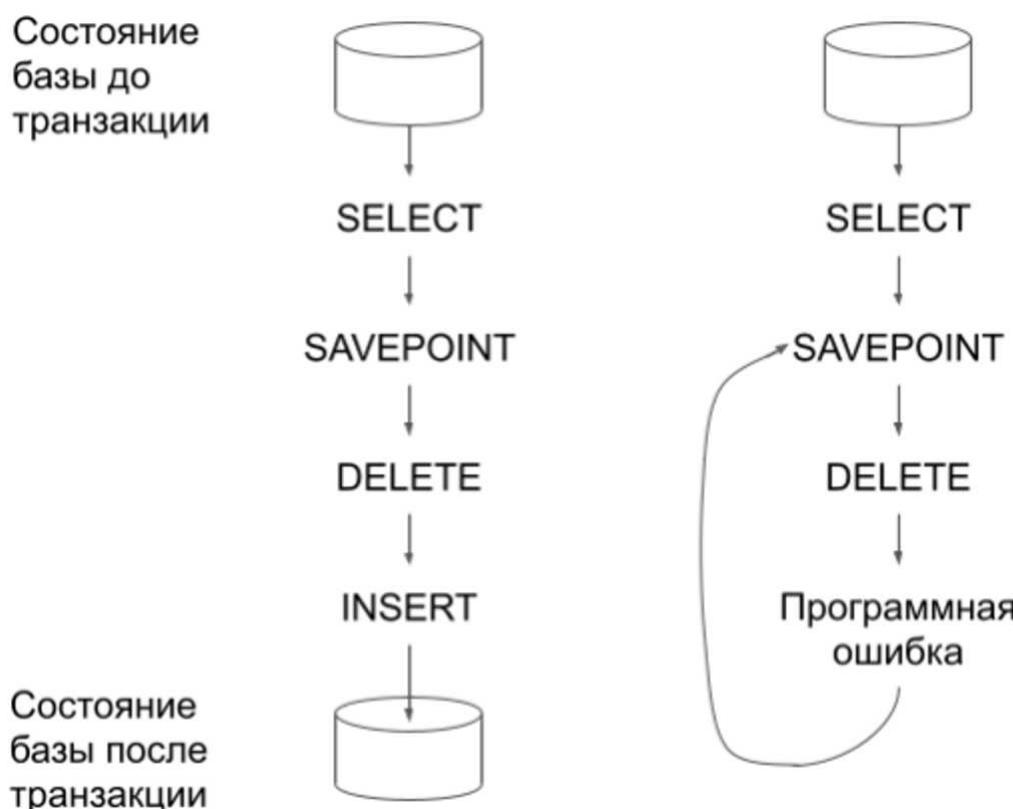
- CREATE INDEX
- DROP INDEX
- CREATE TABLE
- DROP TABLE
- TRUNCATE TABLE
- ALTER TABLE
- RENAME TABLE
- CREATE DATABASE
- DROP DATABASE
- ALTER DATABASE

Не помещайте их в транзакции с другими операторами. Кроме того, существует ряд операторов, которые неявно завершают транзакцию, как если бы был вызван оператор COMMIT:

- ALTER TABLE
- BEGIN

- CREATE INDEX
- CREATE TABLE
- CREATE DATABASE
- DROP DATABASE
- DROP INDEX
- DROP TABLE
- DROP DATABASE
- LOAD MASTER DATA
- LOCK TABLES
- RENAME
- SET AUTOCOMMIT=1
- START TRANSACTION
- TRUNCATE TABLE

В случае сбоя в транзакции откат можно делать до некой точки сохранения - SAVEPOINT.



Точка сохранения представляет собой место в последовательности событий транзакции, которое может выступать в качестве промежуточной точки восстановления. Откат текущей транзакции может быть выполнен не к началу транзакции, а к точке сохранения.

Для работы с точками сохранения предназначены два оператора:

- SAVEPOINT
- ROLLBACK TO SAVEPOINT

```

START TRANSACTION; SELECT total FROM accounts WHERE user_id = 4;
SAVEPOINT accounts_4; UPDATE accounts SET total = total - 2000
WHERE user_id = 4; -- Допустим мы хотим отменить транзакцию и
                   вернуться в точку сохранения. В этом случае мы можем --
                   воспользоваться оператором ROLLBACK TO SAVEPOINT: ROLLBACK TO
  
```

использоваться оператором ROLLBACK TO SAVEPOINT. ROLLBACK TO
SAVEPOINT accounts_4; SELECT * FROM accounts;

Допускается создание нескольких точек сохранения. Если текущая транзакция имеет точку сохранения с таким же именем, старая точка удаляется и устанавливается новая. Все точки сохранения транзакций удаляются, если выполняется оператор COMMIT или ROLLBACK без указания имени точки сохранения.

Транзакций недостаточно, если система не удовлетворяет принципу **ACID**.

Аббревиатура ACID

расшифровывается как атомарность, согласованность, изолированность и сохраняемость).

- **Atomicity** — атомарность.
- **Consistency** — согласованность.
- **Isolation** — изолированность.
- **Durability** — сохраняемость.

Атомарность подразумевает, что транзакция должна функционировать как единная неделимая

единица. Вся транзакция была либо выполняется, либо отменяется. Когда транзакции атомарны, не существует такого понятия, как частично выполненная транзакция.

При выполнении принципа **согласованности** база данных должна всегда переходить из одного непротиворечивого состояния в другое непротиворечивое состояние. В нашем примере согласованность гарантирует, что сбой между двумя UPDATE-командами не приведет к исчезновению 2000 рублей со счета пользователя. Транзакция просто не будет зафиксирована, и ни одно из изменений в этой транзакции не будет отражено в базе данных.

Изолированность подразумевает, что результаты транзакции обычно невидимы другим транзакциям, пока она не закончена. Это гарантирует, что, если в нашем примере во время транзакции будет выполнен запрос на извлечение средств пользователя, такой запрос по-прежнему будет видеть 2000 рублей на его счету.

Сохраняемость гарантирует, что изменения, внесенные в ходе транзакции, будучи

захваченными, становятся постоянными. Это означает, что изменения должны быть записаны так, чтобы данные не могли быть потеряны в случае сбоя системы. Транзакции ACID гарантируют, что интернет-магазин не потеряет ваши деньги. Это очень сложно или даже невозможно сделать с помощью логики приложения.

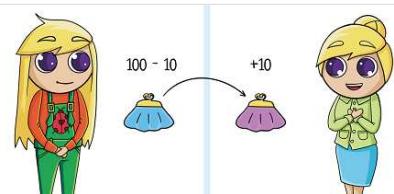
Эффекты с картинками:

Требования ACID на простом языке

Мне нравятся книги из серии Head First O'Reilly — они рассказывают просто о сложном. И я стараюсь делать



<https://habr.com/ru/post/555920/>



Уровни изоляции

Транзакции требуют довольно много ресурсов и замедляют выполнения запросов. Поэтому часто приходится идти на компромиссы и дополнительную настройку транзакций. Изолированность — более сложное понятие, чем кажется на первый взгляд. Стандарт SQL

определяет четыре уровня изоляции с конкретными правилами, устанавливающими, какие изменения видны внутри и вне транзакции, а какие нет:

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

Более низкие уровни изоляции обычно допускают большую степень совместного доступа и вызывают меньше накладных расходов. На первом уровне изоляции, **READ UNCOMMITTED**, транзакции могут видеть результаты незафиксированных транзакций.

На практике **READ UNCOMMITTED** используется редко, поскольку его производительность не намного выше, чем у других. На этом уровне вы видите промежуточные результаты чужих транзакций, т.е. осуществляете грязное чтение.

Уровень **READ COMMITTED** подразумевает, что транзакция увидит только те изменения, которые были уже зафиксированы другими транзакциями к моменту ее начала. Произведенные ею изменения останутся невидимыми для других транзакций, пока она не будет зафиксирована. На этом уровне возможен феномен невоспроизводимого чтения. Это означает, что вы можете выполнить одну и ту же команду дважды и получить различный результат.

Уровень изоляции **REPEATABLE READ** решает проблемы, которые возникают на уровне **READ UNCOMMITTED**. Он гарантирует, что любые строки, которыечитываются в контексте транзакции, будут выглядеть такими же при последовательных операциях чтения в пределах одной и той же транзакции, однако теоретически на этом уровне возможен феномен фантомного чтения (*phantom reads*).

Он возникает в случае, если вы выбираете некоторый диапазон строк, затем другая транзакция вставляет новую строку в этот диапазон, после чего вы выбираете тот же диапазон снова. В результате вы увидите новую фантомную строку. Уровень изоляции **REPEATABLE READ** установлен по умолчанию.

Самый высокий уровень изоляции, **SERIALIZABLE**, решает проблему фантомного чтения, заставляя транзакции выполняться в таком порядке, чтобы исключить возможность конфликта. Уровень **SERIALIZABLE** блокирует каждую строку, которую транзакция читает

Переменные

Часто результаты запроса необходимо использовать в последующих запросах. Для этого полученные данные следует сохранить во временных структурах. Этую задачу решают переменные SQL.

```
SELECT @total := COUNT(*) FROM accounts;
```

Объявление переменной начинается с символа @, за которым следует имя переменной. Значения переменным присваиваются посредством оператора SELECT с использованием оператора присваивания :=. В следующих запросах мы получаем возможность обращаться к переменной:

```
SELECT @total;
```

Переменные также могут объявляться при помощи оператора SET. Команда SET, в отличие от оператора SELECT, не возвращает результирующую таблицу:

```
SET @last = NOW() - INTERVAL 7 DAY; -- от текущей даты отнять 7  
дней SELECT CURDATE(), @last;
```

Временная таблица

Временная таблица автоматически удаляется по завершении соединения с сервером, а ее имя действительно только в течение данного соединения. Это означает, что два разных клиента могут использовать временные таблицы с одинаковыми именами без конфликта друг с другом или с существующей таблицей с тем же именем.

В MySQL временная таблица — это особый тип таблицы, который позволяет вам сохранить временный набор результатов, который вы можете повторно использовать несколько раз в одном сеансе. Временная таблица очень удобна, когда невозможно запрашивать данные, требующие одного SELECT оператора с JOIN предложениями.

Временная таблица создается с помощью CREATE TEMPORARY TABLE. Обратите внимание, что ключевое слово TEMPORARY добавлено между

```
CREATE TEMPORARY TABLE temp (id INT, name VARCHAR(255)); DESCRIBE  
temp; -- Показ всех столбцов в таблице temp
```

Хранимые процедуры и функции

Хранимые процедуры и функции позволяют сохранить последовательность SQL-операторов и

вызывать их по имени функции или процедуры:

- CREATE PROCEDURE procedure_name
- CREATE FUNCTION function_name

Разница между процедурой и функцией заключается в том, что функции возвращают значение и их можно встраивать в SQL-запросы, в то время как хранимые процедуры вызываются явно.

Процедуры.

Синтаксис:

```
DELIMITER {custom delimiter} CREATE PROCEDURE {proc_name}([optional  
parameters]) BEGIN // procedure body... // procedure body... END  
{custom delimiter}
```

- **delimiter** - это команда, которая необходима для изменения разделителя SQL-инструкций с ; на // во время определения процедуры. Это позволяет разделитель ; использовать в теле процедуры для передачи на сервер.
- **proc_name** - уникальное имя хранимой процедуры, длиной не более 64 символа. Имена процедур не чувствительны к регистру, поэтому в одной схеме не может быть двух событий с именами procname и ProcName ;
- в процедуру можно передать параметры (optional_params)

IN - Параметр может ссылаться на процедуру. Значение параметра не может быть перезаписано процедурой.

OUT - Параметр не может ссылаться на процедуру, но значение параметра может быть перезаписано процедурой.

IN OUT - Параметр может ссылаться на процедуру, и значение параметра может быть перезаписано процедурой.

Для создания хранимой процедуры предназначен оператор CREATE PROCEDURE, после которого указывается имя процедуры. Давайте создадим процедуру, которая выводит текущую версию MySQL-сервера:

```
DELIMITER // CREATE PROCEDURE my_version () BEGIN SELECT VERSION();  
END // -- CALL proc_name; CALL my_version (); -- Вызов процедуры
```

После команды CREATE PROCEDURE указывается имя процедуры и круглые скобки, в которых обычно указывают входящие и исходящие параметры. Мы рассмотрим их чуть позже в рамках текущего урока. Между ключевыми словами BEGIN и END размещаются SQL-команды, которые выполняются всякий раз при вызове хранимой процедуры.

Чтобы воспользоваться только что созданной хранимой процедурой, используем команду CALL, после которой указываем имя вызываемой процедуры:

```
CALL my_version();
```

Мы получили текущую версию MySQL-сервера. Чтобы получить список хранимых процедур, можно воспользоваться командой:

```
SHOW PROCEDURE STATUS; -- Все процедуры SHOW PROCEDURE STATUS LIKE  
'my_version%'; -- Конкретная процедура -- При использовании  
ключевого слова -- LIKE можно вывести информацию только о тех  
процедурах, -- имена которых удовлетворяют шаблону
```

Для удаления хранимых процедур и функций предназначены операторы DROP PROCEDURE и DROP FUNCTION. Давайте удалим процедуру my_version:

```
DROP PROCEDURE my_version;
```

Синтаксис команды допускает использование ключевого слова IF EXISTS:

```
DROP PROCEDURE IF EXISTS my_version;
```

В этом случае, если хранимой процедуры уже не существует, команда завершается без сообщения об ошибке.

Функции

Синтаксис:

```
DELIMITER {custom delimiter} CREATE FUNCTION function_name [
```

```
(parameter datatype [, parameter datatype]) ] RETURNS  
return_datatype BEGIN declaration_section executable_section END;
```

Пример:

```
CREATE FUNCTION get_version () RETURNS TEXT DETERMINISTIC BEGIN  
RETURN VERSION(); END -- Процедура не до конца написана
```

Функция создается командой CREATE FUNCTION, после которой идет имя функции. Хранимая

Функция встраивается в SQL-запросы, как обычная mysql-функция. Она должна возвращать значение. Ключевое слово RETURNS указывает возвращаемый тип, например TEXT мы можем заменить на VARCHAR(255). Ключевое слово DETERMINISTIC (дэтерминистик) сообщает, что результат функции детерминирован, т.е., при каждом вызове будет возвращаться одно и то же значение, и если его закешировать в рамках запроса, ничего страшного не произойдет. Если значения, которые возвращает функция, каждый раз различны, то перед DETERMINISTIC (дэтерминистик) следует добавить отрицание NOT. Далее следует тело функции, которое размещается между ключевыми словами BEGIN и END. Внутри тела обязательно должно присутствовать ключевое слово RETURN, которое возвращает результат вычисления. В данном случае мы просто возвращаем результат вызова mysql-функции VERSION().

Для вызова хранимой функции не требуется специальной команды, как в случае хранимых процедур. Порядок их вызова совпадает с порядком вызова встроенных функций MySQL:

```
SELECT get_version();
```

Основная трудность, которая возникает при работе с хранимыми процедурами и функциями,

заключается в том, что символ точки с запятой (;) используется в теле запроса для разделения

SQL-команд. Создание хранимой процедуры или функции — это тоже команда, которая тоже должна завершаться точкой с запятой. В результате возникает конфликт.

Чтобы его избежать, во всех клиентах предусмотрена возможность переназначать признак окончания запроса, в консольном клиенте mysql это осуществляется при помощи команды DELIMITER.

Давайте снова назначим разделителем два слеша:

```
DELIMITER //
```

Теперь мы можем воспользоваться новым разделителем:

```
-- Рабочий вариант DROP FUNCTION IF EXISTS get_version; DELIMITER
// CREATE FUNCTION get_version () RETURNS TEXT DETERMINISTIC BEGIN
RETURN VERSION(); END// SELECT get_version();
```

Параметры процедур и функций

Хранимые процедуры и функции могут использовать параметры. Параметры могут передавать значения внутрь функции и извлекать результаты вычисления. Для этого каждый из параметров снабжается одним из атрибутов: IN, OUT или INOUT.

Параметр param_name предваряет одно из ключевых слов IN, OUT, INOUT, которые позволяют задать направление передачи данных:

- IN — данные передаются строго внутрь хранимой процедуры, но если параметру с данным модификатором внутри функции присваивается новое значение, по выходу из нее оно не сохраняется и параметр принимает значение, которое он имел до вызова процедуры.
- OUT — данные передаются строго из хранимой процедуры. Даже если параметр имеет какое-то начальное значение, внутри хранимой процедуры оно не принимается во внимание.

С другой стороны, если параметр изменяется внутри процедуры, после ее вызова он имеет значение, присвоенное ему внутри процедуры.

- INOUT — значение этого параметра как принимается во внимание внутри процедуры, так и сохраняет свое значение по выходу из нее.

Атрибуты IN, OUT и INOUT доступны лишь для хранимой процедуры, в хранимой функции все параметры всегда имеют атрибут IN.

Давайте сразу установим в качестве разделителя два слеша, для этого используем команду

DELIMITER. Создадим простейшую процедуру get_x, которая принимает единственный параметр value и устанавливает переменную.

```
DELIMITER // DROP PROCEDURE IF EXISTS set_x// CREATE PROCEDURE
set_x (IN value INT) BEGIN SET @x = value; SET value = value -
1000; END// SET @y = 10000// CALL set_x(@y)// SELECT @x, @y// x =
10000, y = 10000
```

спользование ключевого слова IN не обязательно — если ни один из атрибут не указан, СУБД MySQL считает, что параметр объявлен с атрибутом IN. В теле процедуры мы используем команду SET, чтобы создать пользовательскую переменную @x. Напоминаю, что наши собственные переменные создаются с использованием символа @. При вызове хранимой процедуры мы можем передать в круглых скобках значение, которое будет использоваться вместо параметра внутри хранимой функции.

В отличие от пользовательской переменной @x, которая является глобальной и доступна как внутри хранимой процедуры set_x(), так и вне ее, параметры функции локальны и доступны для использования только внутри функции.

Хранимая процедура set_x() принимает единственный IN-параметр value, при помощи оператора SET его значение изменяется внутри функции. Однако после выполнения хранимой процедуры значение пользовательской переменной @y, переданной функции в качестве параметра, не изменяется. Если требуется, чтобы значение переменной менялось, необходимо объявить параметр процедуры с модификатором OUT.

При использовании модификатора OUT любые изменения параметра внутри процедуры отражаются на аргументе. Передача в качестве значения пользовательской переменной позволяет использовать результат процедуры для дальнейших вычислений. Однако передать значение внутрь функции при помощи OUT-параметра уже не получится.

```
DELIMITER // DROP PROCEDURE IF EXISTS set_x// CREATE PROCEDURE
set_x (OUT value INT) BEGIN SET @x = value; SET value = 1000; END// 
SET @y = 10000// CALL set_x(@y)// SELECT @x, @y// x = NULL, y =
1000
```

Чтобы через параметр можно было и передать значение внутрь процедуры, и получить значение, которое попадает в параметр в результате вычислений внутри процедуры, его следует объявить с атрибутом INOUT:

```
DELIMITER // DROP PROCEDURE IF EXISTS set_x// CREATE PROCEDURE
set_x (INOUT value INT) BEGIN SET @x = value; SET value = value -
1000; END// SET @y = 10000// CALL set_x(@y)// SELECT @x, @y// -- x
= 10000, y = 9000
```

До этого момента локальные переменные в хранимой процедуре или функции объявлялись как входящие или исходящие параметры, однако это не всегда удобно.

Часто требуется локальная переменная без необходимости передавать или

возвращать с ее
помощью какие-либо значения.

Объявить такую переменную можно при помощи команды DECLARE. Один оператор DECLARE

позволяет объявить сразу несколько переменных одного типа, причем необязательное слово

DEFAULT позволяет назначить инициирующее значение. Те переменные, для которых не указывается ключевое слово DEFAULT, можно инициировать при помощи команды SET. Позже остановимся на этом подробнее. Команда DECLARE может появляться только внутри блока BEGIN...END, область видимости объявленной переменной также ограничена этим блоком.

```
DELIMITER // DROP PROCEDURE IF EXISTS declare_var// CREATE
PROCEDURE declare_var () BEGIN DECLARE var TINYTEXT DEFAULT
'внешняя переменная'; BEGIN DECLARE var TINYTEXT DEFAULT
'внутренняя переменная'; SELECT var; END; SELECT var; END// CALL
declare_var()//
```

Это означает, что в разных блоках BEGIN...END могут быть объявлены переменные с одинаковым именем, и действовать они будут только в рамках одного блока, не пересекаясь с переменными других.

Ветвление

Оператор IF позволяет реализовать ветвление программы по условию. IF принимает значение либо TRUE (истину), либо FALSE (ложь). В MySQL TRUE и FALSE — константы для целочисленных значений 1 и 0. Если логическое выражение истинно, IF выполняет SQL-выражения, которые размещаются в теле команды между ключевыми словами THEN и END IF.

```
DELIMITER // DROP PROCEDURE IF EXISTS format_now// CREATE PROCEDURE
format_now (format CHAR(4)) BEGIN IF(format = 'date') THEN SELECT
DATE_FORMAT(NOW(), "%d.%m.%Y") AS format_now; END IF; IF(format =
'time') THEN SELECT DATE_FORMAT(NOW(), "%H:%i:%s") AS format_now;
END IF; END// CALL format_now('date')// CALL format_now('time')//
```

Команда IF поддерживает ключевое слово ELSE. Давайте перепишем процедуру format_now с использованием ELSE:

```
DELIMITER // DROP PROCEDURE IF EXISTS format_now// CREATE PROCEDURE
format_now (format CHAR(4)) BEGIN IF(format = 'date') THEN SELECT
DATE_FORMAT(NOW(), "%d.%m.%Y") AS format_now; ELSE SELECT
DATE_FORMAT(NOW(), "%H:%i:%s") AS format_now; END IF; END// CALL
format_now('date')// CALL format_now('time')//
```

Если параметр format равен 'date', условие в IF является истинным, выполняется первый блок, выводящий текущую дату. Если аргумент принимает любое другое условие, условие в IF является ложным и запрос выполняется в блоке ELSE.

Циклы

Циклы являются важнейшей конструкцией, без которой хранимые процедуры и функции не имели бы достаточно функциональности. MySQL предоставляет три цикла: **while**, **repeat** и **loop**. Их можно использовать в теле хранимой процедуры или функции, т.е., между ключевыми словами BEGIN и END.

```
DELIMITER // CREATE PROCEDURE while_cycle () BEGIN DECLARE i INT  
DEFAULT 3; WHILE i > 0 DO SELECT NOW(); SET i = i - 1; END WHILE;  
END//
```

Здесь в процедуре while_cycle используется цикл WHILE для трехкратного вывода даты и времени. Цикл начинается с ключевого слова WHILE, после которого следует условие. Условие вычисляется на каждой итерации цикла: если оно возвращает истину (TRUE), очередная итерация выполняется, если при очередной проверке оно будет ложным (FALSE), цикл завершит работу. Чтобы не создать бесконечный цикл, условие подбирается таким образом, чтобы рано или поздно оно становилось ложным и цикл прекращал свою работу. Цикл while, в свою очередь, сам имеет тело, начало которого обозначается ключевым словом DO, а завершение — ключевым словом END WHILE.

Все команды, которые располагаются между этими ключевыми словами, выполняются на каждой итерации цикла. Обратите внимание: перед циклом мы заводим переменную i, которой при помощи ключевого слова DEFAULT устанавливаем значение 3.

На каждой итерации мы уменьшаем значение i на единицу: пока i больше нуля, условие цикла остается истинным. Как только значение уменьшается до 0, условие возвращает FALSE и цикл завершает работу. Таким образом, текущая дата будет выведена только три раза. Давайте в этом убедимся.

```
CALL while_cycle()// --2023-03-19 15:37:53
```

Количество повторов не обязательно задавать внутри хранимой процедуры. Например, мы можем задать его в качестве входящего параметра.

```
DELIMITER // DROP PROCEDURE IF EXISTS while_cycle; CREATE PROCEDURE  
while_cycle (IN num INT) BEGIN DECLARE i INT DEFAULT 0; IF (num >
```

```
0) THEN WHILE i < num DO SELECT NOW(); SET i = i + 1; END WHILE;
ELSE SELECT 'Ошибканое значение параметра'; END IF; END// CALL
while_cycle(2)//
```

Итак, у нас выводится только две даты. Для досрочного выхода из цикла предназначен оператор LEAVE. Давайте ограничим цикл в процедуре NOWN только двумя итерациями, т.е., сколько бы выводов пользователь ни заказывал, максимальное количество, которое будет доступно — 2 . В тело цикла добавляется дополнительное if-условие, не допускающее достижение счетчика i значения 2. Как только условие срабатывает, выполняется команда LEAVE. Циклы можно вкладывать друг в друга, поэтому, чтобы команда LEAVE понимала, какой из циклов следует останавливать, ей всегда передается метка цикла, в данном случае cycle. Эту метку мы должны поместить перед ключевым словом WHILE и после ключевого слова END WHILE. Давайте запросим заведомо огромное значение, например 1000:

```
DELIMITER // DROP PROCEDURE IF EXISTS while_cycle// CREATE
PROCEDURE while_cycle (IN num INT) BEGIN DECLARE i INT DEFAULT 0;
IF (num > 0) THEN cycle: WHILE i < num DO IF i >= 2 THEN LEAVE
cycle; END IF; SELECT NOW(); SET i = i + 1; END WHILE cycle; ELSE
SELECT 'Ошибканое значение параметра'; END IF; END// CALL
while_cycle(1000)//
```

Как видим, выводятся только две даты, у нас сработал досрочный выход из цикла.

Еще один оператор, выполняющий досрочное прекращение цикла — ITERATE. В отличие от оператора LEAVE, ITERATE не прекращает выполнение цикла, он лишь досрочно прекращает текущую итерацию.

Давайте создадим хранимую процедуру, которая продемонстрирует ITERATE на практике:

```
DELIMITER // DROP PROCEDURE IF EXISTS numbers_string// CREATE
PROCEDURE numbers_string (IN num INT) BEGIN DECLARE i INT DEFAULT
0; DECLARE bin TINYTEXT DEFAULT ''; IF (num > 0) THEN cycle : WHILE
i < num DO SET i = i + 1; SET bin = CONCAT(bin, i); IF i >
CEILING(num / 2) THEN ITERATE cycle; -- CEILING: возвращает
наименьшее целое число END IF; SET bin = CONCAT(bin, i); END WHILE
cycle; SELECT bin; ELSE SELECT 'Ошибканое значение параметра'; END
IF; END// CALL numbers_string(9)//
```

Внутри цикла счетчик i пробегает значения от 1 до 9, на каждой итерации значение счетчика добавляется к строке bin. Если if-условие ложное, то значение добавляется два раза, если истинное, срабатывает оператор ITERATE и текущая итерация

завершается досрочно. Именно поэтому в результатах мы видим удвоенные цифры до 5 и одиночные цифры после 5.

Оператор **REPEAT** похож на оператор WHILE.

```
DELIMITER // DROP PROCEDURE IF EXISTS repeat_cycle// CREATE
PROCEDURE repeat_cycle () BEGIN DECLARE i INT DEFAULT 3; REPEAT
SELECT NOW(); SET i = i - 1; UNTIL i <= 0 END REPEAT; END// CALL
repeat_cycle()//
```

Однако условие для покидания цикла располагается не в начале тела цикла, а в конце. В результате тело цикла в любом случае выполняется хотя бы один раз. В конце цикла после ключевого слова UNTIL располагается условие; если оно истинно, работа цикла прекращается, если ложно, происходит еще одна итерация. Эта хранимая процедура должна выполняться в теле цикла три раза.

Цикл **LOOP**, в отличие от операторов WHILE и REPEAT, не имеет условий выхода. Поэтому он должен обязательно иметь в составе оператор LEAVE.

```
DELIMITER // DROP PROCEDURE IF EXISTS loop_cycle// CREATE PROCEDURE
loop_cycle () BEGIN DECLARE i INT DEFAULT 3; cycle: LOOP SELECT
NOW(); SET i = i - 1; IF i <= 0 THEN LEAVE cycle; END IF; END LOOP
cycle; END// CALL loop_cycle()//
```

Так как мы используем оператор LEAVE, мы должны разместить перед ключевым словом LOOP и после END LOOP метку. Здесь она называется cycle.

Используемые источники

1. <https://dev.mysql.com/doc/refman/5.7/en/sql-syntax-transactions.html>
2. <https://dev.mysql.com/doc/refman/5.7/en/server-system-variables.html>
3. <https://dev.mysql.com/doc/refman/5.7/en/create-temporary-table.html>
4. <https://dev.mysql.com/doc/refman/5.7/en/sql-syntax-prepared-statements.html>
5. <https://dev.mysql.com/doc/refman/5.7/en/create-view.html>
6. Линн Бейли. Head First. Изучаем SQL. — СПб.: Питер, 2012. — 592 с.
7. 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.
8. Дейт К. Дж. SQL и реляционная теория. Как грамотно писать код на SQL. —
Пер. с англ. —
СПб.: Символ-Плюс, 2010. — 480 с.
9. Кузнецов М.В., Симдянов И.В. MySQL на примерах. — СПб.: БХВ-
Петербург, 2007. — 592с.

10. Кузнецов М.В., Симдянов И.В. MySQL 5. — СПб.: БХВ-Петербург, 2006. — 1024с.
11. Дейт К. Дж. Введение в системы баз данных, 8-е издание.: Пер. с англ. — М.: Издательский дом "Вильямс", 2005. — 1328 с.
12. Карвин Б. Программирование баз данных SQL. Типичные ошибки и их устранение. — Рид Групп, 2011. — 336 с.

Книги:

- “Изучаем SQL”, книга Бейли Л.
 - Алан Бьюли "Изучаем SQL" (2007)
 - Энтони Молинаро "SQL. Сборник рецептов" (2009)
1. Виктор Гольцман “MySQL 5.0. Библиотека программиста”
 2. “Изучаем SQL”, книга Бейли Л.
 3. <https://www.webmasterwiki.ru/MySQL>
 4. Поль Дюбуа “MySQL. Сборник рецептов”



Индексы и ключи

Доброго времени суток, уважаемые студенты! Небольшая дополнительная лекция по индексам и ключам

Небольшое вступление

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.

1	
2	
3	
4	
5	
6	
	4 Модули памяти
	6 Блоки питания
	2 Материнские платы
	5 Жесткие диски и SSD
	1 Процессоры
	3 Видеокарты

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;
```

	Field	Type	Null	Key	Default	Extra
▶	id	int unsigned	NO	PRI	NULL	
	name	varchar(255)	YES		NULL	

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.

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

	Field	Type	Null	Key	Default	Extra
▶	id	int unsigned	NO	PRI	NULL	
	name	varchar(255)	NO	PRI	NULL	

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

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

	id	name
▶	1	Процессоры
	2	Мат.платы
	3	Видеокарты
*	NULL	NULL

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 =
'Tоварные позиции'; DESCRIBE products;
```

	Field	Type	Null	Key	Default	Extra
▶	id	bigint unsigned	NO	PRI	NULL	auto_increment
	name	varchar(255)	YES		NULL	
	desription	text	YES		NULL	
	price	decimal(11,2)	YES		NULL	
	catalog_id	int unsigned	YES	MUL	NULL	
	created_at	datetime	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED
	updated_at	datetime	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED on update CURRENT_TI...

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**:

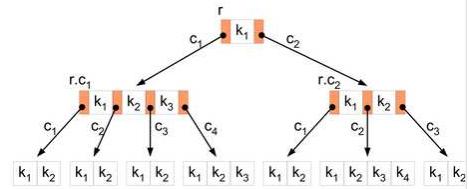
```
DROP INDEX index_of_catalog_id ON products;
```

Устройство индекса

В-дерево

В-дерево — структура данных, дерево поиска. С точки зрения внешнего логического представления,

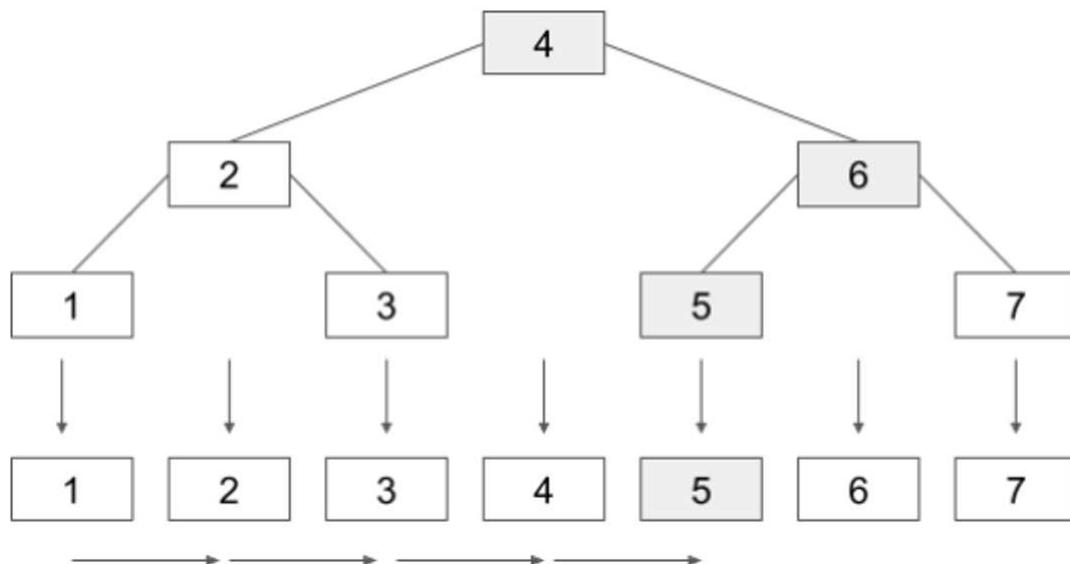
☞ <https://ru.wikipedia.org/wiki/В-дерево>



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.

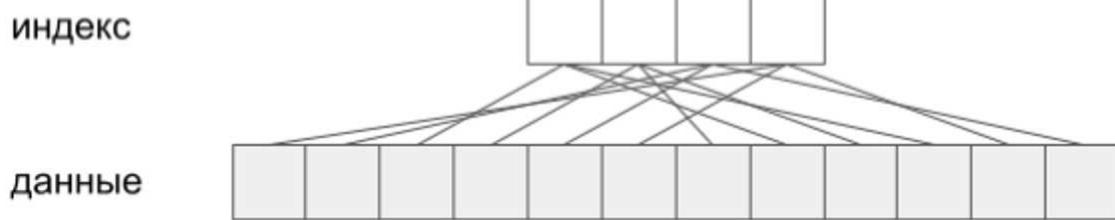


Поскольку узлы дерева отсортированы, их можно использовать как для поиска значений, так и в запросах с сортировкой при помощи **ORDER BY**.

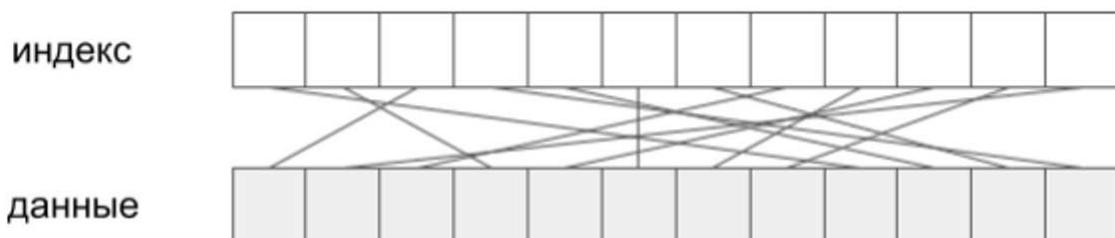
```
CREATE INDEX index_of_catalog_id USING BTREE ON products  
(catalog_id); CREATE INDEX index_of_catalog_id USING HASH ON  
products (catalog_id);
```

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.

Обычный индекс



Уникальный индекс



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:

```
DROP TABLE IF EXISTS products; CREATE TABLE products ( id SERIAL
PRIMARY KEY, name VARCHAR(255) COMMENT 'Название', description 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

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

1990	Борисов	Игорь
1990	Сергеев	Вячеслав
1991	Антонов	Александр
1991	Ковалев	Сергей
1991	Трофимов	Антон

```
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';
```

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

Книги:

- “Изучаем SQL”, книга Бейли Л.
 - Алан Бьюоли "Изучаем SQL" (2007)
 - Энтони Молинаро "SQL. Сборник рецептов" (2009)
1. Виктор Гольцман “MySQL 5.0. Библиотека программиста”
 2. “Изучаем SQL”, книга Бейли Л.
 3. <https://www.webmasterwiki.ru/MySQL>
 4. Поль Дюбуа “MySQL. Сборник рецептов”

