**Урок 1**

Быстрый старт

[Быстрый старт](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.qm7crjtv7mpb)

[Данные и программы](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.1t3h5sf)

[История развития СУБД](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.3tb0lm9pg2u0)

[Иерархические базы данных](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.h727mdfurd3m)

[Сетевые базы данных](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.wewqj08qo8nf)

[Реляционные базы данных](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.g2mfcb94hoa1)

[Индекс популярности баз данных](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.dabm1dkdiptc)

[NoSQL базы данных](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.qwgpx1onbknp)

[Основы реляционных баз данных](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.4d34og8)

[Таблицы](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.r87za5j1xbqi)

[Строки](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.vn0q8dl8qx5t)

[Столбцы](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.fdieubk5eicd)

[Пустая таблица](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.cuftpojsj4cd)

[CAP-теорема](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.h0aj31s6ax2d)

[Архитектура MySQL](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.lubbpltltuvl)

[Клиенты MySQL](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.4qti5ad1wjcg)

[Управление базами данных](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.6e5yebhytxme)

[Создание таблиц](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.2jin0gsygh5f)

[Информационная схема](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.ku7aj3r90e23)

[Документация](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.aqdvxtsbffpe)

[Дополнительные материалы](https://docs.google.com/document/d/1D0sg3uFZhO_j55bOaQQHTOg392LnsaH0xCKU18lmWEc/edit#heading=h.2xcytpi)

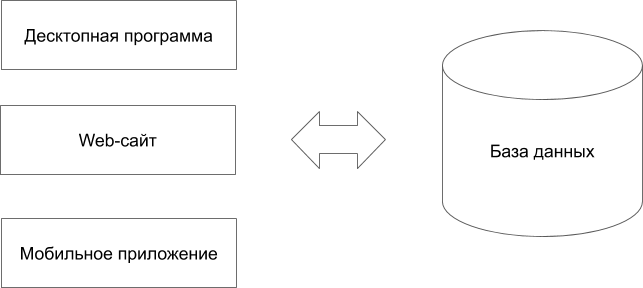
**Быстрый старт**

Знакомство с реляционными базами данных будет вестись на примере MySQL.

**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 является базой данных, пусть и очень примитивной.

**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 убирается 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 пробежимся по истории СУБД, от первых иерархических баз данных до современных NoSQL-решений.

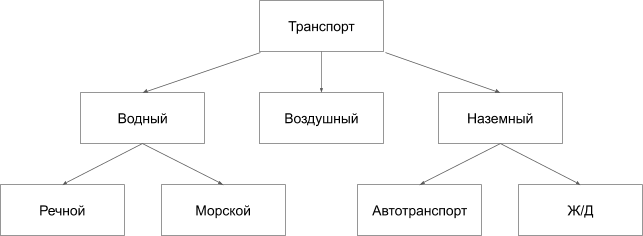
**Иерархические базы данных**

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.

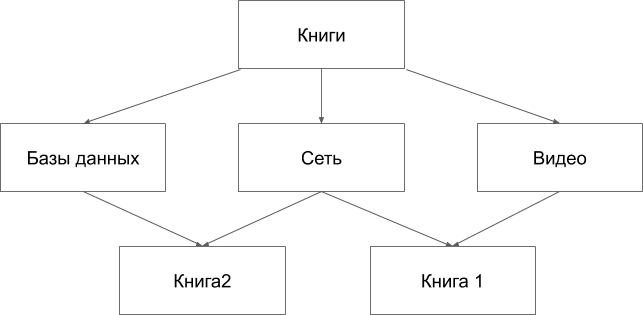
Иерархии очень наглядны и хорошо описываются деревьями, у которых прекрасно изученный мат. аппарат.



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

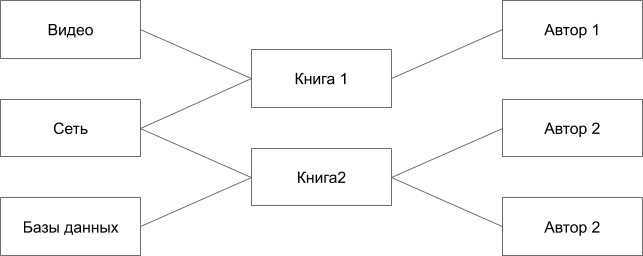
Основной недостаток иерархической структуры базы данных — невозможность реализовать отношения «многие ко многим».

Например, если мы создаём каталог книг, одна книга может относится 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».

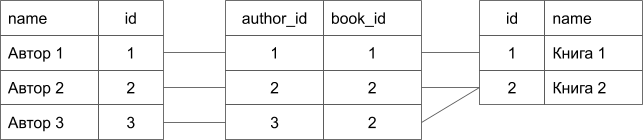


В связи с развитием социальных сетей эти базы данных получили второе дыхание в виде графовых СУБД, которые относятся к современному NoSQL-течению. Только в современной интерпретации ценность приобретают не сами данные, а связи между узлами. Тем не менее, что примечательно, многие идеи, которые появляются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



Теоретические основы новой реляционной модели данных впервые были описанны 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 компания 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.

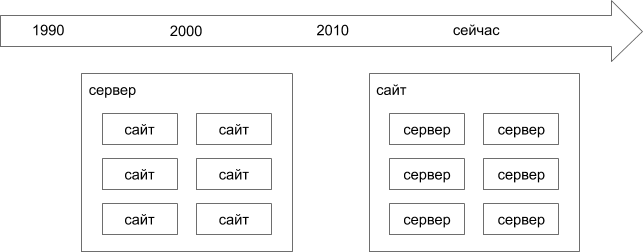
Наиболее популярные СУБД на сегодняшний день — Oracle, MS SQL и DB2 среди коммерческих, MySQL, PostgreSQL и Firebird среди свободных.

**Индекс популярности баз данных**

Следить за индексом популярности баз данных можно по рейтингу на сайте [db-engines.com](https://db-engines.com/en/ranking). 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**

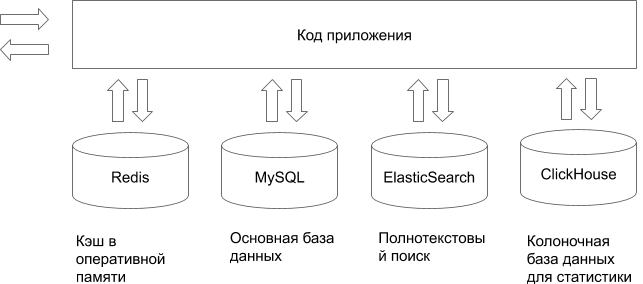
После стремительного развития интернета в принципах построения баз данных произошел переломный момент. Первые веб-сайты и сервисы были очень невелики: зачастую на одном сервере убирались сотни сайтов. Однако по мере вовлечения все новых и новых пользователей проекты начали укрупняться и очень скоро им 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, предоставляющие 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 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 дата-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 позволят 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 кэш. 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.

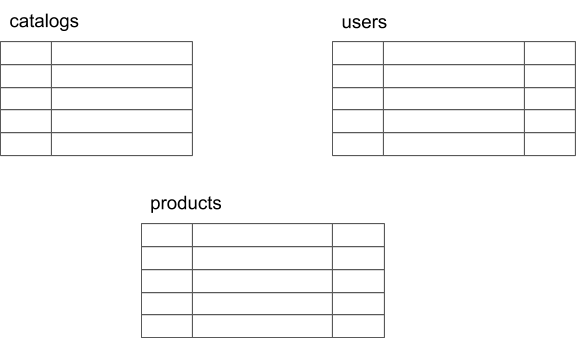
На протяжении всего курса мы будем рассматривать реляционные базы данных на примере MySQL. Последнее занятие нашего курса будет посвящено современным NoSQL-решениям.

**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.



Давайте 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.



**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.

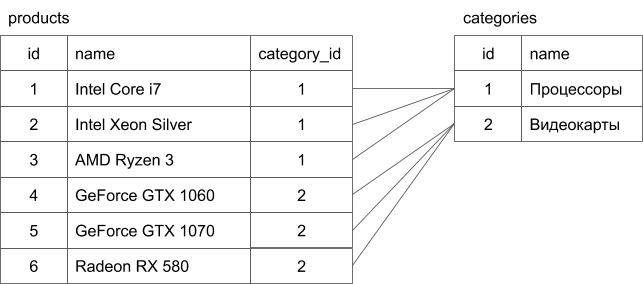


**Связи между таблицами**

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.

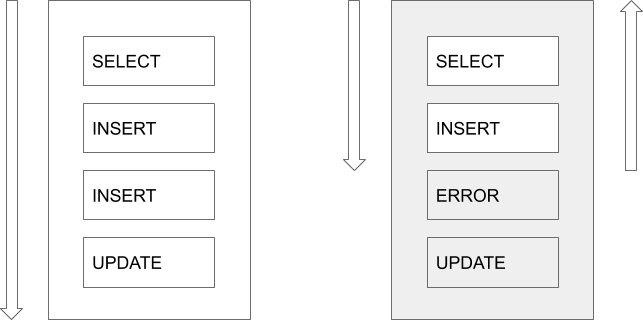
**Транзакции**

В приложении многое может пойти не так:

* может отказать аппаратное обеспечение;
* может произойти исключительная ситуация внутри приложения, в том числе когда последовательность операций выполнена наполовину;
* разрыв сети может отрезать приложение от базы данных;
* несколько клиентов могут одновременно изменять одни и те же данные.

Все эти проблемы обычно решаются при помощи транзакций.

Транзакции — это способ группировки приложением нескольких операций записи и чтения в одну логическую единицу. По сути все операции записи и чтения в ней выполняются как одна: вся транзакция либо целиком выполняется успешно (с фиксацией изменений) или целиком завершается неудачно (с прерыванием и откатом к исходному состоянию). Транзакции значительно упрощают для приложения обработку ошибок, так как нет нужды заботиться о частичных отказах, когда часть операций завершилась успешно, а часть — нет.



**Принцип ACID**

Практически все реляционные базы данных поддерживают транзакции, которые остались почти неизменными с самых первых реляционных баз данных.

Появившиеся в конце 2000-х годов нереляционные NoSQL-базы данных ставили целью улучшить положение дел с реляционными базами данных c помощью новых моделей данных, репликации и секционирования. Транзакции стали главной жертвой этого новшества. Многие базы данных нового поколения полностью от них отказались. Или поменяли значение термина: теперь он стал означать намного более слабый набор функциональных гарантий, чем ранее.

Гарантии безопасности, которые обеспечивают транзакции, обычно принято обозначать аббревиатурой ACID — атомарность, согласованность, изоляция и сохраняемость.

Атомарность обозначает реакцию базы данных на сбой, когда в рамках транзакции операции записи выполнены лишь наполовину. Если транзакцию не удается завершить (зафиксировать изменения), то она прерывается и базе данных приходится откатить все уже выполненные в рамках этой транзакции операции записи. Без атомарности сложно было бы понять, какие операции уже выполнены, а какие — нет. Если попробовать выполнить операции повторно, возникает риск, что часть операций будет продублирована.

Согласованность означает выполнение бизнес-правил. Не должно быть отрицательного возраста или записей, которые ссылаются на уже удаленные записи. В системе бухгалтерского учета кредит должен сходиться с дебетом. Если транзакция начинается с состояния базы данных, которое удовлетворяет этим правилам, то после выполнения транзакции база данных должна также удовлетворять им. Таким образом данные переходят из одного корректного состояния в другое, тоже корректное, не оставляя стороннему наблюдателю возможности увидеть недопустимую совокупность промежуточных значений.

Сейчас бизнес-правила очень часто хранятся в приложении, база данных может ничего не знать о сложных правилах бизнес-логики. Она способна проверять лишь некоторые специальные виды правил, например, ограничения внешнего ключа или уникальности. Однако в целом допустимость или недопустимость данных определяется приложением — база данных лишь обеспечивает хранение. Есть мнение, что букву С в эту аббревиатуру добавили для красоты, чтобы получилось красивое и запоминающееся слово acid (кислота).

Изолированность означает, что параллельно выполняемые транзакции изолированы друг от друга — они не могут помешать друг другу. Т. е., каждая транзакция выполняется так, будто она единственная во всей базе данных. База данных гарантирует, что результат фиксации транзакций такой же, как если бы они выполнялись последовательно, одна за другой, хотя в реальности они могут выполняться параллельно.

Задача СУБД — предоставить надежное место для хранения данных.

Сохраняемость — это обязательство базы данных не терять записанных данных, даже в случае сбоя аппаратного обеспечения или фатального сбоя самой базы данных.

Правила довольно неплохие, почему современные NoSQL-базы данных от них отказываются?

С развитием веба мы получили не просто популярные приложения, а приложения, к которым одновременно обращаются тысячи клиентов. Это приводит к проблема масштабируемости баз данных. Даже в небольших реляционных базах данных никуда не деться от операций соединения, которые могут оказаться медленными.

Под транзакциями подразумевают блокировку некоторой части данных, из-за чего они становятся недоступными другим клиентам. При высоких нагрузках это может стать неприемлемым, так как из-за блокировок запросы пользователей выстраиваются в очередь.

Проблема в том, что мы не можем бесконечно долго заниматься вертикальным масштабированием, увеличивая мощность серверов, на которых расположена СУБД. Рано или поздно нам приходится размещать базу данных на нескольких серверах. Делается это при помощи механизма репликации, с которым мы обязательно познакомимся в рамках курса. Таким образом, база данных оказывается на разных машинах или даже в разных дата-центрах.

Как только база данных становится распределенной, нам необходимо обеспечивать ее устойчивость к сетевым сбоям.

**CAP-теорема**

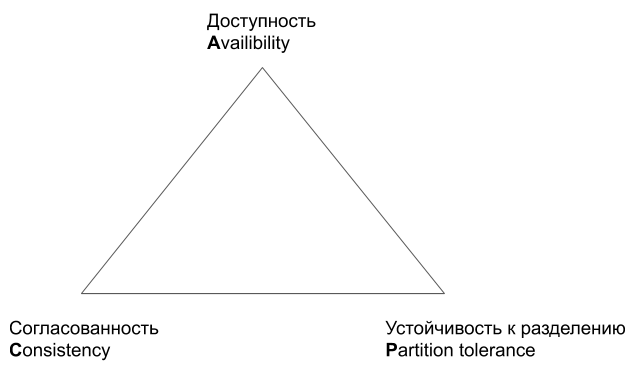
Противоречия обеспечения согласованности и распределенных систем часто описывает CAP-теорема, которую в 2000 году сформулировал Эрик Брюер. Теорема касается трех взаимосвязанных требований, существующих в большой распределенной системе данных:

**Согласованность** означает, что все клиенты должны прочитать одно и то же значение в ответ на один и тот же запрос, даже если одновременно с этим производятся обновления.

**Доступность** соответствует тому, что все клиенты базы данных всегда имеют возможность читать и записывать данные. Мы не можем ждать для этого завершения каких-либо операций.

**Устойчивость** к разделению означает, что базу данных можно разделить между несколькими машинами; более того, она продолжит работу даже в случае отказа сегмента сети.

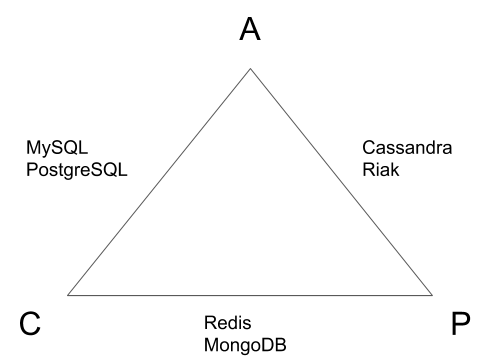
CAP-теорема утверждает, что в любой системе можно гарантированно обеспечить выполнение только двух из этих трех требований. Это аналог поговорки «Программа может быть хорошей, быстрой и дешевой — выбирай любые два свойства».



Чем большая согласованность требуется от системы, тем меньше будет ее устойчивость к разделению.

Однако в распределенной системе сетевые сбои очень вероятны. Мы не можем избежать задержек и потерь пакетов, обрыва связанности, вплоть до того, что часть распределенной базы данных оказывается изолированной от другой. Поэтому избежать узла устойчивости к разделению в современных веб-приложениях почти невозможно. Таким образом, любая СУБД может располагаться лишь на одной грани CAP-треугольника. Точнее это зависит от режима работы СУБД, так как часто многие СУБД могут располагаться на той или иной грани.

Традиционные реляционные СУБД, спроектированные для работы на одном компьютере, обеспечивают согласованность и доступность. Они с трудом могут обеспечивать устойчивость к разделению. Как видите, MySQL, PostgreSQL, реляционные базы данных находятся на грани CA (согласованность и доступность). Существуют сторонние решения и от Google и от компании Percona, которые позволяют перевести базу данных MySQL на грань CP, т. е., обеспечить согласованность данных и устойчивость к разделению. Однако в этом случае страдает доступность: мы вынуждены дожидаться операции обновления базы данных.



Существует большое количество самых разнообразных NoSQL-решений, которые находятся на разных гранях этого треугольника. Они специально созданы для решения проблемы CAP-теоремы.

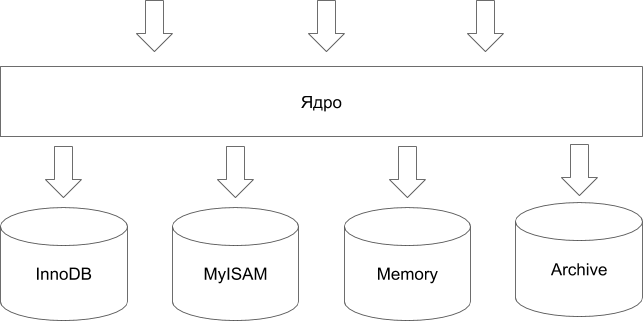
Укрупнение проектов, распределенная структура современных веб-решений приводит к тому, что требуются новые решения, новые базы данных, которые, обеспечивая меньшую согласованность, остаются устойчивыми в разделению.

Именно этим вызван интерес к современным NoSQL-базам данных, которые компенсируют слабую приспособленность реляционных баз данных к распределенной работе.

В том числе поэтому в современных приложениях традиционную реляционную СУБД зачастую сопровождают одна или несколько NoSQL-баз данных. Реляционная СУБД выступает в качестве основного хранилища, NoSQL-базы данных решают специализированные задачи или обеспечивают сетевую связность в условиях нестабильной сети.

**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.

**Клиенты MySQL**

Клиентами MySQL-сервера могут выступать консольные утилиты, специализированные программы с графическим интерфейсом и, конечно же, программный код.

Например, перед вами бесплатная программа DBeaver, которая позволяет подключаться к различным базам данных и доступен в основных операционных системах (Windows, Mac OS X, Linux).

Однако в течение курса мы будем использовать в основном консольный клиент MySQL, прибегая к графическому интерфейсу только для удобства просмотра объемных выборок.

Дело в том, что именно с консольными клиентами вам чаще придется иметь дело на удаленных серверах, доступ к которым может быть ограничен, например, подсетью дата-центра или локальной машиной.

Владея командной строкой, вы легко сможете освоить любой графический клиент, однако в обратную сторону это не работает. Навигация по базам данных, таблицам, системным параметрам в консольных утилитах требует ввода команд. В графических интерфейсах эти команды скрыты за выбором раскрывающихся и обычных списков, прокруткой.

Кроме того, консольные утилиты — зачастую самый быстрый способ выполнения больших пакетных операций. Например, у вас имеется база данных на два десятка гигабайт и вам необходимо снять резервную копию или, наоборот, развернуть ее на другом сервере. Зачастую консольные утилиты из дистрибутива **mysql** — это самый быстрый способ выполнения задачи.

**Консольный клиент MySQL**

Клиент доступен из командной строки при помощи утилиты **mysql**. В Windows для установки необходимо отметить соответствующую галку при установке MySQL или прописать путь к bin-папке MySQL в переменной окружения PATH. Как вариант, просто найти папку **bin** из дистрибутива **mysql**, перейти в нее и запустить MySQL прямо в ней.

Для соединения с сервером мы набираем название клиента MySQL, далее после ключа **-u** указываем имя MySQL-пользователя и ключ **-p**, чтобы клиент запросил у нас пароль. Сразу после установки пользователь, как правило, один — **root**. Поэтому для получения доступа к серверу достаточно набрать команду:

|  |
| --- |
| mysql -u root -p |

|  |
| --- |
|  |

Итак, у нас выводится приглашение **mysql>**, после которого можно набирать команды, адресованные серверу. Например, выведем фразу Hello world!:

|  |
| --- |
| SELECT "Hello world!"; |

Команды завершаются точкой с запятой. Если команда не убирается на одной строке, возможен переход на другую строку после нажатия клавиши Enter — запрос отправляется серверу только после того, как консольный клиент MySQL встретит символ точки с запятой.

Обратите внимание на результирующую таблицу. В первой строке таблицы с результатами содержатся заголовки столбцов, а в следующих строках — ответ сервера на запрос. Обычно заголовками столбцов становятся имена, полученные из таблиц базы. Если же извлекается не столбец таблицы, а значение выражения (как это происходит в приведенном выше примере), MySQL дает столбцу имя запрашиваемого выражения. После этого сообщается количество возвращаемых строк (1 row in set — одна строка в результате) и время выполнения запроса.

Для ввода ключевых слов можно использовать любой регистр символов, например, **select** можно набрать маленькими буквами:

|  |
| --- |
| select "Hello world!"; |

Если запрос не завершен, вид командной строки после ввода меняется. Таким образом, программа MySQL показывает, что завершенного выражения она пока что не получила и ожидает его полного ввода.

|  |
| --- |
| mysql> SELECT "Hello     "> wold!"; |

Введенные ранее команды не обязательно вводить снова, для этого достаточно их вызвать клавишами <↑> и <↓> (стрелка вверх и стрелка вниз).

Если сервер установлен на удаленном хосте, соединиться с ним можно, указав IP-адрес или домен удаленного сервера в параметре **-h**, в параметре **-P** мы можем указать порт.

|  |
| --- |
| mysql -u root -h 192.168.0.10 -P 3306 |

**Внутренние команды клиента MySQL**

Утилита **mysql** поддерживает различные команды, представленные в таблице ниже.

|  |  |  |
| --- | --- | --- |
| **Команда** | **Сокращение** | **Описание** |
| USE | \u | Выбор базы данных |
| SOURCE | \. | Выполнение SQL-команды из файла |
| SYSTEM | \! | Выполнение команды операционной системы |
| STATUS | \s | Вывод информации о состоянии сервера |
| EXIT | \q | Выход |
|  | \G | Вывод результата в вертикальном формате |

Иногда результирующие таблицы содержат слишком большое число столбцов, в результате их структура нарушается и воспринимать информацию практически невозможно.

Да, в этом случае удобно воспользоваться графическим клиентом, но это не всегда возможно. Выход из ситуации — использование так называемого вертикального режима вывода.

|  |
| --- |
| SELECT \* FROM mysql.user LIMIT 1\G |

В таком режиме каждая строка таблицы выводится при помощи отдельной таблицы, в первом столбце которой перечисляются имена столбцов, а во втором — их значения для данной строки.

Для этого используется внутренняя команда клиента **\G**. Повторимся: это команда клиента — то есть не часть SQL-запроса, сервер о ней ничего не знает. Поэтому отправлять ее из других клиентов или из собственного программного кода не следует — запрос будет выполнен с ошибкой.

Давайте посмотрим, как работают другие команды клиента MySQL. Например, получить текущий статус сервера можно при помощи команды **STATUS**. У этой команды есть сокращенный вариант **\s**.

Выполнить системную команду можно при помощи команды **SYSTEM**. Ниже мы запрашиваем список файлов в текущей директории:

|  |
| --- |
| mysql> SYSTEM |

В Windows команда **SYSTEM** в MySQL-клиенте не поддерживается. Для команды **SYSTEM** существует сокращение **\!**.

Точно такая же ситуация с командой выхода **EXIT**: мы можем использовать вместо нее сокращение **\q**.

Некоторые SQL-команды довольно длинные и в них легко ошибиться. Повторный набор может быть довольно утомителен и неудобен.

Вместо этого команды можно набирать в файлах, используя для этого любой текстовый редактор. После того, как SQL-файл сформирован, его можно выполнить при помощи команды **SOURCE**.

Давайте создадим SQL-запрос, выводящий фразу Hello world! (например, в редакторе DBeaver):

Пример: **hello.sql**.

|  |
| --- |
| SELECT "Hello world!"; |

Давайте сохраним файл с именем **hello.sql** в какой либо папке. После этого в директории с файлом можно запустить MySQL-клиент в диалоговом режиме и выполнить файл при помощи команды **SOURCE**:

|  |
| --- |
| mysql> SOURCE hello.sql |

Таким образом, если вы совершите ошибку в запросе, его можно поправить в любом текстовом редакторе и выполнить повторно.

**Конфигурационный файл .my.cnf**

При входе в диалоговый режим утилиты **mysql** мы каждый раз вынуждены вводить параметры. Если этого не делать, то попытка входа завершится неудачей. Чтобы каждый раз не вводить параметры, можно воспользоваться специальным конфигурационным файлом **.my.cnf**. Этот файл обычно размещается в домашней директории пользователя (это касается их Windows и UNIX-подобных операционных систем).

Файл начинается с точки, в UNIX-системах ведущей точкой помечаются скрытые файлы. Так как MySQL изначально разрабатывалась в UNIX, эта традиция распространяется в том числе на Windows-пользователей.

При запуске любой консольной утилиты из дистрибутива MySQL, если параметры не задаются явно, они будут извлекаться из этого файла.

Пример: **.my.cnf**.

|  |
| --- |
| [mysql] user=root password= |

Мы указываем параметры в полной форме, а не в сокращенной, как при запуске консольного клиента. Если вы указываете здесь настоящий пароль, важно назначить потом файлу права для чтения только вашему пользователю.

После того, как файл готов, вы получаете возможность запуска MySQL-клиента без указания логина и пароля:

|  |
| --- |
| mysql |

**Создание дампа**

Одна из часто встречающихся задач — перенос баз данных с одного сервера на другой или просто создание резервной копии базы данных.

Основным инструментом для создания SQL-дампов служит утилита **mysqldump**. Утилита создает текстовый файл с SQL-инструкциями, воспроизведя которые на другом сервере, вы сможете полностью воссоздать базу данных.

Конфигурационный файл **.my.cnf** будет действовать и на эту утилиту. Для этого секцию **[mysql]** следует поменять на **[client]**. После этого директивы, которые указаны в файле **.my.cnf**, будут распространяться в том числе и на **mysqldump**.

Как правило, для создания резервной копии в качестве параметра утилите **mysqldump** указывается какая-то база данных.

У нас пока нет своих собственных баз данных — это тема следующего ролика, но мы можем сделать дамп системной базы данных **mysql**, которую **MySQL** использует для хранения собственных настроек, имен пользователей, хранимых процедур и т. д.

|  |
| --- |
| mysqldump mysql > mysql.sql |

Если мы захотим развернуть дамп, то должны уже воспользоваться утилитой **mysql**. Мы не будем разворачивать дамп системной базы данных, чтобы ничего не повредить, вместо этого мы воспользуемся безобидным файлом **hello.sql**, который не вносит изменений.

|  |
| --- |
| mysql < hello.sql |

**Управление базами данных**

Создание базы данных средствами SQL осуществляется при помощи оператора **CREATE DATABASE**. Давайте создадим базу данных **shop**.

|  |
| --- |
| mysql> CREATE DATABASE shop; |

Чтобы посмотреть список существующих баз данных, мы можем воспользоваться оператором.

|  |
| --- |
| mysql> SHOW DATABASES; |

Как видим, возвращается список баз данных. Причем помимо нашей базы данных **shop**, можно видеть уже существующие системные базы данных. С базой данных **mysql** мы уже знакомы. Базы данных **performance\_schema** и **sys** предназначены для исследования параметров производительности. Информационная схема **information\_schema** предоставляется СУБД для описания структуры данных, мы немного затронем ее в текущем ролике.

База данных в MySQL — это подкаталог на жестком диске, который можно физически обнаружить в каталоге данных. Это каталог операционной системы, где физически располагаются базы данных, такие как таблицы, логи, журналы транзакций и т. п.

Узнать, где он расположен, можно при помощи запроса:

|  |
| --- |
| SHOW VARIABLES LIKE 'datadir'; |

Давайте выйдем из клиента **mysql** и перейдем внутрь каталога (у вас путь может отличаться).

|  |
| --- |
| cd /usr/local/var/mysql/ |

Здесь находится множество файлов, со многими из них мы с вами познакомимся во время курса. Сейчас нас будут интересовать директории. Каждая директория здесь — база данных, в том числе и созданная нами директория **shop**.

Здесь нет только одной базы данных, соответствующей информационной схеме **information\_schema**, так как это виртуальная база данных, которая формируется на лету.

Давайте заглянем внутрь нашей базы данных **shop**. Для этого переходим внутрь каталога.

|  |
| --- |
| cd shop |

Там мы можем обнаружить единственный файл **db.opt**, если мы заглянем внутрь, то увидим, что это текстовый файл, содержащий сведения о кодировке по умолчанию.

В курсе будем работать с единственной кодировкой — UTF8.Именно она по умолчанию выставляется в последних версиях MySQL, поэтому подробно на кодировках останавливаться не будем.

Если база данных — это подкаталог в каталоге данных, то мы можем создать его и руками. Давайте это сделаем. Для этого можно воспользоваться UNIX-командой **cp** и параметром **-r** для рекурсивного копирования всего содержимого директории.

Вы можете воспользоваться любым способом копирования, в том числе и графическим интерфейсом вашей операционной системы.

|  |
| --- |
| cp -r shop foo |

Итак, мы создали новую базу данных **foo** путем копирования ее из базы данных **shop**. Давайте вернем в консоль **mysql** и убедимся, что у нас появилась новая база данных. Запускаем команду **SHOW DATABASE**, и видим новую базу данных **foo**.

|  |
| --- |
| mysql> SHOW DATABASES; |

Разумеется, не следует прибегать к такому приему на практике, так как в разных СУБД организация баз данных сильно отличается. Прием создания базы данных через формирования директории не сработает у вас в другой базе данных. А вот SQL-запрос **CREATE DATABASE** стандартен и будет работать во всех базах данных. Кроме того, варварский способ копирования директории сработает только в отношении пустой базы данных. Если вы поступите так в отношении таблиц, особенно при работающем сервере, вы почти наверняка их повредите. Для копирования баз данных лучше прибегать либо к SQL-дампам, либо к специализированным утилитам.

Удаление баз данных можно осуществить и штатными средствами при помощи оператора **DROP DATABASE**, за которым следует имя базы данных:

|  |
| --- |
| DROP DATABASE foo; SHOW DATABASES; |

Кстати, так как имя базы данных — это имя каталога, чувствительность к регистру определяется конкретной операционной системой. Так, операционной системе Windows имена с заглавной и прописной буквы будут обозначать одну и ту же базу данных, в то время как в UNIX-подобной операционной системе это будут две разные базы данных.

При попытке создания уже существующей базы данных возвращается ошибка:

|  |
| --- |
| CREATE DATABASE shop;  ERROR 1007 (HY000): Can't create database 'shop'; database exists |

Часто такое поведение нежелательно, особенно при работе с объёмными SQL-дампами, которые выполняются в пакетном режиме. Для предотвращения такой ошибки оператор **CREATE DATABASE** можно снабдить конструкцией **IF NOT EXISTS**. Она помогает создать базу данных, если она еще не существует, если же существует — никаких действий не производится.

|  |
| --- |
| DROP DATABASE shop; Query OK, 0 rows affected (0.00 sec) CREATE DATABASE IF NOT EXISTS shop; Query OK, 1 row affected (0.00 sec) CREATE DATABASE IF NOT EXISTS shop; Query OK, 0 rows affected (0.00 sec) |

Как видно, после удаления базы данных **shop** первый оператор **CREATE DATABAS**E возвращает ответ, в котором сообщается, что было произведено одно действие (1 row affected), т. е., база данных создается.

Второй оператор **CREATE DATABASE** возвращает сообщение, что запрос не произвел ни одной операции (0 rows affected), т. е., база данных не создается, однако и ошибка не выдается.

Точно такой же механизм разработан для оператора **DROP DATABASE**: добавление ключевого слова **IF EXISTS** удаляет базу данных, если она существует, и не производит никаких действий, если база данных отсутствует.

|  |
| --- |
| DROP DATABASE IF EXISTS shop; Query OK, 1 rows affected (0.03 sec) DROP DATABASE IF EXISTS shop; Query OK, 0 rows affected (0.00 sec) |

Приступая к работе с базами данных, следует выбрать БД по умолчанию, к таблицам которой будет осуществляться обращение. Каждая клиентская программа решает эту задачу по-своему. Например, в консольном клиенте **mysql** выбрать новую базу данных можно при помощи команды **USE**.

|  |
| --- |
| USE shop |

Если текущая база данных не выбрана, то обращение по умолчанию будет заканчиваться сообщением об ошибке — **No database selected** (База данных не выбрана).

|  |
| --- |
| SHOW TABLES; ERROR 1046 (3D000): No database selected |

Можно не выбирать текущую базу данных, однако в этом случае во всех операторах придётся явно указывать, какая база данных должна использоваться.

|  |
| --- |
| SHOW TABLES FROM mysql; |

При обращении к таблицам потребуется явное указание префикса базы данных. Это правило действует и в отношении таблиц:

|  |
| --- |
| SELECT mysql.User.User, mysql.User.Host FROM mysql.User; |

Здесь мы из таблицы **User** базы данных **mysql** извлекаем поля **User** и **Host**. Такие составные имена называются полными квалификационными именами столбцов и таблиц. В результате явного обращения к базам данных, SQL-запросы становятся достаточно громоздкими и менее гибкими, так как смена имени базы данных требует изменения SQL-запроса.

Другой способ выбрать базу данных при использовании консольного клиента **mysql** — указание имени базы данных в конце списка параметров. Давайте при запуске утилиты **mysql** в качестве текущей базы данных выберем **shop**.

|  |
| --- |
| mysql -u root shop |

**Создание таблиц**

Создать таблицу внутри базы данных можно при помощи оператора **CREATE TABLE**. Мы еще будем более подробно рассматривать его в следующих роликах.

|  |
| --- |
| CREATE [TEMPORARY] TABLE [IF NOT EXISTS] users [(create\_definition,...)] [table\_options] [select\_statement] |

Начинается оператор ключевым словом **CREATE TABLE**, за которым следует имя таблицы, а в круглых скобках — параметры столбцов:

|  |
| --- |
| CREATE TABLE users (k INT); Query OK, 0 rows affected (0.11 sec) CREATE TABLE users (k INT); ERROR 1050: Table 'users' already exists |

Повторное создание таблицы приводит к возникновению ошибки «Таблица users уже существует». Однако необязательная конструкция **IF NOT EXISTS** сообщает, что таблицу необходимо создавать только в том случае, если она не существует, в противном случае никаких действий предпринимать не следует.

|  |
| --- |
| CREATE TABLE IF NOT EXISTS users (k INT); Query OK, 0 rows affected (0.02 sec) |

Такое поведение базы данных бывает удобным при выполнении большого количества SQL-инструкций в пакетном режиме, например из файла. Ошибка вызовет остановку выполнения всего пакетного файла, несмотря на то, что дальнейшее его выполнение в такой ситуации не приведет к сбою.

Для просмотра структуры таблицы можно использовать оператор **DESCRIBE**.

|  |
| --- |
| DESCRIBE users; |

Каждая строка в результирующей таблице соответствует отдельному столбцу.

Вместо имени столбца можно использовать LIKE-шаблон c применением спецсимволов **\_** (заменяет собой любой один символ) и **%** (заменяет собой любое количество символов).

|  |
| --- |
| DESCRIBE users 'us\_r'; |

Ниже приводится запрос, который выводит структура столбцов, имена которых заканчиваются на символ **'e'**. Следует обратить внимание, что для использования шаблона строку необходимо заключать в одинарные кавычки, в противном случае сервер MySQL сообщит об ошибочном SQL-запросе.

|  |
| --- |
| DESCRIBE users '%e'; |

Однако, следует помнить, что это не стандартный оператор, другие базы данных их не реализуют и его возможности довольно ограничены.

**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** (более подробно мы будем его изучать в следующих роликах).

|  |
| --- |
| 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'; |

Информационной схеме будет посвящен отдельный ролик.

**Документация**

Клиент **mysql** поддерживает команду **HELP**, при помощи которой можно ознакомиться с документацией прямо в консоли.

|  |
| --- |
| HELP DESCRIBE; |

Однако гораздо удобнее воспользоваться онлайн-документацией на официальном сайте MySQL: https://dev.mysql.com/doc/refman/5.7/en/.

Помимо онлайн-варианта, можно загрузить документацию в виде PDF-файла или zip-архива.

**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, 2012. — 624 с.
3. Чаллавала Ш., Лакхатария Дж., Мехта Ч., Патель К. MySQL 8 для больших данных. — М.: ДМК Пресс, 2018. — 226с.
4. Поль Дюбуа. MySQL. — Пер. с англ. — М.: ООО "И.Д. Вильямс", 2007. — 1168 с.
5. Поль Дюбуа. MySQL. Сборник рецептов. — Пер. с англ. — М.: Символ-Плюс, 2004. — 1056 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. 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>.

**Урок 2**

Язык запросов SQL

[Введение в SQL](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.mwkmjms8qcg1)

[Достоинства SQL](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.slrqtck6jxu9)

[Недостатки SQL](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.1t3h5sf)

[Элементы языка](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.3tb0lm9pg2u0)

[Комментарии](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.h727mdfurd3m)

[DDL и DML](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.wewqj08qo8nf)

[Структура запроса](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.g2mfcb94hoa1)

[Скалярные выражения](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.dabm1dkdiptc)

[Типы данных](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.4d34og8)

[Атрибуты](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.r87za5j1xbqi)

[Числовые типы](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.vn0q8dl8qx5t)

[Строковые типы](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.fdieubk5eicd)

[Учебная база данных](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.oei09blcrdb6)

[Тип NULL](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.shvwmhf2ei4m)

[Календарные типы](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.80lafunesuh0)

[Коллекционные типы](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.far89q1l7sw7)

[Индексы и ключи](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.1n6uf3xhsoml)

[Виды индексов](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.r02euslnrkbx)

[Устройство индекса](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.vn8fyc9csx7c)

[Введение в CRUD-операции](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.7dd8nuypf6ox)

[Используемые источники](https://docs.google.com/document/d/1Os4W1-eGgSAgF2CXzImHbTkVuaK-zxELLh7FMR83MC0/edit#heading=h.2xcytpi)

# Введение в SQL

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

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

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

## Достоинства SQL

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

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

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

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

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

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

## Недостатки SQL

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

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

SQL плохо ложится на иерархическую и графовую природу объектно-ориентированного программирования, составляющего львиную долю современной разработки. В связи с этим большую популярность стали приобретать различные ORM-системы, которые предоставляют ООП-интерфейс, скрывая за объектами и методами библиотек реляционную природу базы данных.

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

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

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

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

# Элементы языка

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

## Комментарии

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

|  |
| --- |
| -- SELECT "Hello world!"; |

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

|  |
| --- |
| /\* Это комментарий SELECT "Hello world!"; \*/ |

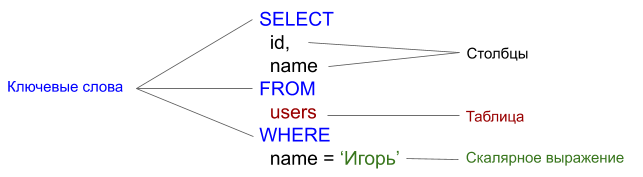
## DDL и DML

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

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

## Структура запроса

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



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

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

## Скалярные выражения

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

|  |
| --- |
| SELECT "Hello world!"; |

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

|  |
| --- |
| SELECT 2 + 2; |

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

|  |
| --- |
| SELECT 'Rubist's world'; |

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

|  |
| --- |
| SELECT 'Rubist\'s world'; |

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

|  |
| --- |
| SELECT "Rubist's world"; |

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

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

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

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

# Типы данных

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

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

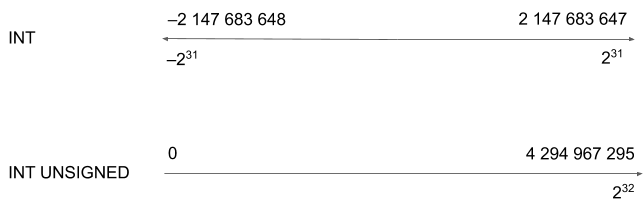
## Атрибуты

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

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

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

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

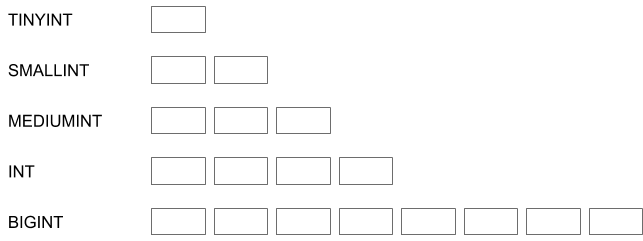


## Числовые типы

Числовые типы можно разделить на:

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

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



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

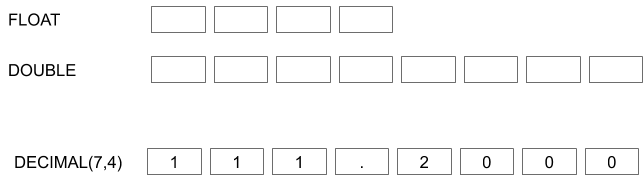
В **TINYINT** один байт, т. е., 8 бит, максимальное значение, которое он может обслуживать — от 0 до 2 в степени 8, т. е., 256. Большое число в поле этого типа поместить не получится. Если при этом не используется атрибут **UNSIGNED**, то эту величину следует поделить пополам и допустимый диапазон —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

|  |
| --- |
| CREATE TABLE tbl (id INT(8)); |

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

|  |
| --- |
| INSERT INTO tbl VALUES (5); SELECT \* FROM tbl;  DROP TABLE IF EXISTS tbl; CREATE TABLE tbl (id INT(8) ZEROFILL); INSERT INTO tbl VALUES (5); INSERT INTO tbl VALUES (500000000); |

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



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

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

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

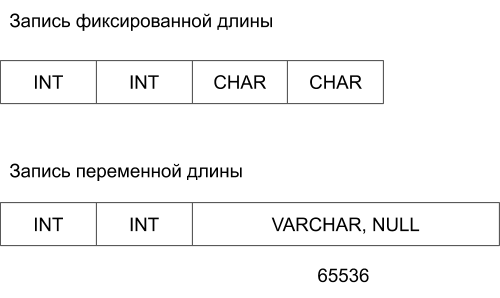
|  |
| --- |
| DROP TABLE IF EXISTS tbl; CREATE TABLE tbl (price DECIMAL(8,4)); INSERT INTO tbl VALUES (111.2); INSERT INTO tbl VALUES (10000.0); ERROR 1264 (22003): Out of range value for column 'price' at row 1 |

## Строковые типы

Строковые типы можно условно разделить на:

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

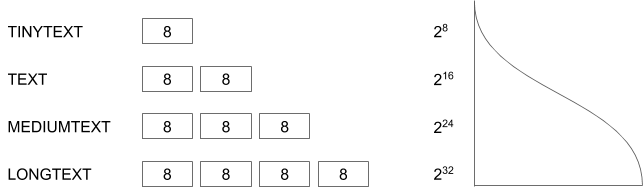
|  |
| --- |
| DROP TABLE IF EXISTS tbl; CREATE TABLE tbl (   name CHAR(10) DEFAULT 'anonymous',   description VARCHAR(255) ); INSERT INTO tbl VALUES(DEFAULT, 'Новый пользователь'); SELECT \* FROM tbl; INSERT INTO tbl (description) VALUES('Еще один пользователь'); INSERT INTO tbl VALUES('Очень длинное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



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

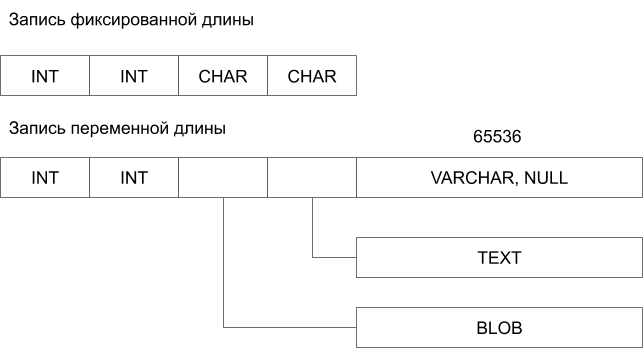
Так как для кодирования строк используется UTF-8, ситуация еще более печальная: для символов, отличных от английских, зачастую используется больше одного байта, например русский текст кодируется двумя байтами.

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



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

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



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

# Учебная база данных

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

|  |
| --- |
| DROP TABLE IF EXISTS catalogs; CREATE TABLE catalogs (   id INT UNSIGNED,   name VARCHAR(255) COMMENT 'Название раздела' ) COMMENT = 'Разделы интернет-магазина';  DROP TABLE IF EXISTS users; CREATE TABLE users (   id INT UNSIGNED,   name VARCHAR(255) COMMENT 'Имя покупателя' ) COMMENT = 'Покупатели';  DROP TABLE IF EXISTS products; CREATE TABLE products (   id INT UNSIGNED,   name VARCHAR(255) COMMENT 'Название',   desription TEXT COMMENT 'Описание',   price DECIMAL (11,2) COMMENT 'Цена',   catalog\_id INT UNSIGNED ) COMMENT = 'Товарные позиции';  DROP TABLE IF EXISTS orders; CREATE TABLE orders (   id INT UNSIGNED,   user\_id INT UNSIGNED ) COMMENT = 'Заказы';  DROP TABLE IF EXISTS orders\_products; CREATE TABLE orders\_products (   id INT UNSIGNED,   order\_id INT UNSIGNED,   product\_id INT UNSIGNED,   total INT UNSIGNED DEFAULT 1 COMMENT 'Количество заказанных товарных позиций' ) COMMENT = 'Состав заказа';  DROP TABLE IF EXISTS discounts; CREATE TABLE discounts (   id INT UNSIGNED,   user\_id INT UNSIGNED,   product\_id INT UNSIGNED,   discount FLOAT UNSIGNED COMMENT 'Величина скидки от 0.0 до 1.0' ) COMMENT = 'Скидки';  DROP TABLE IF EXISTS storehouses; CREATE TABLE storehouses (   id INT UNSIGNED,   name VARCHAR(255) COMMENT 'Название' ) COMMENT = 'Склады';  DROP TABLE IF EXISTS storehouses\_products; CREATE TABLE storehouses\_products (   id INT UNSIGNED,   storehouse\_id INT UNSIGNED,   product\_id INT UNSIGNED,   value INT UNSIGNED COMMENT 'Запас товарной позиции на складе' ) COMMENT = 'Запасы на складе'; |

# Тип NULL

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

|  |
| --- |
| SELECT NULL; |

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

|  |
| --- |
| SELECT NULL + 2; |

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

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

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

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

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

|  |
| --- |
| TRUNCATE tbl; |

Повторяем вызов **ALTER TABLE**:

|  |
| --- |
| ALTER TABLE tbl CHANGE id id INT UNSIGNED NOT NULL; |

И смотрим структуру таблицы:

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

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

|  |
| --- |
| INSERT INTO tbl VALUES(); |

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

# Календарные типы

MySQL поддерживает пять типов календарных типов:

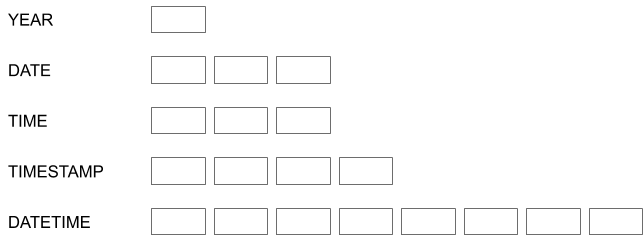
* **TIME** предназначен для хранения времени в течение суток;
* **YEAR** хранит год;
* **DATE** хранит дату с точностью до дня;
* **DATETIME** хранит дату и время;
* **TIMESTAMP** также хранит дату и время, занимает в два раза меньше места, чем **DATETIME**, но может хранить только ограниченные даты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

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

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

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

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



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

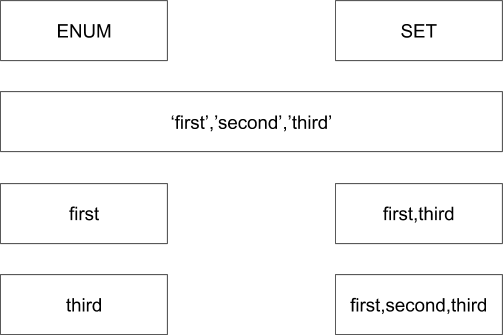
|  |
| --- |
| SELECT '2018-10-01 0:00:00'; |

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

|  |
| --- |
| SELECT '2018-10-01 0:00:00' - INTERVAL 1 DAY; SELECT '2018-10-01 0:00:00' + INTERVAL 1 WEEK; SELECT '2018-10-01 0:00:00' + INTERVAL 1 YEAR; SELECT '2018-10-01 0:00:00' + INTERVAL '1-1' YEAR\_MONTH; |

# Коллекционные типы

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



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

|  |
| --- |
| DESCRIBE tbl; ALTER TABLE tbl ADD collect JSON; DESCRIBE tbl; |

Давайте вставим в это поле JSON-объект:

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

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

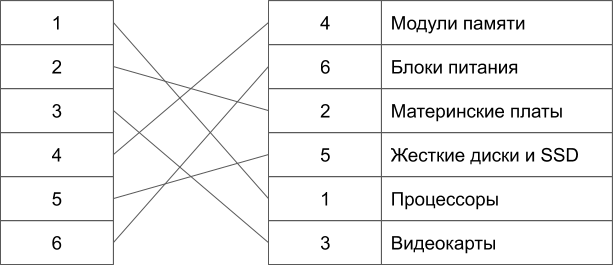
|  |
| --- |
| DROP TABLE IF EXISTS catalogs; CREATE TABLE catalogs (   id INT UNSIGNED NOT NULL,   name VARCHAR(255) COMMENT 'Название раздела' ) COMMENT = 'Разделы интернет магазина';  DROP TABLE IF EXISTS users; CREATE TABLE users (   id INT UNSIGNED NOT NULL,   name VARCHAR(255) COMMENT 'Имя покупателя',   birthday\_at DATE COMMENT 'Дата рождения',   created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,   updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP ) COMMENT = 'Покупатели';  DROP TABLE IF EXISTS products; CREATE TABLE products (   id INT UNSIGNED NOT NULL,   name VARCHAR(255) COMMENT 'Название',   desription TEXT COMMENT 'Описание',   price DECIMAL (11,2) COMMENT 'Цена',   catalog\_id INT UNSIGNED,   created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,   updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP ) COMMENT = 'Товарные позиции';  DROP TABLE IF EXISTS orders; CREATE TABLE orders (   id INT UNSIGNED NOT NULL,   user\_id INT UNSIGNED,   created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,   updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP ) COMMENT = 'Заказы';  DROP TABLE IF EXISTS orders\_products; CREATE TABLE orders\_products (   id INT UNSIGNED NOT NULL,   order\_id INT UNSIGNED,   product\_id INT UNSIGNED,   total INT UNSIGNED DEFAULT 1 COMMENT 'Количество заказанных товарных позиций',   created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,   updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP ) COMMENT = 'Состав заказа';  DROP TABLE IF EXISTS discounts; CREATE TABLE discounts (   id INT UNSIGNED NOT NULL,   user\_id INT UNSIGNED,   product\_id INT UNSIGNED,   discount FLOAT UNSIGNED COMMENT 'Величина скидки от 0.0 до 1.0',   finished\_at DATETIME NULL,   started\_at DATETIME NULL,   created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,   updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP ) COMMENT = 'Скидки';  DROP TABLE IF EXISTS storehouses; CREATE TABLE storehouses (   id INT UNSIGNED NOT NULL,   name VARCHAR(255) COMMENT 'Название',   created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,   updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP ) COMMENT = 'Склады';  DROP TABLE IF EXISTS storehouses\_products; CREATE TABLE storehouses\_products (   id INT UNSIGNED NOT NULL,   storehouse\_id INT UNSIGNED,   product\_id INT UNSIGNED,   value INT UNSIGNED COMMENT 'Запас товарной позиции на складе',   created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,   updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP ) COMMENT = 'Запасы на складе'; |

# Индексы и ключи

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

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

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



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

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

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

## Виды индексов

Существует несколько видов индексов:

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

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

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

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

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

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

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

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

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

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

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

|  |
| --- |
| DROP TABLE IF EXISTS catalogs; CREATE TABLE catalogs (   id INT UNSIGNED NOT NULL PRIMARY KEY AUTO\_INCREMENT,   name VARCHAR(255) COMMENT 'Название раздела' ) COMMENT = 'Разделы интернет магазина';  SELECT \* FROM catalogs; |

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

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

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

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

**SERIAL ==** BIGINT UNSIGNED NOT NULL AUTO\_INCREMENT UNIQUE

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

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

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

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

|  |
| --- |
| DROP TABLE IF EXISTS products; CREATE TABLE products (   id SERIAL PRIMARY KEY,   name VARCHAR(255) COMMENT 'Название',   desription TEXT COMMENT 'Описание',   price DECIMAL (11,2) COMMENT 'Цена',   catalog\_id INT UNSIGNED,   created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,   updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,   KEY index\_of\_catalog\_id(catalog\_id) ) COMMENT = 'Товарные позиции';  DESCRIBE products; |

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

|  |
| --- |
| DROP TABLE IF EXISTS products; CREATE TABLE products (   id SERIAL PRIMARY KEY,   name VARCHAR(255) COMMENT 'Название',   desription TEXT COMMENT 'Описание',   price DECIMAL (11,2) COMMENT 'Цена',   catalog\_id INT UNSIGNED,   created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,   updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP ) COMMENT = 'Товарные позиции'; CREATE INDEX index\_of\_catalog\_id ON products (catalog\_id); |

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

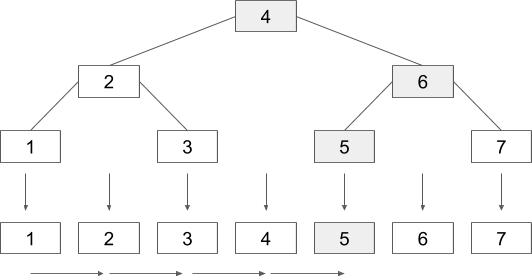
|  |
| --- |
| DROP INDEX index\_of\_catalog\_id ON products; |

## Устройство индекса

MySQL поддерживает два типа индекса:

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

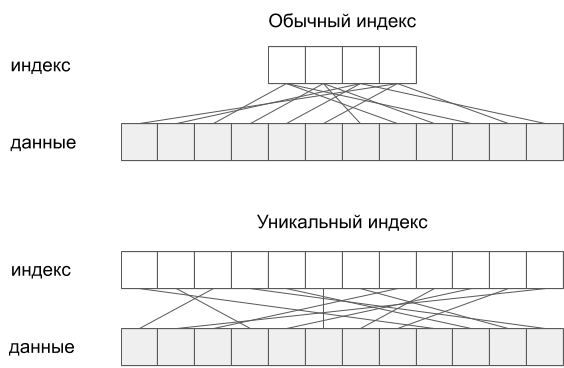
Общая идея бинарного дерева в том, что значения хранятся по порядку, и все листовые страницы находятся на одинаковом расстоянии от корня. BTREE-индекс ускоряет доступ к данным, поскольку подсистеме хранения не нужно сканировать всю таблицу для поиска нужной информации. В BTREE-индексах индексированные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



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

|  |
| --- |
| CREATE INDEX index\_of\_catalog\_id USING BTREE ON products (catalog\_id); CREATE INDEX index\_of\_catalog\_id USING HASH ON products (catalog\_id); DESCRIBE products; |

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



Селективность индекса — это отношение количества проиндексированных значений к общему количеству строк в таблице. Индекс с высокой селективностью хорош тем, что позволяет MySQL при поиске соответствий отфильтровывать больше строк. Уникальный индекс имеет селективность, равную единице.

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

Давайте поправим таблицы учебной базы данных:

|  |
| --- |
| CREATE TABLE products (   id SERIAL PRIMARY KEY,    name VARCHAR(255) COMMENT 'Название',   desription TEXT COMMENT 'Описание',   price DECIMAL (11,2) COMMENT 'Цена',   catalog\_id INT UNSIGNED,   created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,   updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,   KEY index\_of\_catalog\_id(catalog\_id) ) COMMENT = 'Товарные позиции'; DROP TABLE IF EXISTS orders; CREATE TABLE orders (   id SERIAL PRIMARY KEY,   user\_id INT UNSIGNED,   created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,   updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,   KEY index\_of\_user\_id(user\_id) ) COMMENT = 'Заказы'; |

В таблице **orders\_products** у нас два кандидата на индексацию: **order\_id** и **product\_id**. Порядок следования столбцов в индексе имеет значение:



В каждом запросе может использоваться ровно один индекс, если в запросе участвует два поля — **order\_id** и **product\_id** и индекс по двум столбцам сработает. Если у нас два отдельных запроса с участием **order\_id** и **product\_id**, то индекс **order\_id** и **product\_id** может использоваться в запросе с **order\_id**, но его не получится использовать в запросе с **product\_id**. В **discount**, с высокой долей вероятности, **user\_id** и **product\_id** будут использоваться раздельно. Давайте снабдим их индексами.

|  |
| --- |
| DROP TABLE IF EXISTS discounts; CREATE TABLE discounts (   id SERIAL PRIMARY KEY,    user\_id INT UNSIGNED,   product\_id INT UNSIGNED,   discount FLOAT UNSIGNED COMMENT 'Величина скидки от 0.0 до 1.0',   finished\_at DATETIME NULL,   started\_at DATETIME NULL,   created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,   updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,   KEY index\_of\_user\_id(user\_id),   KEY index\_of\_product\_id(product\_id) ) COMMENT = 'Скидки'; |

# Введение в CRUD-операции

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

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

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

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

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

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

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

Такой вариант вставки работает гораздо быстрее.

Для извлечения данных используется оператор **SELECT:**

|  |
| --- |
| SELECT id, name FROM catalogs; |

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

|  |
| --- |
| SELECT name, id FROM catalogs; |

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

|  |
| --- |
| SELECT name FROM catalogs; |

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

|  |
| --- |
| SELECT \* FROM catalogs; |

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

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

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

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

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

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

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

* команда **DELETE** удаляет все или часть записей из таблицы;
* команда **TRUNCATE** удаляет все записи из таблицы и обнуляет счетчики **AUTO\_INCREMENT**.

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

|  |
| --- |
| DELETE FROM catalogs; |

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

|  |
| --- |
| DELETE FROM catalogs LIMIT 2; |

Можем задавать и более сложные условия, например:

|  |
| --- |
| DELETE FROM catalogs WHERE id > 1 LIMIT 1; |

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

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

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

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

|  |
| --- |
| UPDATE catalogs SET name = 'Процессоры (Intel)' WHERE name = 'Процессоры'; SELECT \* FROM catalogs; |

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

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

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

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

|  |
| --- |
| INSERT INTO   cat SELECT   \* FROM   catalogs; SELECT \* FROM cat; |

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

# Используемые источники

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

**Урок 3**

Операторы, фильтрация, сортировка и ограничение

[Операторы](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.wibo77d8dmt9)

[Арифметические операторы](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.w6wiby2lrv2b)

[Операторы сравнения](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.v6xm5tfzblzw)

[Логические операторы](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.m7solcys8ifh)

[Вычисляемые столбцы](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.c84or328wnrq)

[Условная выборка](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.3tb0lm9pg2u0)

[Сортировка](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.cvv8zg68oxw2)

[Ограничения](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.4ps2c18giyyn)

[Уникальные значения](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.yodryo18kj6)

[Предопределенные функции](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.rvyfqb8w33n)

[Календарные функции](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.swaixf3w3hds)

[Случайное значение](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.jtly8tx2lbin)

[Информационные функции](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.jiyqdt8ctp59)

[Математические функции](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.katbev69fij4)

[Строковые функции](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.uk4hmxmaw5ak)

[Логические функции](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.3q3zocl0lbkn)

[Вспомогательные функции](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.ppvckwxvgtvd)

[Используемые источники](https://docs.google.com/document/d/1xIihJbSVT76Nu0pa1b1nlhh5nUd4QYQe0Vuh2C4LPJg/edit#heading=h.2xcytpi)

# Операторы

Под операторами подразумеваются конструкции языка, которые производят преобразование данных, например, операция сложения — **+**, вычитания — **-** и т. п. Данные, над которыми совершается операция, называются операндами.

## Арифметические операторы

К арифметическим операциям относятся сложение (**+**), вычитание (**-**), умножение (**\***),  деление (**/**). Кроме того, выделяют взятие остатка от деления, процент и целочисленное деление **DIV**.

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| 5 + 2 | Сложение |
| 5 - 2 | Вычитание |
| 2 \* 3 | Умножение |
| 2 / 3 | Деление |
| 9 % 3 | Остаток от деления |
| 10 DIV 3 | Целочисленное деление |

Оператор **+** хорошо нам известен из школьного курса:

|  |
| --- |
| SELECT 3 + 5; |

Для получения значения используется оператор **SELECT**. Результатом выступает таблица, состоящая из одного столбца и одной строки. Название столбца совпадает с вычисляемым значением: **3 + 5**. Однако при желании мы можем его переименовать при помощи ключевого слова **AS**:

|  |
| --- |
| SELECT 3 + 5 AS summ; |

Применять арифметические операторы можно не только к обычным числам, но и к столбцам.

|  |
| --- |
| SELECT \* FROM catalogs; UPDATE catalogs SET id = id + 10; SELECT \* FROM catalogs; |

Выше при помощи UPDATE-запроса значения идентификаторов во всех записях увеличивается на десять. Чаще всего такой прием используется для обновления счетчиков, календарных или денежных значений.

Важно отметить, что операция сложения числа с **NULL** снова дает **NULL**:

|  |
| --- |
| SELECT 3 + NULL; |

Такое поведение вполне оправдано. **NULL** обозначает данные, значение которых не определено, поэтому прибавление к такому значению числа приводит опять к неопределенному значению, т. е., к **NULL**.

Если в качестве слагаемых будут выступать строки, они будут автоматически приведены к числам. Результатом в этом случае также будет число:

|  |
| --- |
| SELECT '3' + '5'; |

При этом, если строка не может быть приведена к числу, она интерпретируется как 0:

|  |
| --- |
| SELECT 'abc' + 'dfe'; |

Операция вычитания имеет те же особенности и ограничения, что и операция сложения.

|  |
| --- |
| SELECT 8 - 3; |

Помимо бинарного оператора **-**, который производит вычитание, существует унарный оператор, который меняет знак операнда.

|  |
| --- |
| SELECT -7; |

В операции умножения также нет ничего особенного:

|  |
| --- |
| SELECT 2 \* 3; |

Однако при умножении следует иметь в виду, что очень легко выйти за допустимые границы типа:

|  |
| --- |
| SELECT 18014398509481984 \* 18014398509481984; ERROR 1690 (22003): BIGINT value is out of range in '(18014398509481984 \* 18014398509481984)' |

Если мы выходим за границы типа **BIGINT**, MySQL возвращает ошибку.

В отличие от других языков программирования, деление на ноль не вызывает ошибки синтаксиса и остановки вычислений. В качестве результата возвращается **NULL**.

|  |
| --- |
| SELECT 8 / 0; |

Кроме обычного деления, существует оператор целочисленного деления **DIV**.

|  |
| --- |
| SELECT 5 DIV 2, 5 / 2; |

Результат деления при помощи **DIV** является целочисленным. Дробная часть просто отбрасывается и округления результата не производится. Чтобы получить остаток от деления, необходимо воспользоваться оператором **%**.

|  |
| --- |
| SELECT 5 % 2; |

Результат запроса равен 1, т. к. без остатка на 2 делится только 4 (5 – 4 = 1). Оператор взятия остатка от деления **%** имеет еще две альтернативные формы написания:

* замена **%** на **MOD**;
* встроенная функция **MOD()**.

|  |
| --- |
| SELECT 5 % 2, 5 MOD 2; |

## Операторы сравнения

Как и в любом другом языке программирования, в SQL большое значение имеет логический тип, который может принимать два значения: истину (**TRUE**) или ложь (**FALSE**).

|  |
| --- |
| SELECT TRUE; SELECT FALSE; |

MySQL поддерживает константы **TRUE** и **FALSE**. Однако, точно так же как и **SERIAL**, эти константы являются псевдонимами для 1 и 0 типа **TINYINT**.

Чаще всего логические значения получаются при помощи операторов сравнения, представленных в следующей таблице:

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| > | Больше |
| >= | Больше равно |
| < | Меньше |
| <= | Меньше равно |
| = | Равно |
| !=, <> | Не равно |
| <=> | Безопасное сравнение |

Ниже представлено типичное использование операторов сравнения.

|  |
| --- |
| SELECT 2 > 3; SELECT 2 <= 3;  SELECT 2 = 2, 2 = 3;  SELECT 2 != 3, 2 <> 3; |

Можно использовать отрицание, используя оператор **NOT**.

|  |
| --- |
| SELECT NOT TRUE, NOT FALSE; SELECT ! TRUE, ! FALSE; SELECT NOT 2 != 3, NOT 2 <> 3; |

Сравнение с неопределенным значением **NULL** всегда возвращает **NULL** — неопределенное значение:

|  |
| --- |
| SELECT 2 = NULL, 2 != NULL; |

Впрочем, существует специальный оператор **НЛО <=>**, который позволяет безопасно сравнивать со значением **NULL**:

|  |
| --- |
| SELECT 2 <=> NULL, NULL <=> NULL; |

Более классические операторы сравнения с **NULL** — **IS NULL** и **IS NOT NULL**:

|  |
| --- |
| SELECT 2 IS NULL, 2 IS NOT NULL, NULL IS NULL, NULL IS NOT NULL; |

## Логические операторы

Условия можно комбинировать при помощи логического И:

|  |  |  |
| --- | --- | --- |
| AND | true | false |
| true | true | false |
| false | false | false |

Помимо логического И, поддерживается логическое ИЛИ:

|  |  |  |
| --- | --- | --- |
| OR | true | false |
| true | true | true |
| false | true | false |

# Вычисляемые столбцы

Выражения можно сохранять в виде столбца таблицы. Оператор CREATE TABLE допускает создание столбцов, значение которых автоматически создается как арифметическое выражение других столбцов:

|  |
| --- |
| DROP TABLE IF EXISTS tbl; CREATE TABLE tbl (   x INT,   y INT,   summ INT AS (x + y) ); INSERT INTO   tbl (x, y) VALUES   (1, 1), (5, 6), (11, 12); SELECT \* FROM tbl; |

По умолчанию значения в вычисляемом столбце не сохраняются на жесткий диск, они каждый раз вычисляются снова. Однако если добавить в определение столбца ключевое слово **STORED**, то такой столбец будет сохраняться на жесткий диск.

|  |
| --- |
| DROP TABLE IF EXISTS tbl; CREATE TABLE tbl (   x INT,   y INT,   summ INT AS (x + y) STORED ); INSERT INTO   tbl (x, y) VALUES   (1, 1), (5, 6), (11, 12); SELECT \* FROM tbl; |

# Условная выборка

Ситуация, когда требуется изменить количество выводимых строк, встречается гораздо чаще, чем ситуация, когда требуется изменить число и порядок выводимых столбцов.

Для ввода в SQL-запрос такого рода ограничений в операторе **SELECT** предназначено специальное ключевое слово **WHERE**, после которого следует логическое условие. Если запись удовлетворяет такому условию, она попадает в результат выборки, в противном случае такая запись отбрасывается.

|  |
| --- |
| SELECT \* FROM catalogs WHERE id\_catalog > 2; |

Условие может быть составным и объединяться при помощи логических операторов.

|  |
| --- |
| SELECT \* FROM catalogs WHERE id > 2 AND id <= 4; |

Для выборки записей из определенного интервала используется оператор **BETWEEN**. Ниже представлен запрос, полностью эквивалентный варианту выше с операторами сравнения.

|  |
| --- |
| SELECT \* FROM catalogs WHERE id\_catalog BETWEEN 3 AND 4; |

Существует конструкция, противоположная **BETWEEN**, — **NOT BETWEEN**, которая возвращает записи, не попадающие в интервал.

|  |
| --- |
| SELECT \* FROM catalogs WHERE id\_catalog NOT BETWEEN 3 AND 4; |

Иногда требуется извлечь записи, удовлетворяющие не диапазону, а списку, например, записи с **id** из списка (1,2,5). Для этого предназначена конструкция **IN**.

|  |
| --- |
| SELECT \* FROM catalogs WHERE id IN (1,2,5); |

Конструкция **IN** также возвращает **NULL**, если один из элементов списка равен **NULL**:

|  |
| --- |
| SELECT 2 IN (0,NULL,5,'wefwf'); SELECT 2 IN (2,NULL,5,'wefwf'); |

Конструкция **NOT IN** является противоположной оператору **IN** и возвращает **1** (истина), если проверяемое значение не входит в список, и **0** (ложь), если оно присутствует в списке.

|  |
| --- |
| SELECT \* FROM catalogs WHERE id NOT IN (1,2,5); |

Можно добавить отрицание и перед конструкцией NOT:

|  |
| --- |
| SELECT \* FROM catalogs WHERE NOT id IN (1,2,5); |

В конструкции **WHERE** могут использоваться не только числовые столбцы. Так ниже из таблицы **catalogs** извлекается запись, соответствующая элементу каталога «Процессоры».

|  |
| --- |
| SELECT \* FROM catalogs WHERE name = 'Процессоры'; |

Зачастую условную выборку с участием строк удобнее производить не при помощи оператора равенства **=**, а при помощи оператора **LIKE**, который позволяет использовать простейшие шаблоны. Оператор часто используется в конструкции **WHERE** и возвращает **1** (истину), если шаблон соответствует выражению, и **0** (ложь) в противном случае.

|  |
| --- |
| SELECT 'MySQL' LIKE 'MySQL'; |

Главное преимущество оператора LIKE перед **=** заключается в использовании спецсимволов.

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| % | Любое количество символов или их отсутствие |
| \_ | Ровно один символ |

|  |
| --- |
| SELECT 'Программист' LIKE 'Программ%'; SELECT 'Программа' LIKE 'Программ%'; SELECT 'Программ' LIKE 'Программ%'; |

Спецсимвол **%** заменяет собой любую последовательность символов. Поэтому шаблон «Программ%» удовлетворяет словам «Программист», «Программа» и «Программ» и будет удовлетворять любому слову, начинающемуся с выражения «Программ».

Спецсимвол **%** может быть размещен в любом месте шаблона, как в начале, так и в середине строки:

|  |
| --- |
| SELECT 'Программирование' LIKE 'П%е','Печенье' LIKE 'П%е'; SELECT 'Программирование' LIKE '%ние','Кодирование' LIKE '%ние'; |

Символ подчеркивания соответствует одному любому символу.

|  |
| --- |
| SELECT 'код' LIKE '\_\_\_','рот' LIKE '\_\_\_', 'абв' LIKE '\_\_\_'; |

Так, шаблон из трех знаков подчеркивания **\_\_\_** будет соответствовать любому слову, состоящему из трех символов: «код», «рот», «абв». Чтобы поместить в шаблон сами символы **%** и **\_** без их специальной интерпретации, необходимо экранировать их при помощи обратного слеша:

|  |
| --- |
| SELECT '15 %' LIKE '15 \%', 'my\_sql' LIKE 'my\\_sql'; SELECT '15' LIKE '15 \%', 'my sql' LIKE 'my\\_sql'; |

Рассмотрим использование оператора **LIKE** на примере таблицы catalogs. Извлечем имена каталогов, которые заканчиваются на символ **ы**.

|  |
| --- |
| INSERT INTO catalogs VALUES   (NULL, 'Процессоры'),   (NULL, 'Материнские платы'),   (NULL, 'Видеокарты'),   (NULL, 'Жесткие диски'),   (NULL, 'Оперативная память'); SELECT \* FROM catalogs WHERE name LIKE '%ы'; |

Оператор **NOT LIKE** противоположен по действию оператору **LIKE**:

|  |
| --- |
| SELECT \* FROM catalogs WHERE name NOT LIKE '%ы'; |

Очень часто условия связаны с календарными столбцами. Давайте заполним таблицу пользователей:

|  |
| --- |
| INSERT INTO users (name, birthday\_at) VALUES   ('Геннадий', '1990-10-05'),   ('Наталья', '1984-11-12'),   ('Александр', '1985-05-20'),   ('Сергей', '1988-02-14'),   ('Иван', '1998-01-12'),   ('Мария', '1992-08-29'); |

В таблице пользователей **users** есть поле **birthday\_at**, которое соответствует дню рождения пользователей. Извлечем пользователей, которые родились в 90-е годы:

|  |
| --- |
| SELECT   \* FROM   users WHERE   birthday\_at >= '1990-01-01' AND birthday\_at < '2000-01-01'; |

Точно так же, как и в случае числовых данных, допускается использование оператора **BETWEEN**:

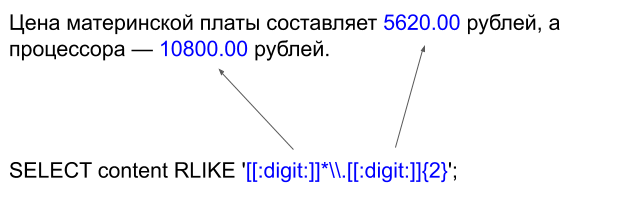
|  |
| --- |
| SELECT   \* FROM   users WHERE   birthday\_at BETWEEN '1990-01-01' AND '2000-01-01'; |

При использовании оператора **LIKE** календарный столбец автоматически преобразуется к строке, поэтому представленный запрос мы можем записать еще более коротким способом:

|  |
| --- |
| SELECT   \* FROM   users WHERE   birthday\_at LIKE '199%'; |

Оператор **RLIKE** (или его синоним **REGEXP**) позволяет производить поиск в соответствии с регулярными выражениями, которые предоставляют значительно более гибкие средства для поиска по сравнению с оператором **LIKE**. Обратной стороной медали является их более медленное выполнение.

Регулярные выражения — это специализированный язык поиска подстрок в тексте. Они оформляются в виде шаблона, который применяется к заданному тексту слева направо. Большая часть символов в таком шаблоне сохраняет свое значение, однако дополнительно вводятся символы, имеющие специальное значение: ограничители, классы, квантификаторы.



Слева от оператора **RLIKE** размещается текст, справа — регулярное выражение.

|  |
| --- |
| SELECT 'грамм' RLIKE 'грам', 'грампластинка' RLIKE 'грам'; |

Так, регулярное выражение, содержащее обычный текст, например «грам», соответствует строке, содержащей такую подстроку («грам»). Например, этому регулярному выражению будут соответствовать строки «программирование», «грамм», «грампластинка» и т. п.

Как видно, регулярное выражение «грам» осуществляет поиск по всему тексту, независимо от того, находится ли подстрока «грам» в начале, середине или конце слова.

Часто необходимо привязать регулярное выражение к началу слова, т. е., нужно, чтобы регулярное выражение «грам» соответствовало строке, начинающемуся с подстроки «грам», например, «грампластинка», но не соответствовало слову «программирование». Для этого используется символ **^**, соответствующий началу строки:

|  |
| --- |
| SELECT 'программирование' RLIKE '^грам', 'грампластинка' RLIKE '^грам'; |

Спецсимвол **$** позволяет привязать регулярное выражение к концу строки:

|  |
| --- |
| SELECT 'грампластинка' RLIKE '^грампластинка$'; |

Т. е., применение символов **^** и **$** позволяет указать, что регулярное выражение должно в точности соответствовать всей строке поиска от начала до конца.

|  |
| --- |
| SELECT 'грампластинка - это вам не программирование' RLIKE '^грампластинка$'; |

Символ вертикальной черты **|** применяется в регулярном выражении для задания альтернативных масок:

|  |
| --- |
| SELECT 'abc' RLIKE 'abc|абв', 'абв' RLIKE 'abc|абв'; |

Если шаблон должен включать символ **|** или любой другой, например рассмотренные выше **^** и **$**, то их необходимо экранировать при помощи двойного обратного слеша **\\** — в этом случае они теряют свое специальное значение и рассматриваются как обычные символы.

Для задания класса символов используются квадратные скобки, которые ограничивают поиск теми символами, которые в них заключены, например **[abc]**.

|  |
| --- |
| SELECT 'a' RLIKE '[abc]' AS a,        'b' RLIKE '[abc]' AS b,        'c' RLIKE '[abc]' AS c; |

Регулярному выражению **[abc]** соответствует подстрока, содержащая один символ: либо **a**, либо **b**, либо **c**.

Так, для создания регулярного выражения, соответствующего всем буквам русского алфавита, можно, конечно, перечислить все буквы в квадратных скобках. Это допустимо, но утомительно и неэлегантно. Более кратко такое регулярное выражение можно записать следующим образом:

|  |
| --- |
| '[а-яё]' |

Это выражение соответствует всем буквам русского алфавита, поскольку любые два символа, разделяемые дефисом, задают соответствие диапазону символов, находящихся между ними.

|  |
| --- |
| SELECT 'Л' RLIKE '[а-яё]' AS a; SELECT 'z' RLIKE '[а-яё]' AS a; |

Точно таким же образом задается регулярное выражение, соответствующее любому числу:

|  |
| --- |
| '[0-9]' |

Это выражение эквивалентно:

|  |
| --- |
| '[0123456789]' |

Кроме классов, которые могут создать разработчики, предусмотрены специальные конструкции классов:

|  |
| --- |
| SELECT '1' RLIKE '[[:digit:]]', 'а' RLIKE '[[:digit:]]'; SELECT '1' RLIKE '[[:alpha:]]', 'а' RLIKE '[[:alpha:]]'; |

Все, что находится в квадратных скобках, соответствует ровно одному символу. Чтобы распространить действие класса на несколько символов, используются квантификаторы, которые указываются сразу после квадратных скобок:

* **?** — символ входит ноль или один раз,
* **\*** — любое количество вхождений, включая ноль,
* **+** — одно или более вхождений символа в строку.

|  |
| --- |
| SELECT '1' RLIKE '^[[:digit:]]+$', '453455234' RLIKE '^[[:digit:]]+$'; SELECT '' RLIKE '^[[:digit:]]+$', '45.3455234' RLIKE '^[[:digit:]]+$'; |

Помимо круглых и квадратных скобок, в регулярных выражениях также применяются фигурные скобки. Они предназначены для указания числа или диапазона повторения элемента.

Давайте создадим регулярное выражение для цены. Целая часть может состоять из любого числа цифр, а дробная часть всегда состоит из двух.

|  |
| --- |
| SELECT '123.90' RLIKE '^[[:digit:]]\*\\.[[:digit:]]{2}$'; SELECT '123' RLIKE '^[[:digit:]]\*\\.[[:digit:]]{2}$'; |

# 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.

|  |
| --- |
| SELECT \* FROM catalogs ORDER BY id; |

Запрос сортирует результат выборки по полю id, можем отсортировать записи и по имени столбца:

|  |
| --- |
| SELECT id, name FROM catalogs ORDER BY name; |

По умолчанию сортировка производится в прямом порядке, однако, добавив после имени столбца ключевое слово **DESC**, можно добиться сортировки в обратном порядке:

|  |
| --- |
| SELECT \* FROM catalogs ORDER BY id DESC; |

Сортировку записей можно производить и по нескольким столбцам. Давайте вставим несколько товарных позиций в таблицу **products**:

|  |
| --- |
| 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); SELECT \* FROM products; |

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; |

В рамках каждого каталога у нас сначала выводятся самые дешевые товарные позиции, а потом дорогие. Если мы захотим изменить порядок сортировки, мы можем добавить ключевое слово **DESC**:

|  |
| --- |
| SELECT id, catalog\_id, price, name FROM products ORDER BY catalog\_id, price DESC; |

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; |

# Ограничения

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.

|  |
| --- |
| SELECT catalog\_id FROM products ORDER BY catalog\_id; |

Данный запрос выдаст множество повторяющихся значений. Иногда удобнее, когда возвращаются только уникальные значения, Для этого перед именем столбца можно использовать ключевое слово **DISTINCT**:

|  |
| --- |
| SELECT DISTINCT catalog\_id FROM products ORDER BY catalog\_id; |

Для ключевого слова **DISTINCT** имеется противоположное слово **ALL**, которое предписывает извлечение всех значений столбца, в том числе и повторяющихся. Поскольку такое поведение установлено по умолчанию, ключевое слово **ALL** часто опускают.

|  |
| --- |
| SELECT ALL catalog\_id FROM products ORDER BY catalog\_id; |

Для решения схожих задача, часто используется группировка значений, при помощи ключевого слова **GROUP BY**, которому будет посвящена следующая тема. Условия и ограничения, которые мы рассмотрели выше, можно применять и в отношении команд обновления **UPDATE** и удаления **DELETE**.

Например, давайте уменьшим цену на 10 % для материнских плат, которые стоят больше 5000 рублей. Давайте сначала найдем все товарные позиции, подходящие к этому условию:

|  |
| --- |
| SELECT   \* FROM   products WHERE   catalog\_id = 2 AND   price > 5000; |

А затем заменим команду **SELECT** на **UPDATE**:

|  |
| --- |
| UPDATE   products SET   price = price \* 0.1 WHERE   catalog\_id = 2 AND   price > 5000; |

Мы можем удалить две самые дорогие товарные позиции из таблицы **products**, для этого необходимо отсортировать таблицу при помощи ключевого слова **ORDER BY**:

|  |
| --- |
| SELECT   \* FROM   products ORDER BY   price DESC; |

Ограничиваем выборку двумя строками:

|  |
| --- |
| SELECT   \* FROM   products ORDER BY   price DESC LIMIT 2; |

А затем заменить команду **SELECT** на **DELETE**:

|  |
| --- |
| DELETE FROM   products ORDER BY   price DESC LIMIT 2; |

Таким образом, всегда можно подобрать подходящие условия при помощи SELECT-запроса, а потом поменять **SELECT** на **UPDATE** или **DELETE**.

# Предопределенные функции

MySQL, как и любая другая база данных, обладает большим числом предопределенных функций, т. е. готовых функций, которые предоставляют систему управления базами данных. Мы как разработчики можем писать и свои собственные функции, однако это тема следующих роликов.

Так же, как и в любом другом языке программирования, функции характеризуются именем и аргументами, которые перечисляются через запятую за именем в круглых скобках. Если аргументы у функции отсутствуют, круглые скобки все равно следует указывать. Результат функции подставляется в место вызова функции.

## Календарные функции

Например, одной из часто используемых является функция **NOW()**, которая позволяет получить текущую дату:

|  |
| --- |
| SELECT NOW(); |

Например, при вставке нового значения в таблицу **users** требуется 3 временных метки:

* дата рождения — **birthday\_at**;
* дата создания записи — **created\_at**;
* дата обновления записи — **updated\_at**.

Таблица у нас создана таким образом, что две последние даты задаются неявно, однако мы могли бы задавать их при помощи функции **NOW()**:

|  |
| --- |
| INSERT INTO users VALUES (NULL, '1986-01-20', NOW(), NOW()); |

Вычисление текущего времени в рамках одного SQL-запроса производится только один раз, сколько бы раз они ни вызывались на протяжении данного запроса. Это приводит к тому, что временное значение в рамках всего запроса остается постоянным.

Календарных функций довольно много. Например, при помощи функции **DATE()** в полях **updated\_at** и **created\_at** можно отсекать время суток, таким образом, в результирующей таблице остаётся только дата.

|  |
| --- |
| SELECT id, name, birthday\_at, DATE(created\_at), DATE(updated\_at) FROM users; |

Обратите внимание на название столбцов с использованием функций **DATE**: они содержат название функций и аргументы. В результате оперировать названиями столбцов очень неудобно. Поэтому их часто переименовывают при помощи ключевого слова **AS**:

|  |
| --- |
| SELECT   id,   name,   birthday\_at,   DATE(created\_at) AS created\_at,   DATE(updated\_at) AS updated\_at FROM   users; |

Допускается не указывать ключевое слово **AS**:

|  |
| --- |
| SELECT   id,   name,   birthday\_at,   DATE(created\_at) created\_at,   DATE(updated\_at) updated\_at FROM   users; |

Эффект получается тот же самый, столбец получает новое название.

Для форматирования календарных типов используется функция **DATE\_FORMAT(date, format)**, которая принимает в качестве первого аргумента время в одном из календарных типов, а в качестве второго — строку форматирования.

|  |
| --- |
| SELECT DATE\_FORMAT('2018-06-12 01:59:59', 'На дворе %Y год'); |

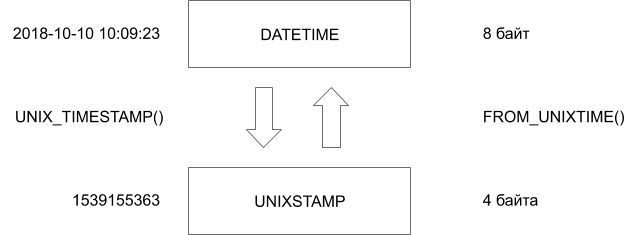
Скалярное значение даты можно заменить вызовом уже рассмотренной функции **NOW()**:

|  |
| --- |
| SELECT DATE\_FORMAT('2018-06-12 01:59:59', 'На дворе %Y год'); |

Последовательность **%Y** отвечает за извлечение года из календарного значения и представления его в строковом виде. Таких последовательностей очень много, желательно познакомиться с ними по [документации](https://dev.mysql.com/doc/refman/5.5/en/date-and-time-functions.html#function_date-format). Например, мы можем отформатировать день рождения пользователей в более привычном формате: день, месяц, год:

|  |
| --- |
| SELECT name, DATE\_FORMAT(birthday\_at, '%d.%m.%Y') AS birthday\_at FROM users; |

Еще часто встречается задача преобразования даты и времени в UNIXSTAMP-формат — количество секунд, которое прошло с полуночи 1 января 1970 года. Так как это целое число, его можно довольно быстро обрабатывать, индексировать и оно занимает мало места. Достаточно 4 байт, если нас устраивают даты от 1970 по 2038 год.



|  |
| --- |
| SELECT   UNIX\_TIMESTAMP('2018-10-10 10:09:23') AS TIMESTAMP,   FROM\_UNIXTIME(1539155363) AS DATETIME; |

Пусть стоит задача вычисления текущего возраста пользователя. Один из вариантов состоит в преобразовании даты рождения и текущей даты в дни при помощи функции **TO\_DAYS()** и делению на число 365.25. В году 365 дней, дробное число 0.25 призвано компенсировать високосные года, которые случаются раз в четыре года.

|  |
| --- |
| SELECT   name,   (TO\_DAYS(NOW()) - TO\_DAYS(birthday\_at))/365.25 AS age FROM   users; |

Чтобы избавиться от дробной части, можно воспользоваться функцией **FLOOR()**.

|  |
| --- |
| SELECT   name,   FLOOR((TO\_DAYS(NOW()) - TO\_DAYS(birthday\_at))/365.25) AS age FROM   users; |

Можно добиться более точного результата, если воспользоваться специальной функцией **TIMESTAMPDIFF()**:

|  |
| --- |
| SELECT   name,   TIMESTAMPDIFF(YEAR, birthday\_at, NOW()) AS age FROM   users; |

## Случайное значение

Использование функций допускается не только после ключевого слова **SELECT()**. Везде, где используется имя столбца, можно задействовать функцию, например, для вывода записей в случайном порядке можно задействовав функцию **RAND()**, передав ее ключевому слову **ORDER BY**:

|  |
| --- |
| SELECT \* FROM users ORDER BY RAND(); |

Получить случайное значение можно, если ограничить выборку при помощи ключевого слова **LIMIT**:

|  |
| --- |
| SELECT \* FROM users ORDER BY RAND() LIMIT 1; |

Существуют и информационные функции, например функция **VERSION()** возвращает текущую версию MySQL-сервера:

|  |
| --- |
| SELECT VERSION(); |

Обратите внимание, что мы часто не используем ключевое слово **FROM** и таблицу, когда нам требуется извлечь только одно значение, например, то, которое возвращает встроенная функция. В таком случае допускается использование псевдотаблицы **DUAL**, которая на самом деле не существует:

|  |
| --- |
| SELECT VERSION() FROM DUAL; |

## Информационные функции

Часто в прикладных программах требуется узнать значение, присвоенное столбцу и снабженное атрибутом **AUTO\_INCREMENT**. Это может потребоваться, чтобы использовать сгенерированное значение первичного ключа в качестве внешнего в другой таблице. Только что сгенерированное значение возвращает встроенная функция MySQL **LAST\_INSERT\_ID()**.

|  |
| --- |
| INSERT INTO catalogs VALUES (NULL, 'Процессоры');  INSERT INTO products   (name, description, price, catalog\_id) VALUES   ('Intel Core i3-8100', 'Процессор Intel.', 7890.00, LAST\_INSERT\_ID()),   ('Intel Core i5-7400', 'Процессор Intel.', 12700.00, LAST\_INSERT\_ID()),   ('AMD FX-8320E', 'Процессор AMD.', 4780.00, LAST\_INSERT\_ID()),   ('AMD FX-8320', 'Процессор AMD.', 7120.00, LAST\_INSERT\_ID());  INSERT INTO catalogs VALUES (NULL, 'Материнские платы');  INSERT INTO products   (name, description, price, catalog\_id) VALUES   ('ASUS ROG MAXIMUS X HERO', 'Z370, Socket 1151-V2, DDR4, ATX', 19310.00, LAST\_INSERT\_ID()),   ('Gigabyte H310M S2H', 'H310, Socket 1151-V2, DDR4, mATX', 4790.00, LAST\_INSERT\_ID()),   ('MSI B250M GAMING PRO', 'B250, Socket 1151, DDR4, mATX', 5060.00, LAST\_INSERT\_ID());  SELECT \* FROM catalogs; SELECT id, name description, price, catalog\_id FROM products; |

К информационным функциям относится также функция **DATABASE()**, которая возвращает текущую базу данных. Если текущая база данных не выбрана, **DATABASE()** возвращает **NULL**:

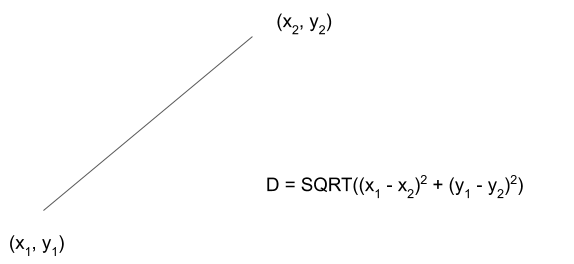
|  |
| --- |
| SELECT DATABASE(); USE shop SELECT DATABASE(); |

Функция **USER()** возвращает аккаунт текущего пользователя:

|  |
| --- |
| SELECT USER(); |

## Математические функции

MySQL предоставляет огромное количество математических функций. С некоторыми, например функцией **RAND()**, предназначенной для получения случайных чисел, мы уже знакомы. Давайте познакомимся с некоторыми другими.



Например, функция **SQRT** позволяет получать квадратный корень числа. Давайте при помощи этой функции вычислим расстояние между двумя точками в декартовой системе координат. Пусть есть две точки с координатами X и Y, расстояние между ними вычисляется как разница квадратов расстояния по представленной на рисунке формуле.

|  |
| --- |
| DROP TABLE IF EXISTS distances; CREATE TABLE distances (   id SERIAL PRIMARY KEY,   x1 INT NOT NULL,   y1 INT NOT NULL,   x2 INT NOT NULL,   y2 INT NOT NULL,   distance DOUBLE AS (SQRT(POW(x1 - x2, 2) + POW(y1 - y2, 2))) ) COMMENT = 'Расстояние между двумя точками';  INSERT INTO distances   (x1, y1, x2, y2) VALUES   (1, 1, 4, 5),   (4, -1, 3, 2),   (-2, 5, 1, 3);  SELECT \* FROM distances; |

В качестве альтернативы можно использовать JSON-поля, отводя под каждую из точек отдельное поле, в котором будет JSON-коллекция, содержащая X и Y:

|  |
| --- |
| DROP TABLE IF EXISTS distances; CREATE TABLE distances (   id SERIAL PRIMARY KEY,   a JSON NOT NULL,   b JSON NOT NULL,   distance DOUBLE AS (SQRT(POW(a->>'$.x' - b->>'$.x', 2) + POW(a->>'$.y' - b->>'$.y', 2))) ) COMMENT = 'Расстояние между двумя точками';  INSERT INTO distances   (a, b) VALUES   ('{"x": 1, "y": 1}', '{"x": 4, "y": 5}'),   ('{"x": 4, "y": -1}', '{"x": 3, "y": 2}'),   ('{"x": -2, "y": 5}', '{"x": 1, "y": 3}');  SELECT \* FROM distances; |

Для некоторых задач требуются тригонометрические функции. Например, если нам известен угол треугольника и длина двух его сторон, с использованием синуса мы можем вычислить его площадь. Давайте создадим таблицу **triangles**, которая будет состоять из трех столбцов:

|  |
| --- |
| DROP TABLE IF EXISTS triangles; CREATE TABLE triangles (   id SERIAL PRIMARY KEY,   a DOUBLE NOT NULL COMMENT 'Сторона треугольника',   b DOUBLE NOT NULL COMMENT 'Сторона треугольника',   angle INT NOT NULL COMMENT 'Угол треугольника в градусах',   square DOUBLE AS (a \* b \* SIN(RADIANS(angle)) / 2.0) ) COMMENT = 'Площадь треугольника'; |

Четвертый столбец мы сделаем вычисляемым, подставив в него формулу вычисления площади треугольника. Так как угол у нас задан в градусах, его потребуется преобразовать в радианы при помощи функции **RADIANS()**:синус будет ожидать значение именно в радианах.

|  |
| --- |
| INSERT INTO   triangles (a, b, angle) VALUES   (1.414, 1, 45),   (2.707, 2.104, 60),   (2.088, 2.112, 56),   (5.014, 2.304, 23),   (3.482, 4.708, 38);  SELECT \* FROM triangles; |

Как видим, результат вычисления имеет до 16 знаков после запятой. Это не всегда удобно для восприятия: результат вычисления можно округлить при помощи помощи функции **ROUND()**.

Давайте при помощи оператора **ALTER TABLE** поменяем определение столбца **square** в таблице, добавив округление результата до четвертого знака после запятой.

|  |
| --- |
| ALTER TABLE triangles CHANGE square square DOUBLE AS (ROUND(a \* b \* SIN(RADIANS(angle)) / 2.0, 4)); SELECT \* FROM triangles; |

Функция **ROUND()** осуществляет математическое округление, т. е., до ближайшего целого числа.

|  |
| --- |
| SELECT ROUND(2.4), ROUND(2.5), ROUND(2.6); |

**ROUND()** — не единственная функция управления дробными числами. Функция **CEILING()** возвращает первое целое число, которое встречает справа от значения аргумента.

|  |
| --- |
| SELECT CEILING(-2.9), CEILING(-2.1), CEILING(2.1), CEILING(2.9); |

Функция **FLOOR(X)** сходна по действию с функцией **CEILING(X)**, но сдвиг происходит в другую сторону.

|  |
| --- |
| SELECT FLOOR(-2.9), FLOOR(-2.1), FLOOR(2.1), FLOOR(2.9); |

## Строковые функции

MySQL предоставляет большое количество функций, которые обслуживают строки. Очень часто требуется выбрать из таблицы не весь текст, а лишь несколько первых символов. Эту задачу удобно решать при помощи функции **SUBSTRING()**:

|  |
| --- |
| SELECT id, SUBSTRING(name, 1, 5) AS name FROM users; |

Нумерация символов в строковых функциях всегда начинается с единицы. Для объединения строк предназначена функция **CONCAT**. Например, давайте выведем имя пользователя и его возраст через пробел:

|  |
| --- |
| SELECT id, CONCAT(name, ' ', TIMESTAMPDIFF(YEAR, birthday\_at, NOW())) AS name FROM users; |

## Логические функции

Логические функции помогают преобразовать результат в зависимости от выполнения того или иного условия.

Давайте выведем слово «совершеннолетний» или «несовершеннолетний», в зависимости от того, достиг пользователь 18 лет или нет. Для этого можно воспользоваться функцией **IF**, которая принимает три аргумента:

* первый — логическое выражение,
* второй — результат, который выводится, если логическое выражение истинное,
* третий — если логическое выражение оказалось ложным.

Давайте посмотрим, как работает функция:

|  |
| --- |
| SELECT IF(TRUE, 'истина', 'ложь'), IF(FALSE, 'истина', 'ложь'); |

Теперь давайте решим задачу определения совершеннолетия пользователя:

|  |
| --- |
| SELECT   name,   IF(  TIMESTAMPDIFF(YEAR, birthday\_at, NOW()) >= 18,  'совершеннолетний',  'несовершеннолетний'   ) AS status FROM   users; |

Если условий больше, можно использовать выражение **CASE**, например, пусть у нас имеется таблица с цветами радуги:

|  |
| --- |
| DROP TABLE IF EXISTS rainbow; CREATE TABLE rainbow (   id SERIAL PRIMARY KEY,   color VARCHAR(255) ) COMMENT = 'Цвета радуги';  INSERT INTO   rainbow (color) VALUES   ('red'),   ('orange'),   ('yellow'),   ('green'),   ('blue'),   ('indigo'),   ('violet');  SELECT   CASE  WHEN color = 'red' THEN 'красный'  WHEN color = 'orange' THEN 'оранжевый'  WHEN color = 'yellow' THEN 'желтый'  WHEN color = 'green' THEN 'зеленый'  WHEN color = 'blue' THEN 'голубой'  WHEN color = 'indigo' THEN 'синий'  ELSE 'фиолетовый'   END AS russian FROM   rainbow; |

## Вспомогательные функции

MySQL предоставляет различные вспомогательные функции. Например функция **INET\_ATON(address)** принимает IP-адрес **address** и представляет его в виде целого числа:

|  |
| --- |
| SELECT INET\_ATON('62.145.69.10'), INET\_ATON('127.0.0.1'); |

Функция **INET\_NTOA** решает обратную задачу:

|  |
| --- |
| SELECT INET\_NTOA(1049707786), INET\_NTOA(2130706433); |

Функция **UUID()** возвращает универсальный уникальный идентификатор. Идентификатор **UUID** реализован в виде числа, которое является глобально уникальным во времени и пространстве.

Два вызова функции **UUID()** вернут два разных значения, если они производятся одновременно на двух разных компьютерах или на одном и том же компьютере в разное время.

|  |
| --- |
| SELECT UUID(); SELECT UUID(); |

Это далеко не все функции, которые предоставляет MySQL. Часть функций, например агрегатные, мы будем рассматривать в следующих уроках.

# Используемые источники

1. <https://dev.mysql.com/doc/refman/5.7/en/functions.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 систе
8. мы баз данных, 8-е издание.: Пер. с англ. — М.: Издательский дом "Вильямс", 2005. — 1328 с.
9. Карвин Б. Программирование баз данных SQL. Типичные ошибки и их устранение. — Рид Групп, 2011. — 336 с.

**Урок 4**

Агрегация данных

[Группировка данных](https://docs.google.com/document/d/1TjaJPA-LI8HZKlWTGpRBtNlbnnmCoqzMuh8BGY4aiSk/edit#heading=h.qjnqizjxpwjm)

[Агрегатные функции](https://docs.google.com/document/d/1TjaJPA-LI8HZKlWTGpRBtNlbnnmCoqzMuh8BGY4aiSk/edit#heading=h.c84or328wnrq)

[Специальные возможности GROUP BY](https://docs.google.com/document/d/1TjaJPA-LI8HZKlWTGpRBtNlbnnmCoqzMuh8BGY4aiSk/edit#heading=h.3tb0lm9pg2u0)

[Используемые источники](https://docs.google.com/document/d/1TjaJPA-LI8HZKlWTGpRBtNlbnnmCoqzMuh8BGY4aiSk/edit#heading=h.2xcytpi)

# Группировка данных

В одном из предыдущих роликов мы с вами сталкивались с результирующими таблицами, которые содержат повторяющиеся значения:

|  |
| --- |
| SELECT catalog\_id FROM products; |

Мы уже знаем механизм получения уникальных значений при помощи ключевого слова **DISTINCT**:

|  |
| --- |
| SELECT DISTINCT catalog\_id FROM products; |

При помощи вычисляемых столбцов мы и сами можем создавать такие повторяющиеся комбинации:

|  |
| --- |
| SELECT id, name, id % 3 FROM products ORDER BY id % 3; |

В языке SQL для работы с такими группами предназначено специальное ключевое слово **GROUP BY**. Например, задачу получения уникальных значений можно решить следующим образом:

|  |
| --- |
| 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.

|  |
| --- |
| SELECT id, name, SUBSTRING(birthday\_at, 1, 3) FROM users; |

Здесь мы преобразуем календарный тип **DATETIME** поля **birthday\_at** к строковому значению и при помощи функции **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; |

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

При помощи конструкции **GROUP BY** мы можем сгруппировать поля по декадам:

|  |
| --- |
| SELECT SUBSTRING(birthday\_at, 1, 3) AS decade FROM users GROUP BY decade; |

Если мы попытаемся вывести имена пользователей, мы потерпим неудачу:

|  |
| --- |
| SELECT id, name, SUBSTRING(birthday\_at, 1, 3) AS decade FROM users GROUP BY decade; |

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; |

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; |

Порядок следования ключевых слов важен: мы не можем размещать ключевое слово **ORDER BY** раньше **GROUP BY**, иначе мы получаем сообщение об ошибке. Это же относится к ключевому слову **LIMIT**, которое всегда должно располагаться после всех остальных ключевых слов.

Когда в запросе мы не используем конструкцию **GROUP BY**, вся таблица рассматривается как одна большая группа. Поэтому, если мы применим функцию **COUNT(\*)** к таблице **users**, мы можем получить количество всех пользователей, без учета года их рождения:

|  |
| --- |
| SELECT COUNT(\*) FROM users; |

Следующий ролик будет полностью посвящен агрегационным функциям. Пока давайте посмотрим, какие задачи можно решать при помощи групп. 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; |

Таким образом, мы можем получить список всех пользователей в каждой из групп.

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; |

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; |

Функция **GROUP\_CONCAT** имеет ограничения: она может извлекать из группы максимум 1000 элементов. Впрочем, это значение можно увеличить за счет изменения параметра сервера **group\_concat\_max\_len**. Как это сделать, мы подробнее будем разбирать в одной из следующих тем.

# Агрегатные функции

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**:

|  |
| --- |
| SELECT COUNT(id) FROM catalogs; |

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; |

Функции вроде **COUNT** называются агрегационными. Дело в том, что их значение изменяется при использовании конструкции **GROUP BY**.  Конструкция **GROUP BY** разбивает таблицу на отдельные группы, например:

|  |
| --- |
| SELECT catalog\_id FROM products; SELECT catalog\_id FROM products GROUP BY catalog\_id; |

Функция **COUNT()** возвращает результат для каждой из этих групп.

|  |
| --- |
| SELECT catalog\_id, COUNT(\*) AS total FROM products GROUP BY catalog\_id; |

Давайте подробнее остановимся на реакции агрегационных функций на NULL-значение. Для этого создадим таблицу, содержащую два столбца — **id** и **value**:

|  |
| --- |
| CREATE TABLE tbl (   id INT NOT NULL,   value INT DEFAULT NULL ); INSERT INTO tbl VALUES (1, 230); INSERT INTO tbl VALUES (2, NULL); INSERT INTO tbl VALUES (3, 405); INSERT INTO tbl VALUES (4, NULL);  SELECT \* FROM tbl; |

Давайте применим функцию **COUNT()** к обоим столбцам:

|  |
| --- |
| SELECT COUNT(id), COUNT(value) FROM tbl; |

Это связано с тем, что **COUNT** игнорирует NULL-поля.

|  |
| --- |
| SELECT COUNT(\*) FROM tbl; |

Если вместо имени столбца используется звездочка, значения **NULL** не влияют на результат:

|  |
| --- |
| SELECT   id,   catalog\_id FROM   products; |

Давайте попробуем подсчитать количество элементов для полей **id** и **catalog\_id**:

|  |
| --- |
| SELECT   COUNT(id) AS total\_ids,   COUNT(catalog\_id) AS total\_catalog\_ids FROM   products; |

Результаты совпадают, однако если мы добавим в функцию **COUNT** ключевое слово **DISTINCT**, мы можем добиться того, что будут подсчитываться только уникальные значения:

|  |
| --- |
| SELECT   COUNT(DISTINCT id) AS total\_ids,   COUNT(DISTINCT catalog\_id) AS total\_catalog\_ids FROM   products; |

Итак, у нас 7 товарных позиций в двух каталогах.

Функции **MIN()** и **MAX()** возвращают минимальное и максимальное значения столбца:

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

Здесь мы извлекаем минимальную и максимальную цену в интернет-магазине. При группировки по полю **catalog\_id** мы получим максимальную и минимальную цену в рамках каждого из разделов каталога:

|  |
| --- |
| SELECT   catalog\_id,   MIN(price) AS min,   MAX(price) AS max FROM   products GROUP BY   catalog\_id; |

Агрегационные функции, можно применять только после ключевого слова **SELECT**. Попытка использования функций **MIN()** и **MAX()** в выражении **WHERE** приведет к ошибке.

|  |
| --- |
| SELECT \* FROM products WHERE price = MAX(price); |

Решить эту задачу проще всего с использованием сортировки, используя ключевое слово **ORDER BY**:

|  |
| --- |
| SELECT id, name, price FROM products ORDER BY price DESC LIMIT 1; |

Функция **AVG()** возвращает среднее значение аргумента. Давайте подсчитаем среднюю цену товара:

|  |
| --- |
| SELECT AVG(price) FROM products; |

Так как мы не производим группировку, средняя цена вычисляется для всех товарных позиций, занесенных в таблицу **products.** При желании мы можем округлить полученное число, например до второго знака после запятой при помощи функции **ROUND**:

|  |
| --- |
| SELECT ROUND(AVG(price), 2) FROM products; |

Если мы добавим **GROUP BY**, например по полю **catalog\_id**, мы получим среднее цены для каждого из разделов:

|  |
| --- |
| SELECT   catalog\_id,   ROUND(AVG(price), 2) AS price FROM   products GROUP BY   catalog\_id; |

Внутри агрегационных функций допускается использовать вычисляемые значения. Например, мы можем увеличить значение цены на 20 %:

|  |
| --- |
| SELECT   catalog\_id,   ROUND(AVG(price \* 1.2), 2) AS price FROM   products GROUP BY   catalog\_id; |

Сумму всех значений столбца можно подсчитать при помощи функции **SUM()**:

|  |
| --- |
| SELECT SUM(price) FROM products; |

Она, так же как и все остальные агрегатные функции, будет подсчитывать только значения, отличные от **NULL**:

|  |
| --- |
| SELECT catalog\_id, SUM(price) FROM products GROUP BY catalog\_id; |

Так мы можем подсчитать сумму всех цен в каждом из разделов.

# Специальные возможности GROUP BY

Каждая строка результирующего запроса с **GROUP BY** представляет собой отдельную группу:

|  |
| --- |
| SELECT   COUNT(\*) AS total,   SUBSTRING(birthday\_at, 1, 3) AS decade FROM   users GROUP BY   decade; |

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**:

|  |
| --- |
| SELECT   COUNT(\*) AS total,   SUBSTRING(birthday\_at, 1, 3) AS decade FROM   users GROUP BY   decade HAVING   total >= 2; |

Допускается использование условия **HAVING** без группировки **GROUP BY**:

|  |
| --- |
| SELECT   \* FROM   users HAVING   birthday\_at >= '1990-01-01'; |

В этом случае каждая строка таблицы рассматривается как отдельная группа. Условие **HAVING** идеально подходит в ситуации, когда требуется обнаружить повторяющиеся значения:

|  |
| --- |
| TRUNCATE products; |

Давайте два раза вставим в таблицу **products** одни и те же значения:

|  |
| --- |
| INSERT INTO products   (name, description, price, catalog\_id) VALUES   ('Intel Core i3-8100', 'Процессор Intel', 7890.00, 1),   ('Intel Core i5-7400', 'Процессор Intel', 12700.00, 1),   ('AMD FX-8320E', 'Процессор AMD', 4780.00, 1),   ('AMD FX-8320', 'Процессор AMD', 7120.00, 1),   ('ASUS ROG MAXIMUS X HERO', 'Z370, Socket 1151-V2, DDR4, ATX', 19310.00, 2),   ('Gigabyte H310M S2H', 'H310, Socket 1151-V2, DDR4, mATX', 4790.00, 2),   ('MSI B250M GAMING PRO', 'B250, Socket 1151, DDR4, mATX', 5060.00, 2); SELECT id, name, catalog\_id FROM products; |

Существует много способов избавиться от таких дублей. Пока мы не знакомились с JOIN-соединениями и многотабличным запросом **DELETE**. Давайте решим эту задачу через промежуточную таблицу. Сформируем запрос на извлечение записей для размещения в промежуточной таблице:

|  |
| --- |
| SELECT   name, description, price, catalog\_id FROM   products GROUP BY   name, description, price, catalog\_id; |

Создадим таблицу **products\_new**, структура которой полностью повторяет структуру таблицы **products**:

|  |
| --- |
| CREATE TABLE products\_new (   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'; |

Теперь сформируем запрос **INSERT ... SELECT ...**, который будет вставлять в таблицу **products\_new** только уникальные значения из таблицы **products**. Так как поля **id**, **created\_at** и **updated\_at** не входят в групповой запрос, нам потребуется сформировать их снова, используя **NULL** и функцию **NOW()**:

|  |
| --- |
| INSERT INTO   products\_new SELECT   NULL, name, description, price, catalog\_id, NOW(), NOW() FROM   products GROUP BY   name, description, price, catalog\_id; |

Давайте посмотрим содержимое промежуточной таблицы:

|  |
| --- |
| SELECT id, name, catalog\_id FROM products\_new; |

Итак, выбраны только уникальные значения, и вместо 14 позиций у нас осталось 7. Теперь давайте уничтожим таблицы **products**:

|  |
| --- |
| DROP TABLE products; |

И переименуем таблицы **products\_new** в **products**. Для этого воспользуемся оператором **ALTER TABLE**.

|  |
| --- |
| ALTER TABLE products\_new RENAME products; SHOW TABLES; SELECT id, name, catalog\_id FROM products; |

Для группировки можно использовать вычисляемые значения. В таблице **users** для каждого из пользователей в поле **birthday\_at** указывается его дата рождения. Давайте извлечем года, на которые приходятся даты рождения, и случайного пользователя:

|  |
| --- |
| SELECT name, birthday\_at FROM users; |

Чтобы было интереснее, давайте добавим несколько записей с таким расчетом, чтобы дни рождения нескольких пользователей приходились на один год:

|  |
| --- |
| INSERT INTO users (name, birthday\_at) VALUES   ('Светлана', '1988-02-04'),   ('Олег', '1998-03-20'),   ('Юлия', '2006-07-12');  SELECT name, birthday\_at FROM users ORDER BY birthday\_at; |

Запрашиваем пользователей и видим, что на 1988, 1998 и 2006 приходится по две даты рождения. Теперь давайте извлечем из таблицы **users** года, на которые приходятся даты рождения:

|  |
| --- |
| SELECT YEAR(birthday\_at) FROM users ORDER BY birthday\_at; |

Избавляемся от повторов:

|  |
| --- |
| SELECT   YEAR(birthday\_at) AS birthday\_year FROM   users GROUP BY   birthday\_year ORDER BY   birthday\_year; |

Если мы сейчас внесем в SELECT-список имя пользователя, то потерпим неудачу. Обойти проблемы мы можем, вернув вместо имени пользователя какое-то агрегационное значение, например максимальное значение, что бы это не значило:

|  |
| --- |
| SELECT   MAX(name),   YEAR(birthday\_at) AS birthday\_year FROM   users GROUP BY   birthday\_year ORDER BY   birthday\_year; |

В данном случае нам все равно, какого мы пользователя вернем. На этот случай в MySQL предусмотрена специальная функция **ANY\_VALUE()**, которая возвращает случайное значение из группы:

|  |
| --- |
| SELECT   ANY\_VALUE(name),   YEAR(birthday\_at) AS birthday\_year FROM   users GROUP BY   birthday\_year ORDER BY   birthday\_year; |

Конструкция **WITH ROLLUP** позволяет добавить еще одну строку с суммой значений всех предыдущих строк. Возвращаясь к таблице пользователей **users**, подсчитываем количество пользователей, родившихся в 80-х, 90-х и 2000-х годах

|  |
| --- |
| SELECT   SUBSTRING(birthday\_at, 1, 3) AS decade,   COUNT(\*) FROM   users GROUP BY   decade; |

При помощи конструкции **WITH ROLLUP** мы можем добавить последнюю результирующую строку с количеством всех пользователей:

|  |
| --- |
| SELECT   SUBSTRING(birthday\_at, 1, 3) AS decade,   COUNT(\*) FROM   users GROUP BY   decade WITH ROLLUP; |

Как видно, в результирующую таблицу добавлена дополнительная строка с количеством всех пользователей и значением **NULL** для столбца **decade**. Значение **NULL** присваивается всем столбцам, кроме сгенерированного агрегатной функцией (в данном случае функцией **COUNT()**).

В MySQL версии 8.0 заменить значение **NULL** можно при помощи функции **GROUPING**, которая принимает в качестве аргумента имя столбца и возвращает **0**, если для него есть значение, и **1**, если в результирующей таблице поле принимает значение **NULL**. Применив функцию **IF**, можно заменить значение **NULL** каким-либо осмысленным текстом.

# 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.

**Урок 5**

Сложные запросы

[Многотабличные запросы](https://docs.google.com/document/d/1Poq5YUiWTSOUFtkC9GXNtwtO9Xwt5q1eH5FiGGcexUM/edit#heading=h.a87pq6wf3svj)

[Объединение UNION](https://docs.google.com/document/d/1Poq5YUiWTSOUFtkC9GXNtwtO9Xwt5q1eH5FiGGcexUM/edit#heading=h.is0w7v781akx)

[Вложенные запросы](https://docs.google.com/document/d/1Poq5YUiWTSOUFtkC9GXNtwtO9Xwt5q1eH5FiGGcexUM/edit#heading=h.c84or328wnrq)

[JOIN-соединения таблиц](https://docs.google.com/document/d/1Poq5YUiWTSOUFtkC9GXNtwtO9Xwt5q1eH5FiGGcexUM/edit#heading=h.3tb0lm9pg2u0)

[Внешние ключи и ссылочная целостность](https://docs.google.com/document/d/1Poq5YUiWTSOUFtkC9GXNtwtO9Xwt5q1eH5FiGGcexUM/edit#heading=h.1btdn6gelt3j)

[Используемые источники](https://docs.google.com/document/d/1Poq5YUiWTSOUFtkC9GXNtwtO9Xwt5q1eH5FiGGcexUM/edit#heading=h.fs11wcbo4wqz)

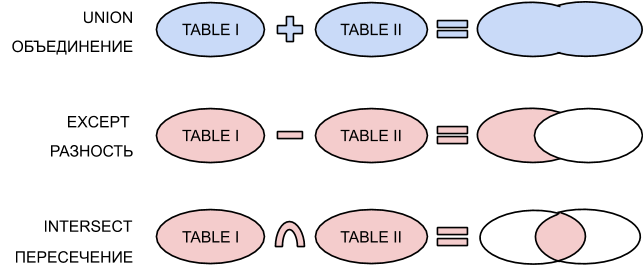
# Многотабличные запросы

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

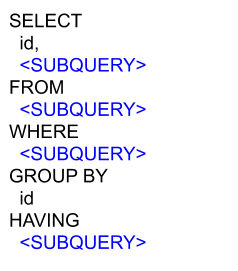
Многотабличные запросы условно можно поделить на три большие группы:

* 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.

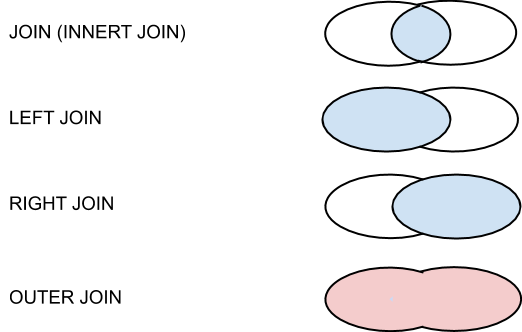


SQL поддерживает все эти операции, предоставляя операторы **UNION**, **EXCEPT** и **INTERSECT**. К сожалению, MySQL поддерживает только **UNION**, поддержка **EXCEPT** и **INTERSECT** не реализована.



Еще один тип запросов — это вложенные запросы. Вложенный запрос позволяет использовать результат, возвращаемый одним запросом, в другом. Здесь синим цветом представлены точки в запросе, где мы можем использовать вложенные запросы.

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».



# 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 давайте создадим таблицу **rubrics**, структура которой полностью совпадает с таблицей **catalogs**:

|  |
| --- |
| 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'); |

Выведем содержимое таблицы **catalogs**:

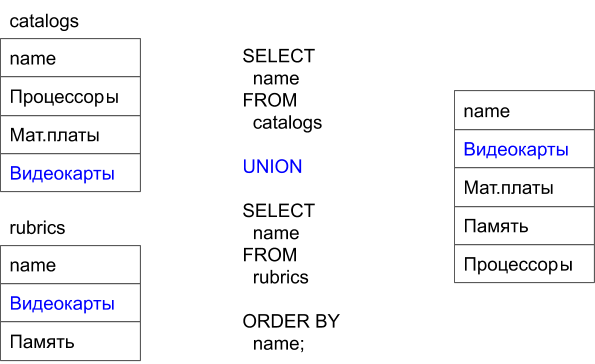
|  |
| --- |
| SELECT \* FROM catalogs; |

И таблицы **rubrics**:

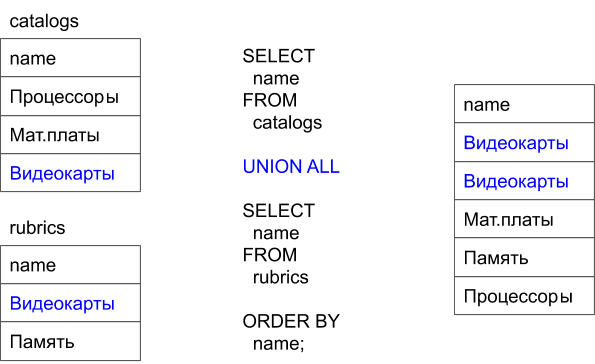
|  |
| --- |
| SELECT \* FROM rubrics; |

Чтобы объединить два результирующих запроса в один, воспользуемся **UNION**.

|  |
| --- |
| SELECT name FROM catalogs UNION SELECT name FROM rubrics; |



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

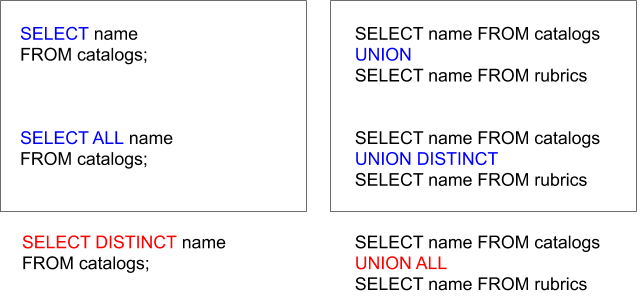


Чтобы предотвратить такое поведение, после **UNION** следует указать ключевое слово **ALL**. В этом случае в результирующий запрос будет содержать ровно столько записей, сколько находится в обеих исходных таблицах, несмотря на их дублирование.

Давайте в консоли убедимся, что запрос **UNION ALL** работает именно так, как мы описали.

|  |
| --- |
| SELECT name FROM catalogs UNION ALL SELECT name FROM rubrics; |

У нас появилось две записи «Видеокарты»: одна из таблицы **catalogs**, другая — из таблицы **rubrics**:



Ключевые слова **ALL** и **DISTINCT** являются взаимозаменяемыми: если вы видите где-то в SQL-синтаксисе **ALL**, его можно заменить на **DISTINCT**. И наоборот часто SQL-команда по умолчанию выбирает то или иное поведение, например, выводить все, включая дубли (**ALL**) или выводить только уникальные значения (**DISTINCT**).

В том случае, если ключевое слово можно опустить, его почти всегда опускают. Например, мы не пишем **SELECT ALL**, если можно писать просто **SELECT**. Точно так же в случае **UNION**, мы используем **UNION ALL**, когда хотим получить все записи. Однако, для получения уникальных значений используем краткую форму **UNION**, хотя могли бы писать **UNION DISTINCT**.

Если мы используем ключевое слово **ORDER BY** для сортировки, оно действует на весь результат запроса, а не на отдельные таблицы:

|  |
| --- |
| SELECT name FROM catalogs UNION ALL SELECT name FROM rubrics ORDER BY name; |

Или в обратном порядке:

|  |
| --- |
| SELECT name FROM catalogs UNION ALL SELECT name FROM rubrics ORDER BY name DESC; |

То же самое касается ключевого слова **LIMIT**: сначала происходит объединение результатов и лишь затем применяется ограничение **LIMIT**:

|  |
| --- |
| SELECT name FROM catalogs UNION ALL SELECT name FROM rubrics ORDER BY name DESC LIMIT 2; |

Обойти это ограничение в рамках синтаксиса **UNION** нельзя. В рамках вложенных запросов мы можем использовать сначала сортировку и ограничение и лишь затем использовать полученные результаты в **UNION**.

Чтобы превратить запросы во вложенные, SELECT-команды следует поместить в круглые скобки:

|  |
| --- |
| (SELECT name FROM catalogs ORDER BY name DESC LIMIT 2)  UNION ALL  (SELECT name FROM rubrics ORDER BY name DESC LIMIT 2); |

Если структура таблиц не совпадает, объединить их при помощи **UNION** не получится.

|  |
| --- |
| SELECT \* FROM catalogs UNION SELECT \* FROM products; |

Попытавшись объединить таблицы **catalogs** и **products**, мы получаем сообщение об ошибке, в котором говорится, что в этих таблицах разное количество столбцов. Однако мы можем подобрать условия таким образом, что содержимое таблиц будет объединено:

|  |
| --- |
| SELECT \* FROM catalogs UNION SELECT id, name FROM products; |

Следует иметь в виду, что первый SELECT-запрос определяет название столбцов.

|  |
| --- |
| SELECT \* FROM catalogs UNION SELECT id, name AS 'product' FROM products; |

Как бы ни назывались столбцы второй таблицы, в результирующей таблице для названия столбцов будет использоваться первая. Так как мы объединяем между собой не сами таблицы, а результаты запроса к ним, мы можем объединять полностью эквивалентные запросы.

|  |
| --- |
| SELECT \* FROM catalogs UNION ALL SELECT \* FROM catalogs; |

В **UNION** не обязательно должно участвовать только две таблицы. Используя несколько ключевых слов **UNION**, можно объединять три и более таблиц.

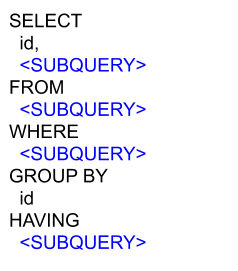
|  |
| --- |
| SELECT \* FROM catalogs UNION SELECT id, name FROM products UNION SELECT id, name FROM users; |

Не следует злоупотреблять UNION-объединениями. Как правило, UNION-запросы довольно медленно выполняются. Промежуточная таблица **UNION** в MySQL почти всегда размещается на жестком диске, поэтому все операции фильтрации и сортировки также будут осуществляться во временном файле.

По возможности после ключевого слова **SELECT** следует указывать только те столбцы, которые нужны в результирующей таблице или для составления запроса. Чем меньше столбцов указано, тем меньше размер промежуточной таблицы и тем быстрее проходят операции с ней.

# 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:



Чтобы СУБД могла отличать основной запрос и подзапрос, последний заключают в круглые скобки.

Пусть мы хотим выяснить список всех товаров, в разделе «Процессоры». Например, у нас имеется таблица разделов каталогов:

|  |
| --- |
| SELECT \* FROM catalogs; |

Она связана через внешний ключ **catalog\_id** с таблицей товарных позиций **products**.

|  |
| --- |
| SELECT id, name, catalog\_id FROM products; |

Если мы хотим извлечь все товары, относящиеся к разделу «Процессоры», мы можем воспользоваться следующим условием:

|  |
| --- |
| SELECT id, name, catalog\_id FROM products WHERE catalog\_id = 1; |

Таким образом, для извлечения списка процессоров нам пришлось выполнить несколько запросов. При помощи вложенных запросов мы можем объединить их в один. Давайте извлечем первичный ключ каталога «Процессоры»:

|  |
| --- |
| SELECT id FROM catalogs WHERE name = "Процессоры"; |

Теперь давайте превратим его во вложенный запрос:

|  |
| --- |
| SELECT   id, name, catalog\_id FROM   products WHERE   catalog\_id = (SELECT id FROM catalogs WHERE name = "Процессоры"); |

При помощи вложенных запросов мы можем решать целый спектр задач. Например, давайте найдем в таблице products товар с самой высокой ценой. Для начала, давайте найдем максимальную цену при помощи функции MAX():

|  |
| --- |
| SELECT MAX(price) FROM products; |

Теперь мы можем сформировать вложенный запрос:

|  |
| --- |
| SELECT   id, name, catalog\_id FROM   products WHERE   price = (SELECT MAX(price) FROM products); |

Для вложенных запросов, которые возвращают единичное значение, можно использовать не только оператор равенства, но и любой другой логический оператор. Например, давайте найдем все товары, чья цена ниже среднего:

|  |
| --- |
| SELECT   id, name, catalog\_id FROM   products WHERE   price < (SELECT AVG(price) FROM products); |

Для этого удобно воспользоваться агрегатной функцией **AVG**, которая возвращает среднее значение. Вложенные запросы можно использовать не только в условиях, но и, например, после ключевого слова **SELECT**.

Давайте для каждого из товаров извлечем название каталога. Для начала выведем список товарных позиций:

|  |
| --- |
| SELECT   id, name, catalog\_id FROM   products; |

Для замены внешнего ключа разделом, к которому принадлежит товар, мы можем воспользоваться следующим запросом:

|  |
| --- |
| SELECT name FROM catalogs WHERE id = 1; |

Только вместо единицы мы должны подставить **catalog\_id**:

|  |
| --- |
| SELECT   id,   name,   (SELECT name FROM catalogs WHERE id = catalog\_id) AS 'catalog' FROM   products; |

Обратите внимание, что в WHERE-условии столбец **id** принадлежит таблицы **catalogs**, а столбец **catalog\_id** таблице **catalogs**. В случае конфликтов мы можем явно использовать квалификационные имена:

|  |
| --- |
| SELECT   products.id,   products.name,   (SELECT  catalogs.name    FROM  catalogs    WHERE  catalogs.id = products.catalog\_id) AS 'catalog' FROM   products; |

Если подзапрос использует столбец из внешнего запроса, его называют коррелированным. Особенность коррелированных запросов — СУБД вынуждена их вычислять для каждой строки внешнего запроса. Это может быть довольно накладно для объемных таблиц.

|  |
| --- |
| SELECT   products.id,   products.name,   (SELECT MAX(price) FROM products) AS 'max\_price' FROM   products; |

Вложенный запрос, который вычисляет максимальную цену товара не является коррелированным, СУБД выполнит его один раз в начале и в каждую строку будет добавлен результат из заранее выполненного запроса.

До этого момента мы рассматривали вложенные запросы, которые всегда возвращали лишь одно значение. Если там, где СУБД ожидает одно значение, мы попробуем передать несколько, MySQL вернет сообщение об ошибке:

|  |
| --- |
| SELECT   id, name, catalog\_id FROM   products WHERE   catalog\_id = (SELECT id FROM catalogs); ERROR 1242 (21000): Subquery returns more than 1 row |

Чтобы воспользоваться вложенным запросом в таких условиях нам потребуется воспользоваться специальными ключевыми словами, например, **IN**:

|  |
| --- |
| SELECT   id, name, catalog\_id FROM   products WHERE   catalog\_id IN (1, 2); |

Содержимое скобок в таком запросе можно заменить на вложенный запрос:

|  |
| --- |
| SELECT   id, name, catalog\_id FROM   products WHERE   catalog\_id IN (SELECT id FROM catalogs); |

Оператор **IN** используется, если необходимо применить оператор равенства в отношении множеств. Однако, помимо оператора равенства, могут использоваться другие логические операторы: больше, меньше, больше-равно, меньше-равно. Для реализации таких сравнений используется ключевое слово **ANY**.

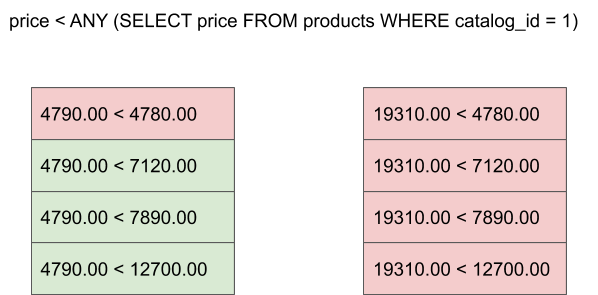
Например, давайте выясним, есть ли среди товаров раздела «Материнские платы» товарные позиции, которые дешевле любой позиции из раздела «Процессоры».

|  |
| --- |
| SELECT   id, name, price, catalog\_id FROM   products WHERE   catalog\_id = 2 AND   price < ANY (SELECT price FROM products WHERE catalog\_id = 1); |

Мы получили только две позиции. Давайте выведем все:

|  |
| --- |
| SELECT id, name, price, catalog\_id FROM products ORDER BY catalog\_id, price; |

Как видим, в результат предыдущего запроса попали позиции 6 и 7, а позиция 5 не попала:



Подзапрос возвращает 4 цены из раздела «Процессоры» и сравнивает каждую из цен с этим списком, если хотя бы в одном случае условие срабатывает, ANY-выражение считается истинным. Для ключевого слова **ANY** существует синоним **SOME**:

|  |
| --- |
| SELECT   id, name, price, catalog\_id FROM   products WHERE   catalog\_id = 2 AND   price < SOME (SELECT price FROM products WHERE catalog\_id = 1); |

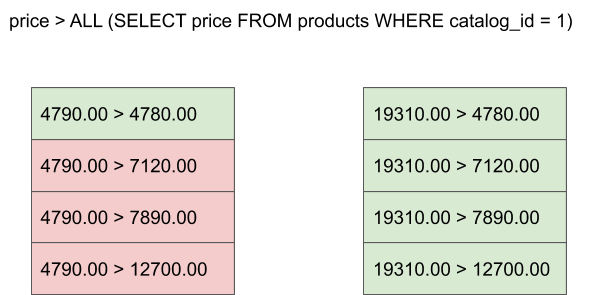
В ключевых словах **ANY** и **SOME** фактически работает логика «или»: если срабатывает хотя бы одно сравнения со множеством значений, выражение считается истинным.

Иногда требуется логика И, когда выражение должно быть истинным, когда все сравнения возвращают истину усли хотя бы одно из сравнению оказалось ложным, весь результат считается ложным. В этом случае используется ключевое слово ALL.

Давайте найдем все товары из раздела «Материнские платы», которые дороже любого товара из раздела «Процессоры»:

|  |
| --- |
| SELECT   id, name, price, catalog\_id FROM   products WHERE   catalog\_id = 2 AND   price > ALL (SELECT price FROM products WHERE catalog\_id = 1); |

Мы получаем единственную товарную позицию с идентификатором 5.



Каждая цена из раздела «Материнские платы» сравнивается с каждой из цен раздела «Процессоры». Если хотя бы одно из выражений является ложным, как в столбце слева, такой товар отбрасывается. В конечную выборку попадает только те товарные позиции, для которых все сравнения оказываются истинными.

Результирующая таблица, которая возвращается вложенным запросом, может быть пустой, т. е., не содержать ни одной строки. Для проверки этого используются ключевые слова **EXISTS** и **NOT EXISTS**.

Давайте извлечем те разделы каталога, для которых имеется хотя бы одна товарная позиция:

|  |
| --- |
| SELECT \* FROM catalogs WHERE EXISTS (SELECT \* FROM products WHERE catalog\_id = catalogs.id); |

Если вложенный запрос возвращает более одной строки, **EXISTS** возвращает истину. **EXISTS** в действительности не использует результаты вложенного запроса, проверяется только число возвращаемых строк.

Это означает, что в списке столбцов, следующих после ключевого слова **SELECT** вложенного запроса, вместо символа **\*** может быть расположено любое допустимое имя — например просто цифра 1:

|  |
| --- |
| SELECT \* FROM catalogs WHERE EXISTS (SELECT 1 FROM products WHERE catalog\_id = catalogs.id); |

Такой запрос выполняется гораздо быстрее.

Допускается использование отрицания **NOT EXISTS**. Давайте извлечем каталоги, для которых нет ни одной товарной позиции:

|  |
| --- |
| SELECT \* FROM catalogs WHERE NOT EXISTS (SELECT 1 FROM products WHERE catalog\_id = catalogs.id); |

До сих пор рассматривались вложенные запросы, возвращающие единственный столбец. Однако в СУБД MySQL реализованы так называемые строчные запросы, которые возвращают более одного столбца.

|  |
| --- |
| SELECT id, name, price, catalog\_id FROM products WHERE (catalog\_id, 5060.00) IN (SELECT id, price FROM catalogs); |

Выражение в скобках перед **IN** называется конструктором строки, его можно записывать с использованием ключевого слова **ROW**:

|  |
| --- |
| SELECT id, name, price, catalog\_id FROM products WHERE ROW(catalog\_id, 5060.00) IN (SELECT id, price FROM catalogs); |

Однако, как правило, это ключевое слово **ROW** опускают.

Вложенные запросы возвращают результирующую таблицу, которая становится предметом дальнейших запросов. Стандарт **SQL** разрешает использование вложенных запросов везде, где допускаются ссылки на таблицы. В частности, вложенный запрос может указываться вместо имени таблицы в предложении **FROM.**

Давайте получим товарные позиции из раздела «Процессоры» следующим образом:

|  |
| --- |
| SELECT \* FROM products WHERE catalog\_id = 1; |

Этот запрос может стать своеобразной промежуточной таблицей, например, давайте извлечем среднюю цену по разделу

|  |
| --- |
| SELECT   AVG(price) FROM   (SELECT \* FROM products WHERE catalog\_id = 1) AS prod; |

Обратите внимание, что в ключевом слове **FROM** мы обязаны назначать вложенному запросу псевдоним при помощи ключевого слова **AS**. Это задачу мы могли бы выполнить и без вложенного запроса:

|  |
| --- |
| SELECT AVG(price) FROM products WHERE catalog\_id = 1; |

Как правило, к вложенным запросам прибегают, когда без них обойтись сложно. Например, если нам требуется вычислить минимальные цены в разделах и получить среднюю минимальную цену. В этом случае мы можем сначала получить минимальные цены разделов:

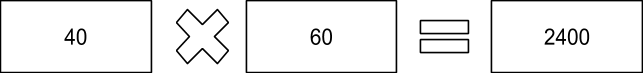
|  |
| --- |
| SELECT catalog\_id, MIN(price) FROM products GROUP BY catalog\_id; |

А потом использовать полученную результирующую таблицу в ключевом слове FROM:

|  |
| --- |
| SELECT   AVG(price) FROM   (SELECT MIN(price) AS price    FROM products    GROUP BY catalog\_id) AS prod; |

# JOIN-соединения таблиц

При соединении получается промежуточная таблица, в котором каждая строка одной таблицы объединяется с каждой строкой другой, создавая тем самым все возможные комбинации строк обеих таблиц.



Результирующая таблица содержит число столбцов, равное сумме столбцов в объединяемых таблицах. Если в первой таблице у нас будет 40 строк, а во второй — 60, то результирующая таблица будет содержать 2400 строк.

Давайте создадим две таблицы — **tbl1** и **tbl2**, которые будут содержать единственный столбец **value**:

|  |
| --- |
| CREATE TABLE tbl1 (   value VARCHAR(255) ); INSERT INTO tbl1 VALUES ('fst1'), ('fst2'), ('fst3');  CREATE TABLE tbl2 (   value VARCHAR(255) ); INSERT INTO tbl2 VALUES ('snd1'), ('snd2'), ('snd3'); |

Давайте посмотрим содержимое таблиц.

|  |
| --- |
| SELECT \* FROM tbl1; SELECT \* FROM tbl2; |

Чтобы создать соединение этих двух таблиц, их имена следует перечислить после ключевого слова **FROM** через запятую.

|  |
| --- |
| SELECT \* FROM tbl1, tbl2; |

Вместо запятой можно использовать ключевое слово **JOIN**:

|  |
| --- |
| SELECT \* FROM tbl1 JOIN tbl2; |



На рисунке видна логика соединения двух таблиц: каждой строке одной таблицы (синий цвет) сопоставляется строка другой таблицы (красный).

У нас в каждой таблице хранится по 3 записи, поэтому результирующая таблица запроса содержит 9 записей. Синим цветом показаны значения из первой таблицы, красным — из второй.

Если мы попробуем явно запросить поле **value**, мы получим сообщение об ошибке:

|  |
| --- |
| SELECT value FROM tbl1, tbl2; |

СУБД не может определить, столбец какой таблицы — **tbl1** или **tbl2** — имеется в виду. Чтобы исключить неоднозначность, можно использовать квалификационные имена:

|  |
| --- |
| SELECT tbl1.value, tbl2.value FROM tbl1, tbl2; |

Для символа звездочки также можно использовать квалификационное имя:

|  |
| --- |
| SELECT tbl1.\*, tbl2.\* FROM tbl1, tbl2; |

В этом случае будут выводиться столбцы всех соединяемых таблиц. Таблицам можно назначать псевдонимы при помощи ключевого слова **AS**:

|  |
| --- |
| SELECT t1.value, t2.value FROM tbl1 AS t1, tbl2 AS t2; |

Такой подход позволяет использовать в качестве имен таблиц более короткие имена. Содержимое промежуточной таблицы можно фильтровать, например при помощи WHERE-условия. Давайте в качестве демонстрации попробуем соединить таблицы **catalogs** и **products**:

|  |
| --- |
| SELECT   p.name, p.price, c.name FROM   catalogs AS c JOIN   products AS p; |

Нам редко требуется выводить всевозможные комбинации строк соединяемых таблиц. Чаще количество строк в результирующей таблице ограничивается при помощи условия.

|  |
| --- |
| SELECT   p.name,   p.price,   c.name FROM   catalogs AS c JOIN   products AS p WHERE   c.id = p.catalog\_id; |

При использовании соединения вместо WHERE-условия используется **ON**:

|  |
| --- |
| SELECT   p.name,   p.price,   c.name FROM   catalogs AS c JOIN products AS p ON   c.id = p.catalog\_id; |

Разница в том, что ON-условие работает в момент соединения, т. е., у нас промежуточная таблица сразу получается небольшой. WHERE-условие всегда действует после соединения, т. е., сначала получается промежуточная таблица с декартовым произведением исходных таблиц и лишь затем осуществляется фильтрация. Поэтому фильтрацию соединений по возможности следует проводить при помощи ON-фильтрации.

Можно делать запросы с участием одной и той же таблицы, назначая ей разные псевдонимы.

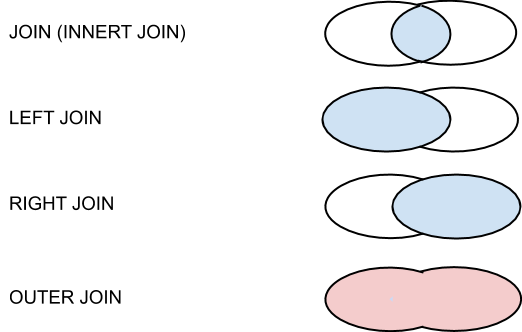
|  |
| --- |
| SELECT   \* FROM   catalogs AS fst JOIN   catalogs AS snd; |

Такие запросы называют самообъединением таблицы. Давайте избавимся от повторов:

|  |
| --- |
| SELECT   \* FROM   catalogs AS fst JOIN   catalogs AS snd ON   fst.id = snd.id; |

Обратите внимание, что названия столбцов в ON-условии совпадают. Различаются только названия таблиц. В этом случае допускается использования ключевого слова **USING**:

|  |
| --- |
| SELECT   \* FROM   catalogs AS fst JOIN   catalogs AS snd USING(id); |



При использовании условий у нас появляется ограничение, согласно которому строки одной таблицы сопоставляются строкам другой. Существует несколько режимов соединения по условию.

Без дополнительных ключевых слов **JOIN** осуществляет перекрестное соединение таблиц, если для записи одной таблицы отсутствует сопоставление в другой таблице. Такая запись отбрасывается.

**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.

**CROSS JOIN** производит соединения записей обеих таблиц, даже если нет подходящего сопоставления. К сожалению, этот тип соединения не поддерживается MySQL. Таким образом, в MySQL различают только три типа соединений.

|  |
| --- |
| SELECT   p.name,   p.price,   c.name FROM   catalogs AS c JOIN   products AS p ON   c.id = p.catalog\_id; |

Давайте рассмотрим **LEFT JOIN** на примере соединения таблиц **catalogs** и **products**:

|  |
| --- |
| SELECT \* FROM catalogs; |

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:

|  |
| --- |
| SELECT   p.name,   p.price,   c.name FROM   catalogs AS c LEFT JOIN   products AS p ON   c.id = p.catalog\_id; |

Однако, если мы заменим **JOIN** на **LEFT JOIN**, в результате появляется раздел «Видеокарты», несмотря на то, что для него нет соответствующих строк в таблице **products**. Недостающие поля заполняются неопределенным значением **NULL**. Порядок таблиц имеет значение, таблица **catalogs** должна располагаться слева от **LEFT JOIN**. Если мы поменяем местами таблицы **catalogs** и **products**, для получения такого результата нам потребуется использовать соединение RIGHT JOIN:

|  |
| --- |
| SELECT   p.name,   p.price,   c.name FROM   products AS p RIGHT JOIN   catalogs AS c ON   c.id = p.catalog\_id; |

Многотабличные запросы можно использовать не только для извлечения, но и для обновления данных, например, если мы захотим снизить цены на 10 % для материнских плат, мы можем воспользоваться следующим UPDATE-запросом:

|  |
| --- |
| UPDATE   catalogs JOIN   products ON   catalogs.id = products.catalog\_id SET   price = price \* 0.9 WHERE   catalogs.name = 'Мат.платы'; |

Схожим образом действует и многотабличное удаление. Однако в нем необходимо явно указать, из каких таблиц мы будем удалять записи:

|  |
| --- |
| DELETE   products, catalogs FROM   catalogs JOIN   products ON   catalogs.id = products.catalog\_id WHERE   catalogs.name = 'Мат.платы'; |

Таким запросом мы удалили 4 записи: 3 товара из таблицы **products** и один раздел «Материнские платы» из таблицы **catalogs**.

Если мы не хотим удалять из таблицы **catalogs** записи, то после ключевого слова **DELETE** мы должны указать только одну таблицу **products.** Давайте удалим процессоры

|  |
| --- |
| DELETE   products FROM   catalogs JOIN   products ON   catalogs.id = products.catalog\_id WHERE   catalogs.name = 'Процессоры'; |

В таблице **products** у нас должны исчезнуть все записи. При этом таблица **catalogs** должна остаться нетронутой.

# Внешние ключи и ссылочная целостность

В SQL довольно много механизмов поддержания целостности данных. Одним из самых важных механизмов является ограничении внешнего ключа.

У нас таблицы **catalogs** и **products** связаны отношением «один ко многим». Одному каталогу могут соответствовать множество товарных позиций, в то время как у каждой товарной позиции может быть только один каталог.

Для такой связи в таблице **products** в поле **catalog\_id** мы храним значение первичного ключа из таблицы **catalogs**.

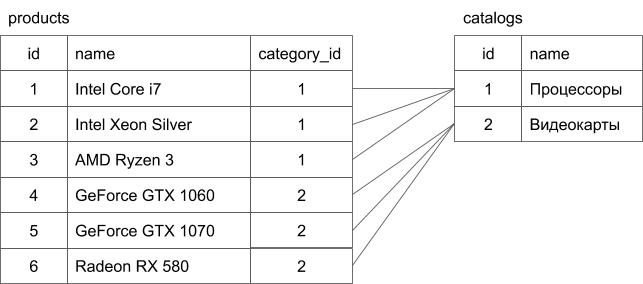
Давайте представим, что мы удаляем каталог с идентификатором 1:

|  |
| --- |
| DELETE FROM catalogs WHERE id = 1; SELECT id, name, price, catalog\_id FROM products; |

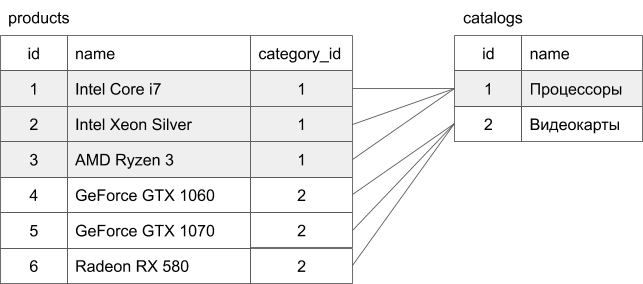
При этом в таблице **products** остаются записи, которые соответствуют данному каталогу. Таким образом, запрос на удаление из таблицы **catalogs** привел к тому, что база данных перестала быть согласованной. Мы нарушили целостность данных.

При удалении таблицы **catalogs** нам необходимо помнить, что требуется внести изменения в таблицу **products**. Например, удалить записи с внешним ключом, который ссылается на несуществующую запись или, как вариант, присвоить внешнему ключу значение **NULL.**

СУБД не знает, какую стратегию вы предпочитаете, поэтому по умолчанию не делает ничего, оставляя решение вопроса на откуп разработчику. У нас довольно простая база данных и помнить о необходимости обновления таблицы **products** не сложно. Однако база данных может содержать десятки и сотни таблиц, которые ссылаются на **catalogs**. Даже если мы все их помним, появляется очень высокая вероятность, что мы ошибемся и забудем выполнить запрос к одной из баз данных. Даже если мы выполним все запросы правильно, на момент выполнения запросов нарушается целостность данных.



Для решения такого рода проблем в SQL предназначено ограничение внешнего ключа. В рамках реляционной модели таблицу с первичным ключом **catalogs** называют предком, а таблицу **catalogs** — потомком:



У нас возможны две ситуации, которые могут приводить к нарушению целостности данных: удаление строки-предка и обновление первичного ключа в строке-предке.

Чтобы задать реакцию на эти ситуации, в таблицу добавляется внешний ключ, ограничение которого задается ключевым словом **FOREIGN KEY**.

|  |
| --- |
| FOREIGN KEY [name\_key] (col1, ...) REFERENCES tbl (tbl\_col, ...) [ON DELETE {CASCADE|SET NULL|NO ACTION|RESTRICT|SET DEFAULT}] [ON UPDATE {CASCADE|SET NULL|NO ACTION|RESTRICT|SET DEFAULT}] |

После ключевого слова **FOREIGN KEY** указывается название ключа и в скобках столбцы, которые играют роль внешнего ключа. После ключевого слова **REFERENCES** указывается имя таблицы и в скобках столбцы, которые играют роль первичного ключа.

Необязательные конструкции **ON DELETE** и **ON UPDATE** позволяют задать поведение СУБД при удалении и обновлении строк из таблицы-предка, соответственно.

После ключевых слов **ON DELETE** и **ON UPDATE** указывается, какое действие нужно предпринять при выполнении **DELETE** и **UPDATE** запросов. Всего предусмотрено пять режимов:

* CASCADE
* SET NULL
* NO ACTION
* RESTRICT
* SET DEFAULT

При использовании ключевого слова **CASCADE** при обновлении или удалении записей в таблице-предке, соответствующие записи в таблице-потомке удаляются или обновляются автоматически.

В случае **SET NULL** при удалении или обновлении записи, содержащей первичный ключ, в таблице-потомке значения устанавливаются в **NULL**.

При использовании действия **NO ACTION** при удалении или обновлении записей, содержащих первичный ключ, с таблицей-потомком никаких дополнительных действий не производится. Мы просто обозначаем логическую связь таблиц, не вводя ограничений.

Ограничение **RESTRICT** приводит к тому, что если в таблице-потомке имеются записи, ссылающиеся на первичный ключ таблицы-предка, при удалении или обновлении записей с таким первичным ключом возвращается ошибка.

СУБД не позволяет изменять или удалять запись с первичным ключом, пока не останется ни одной ссылки из таблицы-потомка.

Последнее ключевое слово **SET DEFAULT** очень похоже на **SET NULL**, только вместо **NULL** устанавливается DEFAULT-значение столбца.

Давайте посмотрим на эти случаи на практике. Воспользуемся оператором **SHOW CREATE TABLE**, чтобы посмотреть структуру таблицы **products**:

|  |
| --- |
| SHOW CREATE TABLE catalogs\G |

Итак, в таблице у нас есть внешний ключ **catalog\_id**, предлагаю добавить ограничение внешнего ключа. Для этого можно воспользоваться **ALTER TABLE**:

|  |
| --- |
| ALTER TABLE products ADD FOREIGN KEY (catalog\_id) REFERENCES catalogs (id) ON DELETE NO ACTION ON UPDATE NO ACTION; |

Мы получаем ошибку, связана она с тем, что типы первичного ключа catalogs и products отличаются. Все идентификаторы у нас имеют тип **BIGINT**, в то время как внешние ключи имеют тип **INT**.

Давайте исправим тип у внешнего ключа **catalog\_id**:

|  |
| --- |
| ALTER TABLE products CHANGE catalog\_id catalog\_id BIGINT UNSIGNED DEFAULT NULL; |

Попробуем добавить внешний ключ повторно:

|  |
| --- |
| ALTER TABLE products ADD FOREIGN KEY (catalog\_id) REFERENCES catalogs (id) ON DELETE NO ACTION ON UPDATE NO ACTION; |

Ключ добавлен. Давайте посмотрим на структуру таблицы:

|  |
| --- |
| SHOW CREATE TABLE products\G |

Итак, у нас появилось ограничение внешнего ключа, которое не предпринимает пока никаких действий. Обратите внимание, что имя ключу назначено автоматически **products\_ibfk\_1**. Это имя нам может пригодиться, если мы захотим удалить ограничение из таблицы:

|  |
| --- |
| ALTER TABLE products DROP FOREIGN KEY products\_ibfk\_1; |

Имя для ограничения **FOREIGN KEY** мы можем задавать и явно. Для этого мы можем его указать после необязательного ключевого слова **CONSTRAINT**:

|  |
| --- |
| ALTER TABLE products ADD CONSTRAINT fk\_catalog\_id FOREIGN KEY (catalog\_id) REFERENCES catalogs (id) ON DELETE NO ACTION ON UPDATE NO ACTION; |

Например, тут мы назначаем в качестве имени **fk\_catalog\_id**. Давайте убедимся, что ограничение успешно добавлено в таблицу:

|  |
| --- |
| SHOW CREATE TABLE products\G |

Теперь, если мы захотим изменить или удалить ограничение, мы можем воспользоваться нашим собственным именем **fk\_catalog\_id**, а не назначенным СУБД MySQL. Действие **NO ACTION** не очень интересно по своему функционалу: что бы ни происходило со связанными таблицами, ограничение не будет срабатывать.

Давайте добавим ограничение **CASCADE**, которое позволяет каскадно удалять и обновлять данные. Опять воспользуемся **ALTER TABLE**.

|  |
| --- |
| ALTER TABLE products DROP FOREIGN KEY fk\_catalog\_id; |

Удалим ограничение **fk\_catalog\_id** и создадим его по новой, указав каскадный режим для операций удаления и обновления:

|  |
| --- |
| ALTER TABLE products ADD CONSTRAINT fk\_catalog\_id FOREIGN KEY (catalog\_id) REFERENCES catalogs (id) ON DELETE CASCADE ON UPDATE CASCADE;  SELECT \* FROM catalogs; |

Итак, у нас есть три каталога, давайте изменим первичный ключ для Процессоров с 1 на 4:

|  |
| --- |
| SELECT id, name, price, catalog\_id FROM products; |

При этом у нас в таблице **products** имеются внешний ключ **catalog\_id**, который ссылается на первичный ключ таблицы **products**. Так как у нас включен каскадный режим обновления, у нас должны обновиться ключи в обеих таблицах.

|  |
| --- |
| UPDATE catalogs SET id = 4 WHERE name = 'Процессоры'; |

Давайте убедимся, что изменения внесены.

|  |
| --- |
| SELECT \* FROM catalogs; SELECT id, name, price, catalog\_id FROM products; |

Мы обновили запись в **catalogs**, а изменения каскадно отразились на таблице **products**. Если мы сейчас удалим раздел «Процессоры», все товары из таблицы products тоже будут удалены.

|  |
| --- |
| DELETE FROM catalogs WHERE name = 'Процессоры';  SELECT \* FROM catalogs; SELECT id, name, price, catalog\_id FROM products; |

Раздел «Процессоры» и все относящиеся к нему товары удалены. Давайте посмотрим, как работает поле ограничения **SET NULL**. Давайте удалим текущее ограничение.

|  |
| --- |
| ALTER TABLE products DROP FOREIGN KEY fk\_catalog\_id; |

И добавим новое:

|  |
| --- |
| ALTER TABLE products ADD CONSTRAINT fk\_catalog\_id FOREIGN KEY (catalog\_id) REFERENCES catalogs (id) ON DELETE SET NULL; |

Обратите внимание, что мы можем устанавливать только одно ограничение, например на удаление, а для обновления задать совершенно другой тип ограничения.

Давайте удалим раздел видеокарт:

|  |
| --- |
| DELETE FROM catalogs WHERE name = 'Мат.платы'; |

Посмотрим на результаты:

|  |
| --- |
| SELECT \* FROM catalogs; SELECT id, name, price, catalog\_id FROM products; |

Обратите внимание, что внешний ключ получил значения **NULL**. Ограничение первичного ключа — далеко не единственный механизм поддержания целостности данных.

Часть механизмов, которые описывает стандарт SQL просто не реализованы в MySQL, например, CHECK-ограничения, Часть поддерживается очень хорошо и будет рассмотрена на следующих уроках.

# Используемые источники

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

**Урок 6**

Транзакции, переменные, представления

[Транзакции](https://docs.google.com/document/d/1T1nFhvp15JOsIYck0iYfRz_YRgZVF8O52q5UbTDr3mE/edit#heading=h.a87pq6wf3svj)

[Уровни изоляции](https://docs.google.com/document/d/1T1nFhvp15JOsIYck0iYfRz_YRgZVF8O52q5UbTDr3mE/edit#heading=h.dp1rf2duo4xh)

[Внутренняя реализация транзакций](https://docs.google.com/document/d/1T1nFhvp15JOsIYck0iYfRz_YRgZVF8O52q5UbTDr3mE/edit#heading=h.is0w7v781akx)

[Журнал транзакций](https://docs.google.com/document/d/1T1nFhvp15JOsIYck0iYfRz_YRgZVF8O52q5UbTDr3mE/edit#heading=h.yqzvcaeoppfw)

[Управление режимом сохранения транзакций](https://docs.google.com/document/d/1T1nFhvp15JOsIYck0iYfRz_YRgZVF8O52q5UbTDr3mE/edit#heading=h.6euq94kedz9q)

[Механизм повышения степени конкурентности MVCC](https://docs.google.com/document/d/1T1nFhvp15JOsIYck0iYfRz_YRgZVF8O52q5UbTDr3mE/edit#heading=h.gtw85a9r63x0)

[Переменные](https://docs.google.com/document/d/1T1nFhvp15JOsIYck0iYfRz_YRgZVF8O52q5UbTDr3mE/edit#heading=h.c84or328wnrq)

[Временная таблица](https://docs.google.com/document/d/1T1nFhvp15JOsIYck0iYfRz_YRgZVF8O52q5UbTDr3mE/edit#heading=h.awqt091wxaa0)

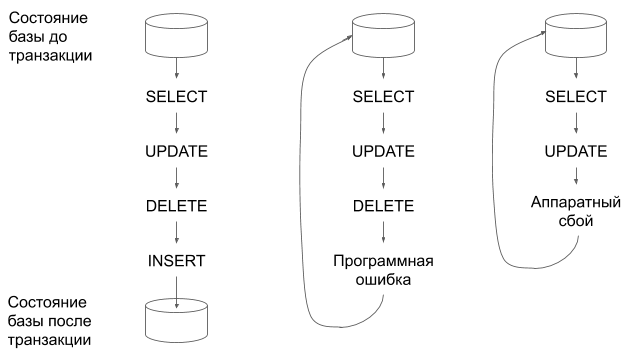
[Динамические запросы](https://docs.google.com/document/d/1T1nFhvp15JOsIYck0iYfRz_YRgZVF8O52q5UbTDr3mE/edit#heading=h.gymf9hkhoy5b)

[Представления](https://docs.google.com/document/d/1T1nFhvp15JOsIYck0iYfRz_YRgZVF8O52q5UbTDr3mE/edit#heading=h.24239ij76vp9)

[Используемые источники](https://docs.google.com/document/d/1T1nFhvp15JOsIYck0iYfRz_YRgZVF8O52q5UbTDr3mE/edit#heading=h.w5vh0mt0cjmc)

# Транзакции

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.



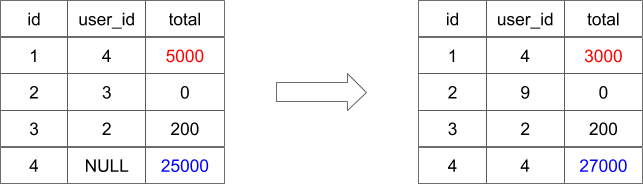
Операции с денежными средствами — классический пример, показывающий, почему необходимы транзакции. Если при оплате покупки происходит перевод от клиента электронному магазину, то счет клиента должен уменьшиться на эту сумму, а счет электронного магазина — увеличиться на нее же.

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.



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 в консоли. Для начала создадим таблицу **accounts**:

|  |
| --- |
| DROP TABLE IF EXISTS accounts; CREATE TABLE accounts (   id SERIAL PRIMARY KEY,   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; |

Перемещаем их на счет интернет-магазина. Давайте посмотрим состояние таблицы **accounts**:

|  |
| --- |
| SELECT \* FROM accounts; |

Изменения в рамках транзакций не сохранены в таблицах. Давайте переключимся в другую консоль и посмотрим состояние таблицы **accounts** глазами другого пользователя:

|  |
| --- |
| SELECT \* FROM accounts; |

Как видим, другие пользователи видят исходное состояние таблицы. Чтобы изменения вступили в силу, мы должны выполнить команду **COMMIT**:

|  |
| --- |
| COMMIT; |

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

|  |
| --- |
| SELECT \* FROM accounts; |

Если в момент, когда мы выполняли транзакцию, какая-то другая транзакция уже изменила счет пользователя, то команда COMMIT завершится ошибкой и все изменения в рамках текущей транзакции будут аннулированы. Мы можем и самостоятельно отменять транзакции.

Например, давайте спишем еще 2000 рублей со счета пользователя:

|  |
| --- |
| START TRANSACTION; |

Начинаем транзакцию:

|  |
| --- |
| 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; |

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

|  |
| --- |
| ROLLBACK; |

Давайте посмотрим состояние таблицы **accounts**:

|  |
| --- |
| SELECT \* FROM accounts; |

Можем видеть, что изменения не отразились на состоянии таблицы.

Для некоторых операторов нельзя выполнить откат при помощи оператора **ROLLBACK**. К их числу относят следующие команды:

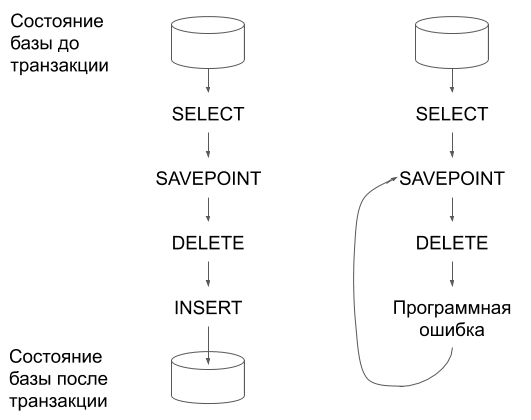
* **CREATE INDEX**
* **DROP INDEX**
* **CREATE TABLE**
* **DROP TABLE**
* **TRUNCATE TABLE**
* **ALTER TABLE**
* **RENAME TABLE**
* **CREATE DATABASE**
* **DROP DATABASE**
* **ALTER DATABASE**

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**

Транзакции не могут быть вложенными, потому что любой оператор, начинающий транзакцию, приводит к завершению предыдущей.



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 SAVEPOINT accounts\_4; |

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.

Пока мы не используем оператор **START TRANSACTION**. В MySQL каждый запрос рассматривается как транзакция. Обычно говорят, что MySQL работает в режиме автозавершения транзакций. Мы можем отключить такой режим при помощи команды:

|  |
| --- |
| SET AUTOCOMMIT=0; |

В этом случае любая последовательность команд будет рассматриваться как транзакция:

|  |
| --- |
| 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; SELECT \* FROM accounts; |

Чтобы сохранить изменения, нужно выполнить команду **COMMIT** или отменить команды при помощи **ROLLBACK**. Например, давайте отменим транзакцию:

|  |
| --- |
| ROLLBACK; SELECT \* FROM accounts; |

Чтобы включить режим автоматического завершения транзакций, необходимо выполнить оператор **SET AUTOCOMMIT=1**.

Транзакций недостаточно, если система не удовлетворяет принципу ACID. Аббревиатура ACID расшифровывается как атомарность, согласованность, изолированность и сохраняемость).

* Atomicy — атомарность.
* Consistency — согласованность.
* Isolation — изолированность.
* Durability — сохраняемость.

Атомарность подразумевает, что транзакция должна функционировать как единая 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 зафиксированными, 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.

## 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, которые считываются 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.

На этом уровне может возникать множество задержек и конфликтов при блокировках. На практике данный уровень изоляции применяется достаточно редко. Изменить уровень изоляции можно при помощи команды **SET TRANSACTION**:

|  |
| --- |
| SET TRANSACTION ISOLATION LEVEL READ COMMITTED; |

# Внутренняя реализация транзакций

Давайте в двух разных сессиях начнем транзакции:

|  |
| --- |
| START TRANSACTION; UPDATE accounts SET total = total - 2000 WHERE user\_id = 4; UPDATE accounts SET total = total + 2000 WHERE user\_id IS NULL; |

|  |
| --- |
| START TRANSACTION; UPDATE accounts SET total = total - 1000 WHERE user\_id = 4; UPDATE accounts SET total = total + 1000 WHERE user\_id IS NULL; |

Как видите, вторая транзакция не выполняет запрос, так как одна транзакция ожидает завершения другой. Только после того как мы введем **COMMIT** или **ROLLBACK**,транзакция продолжит работу:

|  |
| --- |
| COMMIT; |

Возможна ситуация взаимоблокировки, когда две или более транзакции запрашивают блокировку одних и тех же ресурсов. В результате образуется циклическая зависимость: одна транзакция будет до бесконечности ожидать окончания другой, и конфликт не разрешится до тех пор, пока не произойдет какое-то событие, которое снимет взаимную блокировку.

Для разрешения этой проблемы в системах баз данных реализованы различные формы обнаружения взаимоблокировок и таймаутов.

## Журнал транзакций

В базах данных транзакции редко записываются сразу в таблицу. Вместо этого они помещаются в журнал транзакций. Когда происходит какое-либо изменение, оно изменяет не таблицу на жестком диске, а находящуюся в памяти копию данных. Это происходит очень быстро.

Затем подсистема хранения записывает сведения об изменениях в журнал транзакции, который хранится на диске и потому долговечен. Это также относительно быстрая операция, поскольку событие добавляется в конец журнала вместо операций ввода-вывода в разных местах таблицы.

Впоследствии фоновый процесс обновит таблицу на диске. Таким образом, большинство СУБД, использующих журнал транзакций, сохраняют изменения на диске дважды.

Если происходит сбой после того, как внесена соответствующая запись в журнал транзакции, но до того, как обновлены сами данные, подсистема хранения может восстановить изменения после перезапуска сервера.

Запросить параметры журнала транзакций можно при помощи следующего запроса:

|  |
| --- |
| SHOW VARIABLES LIKE 'innodb\_log\_%'; |

Из полученного отчета можно увидеть, что журнал транзакций имеет два файла, размером 50 Мб. По умолчанию файлы журнала транзакций располагаются в каталоге данных.

Получить путь к каталогу данных в вашей системе можно при помощи следующего запроса:

|  |
| --- |
| SHOW VARIABLES LIKE 'datadir'; |

Давайте выйдем из диалогового режима MySQL в консоль командной строки и перейдем в каталог данных:

|  |
| --- |
| cd /usr/local/var/mysql/ ls -la |

Среди файлов мы можем обнаружить два: **ib\_logfile0** и i**b\_logfile1** — это и есть файлы журнала транзакций, все они сначала помещаются сюда. По умолчанию мы используем движок InnoDB, поэтому фактически в настоящий момент мы исследуем его устройство и устройство его реализации транзакций.

InnoDB хранит таблицы всех баз данных в едином табличном пространстве в файле **ibdata1**. Физически единое табличное пространство может располагаться в нескольких файлах. Более того, мы можем выделить отдельное табличное пространство под каждую из таблиц.

Транзакции помещаются в файлы **ib\_logfile0** и **ib\_logfile1** и потом перегоняются в файлы единого табличного пространства: если сервер MySQL останавливается штатно, все транзакции из журнала сохраняются в таблицу. Если происходит сбой и сервер останавливается, например из-за отсутствия питания, перед стартом MySQL проверяет журнал транзакций и перегоняет в единое табличное пространство все транзакции которые не были сохранены в таблицах. Таким образом, потерять сохраненные транзакции невозможно.

Увеличивая размер журнала транзакций, можно укоротить вставку записей при штатной работе MySQL. Однако, чем больше журнал транзакций, тем медленнее будет стартовать сервер MySQL; лучше всего оставить это значение по умолчанию.

## Управление режимом сохранения транзакций

За режим управления сохранения транзакций отвечает переменная **innodb\_flush\_log\_at\_trx\_commit**, которая может принимать следующие значения:

* 0 — сохранение журнала раз в секунду,
* 1 — сохранение после каждой транзакции,
* 2 — сохранение журнала раз в секунду и после каждой транзакции.

Давайте запросим ее состояние:

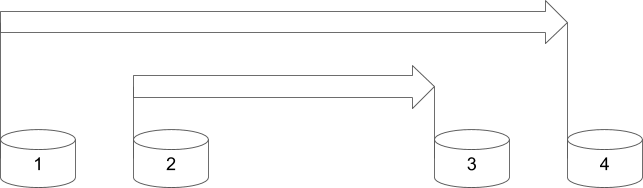
|  |
| --- |
| SHOW VARIABLES LIKE 'innodb\_flush\_log\_at\_trx\_commit'; |

Установить новое значение:

|  |
| --- |
| SET GLOBAL innodb\_flush\_log\_at\_trx\_commit = 0; |

## Механизм повышения степени конкурентности MVCC

Механизм транзакций в MySQL использует не просто механизм блокировки строк, но и механизм повышения степени конкурентности под названием MVCC (multiversion concurrency control — многоверсионное управление конкурентным доступом).



Этот механизм характерен не только для MySQL, но и вообще для реляционных СУБД: его также использует PostgreSQL и Oracle.

Когда мы упоминали уровни изоляции, мы отмечали, что возможны случаи, когда в ходе работы транзакции могут происходить фантомные или невоспроизводимые чтения. Если же мы будем жестко блокировать записи, как это происходит на уровне **SERIALIZABLE**, нам не избежать блокировок даже на операцию чтения.

Механизм MVCC позволяет во многих случаях вообще отказаться от блокировки и способен значительно снизить накладные расходы. Идея механизма MVCC состоит в создании мгновенных снимков состояния. Каждая транзакция читает данные из согласованного снимка состояния, то есть видит данные, которые были зафиксированы в базе на момент начала транзакции.

Даже если данные затем были изменены, каждая транзакция видит только старые данные по состоянию на конкретный момент времени. Выполняющая операцию записи транзакция может блокировать выполнение другой, записывающей в тот же объект. Однако операции чтения не требуют блокировок. Чтение никогда не блокирует запись, а запись — чтение. Благодаря этому база данных способна выполнять длительные запросы на чтение, продолжая в то же время обработку операций записи без какой-либо конкуренции блокировок между ними.

Многоверсионное управление конкурентным доступом MVCC работает только на уровнях изоляции **REPEATABLE READ** и **READ COMMITTED**. Уровень **READ UNCOMMITTED** несовместим с MVCC, поскольку запросы не считывают версию строки, соответствующую их версии транзакции. Они читают самую последнюю версию, несмотря ни на что. Уровень **SERIALIZABLE** несовместим с MVCC, поскольку операции чтения блокируют каждую возвращаемую строку.

# Переменные

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

|  |
| --- |
| SELECT @total := COUNT(\*) FROM products; |

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; |

Переменная будет доступна только в текущей сессии.

|  |
| --- |
| SELECT @total; |

Если мы откроем новую mysql-консоль и попробуем обратиться к переменной **@total**, мы получим неопределенное значение **NULL**. Переменные можно использовать не только в SELECT-списке, но и в WHERE-условии. Давайте извлечем товарную позицию с самой высокой ценой:

|  |
| --- |
| SELECT @price := MAX(price) FROM products; |

Для этого воспользуемся функцией **MAX()** и сохраним максимальную цену товара в переменной **@pirce**.

|  |
| --- |
| SELECT \* FROM products WHERE price = @price; |

После этого мы можем использовать переменную в следующем запросе. Если в качестве значения переменной передается имя столбца, содержащего множество значений, то переменная получит последнее значение:

|  |
| --- |
| SELECT @id := id FROM products; SELECT @id; |

Имена переменных нечувствительны к регистру:

|  |
| --- |
| SELECT @id := 5, @ID := 3; SELECT @id, @ID; |

Несмотря на то, что может сложиться впечатление, что мы присваиваем значения двум разным переменным, мы работаем с одной и той же, и она получает значение 3. Переменные также могут объявляться при помощи оператора **SET**:

|  |
| --- |
| SET @last = NOW() - INTERVAL 7 DAY; SELECT CURDATE(), @last; |

Команда **SET**, в отличие от оператора **SELECT**, не возвращает результирующую таблицу. Переменные можно использовать для нумерации записей в таблице.

Допустим, есть таблица **tbl1** с единственным столбцом **value**, без первичного ключа:

|  |
| --- |
| SELECT \* FROM tbl1; |

И пусть требуется при выводе содержимого таблицы **tbl** пронумеровать строки. Мы можем завести переменную **@start**:

|  |
| --- |
| SET @start := 0; |

И увеличивать значение на единицу по мере вывода записей:

|  |
| --- |
| SELECT @start := @start + 1 AS id, value FROM tbl1; |

Помимо пользовательских переменных, сервер MySQL поддерживает большое количество системных переменных, с помощью которых можно осуществлять его тонкую настройку и масштабирование.

С некоторыми из переменных мы уже имели дело в предыдущих роликах. Получить их полный список можно при помощи оператора **SHOW VARIABLES**:

|  |
| --- |
| SHOW VARIABLES; |

Оператор SHOW поддерживает ключевое слово LIKE, которое ведет себя так же, как в SELECT-запросах. При помощи этого ключевого слова можно отфильтровать выборку

|  |
| --- |
| SHOW VARIABLES LIKE 'read\_buffer\_size'; |

Сервер MySQL поддерживает два типа системных переменных:

* глобальные, которые влияют на сервер,
* и сеансовые, которые влияют на текущее соединение клиента с сервером.

При старте сервера происходит инициализация глобальных переменных значениями по умолчанию. Оператор SET позволяет изменять значения глобальных переменных уже после старта:

|  |
| --- |
| SET GLOBAL read\_buffer\_size = 2097152; |

Для этого перед именем системной переменной устанавливается ключевое слово **GLOBAL**. Вместо ключевого слова **GLOBAL** можно перед названием переменной указывать два символа алеф:

|  |
| --- |
| SET @@global.read\_buffer\_size = 2097152; |

Помимо глобальных переменных, сервер MySQL поддерживает набор сеансовых переменных для каждого соединения клиента с сервером. При установке соединения с сервером сеансовые переменные получают значения, заданные для глобальных переменных. Однако для тех сеансовых переменных, которые являются динамическими, клиент при помощи оператора **SET** может выставлять новые значения:

|  |
| --- |
| SET SESSION read\_buffer\_size = 2097152; |

Для этого перед именем системной переменной устанавливается ключевое слово SESSION. Вместо ключевого слова GLOBAL можно перед названием переменной указывать два символа алеф, после которых указывает префикс session.

|  |
| --- |
| SET @@session.read\_buffer\_size = 2097152; |

Чтобы установить локальной переменной значение глобальной, достаточно присвоить локальной переменной ключевое слово DEFAULT.

|  |
| --- |
| SET read\_buffer\_size = DEFAULT; |

# 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.

Создание временных таблиц осуществляется при помощи необязательного ключевого слова **TEMPORARY**:

|  |
| --- |
| CREATE TEMPORARY TABLE temp (id INT, name VARCHAR(255)); SHOW TABLES; DESCRIBE temp; |

Временные таблицы хранятся в файле **ibtmp1** в каталоге данных.

# Динамические запросы

Динамическими называют запросы, которые, подобно пользовательским переменным, могут быть сохранены под конкретным именем и вызваны позже в течении сессии. Для объявления динамического запроса используется команда **PREPARE**:

|  |
| --- |
| PREPARE ver FROM 'SELECT VERSION()'; |

Выполняется такой динамический запрос при помощи команды EXECUTE

|  |
| --- |
| EXECUTE ver; |

Динамические запросы имеют время жизни только в течение текущего сеанса. Т.е., после того, как соединение с сервером закрыто, динамический запрос перестаёт существовать. Если необходимо, чтобы сохранённый ранее запрос существовал более длительное время, необходимо прибегнуть к представлениям, которые мы рассмотрим на следующем уроке.

Запросы можно параметризовать, используя для этого символ вопросительного знака. Давайте создадим динамический запрос, который извлекает товарные позиции одного из разделов интернет-магазина:

|  |
| --- |
| PREPARE prd FROM 'SELECT id, name, price FROM products WHERE catalog\_id = ?'; |

Как видно, вместо значения внешнего ключа **catalog\_id** используется знак вопроса. Давайте зададим идентификатор раздела при помощи переменной:

|  |
| --- |
| SET @catalog\_id = 1; |

При вызове запроса при помощи оператора **EXECUTE** передать значение параметра можно при помощи конструкции **USING**.

|  |
| --- |
| EXECUTE prd USING @catalog\_id; |

Динамический запрос может иметь более одного параметра, в этом случае они перечисляются после ключевого слова **USING** через запятую в том порядке, в котором они встречаются в динамическом запросе.

Динамические запросы имеют ряд ограничений: не допускается использование вложенных динамических запросов, а также нескольких сразу. Т.е., динамический запрос всегда представляет лишь один запрос. Параметр всегда передаёт строку, т.е., динамически задать имя таблицы или столбца не получится.

Удалить динамический запрос можно при помощи оператора **DROP PREPARE**:

|  |
| --- |
| DROP PREPARE prd; |

# Представления

Основными структурными единицами в реляционных базах данных являются таблицы. Однако язык запросов SQL предоставляет еще один способ организации данных — представления.

Представление — это запрос на выборку (**SELECT**), которому присваивается уникальное имя и который можно сохранять или удалять из базы данных как обычную хранимую процедуру.

Представления позволяют увидеть результаты сохраненного запроса таким образом, как будто это полноценная таблица базы данных. Представления позволяют более гибко управлять правами доступа к таблицам: можно запретить прямое обращение пользователей к таблицам, и разрешить доступ только к представлениям.

Представления позволяют также обеспечить обратную совместимость для программ, ориентирующихся на старую структуру базы данных — достаточно создать представления со структурой, соответствующей старым таблицам.

|  |
| --- |
| SELECT \* FROM catalogs; |

Давайте создадим представления таблицы **catalogs**, в котором записи будут поддерживаться в отсортированном состоянии.

|  |
| --- |
| CREATE VIEW cat AS SELECT \* FROM catalogs ORDER BY name; |

Для создания представления используется команда **CREATE VIEW**, после которой мы указываем имя представления **cat**. Затем после ключевого слова **AS** пишем запрос представления.

К представлению мы можем обращаться как к обычной таблице:

|  |
| --- |
| SELECT \* FROM cat; |

Мы получили список разделов интернет-магазина в отсортированном состоянии:

|  |
| --- |
| SHOW TABLES; |

Представление рассматривается MySQL как полноценная таблица, оно появляется в списке таблиц команды **SHOW TABLES**.

В полученном представлении мы использовали **\***, поэтому оно принимает структуру таблицы **catalogs.** Однако при создании представления можно явно указать список столбцов и даже изменить их названия и порядок следования:

|  |
| --- |
| CREATE VIEW cat\_reverse (catalog, catalog\_id) AS SELECT name, id FROM catalogs; SELECT \* FROM cat\_reverse; |

Мы поменяли название столбца **name** на **catalog**, а **id** — на **catalog\_id**. При этом порядок следования столбцов меняется на обратный. При формировании списка столбцов представления задаются только имена столбцов, а тип данных, их размер и другие характеристики берутся из определения столбца исходной таблицы.

В качестве столбцов представления могут выступать вычисляемые столбцы. Создадим представление **namecat**, которое выводит столбцы **id** и **name** таблицы **catalogs**, и дополнительно столбец **total,** в который будет помещаться количество символов в имени каталога.

|  |
| --- |
| CREATE OR REPLACE VIEW namecat (id, name, total) AS SELECT id, name, LENGTH(name) FROM catalogs; |

Обратите внимание мы используем команду **CREATE OR REPLACE VIEW**, чтобы заменить уже существующее представление:

|  |
| --- |
| SELECT \* FROM namecat ORDER BY total DESC; |

Ключевое слово **ALGORITHM** определяет способ формирования конечного запроса с участием представления и может принимать три значения:

* **MERGE** — при использовании данного алгоритма запрос объединяется с представлением таким образом, что представление заменяет собой соответствующие части в запросе;
* **TEMPTABLE** — результирующая таблица представления помещается во временную, которая затем используется в конечном запросе;
* **UNDEFINED** — в данном случае СУБД MySQL самостоятельно пытается выбрать алгоритм, предпочитая использовать подход **MERGE** и прибегая к алгоритму **TEMPTABLE** (создание временной таблицы) только в случае необходимости, т. к. метод **MERGE** более эффективен.

Если ни одно из значений **ALGORITHM** не указано, по умолчанию назначается **UNDEFINED**.

|  |
| --- |
| CREATE ALGORITHM = TEMPTABLE VIEW cat2 AS SELECT \* FROM catalogs; |

В созданном представлении мы требуем от MySQL при каждом обращении к представлению создавать временную таблицу.

|  |
| --- |
| DESCRIBE products; |

Следует отметить, что представления способны скрывать ряд столбцов за счёт того, что SELECT-запросы могут извлекать не все столбцы таблицы. Такие представления называются вертикальными представлениями.

Давайте создадим такое представление для таблицы **products**:

|  |
| --- |
| CREATE OR REPLACE VIEW prod AS SELECT id, name, price, catalog\_id FROM products ORDER BY catalog\_id, name;  SELECT \* FROM prod; |

Запросы к представлениям сами могут содержать условия **WHERE** и собственные сортировки. Правильное составление запроса к исходным таблицам — это забота MySQL.

|  |
| --- |
| SELECT \* FROM prod ORDER BY name DESC; |

Наряду с вертикальными используются горизонтальные представления, которые ограничивают доступ пользователей к строкам таблиц, делая видимыми только те строки, с которыми они работают.

Давайте создадим представление, которое извлекает из таблицы **products** только процессоры:

|  |
| --- |
| CREATE OR REPLACE VIEW processors AS SELECT id, name, price, catalog\_id FROM products WHERE catalog\_id = 1;  SELECT \* FROM processors; |

В реальной практике могут встречаться смешанные представления, которые ограничивают таблицу и по горизонтали, и по вертикали. Термины «горизонтальное представление» и «вертикальное представление» являются условными и предназначены, чтобы лучше понять, как из исходной таблицы формируется представление.

Чтобы в представление можно было вставлять новые записи при помощи команды **INSERT** и обновлять существующие записи при помощи команды **UPDATE**, необходимо при создании представления использовать конструкцию **WITH CHECK OPTION**.

Во время вставки происходит проверка, чтобы вставляемые данные удовлетворяли WHERE-условию SELECT-запроса, лежащего в основе представления.

|  |
| --- |
| SELECT \* FROM tbl1; |

Воспользуемся таблицей **tbl1** для создания обновляемого представления:

|  |
| --- |
| CREATE VIEW v1 AS SELECT \* FROM tbl1 WHERE value < 'fst5' WITH CHECK OPTION;  INSERT INTO v1 VALUES ('fst4'); |

Значение успешно вставилось.

|  |
| --- |
| INSERT INTO v1 VALUES ('fst5'); ERROR 1369 (HY000): CHECK OPTION failed 'shop.v1' |

Однако при попытке вставить значение **fst5** срабатывает ограничение WHERE-условия.

Отредактировать представление можно при помощи команды **ALTER**:

|  |
| --- |
| ALTER VIEW v1 AS SELECT \* FROM tbl1 WHERE value > 'fst4' WITH CHECK OPTION; |

Кроме того, можно воспользоваться командой **CREATE OR REPLACE VIEW**:

|  |
| --- |
| CREATE OR REPLACE VIEW v1 AS SELECT \* FROM tbl1 WHERE value > 'fst4' WITH CHECK OPTION; |

Для удаления представлений предназначена команда **DROP VIEW**.

|  |
| --- |
| DROP VIEW cat, cat\_reverse, namecat, prod, processors, v1; |

При попытке удаления не существующего представления возникает ошибка:

|  |
| --- |
| DROP VIEW cat, cat\_reverse, namecat, prod, processors, v1; |

Как и многие команды, **DROP VIEW** поддерживает ключевое слово **IF EXISTS**, позволяющее игнорировать попытки удаления несуществующих представлений:

|  |
| --- |
| DROP VIEW IF EXISTS cat, cat\_reverse, namecat, prod, processors, v1; |

# 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.

**Урок 7**

Администрирование MySQL

[Параметры запуска сервера](https://docs.google.com/document/d/1HyS74SgaPV5IC0VKdRXkIi9JlDTb97Kh0c7Ln0QHjiY/edit#heading=h.a87pq6wf3svj)

[Конфигурационный файл my.cnf](https://docs.google.com/document/d/1HyS74SgaPV5IC0VKdRXkIi9JlDTb97Kh0c7Ln0QHjiY/edit#heading=h.xiva90oaeng2)

[Переменные сервера](https://docs.google.com/document/d/1HyS74SgaPV5IC0VKdRXkIi9JlDTb97Kh0c7Ln0QHjiY/edit#heading=h.meswlw5s6ps)

[Журнальные файлы](https://docs.google.com/document/d/1HyS74SgaPV5IC0VKdRXkIi9JlDTb97Kh0c7Ln0QHjiY/edit#heading=h.38qog18hyw3d)

[Журнал ошибок](https://docs.google.com/document/d/1HyS74SgaPV5IC0VKdRXkIi9JlDTb97Kh0c7Ln0QHjiY/edit#heading=h.10ucuwz795gq)

[Журнал медленных запросов](https://docs.google.com/document/d/1HyS74SgaPV5IC0VKdRXkIi9JlDTb97Kh0c7Ln0QHjiY/edit#heading=h.2vgp5bm8bcwc)

[Бинарный журнал](https://docs.google.com/document/d/1HyS74SgaPV5IC0VKdRXkIi9JlDTb97Kh0c7Ln0QHjiY/edit#heading=h.z2af3zdn66lc)

[Соединения](https://docs.google.com/document/d/1HyS74SgaPV5IC0VKdRXkIi9JlDTb97Kh0c7Ln0QHjiY/edit#heading=h.u459s2h8n9cu)

[Права пользователей](https://docs.google.com/document/d/1HyS74SgaPV5IC0VKdRXkIi9JlDTb97Kh0c7Ln0QHjiY/edit#heading=h.dvak9lfj9ug1)

[Учетная запись](https://docs.google.com/document/d/1HyS74SgaPV5IC0VKdRXkIi9JlDTb97Kh0c7Ln0QHjiY/edit#heading=h.fw38r8xhvaya)

[Управление привилегиями](https://docs.google.com/document/d/1HyS74SgaPV5IC0VKdRXkIi9JlDTb97Kh0c7Ln0QHjiY/edit#heading=h.xiw07q1yh22)

[Репликация](https://docs.google.com/document/d/1HyS74SgaPV5IC0VKdRXkIi9JlDTb97Kh0c7Ln0QHjiY/edit#heading=h.xu5sqt3hntrm)

[Внутреннее устройство репликации](https://docs.google.com/document/d/1HyS74SgaPV5IC0VKdRXkIi9JlDTb97Kh0c7Ln0QHjiY/edit#heading=h.rq5dngp6gcwc)

[Ограничения масштабирования](https://docs.google.com/document/d/1HyS74SgaPV5IC0VKdRXkIi9JlDTb97Kh0c7Ln0QHjiY/edit#heading=h.25m8aig1qwqh)

[Шардирование](https://docs.google.com/document/d/1HyS74SgaPV5IC0VKdRXkIi9JlDTb97Kh0c7Ln0QHjiY/edit#heading=h.o656h829c232)

[Типы репликации](https://docs.google.com/document/d/1HyS74SgaPV5IC0VKdRXkIi9JlDTb97Kh0c7Ln0QHjiY/edit#heading=h.njpwmstlah5r)

[Используемые источники](https://docs.google.com/document/d/1HyS74SgaPV5IC0VKdRXkIi9JlDTb97Kh0c7Ln0QHjiY/edit#heading=h.w5vh0mt0cjmc)

# Параметры запуска сервера

Сервер MySQL — сложный программный продукт. Он должен работать одинаково эффективно в совершенно разных условиях и решать задачи различного характера. Поэтому сервер имеет гибкие настройки, которые позволяют настроить его под свои нужды без перекомпиляции исходного кода.

У сервера MySQL есть два главных способа настройки: это параметры запуска, которые определяют режимы его работы, и переменные состояния, с помощью которых можно его масштабировать.

Часть из переменных состояния могут быть изменены при помощи параметров запуска. В разных операционных системах для просмотра параметров запуска используются разные средства. Например в UNIX-подобных операционных системах для этого используется команда:

|  |
| --- |
| ps -aux | grep mysqld |

Мы можем видеть параметр **--basedir=/usr/local/opt/mysql**.

Каталог данных: **--datadir=/usr/local/var/mysql**.

Лог ошибок **--log-error=mac422rih.local.err**.

Pid файл **--pid-file=mac422rih.local.pid**.

В Windows посмотреть начальные параметры можно в свойствах сервиса, при помощи которого осуществляется управление сервером MySQL. Параметров сервера MySQL достаточно много, поэтому их может накапливаться изрядное количество. Просмотреть их полный список можно, передав серверу MySQL параметры **--verbose --help**.

У части параметров есть сокращения: они начинаются с одного дефиса, а не с двух. Например, параметр **--user** имеет сокращение **-u**.

# Конфигурационный файл my.cnf

Параметры запуска можно не прописывать непосредственно в строке запуска сервера, а поместить в конфигурационный файл my.cnf, из которого сервер прочитает их при старте:

* /etc/my.cnf
* /etc/mysql/my.cnf
* $MYSQL\_HOME/my.cnf
* [datadir]/my.cnf
* ~/.my.cnf

Следует отметить, что внутри конфигурационного файла **my.cnf** параметры указываются без предваряющих дефисов (они не нужны, так как каждый параметр располагается на отдельной строке). Если строка начинается с символа диеза **#** или точки с запятой **;**, то вся строка относится к комментарию. Параметры в конфигурационном файле называют директивами.

В операционной системе Windows конфигурационный файл может иметь как расширение **ini** (такой файл может располагаться в директориях **C:\Windows\my.ini** и в **C:\mysql5\my.ini**), так и расширение **cnf**. В UNIX-подобных операционных системах конфигурационный файл имеет, как правило, расширение **cnf**.

|  |  |
| --- | --- |
| **Название секции** | **Описание** |
| [mysqld] | Сервер MySQL |
| [server] | Сервер MySQL |
| [mysqld-5.7] | Сервер MySQL версии 5.7 |
| [mysqld-5.8] | Сервер MySQL версии 5.8 |
| [client] | Любая клиентская утилита |
| [mysql] | Консольный клиент mysql |
| [mysqldump] | Утилита создания дампов mysqldump |

Конфигурационный файл влияет на работу не только сервера MySQL, но и вспомогательных утилит, таких как консольный клиент mysql, утилита создания SQL-дампов **mysqldump** и т.п. Более того, один конфигурационный файл может управлять работой нескольких серверов MySQL. Поэтому содержимое конфигурационного файла разделено на секции, которые имеют вид [имя\_секции].

Имя секции определяют утилиту или сервер, к которой будут относиться перечисленные ниже директивы, до тех пор, пока не встретится новая секция или конец файла.

Наличие секции **[mysqld]** и специальных секций для разных версий обусловлены тем, что с каждой новой версией появляется всё больше и больше параметров запуска, и, если конфигурационный файл управляет несколькими серверами, то ряд директив будут одинаковыми для всех серверов, а ряд — уникальны для каждой из версий.

Точно такая же ситуация с секцией [client] и секциями для каждой из утилит. Дело в том, что каждая из утилит обладает сходными параметрами (например, параметры соединения с серверами у всех одинаковые) и в то же время имеет уникальные параметры, характерные только для её функциональных возможностей.

|  |
| --- |
| /usr/local/etc/my.cnf  # Порт TCP/IP который прослушивает MySQL сервер, порт 3306 является # стандартным, однако можно назначить любой другой не занятый порт port=3307 |

Давайте запустим MySQL на порту, отличном от 3306. Для остановки сервера можно использовать команду:

|  |
| --- |
| sudo /usr/local/opt/mysql/support-files/mysql.server stop |

Для повторного запуска:

|  |
| --- |
| sudo /usr/local/opt/mysql/support-files/mysql.server start |

для перезапуска можно использовать команду restart

|  |
| --- |
| sudo /usr/local/opt/mysql/support-files/mysql.server restart |

В дистрибутиве Linux Ubuntu для остановки сервера можно воспользоваться командой:

|  |
| --- |
| service mysql stop |

Для запуска:

|  |
| --- |
| service mysql start |

Для перезапуска:

|  |
| --- |
| service mysql start |

В Windows проще всего воспользоваться оснасткой сервисов. В сервисах выбрать тот, который был заведен при установке MySQL и в контекстном меню выбрать соответствующий пункт на запуск, остановку или перезапуск сервера.

Параметрам в конфигурационном файле, как правило, выставлены разумные значения. Поэтому большинство из них можно вообще не упоминать в конфигурационном файле.

# Переменные сервера

Ознакомиться с текущими значениями параметров можно при помощи команды **SHOW VARIABLES**. Например, формат даты и времени заданы переменным **date\_format**, **datetime\_format** и **time\_format**:

|  |
| --- |
| SHOW VARIABLES LIKE 'date%\_format'; SHOW VARIABLES LIKE 'time%\_format'; |

Текущий часовой пояс берется из операционной системы:

|  |
| --- |
| SHOW VARIABLES LIKE 'system\_time\_zone'; |

Однако при необходимости может быть изменен в конфигурационном файле **my.cnf**.

На одном из предыдущих уроков мы с вами знакомились с уровнями изоляции. Узнать текущий уровень мы можем при помощи следующей команды:

|  |
| --- |
| SHOW VARIABLES LIKE 'tx\_isolation'; |

А используя директиву **tx\_isolation**, можно изменить уровень изоляции в конфигурационном файле **my.cnf**.

Чаще всего изменяются параметры различных буферов и кэшей. Например, переменная **tmp\_table\_size** позволяет узнать максимальный размер временной таблицы. Если промежуточная или временная таблица не помещается в этот размер, вместо оперативной памяти она будет размещена на жестком диске.

|  |
| --- |
| SHOW VARIABLES LIKE 'tmp\_table\_size'; |

Давайте изменим этот размер. В конфигурационном файле не обязательно прописывать размер вплоть до байта. Мы можем использовать сокращения, килобайты, мегабайты и гигабайты.

|  |
| --- |
| tmp\_table\_size=32M |

Следует следить, чтобы суффиксы набирались в английской раскладке, иначе сервер не сможет перезапуститься. Итак давайте перезагрузим сервер

|  |
| --- |
| sudo /usr/local/opt/mysql/support-files/mysql.server restart |

Запрашиваем размер временной таблицы:

|  |
| --- |
| SHOW VARIABLES LIKE 'tmp\_table\_size'; |

Изменить размер можно так же при помощи команды SET

|  |
| --- |
| SET SESSION tmp\_table\_size = 16777216; |

Здесь мы устанавливаем размер оперативной памяти, отводимой под временные таблицы, равный 16М, только в рамках текущей сессии. Если мы выйдем из клиента, снова зайдем и запросим результат, сессионные данные обнулятся:

|  |
| --- |
| SHOW VARIABLES LIKE 'tmp\_table\_size'; |

Мы можем установить значение для всего сервера, используя ключевое слово GLOBAL

|  |
| --- |
| SET GLOBAL tmp\_table\_size = 16777216; |

Если мы после этого выйдем из клиента и снова зайдем, значение уже не изменится, оно будет сброшено только после перезапуска сервера.

Давайте посмотрим еще на одну системную переменную — **auto\_increment\_increment**. Системная переменная предназначена для установки приращения в механизме **AUTO\_INCRENENT** в таблицах.

|  |
| --- |
| SHOW VARIABLES LIKE 'auto\_increment\_increment'; |

Переменная может принимать значения от 1 до 65535 и по умолчанию получает значение 1. Установка значение переменной, например, в 10 приводит к тому, что значение автоинкрементного счётчика увеличивается не на единицу, а сразу на 10.

Не все переменные можно устанавливать глобально или сессионно. Какие из них в каком режиме можно устанавливать, следует уточнять в документации.

# Журнальные файлы

MySQL поддерживает несколько видов журнальных файлов, в которых регистрируются различные события, происходящие на MySQL сервере. В журнал ошибок помещаются сообщения обо всех ошибках, происходящих во время запуска, работы или остановки MySQL сервера.

Общий журнал запросов позволяет регистрировать все выполненные запросы.

Бинарный журнал регистрирует все операторы, которые приводят к изменению данных. Данный журнал используется для репликации и восстановления данных.

В журнал медленных запросов заносятся все запросы, выполнение которых потребовало больше времени, чем указано в системной переменной **long\_query\_time**. По умолчанию это 10 секунд.

Среди этих файлов только бинарный журнал не является текстовым, для его просмотра необходима специальная утилита **mysqlbinlog**. Все остальные файлы текстовые и могут быть просмотрены обычными средствами.

## Журнал ошибок

Давайте начнем изучение с журнала ошибок. Получить путь к нему можно, запросив переменную **log\_error**:

|  |
| --- |
| SHOW VARIABLES LIKE 'log\_error'; |

Как видим, журнальный файл ошибок находится в каталоге данных. Давайте выйдем из диалогового режима MySQL в консоль командной строки и перейдем в каталог данных:

|  |
| --- |
| cd /usr/local/var/mysql/ ls -la |

Здесь мы можем обнаружить err-файл **mac422rih.local.err**. У вас название файла может отличаться, так как по умолчанию его имя совпадает с именем компьютера. Посмотрим последние 20 строк этого файла при помощи команды **tail** (можно использовать любое другое средство просмотра):

|  |
| --- |
| tail -20 mac422rih.local.err |

Здесь мы можем видеть последние сообщения от сервера. В штатном режиме тут будут информационные сообщения, сопровождающие перезапуск сервера. Однако, если сервер не стартует именно отсюда, следует начинать расследование причин неполадок.

Общий журнал запросов позволяет регистрировать все соединения клиентов и их SQL-запросы в текстовом формате.

|  |
| --- |
| SHOW VARIABLES LIKE 'general\_log%'; |

По соображениям производительности журнал запросов отключен. Для того, чтобы его включить его ведение в конфигурационном файле my.cnf

|  |
| --- |
| general\_log=ON |

Теперь нужно перезагрузить сервер:

|  |
| --- |
| sudo /usr/local/opt/mysql/support-files/mysql.server restart |

## Журнал медленных запросов

Журнал медленных запросов по умолчанию тоже отключен:

|  |
| --- |
| SHOW VARIABLES LIKE 'slow\_query\_log%'; SHOW VARIABLES LIKE 'long\_query\_time'; |

Запрос считается медленным, если время его выполнения превысило значение, заданное параметром **long\_query\_time**. Значение этого параметра можно при желании изменить в конфигурационном файле или при помощи команды **SET**.

|  |
| --- |
| slow\_query\_log=ON |

Команда записывается в журнал медленных запросов только после того, как она была выполнена, и после того, как сняты все блокировки, поэтому порядок размещения операторов в журнале может отличаться от порядка, в котором они выполнялись.

Журнал медленных запросов может использоваться с целью выявления запросов, на выполнение которых ушло слишком много времени, а значит требующих оптимизации.

Давайте воспользуемся функцией BENCHMARK(), чтобы организовать медленный запрос. Функция принимает в качестве первого аргумента число, в качестве второго — запрос. Этот запрос выполняется столько раз, сколько указано в первом аргументе:

|  |
| --- |
| SELECT BENCHMARK(1000000000, (SELECT COUNT(\*) FROM products)); |

Итак, запрос выполняется 25 секунд, давайте посмотрим содержимое журнала медленных запросов.

|  |
| --- |
| cat mac422rih-slow.log |

Здесь у нас только один запрос, когда их много, разобраться в журнале медленных запросов не так просто. Поэтому предварительно лучше прогнать журнал через команду **mysqldumpslow**, которая отобразит список всех встречающихся в журнале запросов.

|  |
| --- |
| mysqldumpslow mac422rih-slow.log |

По умолчанию сервер размещает журналы общих и медленных запросов в файлах. Однако журналы могут быть записаны и в специальные таблицы **general\_log** и **slow\_log** системной базы данных **mysql**.

|  |
| --- |
| SELECT \* FROM mysql.general\_log; SELECT \* FROM mysql.slow\_log; |

Сейчас таблицы пусты. MySQL принимает решение о том размещать журналы в файлах или в базе данных, ориентируясь на значение системной переменной **log\_output.**

|  |
| --- |
| SHOW VARIABLES LIKE 'log\_output'; |

По умолчанию переменная принимаете значение **FILE:**

* TABLE
* FILE
* NONE

Однако журналы могут быть записаны как в файл, так и в специальные таблицы **general\_log** и **slow\_log** системной базы данных **mysql**. Если присвоить переменной значение **TABLE**, записи журналов общих и медленных запросов будут помещаться в таблицах **general\_log** и **slow\_log** базы данных **mysql**. Если присвоить значение **FILE**, как сейчас, записи будут вестись в текстовые файлы. Если присвоить значение **NONE**, записи не сохраняются.

Давайте отредактируем конфигурационный файл **my.cnf**:

|  |
| --- |
| log\_output=TABLE |

Снова выполним наш медленный запрос:

|  |
| --- |
| SELECT BENCHMARK(1000000000, (SELECT COUNT(\*) FROM products)); |

Обратимся к таблице slow\_log базы данных mysql:

|  |
| --- |
| SELECT \* FROM mysql.slow\_log\G |

Теперь все запросы будут фиксироваться в системной базы данных **mysql**, а не в файлах каталога данных.

## Бинарный журнал

Бинарный журнал регистрирует все запросы, которые приводят к изменению состояния базы данных. Однако в журнал заносятся только те запросы, которые действительно обновляют данные.

Так, операторы **UPDATE** и **DELETE**, конструкция **WHERE** которых не соответствует ни одной строке, в журнале не регистрируются. Также пропускаются даже те операторы **UPDATE**, которые присваивают столбцу уже имеющиеся у него значение.

Основное назначение бинарного журнала — предоставить возможность обновить базу данных во время операции восстановления настолько подробно, насколько это возможно. Так как именно в бинарном журнале будут содержаться все изменения, произошедшие после того, как была сделана резервная копия всей системы.

Обычно резервную копию с базы данных снимают раз в сутки. Если происходит сбой, база данных восстанавливается из резервной копии, а изменения, которые были сделаны с момента возникновения сбоя беруться из бинарного журнала.

Узнать, включены ли бинарные логи, можно при помощи переменной **log\_bin**:

|  |
| --- |
| SHOW VARIABLES LIKE 'log\_bin'; |

Давайте их включим в конфигурационном файле my.cnf

|  |
| --- |
| log\_bin=ON server-id=1 |

Кроме того, нам потребуется присвоить серверу уникальный идентификатор при помощи директивы **server-id**. Этот идентификатор используется, чтобы отличать один сервер от другого, когда они связаны репликацией. Репликацию мы более подробно рассмотрим далее на уроке.

В каталоге данных у нас появилось два файла:

* ON.index
* ON.000001

Посмотрим содержимое индексного файла.

|  |
| --- |
| cat ON.index |

Индексный файл содержит список бинарных файлов, которых может быть несколько. Сервер MySQL присоединяет в конец имени файла бинарного журнала расширение в виде номера. Этот номер увеличивается каждый раз при запуске сервера или выполнении команд по очистке журналов.

Новый бинарный журнал создается автоматически, когда размер текущего журнала достигает максимума, заданного в системной переменной **max\_binlog\_size**. В бинарный журнал записываются все изменения базы данных.

Давайте запустим на выполнение SQL-дамп **shop.sql**:

|  |
| --- |
| ls -la | grep ON |

Размер файла **ON.000001** вырос. Файл бинарный, просто так посмотреть его содержимое мы не можем. Однако мы можем преобразовать его в текстовую форму при помощи утилиты **mysqlbinlog**:

|  |
| --- |
| mysqlbinlog ON.000001 |

Мы получаем тестовое содержимое, готовое к выполнению в клиенте **mysql**. Вся вспомогательная информация оформлена в виде SQL-комментариев.

## Соединения

Количество соединений, которые может одновременно обслуживать MySQL, ограничено системной переменной **max\_connections**:

|  |
| --- |
| SHOW VARIABLES LIKE 'max\_connections'; |

При желании ее можно изменить в конфигурационном файле **my.cnf**. Однако делать это следует осмотрительно, так как как каждое соединение MySQL потребляет довольно много оперативной памяти. Иногда это мегабайты или даже десятки мегабайт на одно соединение. Если выставить большое значение и количество соединений будет слишком велико, можно исчерпать всю оперативную память сервера. Отслеживать текущие соединения можно при помощи команды:

|  |
| --- |
| SHOW PROCESSLIST; |

Каждая строка результирующей таблицы, возвращаемой оператором **SHOW PROCESSLIST**, соответствует одному соединению с сервером.

Столбец **Time** указывает время соединения. Большое время, как тут, обычно указывает на постоянные соединения от консольных и графических клиентов, либо на зависшие или заблокированные запросы, которые ожидают освобождения ресурсов.

В столбце **Info** указывается статус соединения. Если мы хотим снять какое-то соединение, мы можем воспользоваться командой **KILL**, передав ей идентификатор соединения из первого столбца:

|  |
| --- |
| KILL 1 |

# Права пользователей

СУБД MySQL — многопользовательская среда. Это означает, что для доступа к таблицам базы данных могут быть созданы различные учетные записи с разным уровнем привилегий. Создать новую учетную запись можно при помощи оператора **CREATE USER**:

|  |
| --- |
| CREATE USER foo; |

Если пароль не указывается, по умолчанию в его качестве выступает пустая строка. После того, как новая учетная запись создана, в любом клиенте можно авторизоваться с именем пользователя.

|  |
| --- |
| $ mysql -u foo mysql> SELECT USER(); |

Чтобы задать пароль, нужно использовать ключевое слово **IDENTIFIED BY**, за которым в одиночных кавычках следует пароль.

|  |
| --- |
| CREATE USER shop IDENTIFIED WITH sha256\_password BY 'pass'; $ mysql -u shop -p mysql> SELECT USER(); |

Однако созданные при помощи оператора **CREATE USER** учетные записи не обладают привилегиями. C использованием такой учетной записи невозможно просматривать таблицы и осуществлять запросы. Для наделения учетной записи привилегиями необходимо воспользоваться оператором **GRANT**. Получить список пользователей можно, обратившись к таблице users системной базы данных shop:

|  |
| --- |
| SELECT Host, User FROM mysql.user; |

Оператор **DROP USER** позволяет удалить учетную запись:

|  |
| --- |
| DROP USER foo; SELECT Host, User FROM mysql.user; |

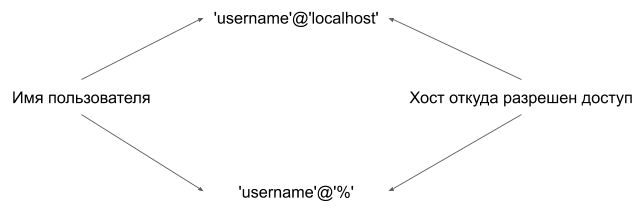
Оператор **DROP USER** не закрывает автоматически соединение удаляемого пользователя, который сможет работать с базой данных, до тех пор, пока не будет закрыто его соединение или связь с сервером не оборвется.

Изменять имя уже существующей учётной записи можно при помощи оператора **RENAME USER**:

|  |
| --- |
| RENAME USER shop TO foo; SELECT Host, User FROM mysql.user; |

## Учетная запись

Следует отметить, что учетная запись является составной и принимает форму **'username'@'host'**, где **username** — имя, а **host** — наименование хоста, с которого пользователю **username** разрешено обращаться к серверу MySQL.



Если к IP-адресу хоста привязано доменное имя, его можно использовать вместо этого хоста.

Количество адресов, с которых необходимо обеспечить доступ пользователю, может быть значительным и включать целые диапазоны. Для задания диапазона в имени хоста используется специальный символ **%**, выполняющий ту же функцию, что и в операторе **LIKE**. Так, учетная запись **'username'@'%'** разрешает пользователю **'username'** обращаться к серверу MySQL с любых компьютеров сети.

## Управление привилегиями

Рассмотренные команды позволяют создавать, удалять и редактировать учетные записи, однако они не разрешают изменять привилегии пользователя, т. е., сообщать СУБД MySQL, какой пользователь имеет право только на чтение информации, какой — на чтение и редактирование, а кому предоставлены права изменять структуру базы данных и создавать учетные записи для остальных пользователей.

Для решения этих задач предназначены операторы **GRANT** и **REVOKE**: оператор **GRANT** назначает привилегии пользователю, а **REVOKE** — удаляет.

Если учетная запись, которая появляется в команде **GRANT**, не существует, она автоматически создается. Однако удаление всех привилегий с помощью команды **REVOKE** не приводит к автоматическому уничтожению учетной записи. Для полного удаления пользователя необходимо либо воспользоваться оператором **DROP USER**.

Давайте воспользуемся командой **GRANT**:

|  |
| --- |
| GRANT ALL ON \*.\* TO 'foo'@'localhost' IDENTIFIED WITH sha256\_password BY 'pass'; |

Запрос создает пользователя с именем **foo** и паролем **pass**. Этот пользователь может обращаться к серверу MySQL с локального хоста (**localhost**) и имеет все права (**ALL**) для всех баз данных (**\*.\***). Если такой пользователь уже существует, то его привилегии будут изменены на **ALL**.

Отобрать права у пользователя можно при помощи команды **REVOKE**:

|  |
| --- |
| REVOKE ALL ON \*.\* FROM 'foo'@'localhost'; |

Существует большое количество самых разнообразных привилегий.

|  |
| --- |
| GRANT SELECT, INSERT, DELETE, UPDATE ON \*.\* TO foo; |

Например, тут пользователю **foo** разрешается полный доступ на просмотр, заполнение, редактирование и удаление таблиц.

|  |
| --- |
| GRANT ALL ON \*.\* TO foo; GRANT GRANT OPTION ON \*.\* TO foo; |

Чтобы назначить все привилегии сразу, следует назначить привилегию **ALL** и **GRANT OPTION**. **ALL** разрешает все операции, кроме назначение прав себе и другим пользователям, за это отвечает специальная привилегия **GRANT OPTION**.

Выполнить операции по наделению пользователя всеми правами в одном запросе не получиться, ключевое слово **ALL** всегда употребляется отдельно и не должно использоваться совместно с другими ключевыми словами:

|  |
| --- |
| GRANT ALL, GRANT OPTION ON \*.\* TO foo; |

Однако использование ключевого слова **USEAGE**, означающего полное отсутствие привилегий, совместимо с другими ключевыми словами:

|  |
| --- |
| GRANT USAGE, SELECT ON \*.\* TO foo; |

Ключевое слово ON в операторе GRANT задает уровень привилегий:

* GRANT USAGE ON \*.\* TO foo;
* GRANT USAGE ON \* TO foo;
* GRANT USAGE ON shop.\* TO foo;
* GRANT USAGE ON shop.catalogs TO foo;
* GRANT SELECT (id, name), UPDATE (name) ON shop.catalogs TO foo.

Глобальный уровень — касается всех баз данных и таблиц, входящих в их состав. Таким образом, пользователь с полномочиями на глобальном уровне может обращаться ко всем базам данных.

Если текущая база данных не была выбрана при помощи оператора **USE**, данное предложение эквивалентно **ON \*.\***, если произведен выбор текущей базы данных, то устанавливаемые при помощи оператора **GRANT** привилегии относятся ко всем таблицам текущей базы данных

Уровень базы данных — данное предложение означает, что привилегии распространяются на таблицы базы данных **db.**

Уровень таблицы — данное предложение означает, что привилегии распространяются на таблицу **tbl** базы данных **db**.

Уровень столбца — привилегии уровня столбца касаются отдельных столбцов в таблице **tbl** базы данных **db**.

Список столбцов указывается в скобках через запятую после ключевых слов **SELECT**, **INSERT,** **UPDATE**.

Для просмотра существующих привилегий необходимо было выполнить оператор **SHOW GRANTS**:

|  |
| --- |
| SHOW GRANTS; |

Ключевое слово **WITH** оператора **GRANT**, помимо привилегии **GRANT OPTION**, позволяет наложить ограничения на число подключений, запросов, обновлений и параллельных запросов в час:

|  |
| --- |
| GRANT ALL ON shop.\* TO 'foo'@'localhost' IDENTIFIED WITH sha256\_password BY 'pass' WITH MAX\_CONNECTIONS\_PER\_HOUR 10  MAX\_QUERIES\_PER\_HOUR 1000  MAX\_UPDATES\_PER\_HOUR 200  MAX\_USER\_CONNECTIONS 3; |

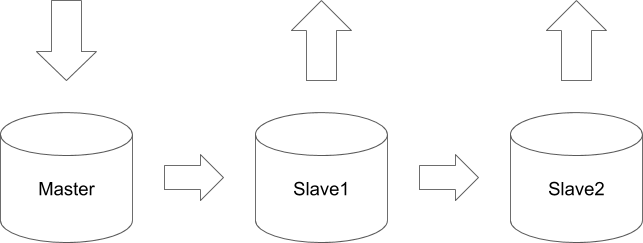
Запрос устанавливает для пользователя **foo** полный доступ к базе данных **shop**, но с ограничением — не более 10 подключений к серверу MySQL в час и не более 1000 запросов, из которых только 200 могут быть операциями обновления **UPDATE**, а 3 запроса — протекать одновременно.

Если значение каждой из опции равно 0 (по умолчанию) — это означает, что у пользователя нет ограничений по этому параметру.

# Репликация

Репликация (replication) означает хранение копий одних и тех же данных на нескольких машинах, соединенных с помощью сети. Есть несколько возможных причин репликации данных:

* ради хранения данных географически близко к пользователям (и сокращения, таким образом, задержек);
* чтобы система могла продолжать работать при отказе некоторых ее частей (и повышения, таким образом, доступности);
* для горизонтального масштабирования количества машин, обслуживающих запросы на чтение (и повышения, таким образом, пропускной способности по чтению).



Данные записываются в один из серверов, который называется мастером, и передаются за счет копирования бинарного журнала на связанные с ним, которые называют слейв-серверами. При классической репликации набор данных, которые хранит база данных, должен убираться на одной машине. Каждая из машин, участвующих в репликации, должна хранить у себя копию всех данных.

Репликация по определению может масштабировать только операцию чтения, операция записи не масштабируется. Все сервера, сколько бы их ни было, должны выполнить запись. Выгоду мы приобретаем, только если можем разделить запросы на чтение среди нескольких серверов.

Чтобы продемонстрировать репликацию, нам потребуется запустить два MySQL-сервера. Один будет выступать master-сервером, принимающим запросы на вставку и обновление данных, другой — слейв-сервером, который будет получать изменения с master-сервера.

Для этого предлагаю воспользоваться утилитой **mysqld\_multi**, которая позволяет запускать несколько экземпляров mysql-сервера на одном хосте.

Предварительно останавливаем сервер и инициируем дополнительный каталог данных

|  |
| --- |
| bin/mysqld --initialize --user=mysql --basedir=/usr/local/opt/mysql --datadir=/usr/local/var/mysql1 |

Далее редактируем конфигурационный файла **my.cnf**. Давайте уберем все предыдущие изменения и добавим две секции: **mysql1** и **mysql2**.

|  |
| --- |
| [mysqld] bind-address = 127.0.0.1 skip-grant-tables  [mysqld1] socket      = /tmp/mysql.sock1 port        = 3306 pid-file    = /usr/local/var/mysql1/mysqld1.pid datadir     = /usr/local/var/mysql1  [mysqld2] socket      = /tmp/mysql.sock2 port        = 3307 pid-file    = /usr/local/var/mysql2/mysqld2.pid datadir     = /usr/local/var/mysql2 |

Мы изменяем каталог данных, каждый из серверов будет смотреть в разный каталог данных, кроме того, мы указываем директиву **skip-grant-tables**, которая требует, чтобы настройки привилегий пользователей игнорировались и мы могли бы обратиться к серверам, не указывая пароль.

Кроме того, мы явно указываем разные порты и сокеты. По умолчанию, если вы обращаетесь к mysql-серверу с текущей машины, клиент **mysql** будет пытаться использовать сокет. Если обращение идет с внешней машины, вам волей-неволей придется использовать IP-адрес и порт.

Запустим сервер при помощи команды **mysqld\_multi**:

|  |
| --- |
| mysqld\_multi start |

При старте MySQL инициализирует новые каталоги данных, для каждого из серверов указывается временный пароль root-пользователя. Однако он нам не потребуется, так как при помощи директивы **skip-grant-tables** мы явно отключили проверки паролей и привилегий.

|  |
| --- |
| mysqld\_multi stop mysqld\_multi start 1 mysqld\_multi start 2  mysql *--socket=/tmp/mysql.sock1 -u root* mysql *--socket=/tmp/mysql.sock2 -u root*  mysql> FLUSH PRIVILEGES; mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY '';  mysqld\_multi stop ps aux | grep mysqld  mysql *--socket=/tmp/mysql.sock2 -u root* mysql *--socket=/tmp/mysql.sock2 -u root* |

Давайте сервер **mysql1** у нас будет master-сервером, а **mysql2** — slave-сервером. Для настройки сервера в качестве главного необходимо включить бинарный журнал (**binary log**) и назначить уникальный идентификатор сервера.

|  |
| --- |
| [mysqld1] socket    = /tmp/mysql.sock1 port      = 3306 pid-file  = /usr/local/var/mysql1/mysqld1.pid datadir   = /usr/local/var/mysql1 log-bin   = master-bin log-bin-index = master-bin.index server-id = 1 |

Идентификатор сервера нужен, чтобы отличить один от другого. В параметре **log-bin** задано базовое имя для всех файлов, созданных в двоичном журнале. В параметре **log-bin-index** задано базовое имя индексного файла двоичного журнала, в котором содержится список всех файлов двоичного журнала.

Можно не указывать имя файла в параметре **log-bin**. В этом случае значение будет назначено по умолчанию. Однако если имя хоста будет изменено, то имена файлов бинарного журнала тоже обновятся. Поэтому лучше их назначить сразу и явно. Перезапускаем сервер.

Перейдем в каталог данных первого сервера:

|  |
| --- |
| cd /usr/local/var/mysql1 ls -la |

Видим индексный и первый файлы бинарного журнала. Подчиненный сервер посылает запрос на главный на получение всех изменений. Чтобы подключить подчиненный сервер, на главном должен быть пользователь с особыми полномочиями репликации.

|  |
| --- |
| mysql *--socket=/tmp/mysql.sock1 -u root* mysql> CREATE USER repl\_user; mysql> GRANT REPLICATION SLAVE ON \*.\* TO repl\_user IDENTIFIED BY '321321'; |

Привилегия **REPLICATION SLAVE** позволяет пользователю получать доступ к двоичному журналу. Теперь давайте настроим **slave**, или подчиненный сервер.

Как и главному, подчиненному серверу нужно присвоить уникальный идентификатор. Кроме того, в файл **my.cnf** можно добавить имена файлов журнала ретрансляции и индекса журнала ретрансляции:

|  |
| --- |
| [mysqld2] socket      = /tmp/mysql.sock2 port        = 3307 pid-file    = /usr/local/var/mysql2/mysqld2.pid datadir     = /usr/local/var/mysql2 server-id   = 2 relay-log-index = slave-relay-bin.index relay-log   = slave-relay-bin |

По умолчанию значения параметров **relay-log** и **relay-log-index**, как и для параметров **log-bin** и **log-bin-index**, зависят от имени хоста.

Использование умолчаний чревато проблемой, так как в случае изменения имени узла сервера индексный файл журнала ретрансляции не будет найден, и сервер будет полагать, что файлы журнала ретрансляции пустые.

Осталось сделать заключительный шаг: указать slave-серверу главный, чтобы он знал, откуда выполнять репликацию.

Для этого нужно ответить на четыре вопроса о главном сервере:

* Имя хоста.
* Порт.
* Учетная запись для репликации.
* Пароль к учетной записи.

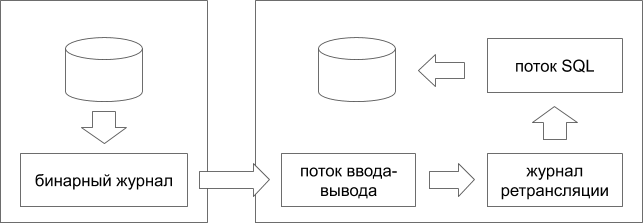
|  |
| --- |
| CHANGE MASTER TO MASTER\_HOST = 'localhost', MASTER\_PORT = 3306, MASTER\_USER = 'repl\_user', MASTER\_PASSWORD = '321321';  START SLAVE; |

Простейшая репликация между главным и подчиненным серверами настроена. Проверить успешность запуска процесса репликации на подчиненном сервере можно при помощи оператора **SHOW SLAVE STATUS**:

|  |
| --- |
| SHOW SLAVE STATUS\G |

## Внутреннее устройство репликации

Главный сервер записывает изменения данных в бинарный журнал. Подчиненный сервер копирует содержимое бинарного журнала в свой журнал ретрансляции. За это отвечает специальный поток ввода данных. Далее подчиненный сервер воспроизводит события из журнала ретрансляции, применяя изменения к собственным данным. Этим занимается еще один поток, который называется SQL.



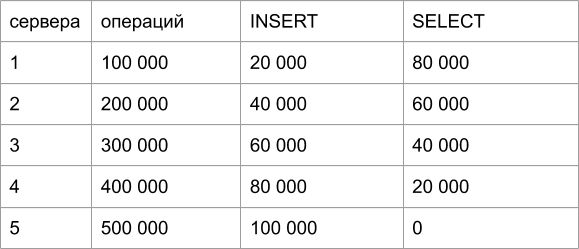
Возвращаясь к отчету команды **SHOW SLAVE STATUS**, мы можем увидеть две строки: **Slave\_IO\_Running** и **Slave\_SQL\_Running**,  именно они показывают статус потоков SQL и ввода-вывода. Если один из потоков не работает, вместо значений **Yes** будет выведено значение **No**.

По значению SQL\_Delay можно судить о том, насколько slave-сервер отстает от мастера: чем меньше это значение, тем лучше.

## Ограничения масштабирования

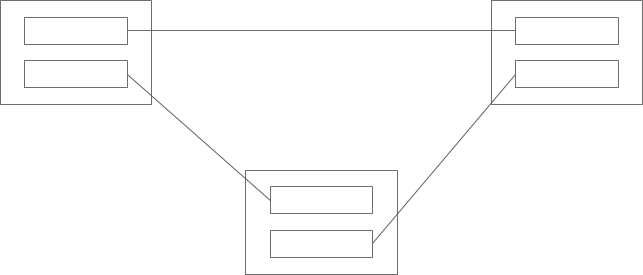
При классической репликации набор данных, которые хранит база, должен убираться на одной машине. Каждая из машин, участвующих в репликации, должна хранить у себя копию всех данных.

Репликация по определению может масштабировать только чтение, запись не масштабируется. Все сервера, сколько бы их ни было, должны выполнить запись. Выгоду мы приобретаем, только если можем разделить запросы на чтение среди нескольких серверов.



## Шардирование

В тех случаях, когда данных не убираются на одной машине, их придется дробить на части и располагать несколько копий одной и той же части на нескольких машинах.



Такой подход называется секционированием или шардированием. Между этими частями, которые располагаются на разных хостах, также устанавливаются отношения репликации.

Только здесь репликация осуществляется не между физическими серверами, а между фрагментами или секциями большой базы данных.

Откровенно говоря, MySQL довольно плохо поддерживает шардирование из коробки. Для него, как правило, выбирают либо базу данных, которая поддерживает шардирование из коробки (PostgreSQL, MongoDB, Cassandra), либо эксплуатируют MySQL в специальном режиме, например режиме кластера.

В любом случае, даже если вы используете шардирование, вы сталкиваетесь со всеми особенностями репликации, которой будут посвящены оставшиеся ролики урока.

## Типы репликации

Различают несколько типов репликации:

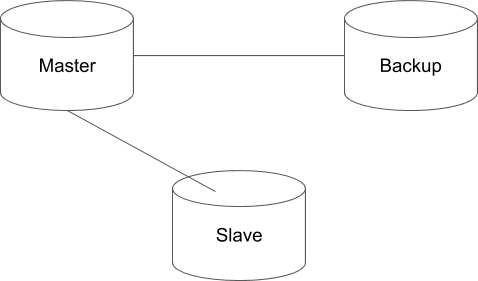
* синхронная,
* асинхронная.

В классической MySQL используется асинхронная репликация, когда изменения применяются на master-сервере, а на подчиненных серверах они доставляются с задержкой. При синхронной репликации мастер-сервер ожидает ответа от всех реплик и считает запрос завершенным только когда данные сохранились на всех хостах, участвующих в репликации.

В случае синхронной репликации резко возрастает время ответа сервера для операций записи, а зачастую и чтения. В синхронной репликации возникают ошибки, связанные с задержками доставки данных, которые могут достигать минуты. Вы публикуете сообщение, хотите его прочитать, но попадаете на slave-сервер, где этого сообщения еще нет, т.е., ваше приложение уверено, что оно должно быть в базе данных, но запись появилась на master-сервере и еще не добралась до slave-сервера.

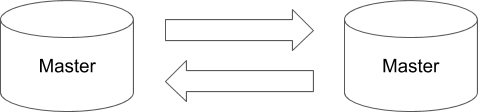
Если вы разрабатываете приложение, базу данных которого планируете в дальнейшем масштабировать при помощи асинхронной репликации, используйте репликацию сразу на этапе разработки. Заменить монолитную базу данных реплицированной потом крайне сложно. Появляется целый класс очень сложно улавливаемых ошибок, когда вы вставляете данные на master-сервере, читаете их со slave-сервера, а их еще нет, они не доставлены.

Когда серверов более чем один, появляются различные варианты их использования.



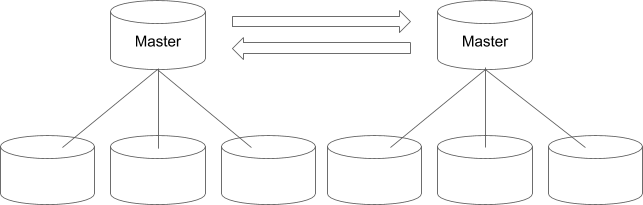
Наиболее простая топология дублирования серверов — это топология горячего резерва. Она состоит из главного сервера и выделенного сервера, называемого «горячим резервом» и дублирующего главный сервер. Сервер горячего резерва подключен к главному как подчиненный, читая и внося все изменения.

Идея состоит в том, что в случае отказа главного сервера горячий резерв — его точная копия — заменит его. Это даст возможность переключить всех клиентов и подчиненные серверы на резервный и продолжить работу.



Довольно популярной топологией является **master-master**. В ней используются два главных сервера, которые реплицируют изменения друг друга. Эта система проста в использовании, поскольку симметрична:

* для перехода на резервный главный сервер не требуется менять конфигурацию главного сервера,
* вернуться на исходный главный сервер при отказе резервного очень просто.



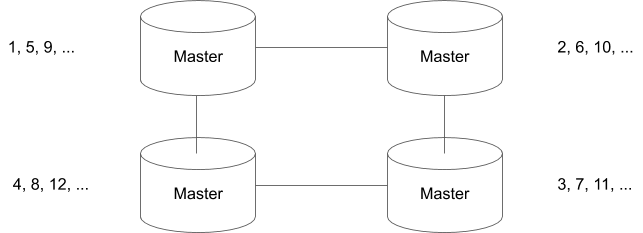
Каждый из master-серверов может выступать в качестве главного для целой группы mysql-серверов. Если на обоих главных серверах будет обновлена одна и та же информация — между обновлениями возникнет конфликт, который может привести к останову репликации.

Частично исключить проблему конфликтов изменений можно, разрешив запросы на запись только на одном из серверов, т. е., сделав пассивным один из серверов. Решить такие конфликты можно, либо записывая информацию в разные таблицы, либо организовать запись таким образом, чтобы исключить конфликты.

Большинство конфликтов связаны со вставкой новых значений в таблицу, так как два master-сервера ведут свой отсчет счетчика **auto\_increment** для первичного ключа.

Сервер MySQL предоставляет две серверные переменные для управления механизмом **AUTO\_INCREMENT**:

* **auto\_increment\_offset** — назначает начальное значение всех столбцов **AUTO\_INCREMENT**;
* **auto\_increment\_increment** — устанавливает приращение, которое используется для вычисления следующего значения в столбце **AUTO\_INCREMENT**.



У любого подчиненного сервера может быть только один главный, поэтому единственный способ этого достичь — настроить репликацию по кругу. Это не рекомендуется, но вполне возможно. Не рекомендуют этот вариант потому, что очень сложно добиться его правильной работы при сбое.

# Используемые источники

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

**Урок 8**

Хранимые процедуры и функции, триггеры

[Хранимые процедуры и функции](https://docs.google.com/document/d/1YtUeOkfXpYgy-33eAhsoIYxcXhISGFirHP5q9XBnnPI/edit#heading=h.a87pq6wf3svj)

[Параметры процедур и функций](https://docs.google.com/document/d/1YtUeOkfXpYgy-33eAhsoIYxcXhISGFirHP5q9XBnnPI/edit#heading=h.pk2umztsa85e)

[Ветвление](https://docs.google.com/document/d/1YtUeOkfXpYgy-33eAhsoIYxcXhISGFirHP5q9XBnnPI/edit#heading=h.kdbs9buvqkp)

[Циклы](https://docs.google.com/document/d/1YtUeOkfXpYgy-33eAhsoIYxcXhISGFirHP5q9XBnnPI/edit#heading=h.d7zokqrs8tuy)

[Обработка ошибок](https://docs.google.com/document/d/1YtUeOkfXpYgy-33eAhsoIYxcXhISGFirHP5q9XBnnPI/edit#heading=h.n3z06p5yptbs)

[Курсоры](https://docs.google.com/document/d/1YtUeOkfXpYgy-33eAhsoIYxcXhISGFirHP5q9XBnnPI/edit#heading=h.wtggwj9y7cu)

[Триггеры](https://docs.google.com/document/d/1YtUeOkfXpYgy-33eAhsoIYxcXhISGFirHP5q9XBnnPI/edit#heading=h.o2lixyvo4vvi)

[Используемые источники](https://docs.google.com/document/d/1YtUeOkfXpYgy-33eAhsoIYxcXhISGFirHP5q9XBnnPI/edit#heading=h.w5vh0mt0cjmc)

# 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.

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

|  |
| --- |
| CREATE PROCEDURE my\_version () BEGIN   SELECT VERSION(); END |

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. Итак, нажимаем **ALT + X**, чтобы выполнить запросы.

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**:

|  |
| --- |
| SHOW PROCEDURE STATUS LIKE 'my\_version%'\G |

Команда возвращает список хранимых процедур и функций. При использовании ключевого слова **LIKE** можно вывести информацию только о тех процедурах, имена которых удовлетворяют шаблону. Для просмотра списка хранимых функций предназначена аналогичная команда **SHOW FUNCTION STATUS**.

Вывод довольно объемный и пользоваться командами **SHOW** не очень удобно, поэтому при наличии прав доступа можно обратиться к системной базе данных mysql, где хранимые процедуры и функции лежат в таблице **proc**.

|  |
| --- |
| SELECT name, type FROM mysql.proc LIMIT 10; |

Пользоваться обычным SELECT-запросом гораздо удобнее, можно извлекать только ту информацию, которая действительно необходима. После того, как хранимая процедура уже создана, посмотреть ее содержимое можно при помощи команды **SHOW CREATE PROCEDURE**:

|  |
| --- |
| SHOW CREATE PROCEDURE my\_version\G |

Для удаления хранимых процедур и функций предназначены операторы **DROP PROCEDURE** и **DROP FUNCTION**. Давайте удалим процедуру **my\_version**:

|  |
| --- |
| DROP PROCEDURE my\_version; |

Попытка удаления несуществующей хранимой процедуры вызывает ошибку:

|  |
| --- |
| 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.

Создание и использование хранимой функции немного отличается от создания и использования хранимой процедуры. Давайте сразу разместим команду удаления хранимой функции, чтобы мы могли многократно вызывать файл **func.sql**:

|  |
| --- |
| CREATE FUNCTION get\_version () RETURNS TEXT DETERMINISTIC BEGIN   RETURN VERSION(); END |

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** (дэтеминистик) 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** (дэтеминистик) 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.

|  |
| --- |
| DROP FUNCTION IF EXISTS get\_version; CREATE FUNCTION get\_version () RETURNS TEXT DETERMINISTIC BEGIN   RETURN VERSION(); END |

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

|  |
| --- |
| DELIMITER // SELECT VERSION()// DELIMITER ; SELECT VERSION(); |

Давайте снова назначим разделителем два слеша:

|  |
| --- |
| DELIMITER // |

Теперь мы можем воспользоваться новым разделителем

|  |
| --- |
| CREATE FUNCTION get\_version () RETURNS TEXT DETERMINISTIC BEGIN   RETURN VERSION(); END// |

# 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**.

|  |
| --- |
| DELIMITER // |

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.

|  |
| --- |
| CREATE PROCEDURE set\_x (IN value INT) BEGIN   SET @x = value; END// |

Использование 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.

|  |
| --- |
| CALL set\_x(123456)// |

Такое значение называется аргументом функции. Результатом работы процедуры будет установленная переменная **@x**:

|  |
| --- |
| SELECT @x// |

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.

|  |
| --- |
| 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// |

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**.

|  |
| --- |
| 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// |

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.

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**:

|  |
| --- |
| 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// |

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:

|  |
| --- |
| CREATE PROCEDURE declare\_var () BEGIN   DECLARE id, num INT(11) DEFAULT 0;   DECLARE name, hello, temp TINYTEXT; END// |

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 в ролике мы остановимся на этом подробнее. Команда **DECLARE** может появляться только внутри блока **BEGIN...END**, область видимости объявленной переменной также ограничена этим блоком.

|  |
| --- |
| 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.

|  |
| --- |
| CREATE PROCEDURE one\_declare\_var () BEGIN   DECLARE var TINYTEXT DEFAULT 'внешняя переменная';   BEGIN  SELECT var;   END;   SELECT var; END// |

Однако переменная, объявленная во внешнем блоке **BEGIN...END**, будет доступна во вложенном блоке, если не будет объявлено экранирующей ее переменной.

|  |
| --- |
| CALL one\_declare\_var()// |

Помимо ключевого слова **DEFAULT**, позволяющего присваивать значение переменной при ее объявлении допускается присвоение значения переменной по мере работы процедуры или функции. Существует два способа назначить переменной новое значение:

* Команда **SET**.
* Команда **SELECT ... INTO ... FROM**.

Ниже приводится типичный способ использования **SET**: переменной **var** сначала присваивается значение 100, а второй оператор **SET** увеличивает значение переменной **var** на единицу.

|  |
| --- |
| SET var = 100; SET var = var + 1; |

Команда **SELECT … INTO … FROM** позволяет сохранять результаты SELECT-запроса без их немедленного вывода и без использования внешних переменных.

|  |
| --- |
| SELECT id, data INTO x, y FROM test LIMIT 1; |

Тут в примере значения полей **id** и **data** из таблицы **test** присваивается локальным переменным **x** и **y**.

Давайте создадим функцию, которая принимает в качестве аргумента количество секунд и возвращает строку, в которой сообщается, сколько дней, часов, минут и секунд входит в интервал.

|  |
| --- |
| DROP FUNCTION IF EXISTS second\_format; CREATE FUNCTION second\_format (seconds INT) RETURNS VARCHAR(255) DETERMINISTIC BEGIN   RETURN ''; END |

Давайте начнем с такой заготовки:

|  |
| --- |
| DROP FUNCTION IF EXISTS second\_format; CREATE FUNCTION second\_format (seconds INT) RETURNS VARCHAR(255) DETERMINISTIC BEGIN   DECLARE days, hours, minutes INT;   RETURN ''; END |

Объявляем дни, часы и минуты:

|  |
| --- |
| DROP FUNCTION IF EXISTS second\_format; CREATE FUNCTION second\_format (seconds INT) RETURNS VARCHAR(255) DETERMINISTIC BEGIN   DECLARE days, hours, minutes INT;    SET days = FLOOR(seconds / 86400);   SET seconds = seconds - days \* 86400;   SET hours = FLOOR(seconds / 3600);   SET seconds = seconds - hours \* 3600;   SET minutes = FLOOR(seconds / 60);   SET seconds = seconds - minutes \* 60;    RETURN CONCAT(days, " days ",             hours, " hours ",             minutes, " minutes ",             seconds, " seconds"); END |

И используем mysql-функцию **CONCAT()**, чтобы соединить результаты в строки:

|  |
| --- |
| SELECT second\_format(123456)// |

Итак, мы познакомились с командой **SET**, теперь давайте более подробно поговорим о команде **SELECT** и особенностях ее использования внутри хранимых процедур и функций.

|  |
| --- |
| DROP PROCEDURE IF EXISTS numcatalogs// CREATE PROCEDURE numcatalogs (OUT total INT) BEGIN   SELECT COUNT(\*) INTO total FROM catalogs; END// |

Давайте создадим процедуру **numcatalogs**, которая возвращает количество строк в таблице **catalogs**. Хранимая процедура **numcatalogs()** имеет один целочисленный (**INT**) параметр **total**, в который сохраняется количество записей в таблице **catalogs**.

Осуществляется это при помощи команды **SELECT** и дополнительного ключевого слова **INTO**, который позволяет сохранять результаты непосредственно в выходном параметре **total** функции **numcatalogs**.

|  |
| --- |
| CALL numcatalogs(@a)// SELECT @a// |

Если команда **SELECT** возвращает несколько значений, их можно принять в несколько переменных; для этого их достаточно перечислить через запятую.

# 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**.

|  |
| --- |
| 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// |

Давайте вызовем процедуру:

|  |
| --- |
| CALL format\_now('date')// CALL format\_now('time')// |

Команда **IF** поддерживает ключевое слово **ELSE**. Давайте перепишем процедуру **format\_now** c использованием **ELSE**:

|  |
| --- |
| 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// |

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**.

Оператор **IF** позволяет выбрать и большее число альтернатив. Для этого используются дополнительные условия после ключевого слова **ELSEIF**:

|  |
| --- |
| 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;   ELSEIF (format = 'time') THEN  SELECT DATE\_FORMAT(NOW(), "%H:%i:%s") AS format\_now;   ELSE  SELECT UNIX\_TIMESTAMP(NOW()) AS format\_now;   END IF; END// |

Процедуру **format\_now()** можно изменить таким образом, чтобы она выводила количество секунд, прошедших с полуночи первого января 1970 года. Таким образом, аргумент **format** по-прежнему может принимать значения **'date'** и **'time'** для вывода даты и времени в строковом представлении. Если задан любой другой аргумент — выводится количество секунд, прошедших с полуночи первого января 1970 года:

|  |
| --- |
| CALL format\_now('secs')// |

Под множественный выбор в MySQL предназначен оператор **CASE**:

|  |
| --- |
| DROP PROCEDURE IF EXISTS format\_now// CREATE PROCEDURE format\_now (format CHAR(4)) BEGIN   CASE format  WHEN 'date' THEN   SELECT DATE\_FORMAT(NOW(), "%d.%m.%Y") AS format\_now;  WHEN 'time' THEN   SELECT DATE\_FORMAT(NOW(), "%H:%i:%s") AS format\_now;  WHEN 'secs' THEN   SELECT UNIX\_TIMESTAMP(NOW()) AS format\_now;  ELSE   SELECT 'Ошибка в параметре format';   END CASE; END//  CALL format\_now ('date')// CALL format\_now ('secs')// CALL format\_now ('four')// |

# Циклы

Циклы являются важнейшей конструкцией, без которой хранимые процедуры и функции не имели бы достаточно функциональности. Таблицы, как правило, имеют множество записей, поэтому циклическая обработка данных встречается в SQL-программировании достаточно часто.

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**.

|  |
| --- |
| CREATE PROCEDURE NOW3 () BEGIN   DECLARE i INT DEFAULT 3;   WHILE i > 0 DO  SELECT NOW();  SET i = i - 1;   END WHILE; END// |

Здесь в процедуре **NOW3** 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 NOW3()// |

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

|  |
| --- |
| CREATE PROCEDURE NOWN (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 'Ошибочное значение параметра';   END IF; END// |

При помощи оператора **IF** можем убедиться, что заданное значение больше 0 и использовать его в условии оператора **while**. Обратите внимание: на этот раз локальная переменная **i** пробегает значение от 0 до заданного в параметре **num**. Как только оно достигает заданного пользователем значения, условие становится ложным и цикл прекращает работу.

Давайте запустим процедуру на выполнение:

|  |
| --- |
| CALL NOWN(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.

|  |
| --- |
| DROP PROCEDURE IF EXISTS NOWN// CREATE PROCEDURE NOWN (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 'Ошибочное значение параметра';   END IF; END// |

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:

|  |
| --- |
| CALL NOWN(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:

|  |
| --- |
| 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;   END IF;   SET bin = CONCAT(bin, i);  END WHILE cycle;  SELECT bin;   ELSE  SELECT 'Ошибочное 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**.

|  |
| --- |
| DROP PROCEDURE IF EXISTS NOW3// CREATE PROCEDURE NOW3 () BEGIN   DECLARE i INT DEFAULT 3;   REPEAT  SELECT NOW();  SET i = i - 1;   UNTIL i <= 0   END REPEAT; END// |

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. Давайте в этом убедимся.

|  |
| --- |
| CALL NOW3()// |

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

|  |
| --- |
| DROP PROCEDURE IF EXISTS NOW3// CREATE PROCEDURE NOW3 () 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// |

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

Запускаем процедуру на выполнение.

|  |
| --- |
| CALL NOW3()// |

# Обработка ошибок

Во время выполнения хранимых процедур и функций могут происходить самые разнообразные ошибки. Поэтому СУБД MySQL поддерживает обработчики ошибок, которые позволяют каждой ошибке назначить свой собственный обработчик.

Кроме того, обработчик, в зависимости от ситуации и серьезности ошибки, может как прекратить, так и продолжить выполнение процедуры.

|  |
| --- |
| SHOW CREATE TABLE catalogs\G |

Давайте смоделируем ошибку. Так как поле **id** объявлено первичным ключом, его значения обязаны быть строго уникальными. Добавление в таблицу значений, совпадающих с одним из тех, которые в ней уже есть, приведет к возникновению ошибочной ситуации.

|  |
| --- |
| SELECT \* FROM catalogs// INSERT INTO catalogs VALUES (1, 'Процессоры')// |

Номер 23000 — код ошибки, возникающей при попытке вставить уже существующее значение в уникальный столбец. Мы можем обработать код ошибки при помощи команды **DECLARE ... HANDLER FOR**. Эта команда может появляться только в теле хранимых функций и процедур.

|  |
| --- |
| DROP PROCEDURE IF EXISTS insert\_to\_catalog// CREATE PROCEDURE insert\_to\_catalog (IN id INT, IN name VARCHAR(255)) BEGIN   DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @error = 'Ошибка вставки значения';   INSERT INTO catalogs VALUES(id, name);   IF @error IS NOT NULL THEN  SELECT @error;   END IF; END//  SELECT \* FROM catalogs//  CALL insert\_to\_catalog(4, 'Оперативная память')// CALL insert\_to\_catalog(1, 'Процессоры')// |

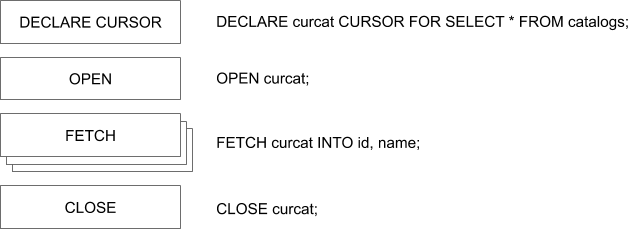
Если результирующий запрос возвращает одну запись, поместить результаты в промежуточные переменные можно при помощи оператора **SELECT...INTO...FROM**. Однако результирующие таблицы чаще содержат несколько записей, и использование такого запроса совместно с оператором **SELECT...INTO...FROM** приводит к возникновению ошибки:

|  |
| --- |
| DROP PROCEDURE IF EXISTS catalog\_id// CREATE PROCEDURE catalog\_id (OUT total INT) BEGIN   SELECT id INTO total FROM catalogs; END//  CALL catalog\_id(@total)// |

Избежать возникновения ошибки можно, добавив **LIMIT 1** или назначив обработчик ошибок. Однако функция будет реализовывать совсем не то поведение, которое ожидает пользователь. Чаще всего требуется обработать именно многострочную результирующую таблицу.

# Курсоры

Решить эту задачу можно при помощи курсоров, которые представляют собой своеобразные циклы, специально предназначенные для обхода результирующих таблиц.



Работа с курсорами похожа на работу с файлами — сначала происходит открытие курсора, затем чтение и после закрытие.

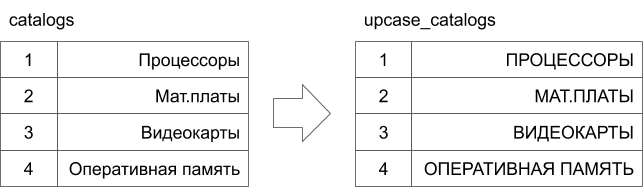
1.  При помощи инструкции **DECLARE CURSOR** имя курсора связывается с выполняемым запросом.

2.  Оператор **OPEN** выполняет запрос, связанный с курсором, и устанавливает курсор перед первой записью результирующей таблицы.

3.  Оператор **FETCH** помещает курсор на первую запись результирующей таблицы и извлекает данные из записи в локальные переменные хранимой процедуры. Повторный вызов оператора **FETCH** приводит к перемещению курсора к следующей записи, и так до тех пор, пока записи в результирующей таблице не будут исчерпаны. Эту операцию удобно осуществлять в цикле.

4.  Оператор **CLOSE** прекращает доступ к результирующей таблице и ликвидирует связь между курсором и результирующей таблицей.

Давайте в качестве примера создадим копию таблицы **catalogs** учебной базы данных **shop**. Эту дублирующую таблицу назовем **upcase\_catalogs** и поместим в нее записи из оригинальной таблицы, только приведем названия разделов к верхнему регистру.



Мы не знаем заранее, сколько записей может быть в таблице **catalogs**, поэтому проще всего обойти ее при помощи курсора.

Давайте создадим таблицу.

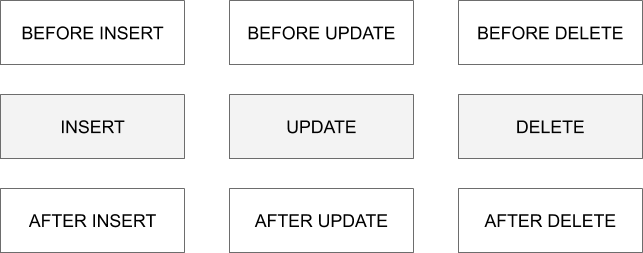
|  |
| --- |
| DROP TABLE IF EXISTS upcase\_catalogs// CREATE TABLE upcase\_catalogs (   id SERIAL PRIMARY KEY,   name VARCHAR(255) COMMENT 'Название раздела' ) COMMENT = 'Разделы интернет-магазина'//  DROP PROCEDURE IF EXISTS copy\_catalogs// CREATE PROCEDURE copy\_catalogs () BEGIN   DECLARE id INT;   DECLARE is\_end INT DEFAULT 0;   DECLARE name TINYTEXT;    DECLARE curcat CURSOR FOR SELECT \* FROM catalogs;   DECLARE CONTINUE HANDLER FOR NOT FOUND SET is\_end = 1;    OPEN curcat;    cycle : LOOP  FETCH curcat INTO id, name;  IF is\_end THEN LEAVE cycle;  END IF;  INSERT INTO upcase\_catalogs VALUES(id, UPPER(name));   END LOOP cycle;    CLOSE curcat; END//  Внутри хранимой процедуры мы */\* Объявляем локальные переменные \*/* */\* Объявляем курсор \*/* */\* Объявляем обработчик для ситуации, когда курсор достигает  конца результирующей таблицы \*/* */\* Открываем курсор при помощи ключевого слова OPEN \*/* */\* В цикле читаем данные из курсора и формируем запись для таблицы upcase\_catalogs \*/* */\* В конце закрываем курсор при помощи команды CLOSE \*/*  CALL copy\_catalogs()// SELECT \* FROM upcase\_catalogs// |

Как хранимая процедура заполняет таблицу **upcase\_catalogs** записями из таблицы **catalogs**, переводя названия разделов в верхний регистр.

# Триггеры

Триггер — специальная хранимая процедура, привязанная к событию изменения содержимого таблицы.

Существуют три события изменения таблицы, к которым можно привязать триггер: это изменение содержимого таблицы при помощи команд **INSERT**, **DELETE** и **UPDATE**. Триггеры могут выполняться до и после каждой из этих команд, поэтому существуют три BEFORE- и три AFTER-триггера, которые на рисунке обозначены белыми прямоугольниками.



|  |
| --- |
| CREATE TRIGGER catalogs\_count AFTER INSERT ON catalogs FOR EACH ROW BEGIN   SELECT COUNT(\*) INTO @total FROM catalogs; END// |

Для создания триггера используется команда **CREATE TRIGGER**. После команды следует имя триггера, далее при помощи ключевого слова **AFTER** указывается, что триггер запускается уже после выполнения команды. В данном случае — после команды **INSERT** для таблицы **catalogs**.

Между ключевыми словами **BEGIN** и **END** располагается тело триггера. Внутри составного тела триггера между ключевыми словами **BEGIN** и **END** допускаются все специфичные для хранимых процедур операторы и конструкции.

В триггере мы извлекаем количество записей в таблице **catalogs** и помещаем это значение в переменную **@total**. Воспользуемся триггером. Для этого достаточно вставить новую запись в таблицу catalogs:

|  |
| --- |
| INSERT INTO catalogs VALUES (NULL, 'Мониторы')// |

Извлечем записи:

|  |
| --- |
| SELECT \* FROM catalogs; |

И давайте убедимся, что переменная **@total** установлена:

|  |
| --- |
| SELECT @total// |

Получить список триггеров можно при помощи команды **SHOW TRIGGERS**:

|  |
| --- |
| SHOW TRIGGERS\G |

За удаление отвечает команда **DROP TRIGGER**. Давайте удалим ранее созданный триггер **catalogs\_count**:

|  |
| --- |
| DROP TRIGGER catalogs\_count// |

Попытка удаления несуществующего триггера завершается неудачей:

|  |
| --- |
| DROP TRIGGER catalogs\_count// |

Чтобы избежать ошибки, как и во многих других командах MySQL, мы можем использовать ключевое слово **IF EXISTS**.

|  |
| --- |
| DROP TRIGGER IF EXISTS catalogs\_count// |

Триггеры очень сложно использовать, не имея доступа к новым записям, которые вставляются в таблицу, или старым записям, которые обновляются или удаляются. Для доступа к новым и старым записям используются префиксы **NEW** и **OLD** соответственно.



То есть, если в таблице обновляется поле **name**, то получить доступ к старому значению можно по имени **OLD.total**, а к новому — **NEW.total**.

Давайте создадим триггер, который при вставке новой товарной позиции в таблицу **products** будет следить за состоянием внешнего ключа **catalog\_id.** Если внешний ключ будет оставаться незаполненным, триггер будет извлекать из таблицы **catalogs** наименьший идентификатор **id** и назначать его записи.

Эти действия нужно выполнить до вставки записи в таблицу, поэтому воспользуемся BEFORE-триггером:

|  |
| --- |
| CREATE TRIGGER check\_catalog\_id\_insert BEFORE INSERT ON products FOR EACH ROW BEGIN   DECLARE cat\_id INT;   SELECT id INTO cat\_id FROM catalogs ORDER BY id LIMIT 1;   SET NEW.catalog\_id = COALESCE(NEW.catalog\_id, cat\_id); END// |

В триггере мы объявляем переменную **id** и извлекаем в нее наименьшее значение идентификатора из таблицы **catalogs**. Далее, если вставляемое значение **catalog\_id** не инициализировано, вместо него вставляется значение переменной **id**. Если пользователь передает значение **catalog\_id**, оно остается неизменным. Для доступа к данным, которые пользователь хочет вставить в таблицу **products**, мы используем ключевое слово **NEW**.

Функция **COALESCE** возвращает первое не NULL-значение и довольно интенсивно используется в SQL-программировании:

|  |
| --- |
| SELECT COALESCE(NULL, NULL, NULL, 1, 2, 3)// SELECT COALESCE(NULL, 3, NULL)// |

Давайте вставим в таблицу **products** записи без указания внешнего ключа **catalog\_id**:

|  |
| --- |
| INSERT INTO products   (name, description, price) VALUES   ('AMD RYZEN 5 1600', 'Процессор AMD', 13200.00)//  SELECT id, name, price, catalog\_id FROM products// |

Как видим, товарная позиция автоматически получает значение 1, в то же время, если мы вставим внешний ключ явно, то:

|  |
| --- |
| INSERT INTO products   (name, description, price, catalog\_id) VALUES   ('ASUS PRIME Z370-P', 'HDMI, SATA3, PCI Express 3.0,, USB 3.1', 9360.00, 2)//  SELECT id, name, price, catalog\_id FROM products// |

В таблицу попадет значение из запроса, триггер не будет вносить коррективы в параметры запроса. Итак, мы добились того, чтобы значение внешнего ключа корректировалось при вставке. Однако мы по-прежнему можем сделать поле **catalog\_id** при помощи команды **UPDATE**:

|  |
| --- |
| UPDATE products SET catalog\_id = NULL WHERE name = 'AMD RYZEN 5 1600'// SELECT id, name, price, catalog\_id FROM products// |

Мы можем создать триггер и для команды **UPDATE**. Давайте при попытке назначить полю **catalog\_id** значение будем оставлять текущее, если оно отлично от **NULL**, или заменять его не NULL-значением. Если и текущее и новое значения принимают значение **NULL**, будем назначать наименьшее значение из таблицы **catalogs**.

|  |
| --- |
| CREATE TRIGGER check\_catalog\_id\_update BEFORE UPDATE ON products FOR EACH ROW BEGIN   DECLARE cat\_id INT;   SELECT id INTO cat\_id FROM catalogs ORDER BY id LIMIT 1;   SET NEW.catalog\_id = COALESCE(NEW.catalog\_id, OLD.catalog\_id, cat\_id); END//  UPDATE products SET catalog\_id = NULL WHERE name = 'AMD RYZEN 5 1600'// SELECT id, name, price, catalog\_id FROM products//  UPDATE products SET catalog\_id = 3 WHERE name = 'MSI B250M GAMING PRO'// SELECT id, name, price, catalog\_id FROM products//  UPDATE products SET catalog\_id = NULL WHERE name = 'MSI B250M GAMING PRO'// SELECT id, name, price, catalog\_id FROM products// |

Триггеры можно использовать, чтобы присваивать другим столбцам вычисляемые значения. Пусть у нас естьь таблица **price**, которая содержит четыре столбца.

|  |
| --- |
| CREATE TABLE prices (   id SERIAL PRIMARY KEY,   processor DECIMAL (11,2) COMMENT 'Цена процессора',   mother DECIMAL (11,2) COMMENT 'Цена мат.платы',   memory DECIMAL (11,2) COMMENT 'Цена оперативной памяти',   total DECIMAL (11,2) COMMENT 'Результирующая цена' )// |

Последний столбец **total** должен содержать сумму трех других, в этом случае мы можем заполнять его автоматически при помощи триггера.

|  |
| --- |
| CREATE TRIGGER auto\_update\_price\_on\_insert BEFORE INSERT ON prices FOR EACH ROW BEGIN   SET NEW.total = NEW.processor + NEW.mother + NEW.memory; END// |

Давайте сразу создадим триггер и для обновления записи

|  |
| --- |
| CREATE TRIGGER auto\_update\_price\_on\_update BEFORE UPDATE ON prices FOR EACH ROW BEGIN   SET NEW.total = NEW.processor + NEW.mother + NEW.memory; END// |

Тело запроса в нем будет точно такое же. Давайте попробуем вставить в таблицу **price** какие-либо записи.

|  |
| --- |
| INSERT INTO prices   (processor, mother, memory) VALUES   (7890.00, 5060.00, 4800.00)//  INSERT INTO prices   (processor, mother, memory) VALUES   (12700.00, 19310.00, 6800.00)// |

Посмотрим содержимое таблицы prices:

|  |
| --- |
| SELECT \* FROM prices// |

Как видим, цена в столбце **total** обновляется автоматически. Впрочем, решить задачу можно с использованием STORED-столбцов. Давайте удалим таблицу **prices**:

|  |
| --- |
| DROP TABLE IF EXISTS prices// |

И создадим ее снова с использованием STORED-столбца:

|  |
| --- |
| CREATE TABLE prices (   id SERIAL PRIMARY KEY,   processor DECIMAL (11,2) COMMENT 'Цена процессора',   mother DECIMAL (11,2) COMMENT 'Цена мат.платы',   memory DECIMAL (11,2) COMMENT 'Цена оперативной памяти',   total DECIMAL (11,2) AS (processor + mother + memory) STORED COMMENT 'Результирующая цена' )// |

Повторно вставим записи:

|  |
| --- |
| INSERT INTO prices   (processor, mother, memory) VALUES   (7890.00, 5060.00, 4800.00)//  INSERT INTO prices   (processor, mother, memory) VALUES   (12700.00, 19310.00, 6800.00)// |

Запросим содержимое таблицы **prices**:

|  |
| --- |
| SELECT \* FROM prices// |

Как видим, добиться точно такого же эффекта можно и без триггеров. Триггеры можно использовать не только для обновления и контроля состояния полей, но и для предотвращения операций.

Например, давайте добьемся, чтобы в таблице **catalogs** всегда присутствовала хотя бы одна запись. Мы просто не будем позволять удалять последнюю запись из таблицы. Для решения этой задачи удобно воспользоваться триггером **BEFORE DELETE**.

|  |
| --- |
| CREATE TRIGGER check\_last\_catalogs BEFORE DELETE ON catalogs FOR EACH ROW BEGIN   DECLARE total INT;   SELECT COUNT(\*) INTO total FROM catalogs;   IF total <= 1 THEN  SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'DELETE canceled';   END IF; END// |

Триггер подсчитывает количество строк в таблице, и если оно меньше или равно единицы, запрещает дальнейшее выполнение запроса. Для этого мы генерируем свою собственную ошибку при помощи команды **SIGNAL SQLSTATE**.

Мы задаем код 45000, который предназначен для пользовательских ошибок. Их невозможно перехватить при помощи обработчиков. Давайте попробуем последовательно удалять записи из таблицы catalogs.

|  |
| --- |
| DELETE FROM catalogs LIMIT 1// DELETE FROM catalogs LIMIT 1// DELETE FROM catalogs LIMIT 1// DELETE FROM catalogs LIMIT 1// ERROR 1644 (45000): DELETE cancelled |

Срабатывает триггер, выбрасывая ошибку с кодом 45000 и сообщением **DELETE cancelled**, которое мы задали внутри триггера. Убедимся, что таблица **catalogs** содержит по меньшей мере одну запись.

|  |
| --- |
| SELECT \* FROM catalogs// |

Так и есть, в таблице остается одна запись, и удалить ее не удастся, пока у нас есть триггер на операцию удаления.

# Используемые источники

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

**Урок 9**

Оптимизация запросов

[Тип таблицы](https://docs.google.com/document/d/1zt2eO0xPfSvo0kePwEurEydmYJz7HCKf2NP_EsI7LiI/edit#heading=h.a87pq6wf3svj)

[Архитектура MySQL](https://docs.google.com/document/d/1zt2eO0xPfSvo0kePwEurEydmYJz7HCKf2NP_EsI7LiI/edit#heading=h.rqxdufghfame)

[Файлы таблицы](https://docs.google.com/document/d/1zt2eO0xPfSvo0kePwEurEydmYJz7HCKf2NP_EsI7LiI/edit#heading=h.63fqqjq5zmff)

[Индексы](https://docs.google.com/document/d/1zt2eO0xPfSvo0kePwEurEydmYJz7HCKf2NP_EsI7LiI/edit#heading=h.pk2umztsa85e)

[Приемы оптимизации](https://docs.google.com/document/d/1zt2eO0xPfSvo0kePwEurEydmYJz7HCKf2NP_EsI7LiI/edit#heading=h.ka93hkvu0lz7)

[Команда EXPLAIN](https://docs.google.com/document/d/1zt2eO0xPfSvo0kePwEurEydmYJz7HCKf2NP_EsI7LiI/edit#heading=h.7xdvbyuszif3)

[Используемые источники](https://docs.google.com/document/d/1zt2eO0xPfSvo0kePwEurEydmYJz7HCKf2NP_EsI7LiI/edit#heading=h.p70879nalwht)

# Тип таблицы

СУБД MySQL поддерживает несколько видов таблиц, каждая из которых имеет свои возможности и ограничения. Задать тип таблицы можно при ее создании при помощи оператора **CREATE TABLE**:

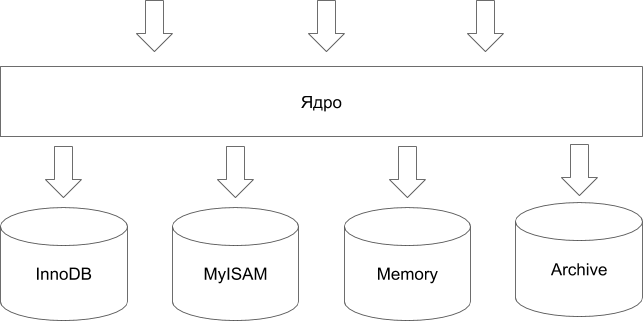
|  |
| --- |
| DROP TABLE IF EXISTS catalogs; CREATE TABLE catalogs (   id SERIAL PRIMARY KEY,   name VARCHAR(255) COMMENT 'Название раздела' ) COMMENT = 'Разделы интернет-магазина'; |

Для этого в конце в параметрах таблицы мы указываем ключевое слово **ENGINE**:

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

# Архитектура MySQL

В СУБД MySQL подсистемы хранения данных и ядро разделены.

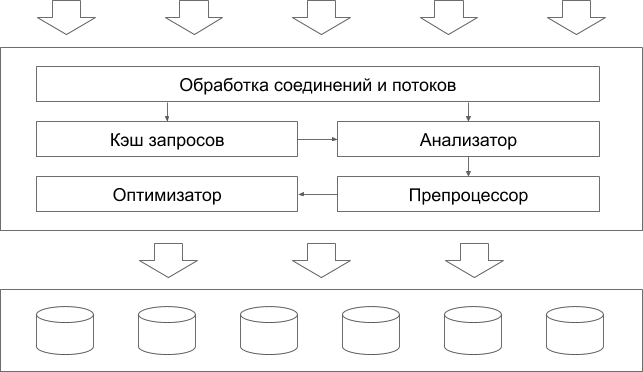


Поддержка соединения с клиентом, применение правил и логики языка SQL, кеширование запросов осуществляет ядро сервера MySQL. За хранение данных, индексирование, выполнение низкоуровневых операций с хранилищем отвечают подсистемы хранения, которые часто называют движками.

Их довольно много, на экране вы видите лишь часть. По умолчанию используется движок InnoDB. До текущего момента использовали его неявно. Так как он используется по умолчанию, нам не обязательно было его указывать при создании таблиц.

Однако вам придется явно указать тип движка, если вы захотите создать таблицу с подсистемой хранения, отличной от InnoDB. Например, можно задействовать таблицу типа **Memory**, полностью разместив данные в оперативной памяти, или использовать таблицу **Archive**, чтобы сжать данные. В этом случае вам придется явно указать тип движка при создании таблицы.

Прежде чем мы окунемся в особенности каждого движка, давайте кратко остановимся на ядре MySQL. Оно поддерживает пул соединений с клиентами, обеспечивая каждого из них сессией и обрабатывая его запросы. В случае, если на сервере включен кеш запросов, каждый SELECT-запрос проверяется сначала в кеше, если он присутствует там, ответ выдается из него.



Далее запрос идет в анализатор, который разбивает его на лексемы и строит дерево разбора. Для интерпретации и проверки запросов анализатор использует грамматику языка SQL. Проверяется, что все лексемы допустимы, следуют в нужном порядке и нет других ошибок, например непарных кавычек.

Затем получившееся дерево разбора передается препроцессору, который контролирует дополнительную семантику, не входящую в компетенцию анализатора. К примеру, проверяется, что указанные таблицы и столбцы существуют, а ссылки на столбцы не допускают неоднозначного толкования. Далее препроцессор проверяет привилегии.

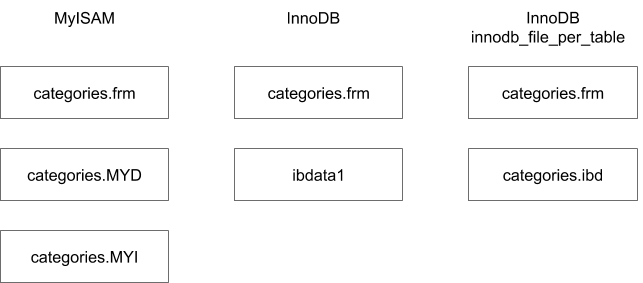
Теперь, когда дерево разбора тщательно проверено, наступает очередь оптимизатора, который превращает его в план выполнения запроса. Часто существует множество способов выполнить запрос, и все они дают один и тот же результат. Задача оптимизатора — выбрать лучший из них.

Язык SQL — декларативный: мы не объясняем как выполнять запрос, мы запрашиваем то, что хотим получить. Поэтому анализатор имеет свободу действий по оптимизации запроса и выбора индексов. Более того, он привлекает статистику сервера для выбора наиболее оптимального пути выполнения запроса.

После того, как план выполнения запроса выбран, происходит его выполнение с использованием API подсистемы хранения, т.е., ядро обращается к низкоуровневым вызовам движка.

# Файлы таблицы

Мы с вами уже исследовали базы данных и выяснили, что это подкаталог в каталоге данных. Что же из себя представляют таблицы? В подкататалоге для каждой из таблиц создается файл с расширением **.frm** и именем, совпадающим с именем таблицы. В этом файле хранится определение таблицы, список столбцов, их типы и порядок следования. Этот файл одинаков для всех подсистем хранения, так как его обрабатывает ядро.



Данные и индексы для каждой системы хранения реализуются по-разному, за их работу отвечают движки таблиц. Например, для типа **MyISAM** данные сохраняются в одноименном файле с расширением **MYD**, а индексы — в файле **MYI**. В случае движка InnoDB данные и индексы всех таблиц и всех баз данных хранятся в едином табличном пространстве.

Впрочем, если в конфигурационном файле **my.cnf** включить директиву **innodb\_file\_per\_table**, можно перевести MySQL в режим, когда для каждой таблицы будет организовываться свое собственное табличное пространство в файле с расширением **idb**.

Давайте перейдем в каталог данных и посмотрим его содержимое, путь к нему можно обнаружить среди системных переменных:

|  |
| --- |
| SHOW VARIABLES LIKE 'datadir'; |

Полученный путь используем в команде **cd**, чтобы перейти в каталог. Для просмотра содержимого каталога можно использовать команду **ls** в UNIX-подобной операционной системе и **dir** в Windows:

|  |
| --- |
| cd /usr/local/var/mysql ls -la |

Итак, мы можем видеть файл **ibdata1** с единым табличным пространством. Файл **ibtmp1** содержит временные таблицы, файлы **ib\_logfile0** и i**b\_logfile1** — журнал транзакций.

Давайте перейдем в каталог **shop**:

|  |
| --- |
| cd shop |

Посмотрим его содержимое:

|  |
| --- |
| ls -la |

Мы можем видеть здесь файлы со структурой **frm** и данными **ibd**, каждая пара соответствует таблице в базе данных **shop**. Кроме того, мы можем видеть несколько файлов с расширением **TRN**, которые предназначены для обеспечения работы триггеров.

Давайте переместимся в системную базу данных **mysql**:

|  |
| --- |
| cd ../mysql |

Посмотрим ее содержимое:

|  |
| --- |
| ls -la |

Как видим, тут есть таблицы **MyISAM**, например таблица **proc**, в которой содержатся хранимые процедуры и функции; для нее есть файл **frm** со структурой таблицы, MYD-файл с данными и **MYI** с индексами.

Чтобы определить, какая подсистема хранения используется для конкретной таблицы, не заглядывая в каталог данных, следует использовать команду **SHOW TABLE STATUS**.

|  |
| --- |
| SHOW TABLE STATUS LIKE 'catalogs'\G |

Как видим, перед нами таблица типа InnoDB. Список доступных движков и их характеристики можно запросить при помощи запроса:

|  |
| --- |
| SHOW ENGINES\G |

InnoDB была разработана для транзакционной обработки и автоматического восстановления после сбоя. Движок поддерживает внешние ключи и блокировку на уровне строк. Этот движок не кеширует информацию о количестве строк в таблице, поэтому в отчетах эта цифра часто приблизительная, а выполнение запросов с участием функции **COUNT(\*)** — очень трудоемкое, так как каждый раз происходит полное сканирование таблицы.

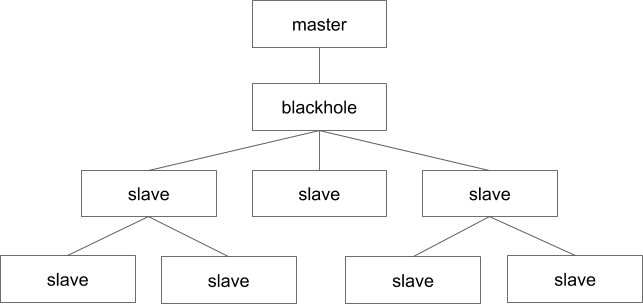
Memory-таблицы полностью расположены в оперативной памяти и не поддерживают транзакции. Поэтому при перезапуске содержимое таблиц обнуляется. За счет расположения в оперативной памяти операции с такой таблицей происходят исключительно быстро. При операциях вставки происходит табличная блокировка, т.е., нельзя одновременно вставлять две записи в таблицу, такие операции выстраиваются в очередь.

Для данного типа таблиц по умолчанию используются хэш-индексы. В настоящий момент эти таблицы редко используются, так как существуют более эффективные NoSQL-решения для таких задач, например, база данных  Redis.

BLACKHOLE-таблицы не содержат данных, операции которые на них выполняются не фиксируются в базе данных. Однако фиксируются в бинарном журнале, поэтому BLACKHOLE-таблицы выступают в качестве ретранслятора бинарного журнала

Когда с этим не справляется реплика?

Например, в топологии «пирамида», когда необходимо к нагруженному мастер-серверу подключить множество подчиненных слейв-серверов. В этом случае сервер, который не выполняет запросов, справляется с передачей бинарного журнала гораздо лучше. На таком сервере таблицам переназначается тип **BLACKHOLE**.



**MyISAM** — файловый движок, не поддерживающий транзакции: при операциях вставки также происходит табличная блокировка. Можно задаться вопросом, зачем нужны движки без транзакций? Дело в том, что транзакции потребляют ресурсы и в тех случаях когда они не важны, а операции вставки не часты и применяются в отношении небольших таблиц, можно добиться значительного выигрыша в производительности.

**MRG\_MYISAM** представляет собой объединение нескольких структурно одинаковых таблиц **MyISAM** в одну виртуальную; это до некоторой степени своеобразный материализованный UNION-запрос.

Система хранения **CSV** позволяет рассматривать текстовые CSV-файлы с разделителями-запятыми как таблицы. Такие файлы легко создаются вручную, любым скриптом или сохраняются из Excel-таблицы.

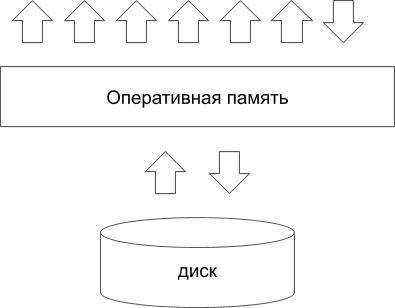
**Archive** позволяет выполнять только команды **INSERT** и **SELECT**. Таблицы этого типа потребляют исключительно мало оперативной памяти, снижают дисковый ввод-вывод, так как данные сжимаются библиотекой **zlib**. Этот тип чаще всего используется для протоколирования и сбора данных. Система хранения **Archive** поддерживает блокировку на уровне строк и специальный системный буфер для вставок с высокой степенью конкурентности. **Archive** не поддерживает транзакции. Она оптимизирована лишь для высокоскоростной вставки и хранения данных в сжатом виде.

# Индексы

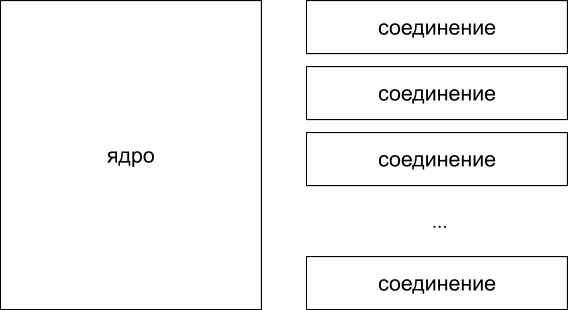
Индексы представляют собой структуры, которые помогают MySQL эффективно извлекать данные. Они критичны для хорошей производительности. Важность индексов увеличивается по мере роста объема данных. Небольшие слабозагруженные базы данных зачастую могут удовлетворительно работать даже без правильно построенных индексов, но по мере роста объемов хранимой в базе данных информации производительность может упасть очень быстро.

Хорошая производительность достигается не только за счет того, что индексы хранят столбцы в заранее отсортированном виде, но и в том, что сами индексы стараются держать в быстрой оперативной памяти вместо хранения их на жестком диске.

Тем не менее, MySQL не может взять без спросу столько оперативной памяти, сколько имеется в системе — память потребуется выделять.

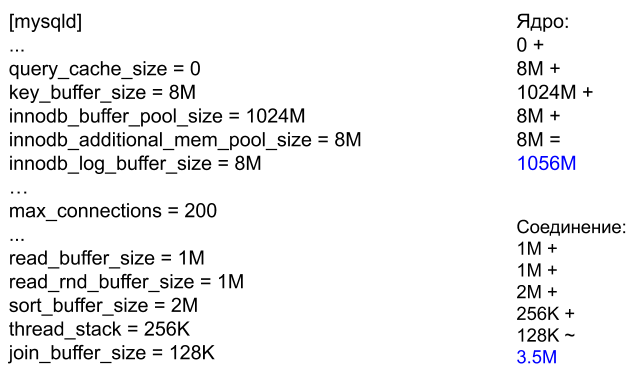


MySQL содержит большое количество самых разнообразных кешей. Основное их назначение — разместить в быстрой оперативной памяти информацию с более медленного диска. В результате данные моментально отдаются из оперативной памяти, вместо того, чтобы извлекаться с более медленного диска.



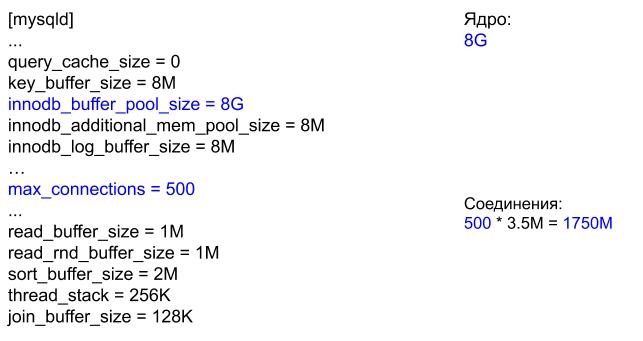
Одни кеши выделяются на ядро и являются общими для всех соединений, часть кешей выделяется на каждое соединение.

Например, кеши под индексы общие для всех клиентов, а кеши под сортировку данных — индивидуальные для каждого из соединений.



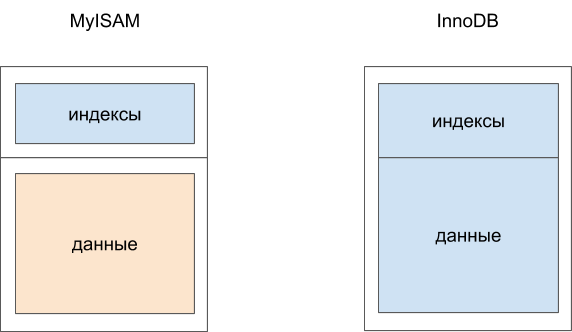
Почти всегда можно подсчитать максимальный размер, который займет MySQL кешами общего назначения и одним соединением. На рисунке можно видеть пример конфигурации MySQL-сервера, в котором ядро может занять чуть больше 1 Гб, а каждое из соединений — порядка 3.5 Мб.

Учитывая, что максимальное количество соединений — 200, при максимальной нагрузке на соединения уйдет около 700 Мб. В реальности, конечно, меньше, так как не все соединения будут использовать кеши. Кеши ядра тоже редко бывают заполнены под завязку.



Если на сервере есть свободная оперативная память, скажем, 16 Гб, можем смело выделить MySQL дополнительную память. Например, увеличить пул буферов InnoDB для кеширования индексов и данных до 8 Гб, а количество соединений — до 500. В результате при максимальной нагрузке MySQL-сервер будет потреблять чуть меньше 10 Гб.

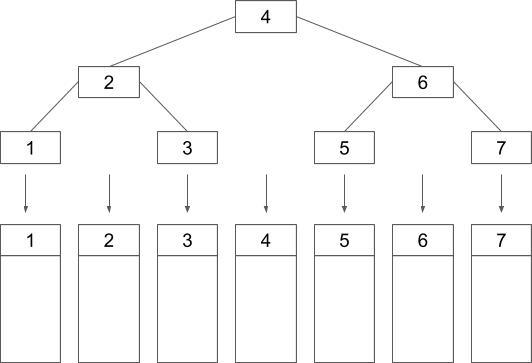
За реализацию индексов отвечают подсистемы хранения, поэтому в разных системах хранения кеширование индексов может осуществляться по-разному. Например, как мы видели в предыдущих роликах, в таблицах типа MyISAM индексы и данные разделены.



Данные всегда находятся на жестком диске, индексы по мере необходимости подтягиваются в оперативную память в кеш индексов.

Данные кешируются средствами операционной системы, поэтому, если MyISAM — основной тип используемых таблиц, важно оставлять на сервере какое-то количество свободной оперативной памяти.

В таблицах InnoDB индексы хранятся в едином табличном пространстве вместе с данными. Поэтому, когда мы говорим о кеше InnoDB, имеется в виду, что в оперативной памяти размещаются и индексы и данные.



Такое различие связано с тем, что для первичного ключа в InnoDB используется кластерный индекс — это фактически то же самое бинарное дерево, как и в случае BTREE-индексов, только в качестве листьев этого дерева выступают строки таблицы.

Таким образом, строки с близкими значениями первичного ключа хранятся по соседству, кроме того, над таблицей можно построить только один кластерный индекс, поскольку невозможно хранить один и тот же индекс в двух местах.

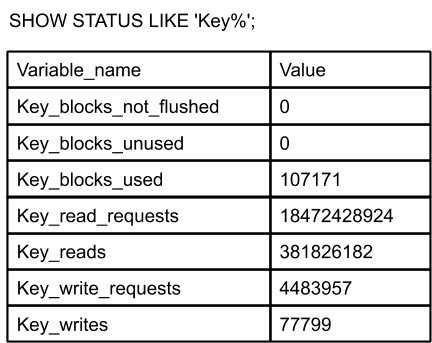
|  |
| --- |
| SHOW VARIABLES LIKE 'Key%'; |

Оценить объем оперативной памяти, выделенной под кеш MyISAM-индексов, можно при помощи команды. Значение **key\_buffer\_size** показывает объем в байтах выделенный под этот кеш: в данном случае это 8 Мб. Это значение можно увеличить через конфигурационный файл **my.cnf**, для этого используется одноименная директива **key\_buffer\_size**, максимально допустимый размер которой 4 Гб.

Мы преимущественно используем InnoDB, поэтому этот кеш не стоит увеличивать слишком сильно, тем не менее совершенно отключать этот кеш не стоит, так как таблицы типа MyISAM используются в системной базе данных MySQL:

|  |
| --- |
| SHOW STATUS LIKE 'Key%'; |

Оценить эффективность данного кеша можно при помощи команды **SHOW STATUS**, отфильтровав результаты по шаблону **Key**. **Key\_blocks\_used** сообщает количество занятых блоков в кеше, **Key\_blocks\_unused** показывает свободное количество блоков. Как видим, на этом сервере MySQL еще далеко до заполнения данного кеша и в его увеличении нет необходимости.



Состояние **Key\_read\_requests** сообщает о количестве блоков, прочитанных из кеша. Состояние **Key\_reads** сообщает о количестве блоков, прочитанных с жесткого диска в кеш.

Таким образом, в приведенном примере, операция обращения к жесткому диску осуществлялась на каждые 50 операций чтения из оперативной памяти.

Состояния **Key\_write\_requests** и **Key\_writes** сообщают аналогичные сведения в отношении операций записи в кеш и на жесткий диск

|  |
| --- |
| SHOW VARIABLES LIKE 'innodb\_buffer\_pool\_size'; |

Оценить объем пула буферов, выделенный под InnoDB, можно при помощи команды **SHOW VARIABLES**, отфильтровав результат по значению **innodb\_buffer\_pool\_size**.

Как видно, переменная принимает значение 128 Мб. Это пул буферов в оперативной памяти и под индексы и под данные InnoDB. Значение **innodb\_buffer\_pool\_size** можно регулировать через одноименную директиву конфигурационного файла **my.cnf**.

Если MySQL — это основное программное обеспечение на сервере, то под пул буферов рекомендуется выделять 50-80 %, т. е., 2, 4, 8, 16 Гб, если это позволяют ресурсы сервера. Так как кешируются не только индексы, но и данные, пул буферов должен быть объемным.

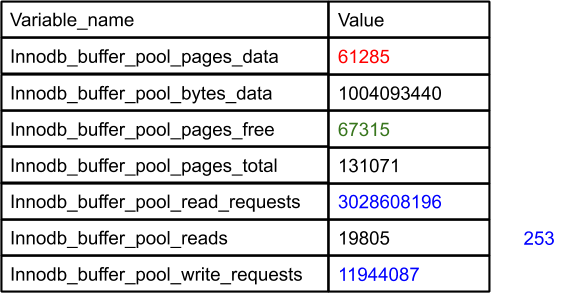
Если ситуация позволяет, желательно, чтобы база данных целиком проваливалась в пул буферов, а к жесткому диску MySQL обращалась бы только для записей транзакций.

Однако бесконечно увеличивать размер **innodb\_buffer\_pool\_size** нельзя. Если оперативная память на сервере будет исчерпана, часть данных из оперативной памяти будет сбрасываться на жесткий диск в **swap**. При обращении к этим данным операционная система будет вынуждена выгрузить другие данные из оперативной памяти. В результате скорость работа системы MySQL-сервера, да и вообще операционной системы, может замедлиться критически. Все параметры MySQL-сервера лучше наращивать постепенно, тщательно отслеживая состояние операционной системы.

Оценить состояние эффективности кеша **InnoDB** можно при помощи команды **SHOW STATUS**:

|  |
| --- |
| SHOW STATUS LIKE 'Innodb\_buffer\_pool\_%'; |

Величина **Innodb\_buffer\_pool\_pages\_total** показывает общение количество блоков в кеше, значение **Innodb\_buffer\_pool\_pages\_free** сообщает о количестве свободных блоков. Если эта величина исчерпана, имеет смысл увеличить объем памяти под InnoDB.



# Приемы оптимизации

При выполнении любого запроса в консольном клиенте mysql всегда выводится время выполнения запроса:

|  |
| --- |
| mysql> SHOW TABLES; +*----------------+* | Tables\_in\_shop | +*----------------+* | hello      | +*----------------+* 1 row in set (0,01 sec) |

Довольно трудно делать какие-то выводы, так как время выполнения запросов на современных компьютерах зачастую составляет несколько миллисекунд. Тем не менее, когда сервер обрабатывает миллионы запросов, их общее время может выливаться в значительную нагрузку.

Оптимизатор старается переработать ваш запрос, переставить условные конструкции местами, чтобы они могли использовать индекс, если есть возможность вычислить значения заранее, вычислить результат по индексам, не используя данные таблицы, и т. д.

Тем не менее, ряд оптимизаций зависит от разработчика. Чем меньше данных возвращает запрос — тем лучше. MySQL затрачивает меньше усилий на их сбор, ответ быстрее пересылается по сети. Это касается и количества возвращаемых запросом строк, и данных в столбцах. Если из всей таблицы вам нужны лишь два-три столбца, не следует извлекать все данные при помощи **\*** в SELECT-запросе. Лучше выбрать конкретные столбцы. Запрос на извлечения всех данных из таблицы пользователей менее эффективен, чем извлечение только части столбцов.

|  |
| --- |
| SELECT id, name FROM users; |

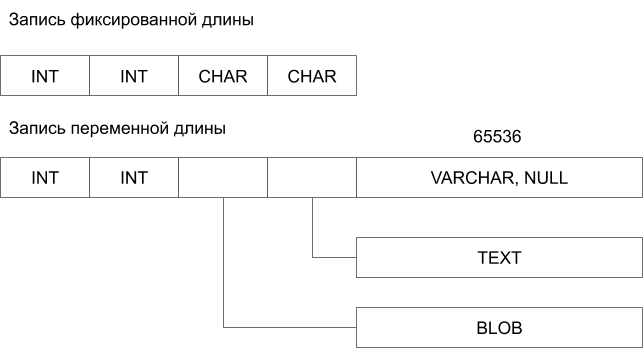
Запрос на извлечение всех данных таблицы менее эффективен, чем извлечение только ее части:

|  |
| --- |
| SELECT id, name FROM users LIMIT 2; |

Например, только двух записей при помощи ключевого слова **LIMIT**.

Значения столбцов типа **VARCHAR** и **NULL** хранятся в специальной области для данных переменной длины. Эти данные обрабатываются менее эффективно, чем столбцы фиксированной длины, например целый тип. Поэтому, если вам не нужно NULL-значение столбца, лучше явно указывать атрибут **NOT NULL**.

Типы данных **BLOB** для хранения бинарных данных и **TEXT**, созданный на его основе, еще медленнее, чем остальные типы данных, так как хранятся отдельно от остальных столбцов таблицы. В силу такой особенности большинство операций с их участием часто протекают с использованием жесткого диска. Трудно построить базу данных совсем без этих типов, но если есть возможность, стоит исключить TEXT-столбцы из запроса.



# Команда EXPLAIN

Основной способ узнать, какие решения принимает оптимизатор — воспользоваться командой EXPLAIN:

|  |
| --- |
| EXPLAIN SELECT id, name FROM catalogs ORDER BY id\G |

Чтобы воспользоваться командой **EXPLAIN**, достаточно добавить слово **EXPLAIN** перед словом **SELECT** в запросе. MySQL пометит этот запрос специальным флагом. Во время его обработки этот флаг заставляет сервер сообщать информацию о каждом шаге плана выполнения, а не исполнять его.

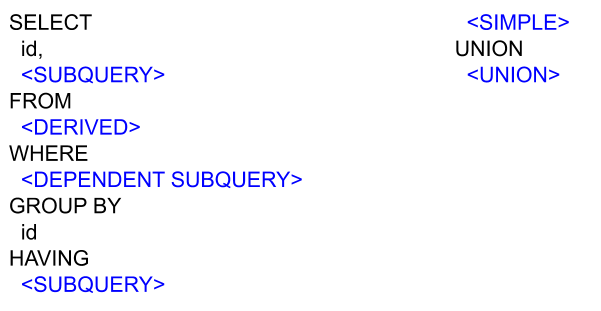
Для каждой встречающейся в запросе таблицы выводится одна строка. Если соединяются две таблицы, то будет выведено две строки.

|  |
| --- |
| EXPLAIN SELECT id, name FROM catalogs UNION ALL SELECT id, name FROM catalogs\G |

Если одна и та же таблица попадается дважды, например при соединении таблицы с собой же, тоже будет выведено две строки. Команду **EXPLAIN** можно применить только для SELECT-запросов.

Для исследования других команд их предварительно следует перевести в SELECT-форму и применить **EXPLAIN**. Результаты, конечно, будут приблизительные, но это лучше, чем ничего.

Результат команды **EXPLAIN** всегда состоит из одних и тех же столбцов, изменяется лишь количество и содержимое строк. Столбец **select\_type** показывает, соответствует ли строка простому или составному запросу **SELECT**.



На экране синим цветом показаны возможные значения поля **select\_type** и где следует искать подзапросы, к которым относятся та или иная строка в отчете команды **EXPLAIN**.

Столбец **table** показывает, к какой таблице относится данная строка. Столбец **type** сообщает метод доступа к таблице: как MySQL будет искать строки в таблице. В данном случае используется метод доступа **ALL**, т.е. будет произведено полное сканирование таблицы. Здесь это не страшно, так как мы и хотим извлечь все содержимое таблицы.

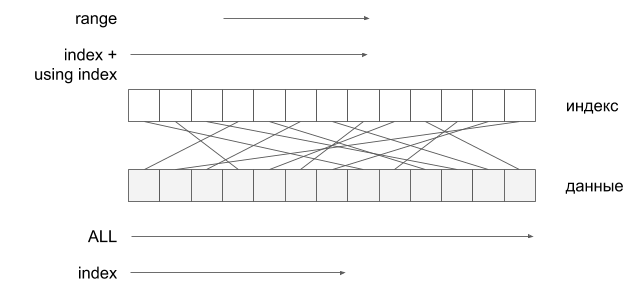
В поле **type** результирующей таблицы команды **EXPLAIN** могут встречаться следующие методы доступа:

* **ALL**,
* **index**,
* **range**,
* **ref**,
* **eq\_ref**,
* **const**,
* **NULL**.

Методы доступа отсортированы от самых худших к наиболее предпочтительным.

**ALL**, как мы уже упоминали, означает полное сканирование таблицы. MySQL должна просмотреть таблицу от начала до конца, чтобы найти нужную строку. Существуют исключения, например запросы с фразой **LIMIT** — в этом случае MySQL прекратит перебор строк, как только будет набранно нужное количество.

**index** означает полное сканирование таблицы, только в порядке, задаваемом индексом. **range** означает просмотр диапазона индекса. Здесь тоже происходит сканирование, но уже индекса. Часто **range** можно увидеть в случае BETWEEN-условий или операций «меньше» или «больше».

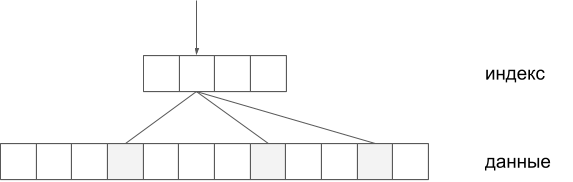


На экране представлены эти таблицы и индекс над ними. Индекс зачастую находится в оперативной памяти, работа с ним идет быстро за счет того, что его значения отсортированы. Данные чаще всего находятся на жестком диске.

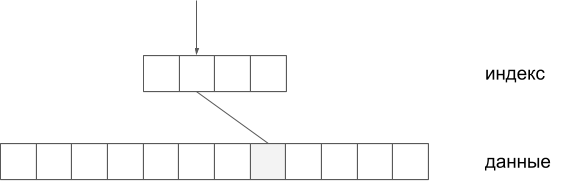
Методы **ALL** и **index** обращаются к данным таблицы. Метод **range** работает с индексом. Это означает, что данных, которые есть в индексе, достаточно для выполнения операции. Однако, потребуется перебор значений индекса в достаточно большом интервале.

Иногда совместно с типом **index** в дополнительном поле **Extra** можно увидеть сообщение **using index**. В этом случае речь идет о покрывающем индексе. Здесь тоже происходит сканирование индекса, но значений индекса достаточно, чтобы дать ответ, нет необходимости использовать данные таблицы с жесткого диска.

Методы доступа по **ref** и **eq\_ref** — это доступ по индексу.



Метод доступа **ref** — это доступ по индексу, в результате которого возвращаются строки, соответствующие единственному заданному значению. Однако таких строк может быть несколько, поэтому поиск сочетается с просмотром. Данный тип доступа возможен лишь в случае неуникального индекса или неуникального префикса ключа в уникальном индексе.



**eq\_ref** — поиск по индексу в случае, когда MySQL точно знает, что будет возвращено не более одного значения.

Если **EXPLAIN** возвращает **const**, это означает, что в процессе оптимизации какую-то часть запроса можно преобразовать в константу. **NULL** означает, что MySQL сумела разрешить запрос на фазе оптимизации, так что в ходе выполнения вообще не потребуется обращаться к таблице или индексу.

Столбец **possible\_keys** показывает, какие индексы можно было бы задействовать для выполнения запроса. Сколько бы индексов ни было создано, в простом запросе может быть задействован только один. Выбранный оптимизатором индекс помещается в столбце **key**.

Столбец **key\_len** показывает, сколько байт индекса использует MySQL. Если задействованы лишь некоторые индексированные столбцы составного индекса, то, зная это значение, можно определить, какие именно.

Столбец **ref** показывает, какие столбцы и константы из предыдущих таблиц используются для поиска в индексе. Столбец **rows** сообщает, сколько строк придется прочитать, чтобы найти запрошенные данные. Чем меньше это значение, тем быстрее будет выполняться запрос.

Столбец **filtered** показывает оценку доил-строк, которые удовлетворяют критерию выборки. Чем меньше этот процент, тем больше данных придется отбрасывать MySQL при сканировании таблицы.

Столбец **Extra** содержит дополнительную информацию, для которой не нашлось места в других столбцах:

* **Using index**.
* **Using where**.
* **Using temporary**.
* **Using filesort**.

Например **Using index** совместно с типом доступа **index** означает, что будет использоваться покрывающий индекс, т. е., данных индекса будет достаточно для выполнения запроса, обращаться к таблице не придется.

**Using where** означает, что сервер произведет дополнительную фильтрацию строк, отобранных подсистемой хранения. Чаще WHERЕ-условия обрабатываются системой хранения, например MyISAM или InnoDB, но иногда их возможностей не хватает и в дело вступает ядро.

**Using temporary** означает, что MySQL будет применять временную таблицу для сортировки результатов запроса. Такая таблица не обязательно размещается на жестком диске, если ее объем позволяет — она будет размещена в оперативной памяти. Максимально допустимый объем временной таблицы задается директивой **tmp\_table\_size**.

**Using filesort** означает, что MySQL прибегнет к обычной сортировке для упорядочения результатов, а не станет читать строки из таблицы в порядке, задаваемом индексом.

# Используемые источники

1. <https://dev.mysql.com/doc/refman/5.7/en/optimization.html>
2. <https://dev.mysql.com/doc/refman/5.7/en/execution-plan-information.html>
3. 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.
4. Чарльз Белл, Мэтс Киндал и Ларс Талманн. Обеспечение высокой доступности систем на основе MySQL / Пер. с англ. — М. : Издательство "Русская редакция"; СПб. : БХВ-Петербург, 2012. — 624 с.
5. Линн Бейли. Head First. Изучаем SQL. — СПб.: Питер, 2012. — 592 с.
6. 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.
7. Дейт К. Дж. SQL и реляционная теория. Как грамотно писать код на SQL. — Пер. с англ. — СПб.: Символ-Плюс, 2010. — 480 с.
8. Кузнецов М.В., Симдянов И.В. MySQL на примерах. — СПб.: БХВ-Петербург, 2007. — 592с.
9. Кузнецов М.В., Симдянов И.В. MySQL 5. — СПб.: БХВ-Петербург, 2006. — 1024с.
10. Дейт К. Дж. Введение в системы баз данных, 8-е издание.: Пер. с англ. — М.: Издательский дом "Вильямс", 2005. — 1328 с.
11. Карвин Б. Программирование баз данных SQL. Типичные ошибки и их устранение. — Рид Групп, 2011. — 336 с.

**Урок 10**

NoSQL

[NoSQL базы данных](https://docs.google.com/document/d/1EoqVvggI0IzF8MUOQLSf5XWOQlp9YvBriT-jXDCXeKE/edit#heading=h.a87pq6wf3svj)

[CAP-теорема](https://docs.google.com/document/d/1EoqVvggI0IzF8MUOQLSf5XWOQlp9YvBriT-jXDCXeKE/edit#heading=h.mk9x003pi3m7)

[Redis](https://docs.google.com/document/d/1EoqVvggI0IzF8MUOQLSf5XWOQlp9YvBriT-jXDCXeKE/edit#heading=h.rqxdufghfame)

[Клиент redis-cli](https://docs.google.com/document/d/1EoqVvggI0IzF8MUOQLSf5XWOQlp9YvBriT-jXDCXeKE/edit#heading=h.koisw3u9figg)

[Справочная система](https://docs.google.com/document/d/1EoqVvggI0IzF8MUOQLSf5XWOQlp9YvBriT-jXDCXeKE/edit#heading=h.2bi1b2mw3nhc)

[Операции с ключами](https://docs.google.com/document/d/1EoqVvggI0IzF8MUOQLSf5XWOQlp9YvBriT-jXDCXeKE/edit#heading=h.9sb9gabg46dk)

[Время жизни ключа](https://docs.google.com/document/d/1EoqVvggI0IzF8MUOQLSf5XWOQlp9YvBriT-jXDCXeKE/edit#heading=h.3wwflkekkci9)

[Типы ключей](https://docs.google.com/document/d/1EoqVvggI0IzF8MUOQLSf5XWOQlp9YvBriT-jXDCXeKE/edit#heading=h.6mdhu6mnmpqg)

[Коллекционные типы](https://docs.google.com/document/d/1EoqVvggI0IzF8MUOQLSf5XWOQlp9YvBriT-jXDCXeKE/edit#heading=h.3z2p1mm37ema)

[Базы данных](https://docs.google.com/document/d/1EoqVvggI0IzF8MUOQLSf5XWOQlp9YvBriT-jXDCXeKE/edit#heading=h.ng427274sape)

[MongoDB](https://docs.google.com/document/d/1EoqVvggI0IzF8MUOQLSf5XWOQlp9YvBriT-jXDCXeKE/edit#heading=h.pk2umztsa85e)

[СУБД полнотекстового поиска ElasticSearch](https://docs.google.com/document/d/1EoqVvggI0IzF8MUOQLSf5XWOQlp9YvBriT-jXDCXeKE/edit#heading=h.7xdvbyuszif3)

[СУБД ElasticSearch](https://docs.google.com/document/d/1EoqVvggI0IzF8MUOQLSf5XWOQlp9YvBriT-jXDCXeKE/edit#heading=h.d5i9rqkjo329)

[Колоночная СУБД ClickHouse](https://docs.google.com/document/d/1EoqVvggI0IzF8MUOQLSf5XWOQlp9YvBriT-jXDCXeKE/edit#heading=h.b8yxth26jgt2)

[Используемые источники](https://docs.google.com/document/d/1EoqVvggI0IzF8MUOQLSf5XWOQlp9YvBriT-jXDCXeKE/edit#heading=h.zcfqxij5sjoo)

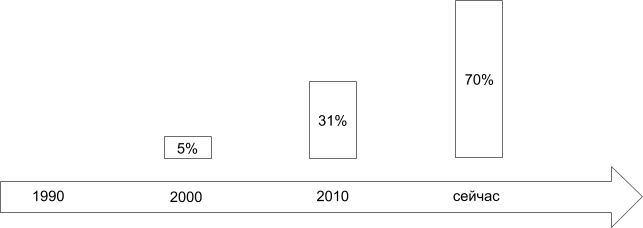
# NoSQL-базы данных

Базы данных решают проблему: «я хочу сохранить некие данные и позже снова их просмотреть». Сейчас нет одного правильного решения этой проблемы, но зато есть много разных подходов, более или менее приемлемых, в зависимости от ситуации.

При реализации базы данных обычно приходится выбирать какой-то один из них. Достаточно сложно получить код, который был бы одновременно надежным, масштабируемым и очень быстрым. Попытка получить все сразу в одном программном продукте почти гарантированно ведет к неудачной реализации.

Поэтому часто выбор той или иной базы данных зависит от обстоятельств. Любая база данных предназначена для конкретного стиля использования. Задача разработчика заключается в том, чтобы разобраться, в каких обстоятельствах те или иные программные продукты проявят себя лучше всего.

Так было не всегда, много десятилетий реляционные базы данных считались хорошим компромиссом для решения почти всех задач. Разработка новых технологий и решений велась от существующих движков, количество игроков на рынке баз данных год от года сокращалось, компании поглощались, вытеснялись конкруентами.



Ситуация резко изменилась в начале 2000-х годов, когда интернет из дорогой игрушки превратился в новую электронную инфраструктуру, и многие операции по общению с госорганами, покупками, обучением перешли из офлайн-сферы в онлайн.

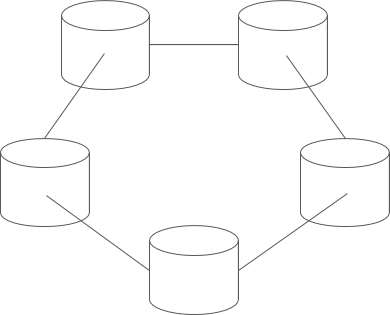
Как следствие, доля людей, которые используют интернет, постоянно росла. В ряде стран, в том числе и в России, она достигла 70–80 % от всего населения. Если раньше большие данные были относительно редким явлением, то сейчас они стали массовыми — даже небольшие компании стали сталкиваться с гигантским объемом информации.

Не важно, что делать с этими потоками: обсчитывать, анализировать и искать корреляции, обучать нейронные сети или даже просто хранить массив данных.

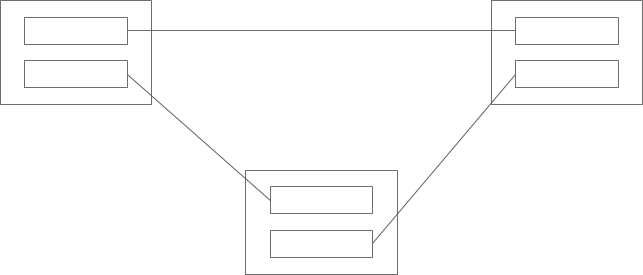
Эти потоки просто появились, причем массово. Проблемы хранения и обработки, которые они стали вызывать, тоже стали массовыми, и возникают не только в больших корпорациях, но и в стартапах. Что это за проблемы?

1. Одного, даже мощного, сервера стало не хватать для обсчета этих потоков данных. Базы данных стали распределенными. Увеличение количества компьютеров, участвующих в работе кластера, приводит к возрастанию отказов отдельных компонентов. В результате необходимо менять подход к отдельным узлам, строить распределенные базы данных таким образом, чтобы потеря нескольких узлов не отражалась на целостности данных.

Распределенные приложения стали все чаще и чаще строиться при помощи кластерных решений, AWS, Kubernetes. В них любая нода может перестать быть доступной в любой момент. Например, нода может оказаться перегруженной или отказать, и контейнер будет выключен и развернут на другой. Т.е., в больших кластерных решениях контейнеры часто становятся недоступными без предупреждения. На это не были рассчитаны старые классические СУБД.

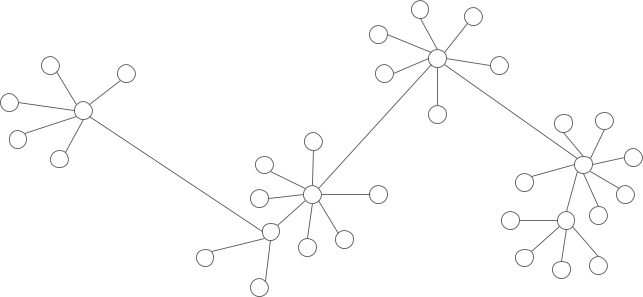


2. Одного сервера стало не хватать для хранения всей базы данных. Ее приходится разбивать на части, шарды, и хранить их на разных серверах. Меняются подходы для построения распределенных приложений.



3. Данных стало очень много, а их формат зачастую неудобен для быстрой обработки в рамках реляционных приложений.

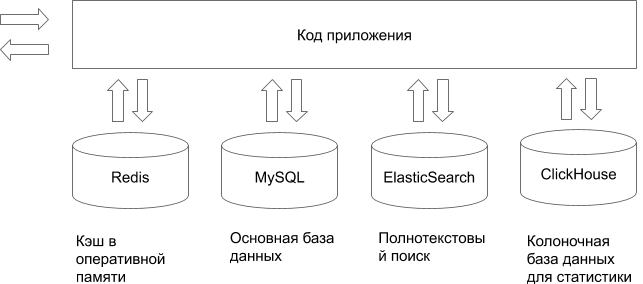
4. Сетевая природа интернета и связей пользователей в социальных сетях потребовала пересмотра реляционной модели, которая зачастую неудобна для организации деревьев или требует слишком объемных вычислительных мощностей. Потребовались другие форматы хранения.



5. Интенсивность потока настолько возросла, что потребовалось масштабировать не только чтение, но и прием данных. Резко увеличилась потребность в очень производительных базах данных.

Эта цепочка событий взорвала уже устоявшийся рынок баз данных. Системы управления базами данных стали меняться, а на рынке стали появляться новые игроки. Сейчас, спустя 40 лет, мы снова находимся в точке перелома, когда базы данных адаптируются к новым правилам игры.

Поэтому, если еще 20 лет назад СУБД от одного поставщика решала весь спектр задач долговременного хранения данных, то сейчас при создании приложения мы зачастую вынуждены пользоваться сразу несколькими СУБД, которые объединяются логикой приложения.



Например, часто требуется интегрировать в базу данных полнотекстовый поиск. Существующего в базах данных полнотекстового поиска иногда недостаточно и может использоваться специализированная база данных. При этом база данных полнотекстового поиска может быть не очень надежной, не поддерживать транзакции. Поэтому во многих приложениях, чтобы удовлетворить все требования, необходимо комбинировать реляционную СУБД для долговременного хранения и СУБД полнотекстового поиска.

Поддержка транзакций и правил SQL может приводить к невысокой производительности СУБД долговременного хранения. Поэтому данные, которые меняются, не часто могут помещаться в специализированную СУБД для кеширования. Благодаря тому, что такая СУБД полностью расположена в оперативной памяти и спроектирована для максимально быстрой работы, можно добиться существенного увеличения производительности системы.

Обсчет статистики в основной базе данных может занимать длительное время, блокировать большие диапазоны строк в таблице. Поэтому для хранения и обсчета статистики лучше использовать колоночные СУБД. Они не позволяют обновлять и удалять данные, поэтому не предназначены для выполнения роли основной базы данных. Однако они позволяют в режиме реального времени получать агрегационные данные. Так как работа будет производиться в отдельной базе данных, то эти вычисления не будут затрагивать остальные хранилища и влиять на производительность системы.

Новые базы, которые пришли на смену реляционным, стали называть NoSQL-базами данных. Термин стал популярен в 2009 году и стал обозначать нереляционные базы данных. Такие базы данных часто поддерживают SQL-подобный синтаксис, поэтому NoSQL сейчас часто расшифровывают, как «Не только SQL».

Базы данных «ключ-значение». В них вместо таблиц используется ключ, с которым ассоциируется один или несколько атрибутов. Все данные ключа хранятся вместе и часто дублируются. К таким базам данных относятся Riak, Dynamo DB, Memcached и Redis.

Столбцовые базы данных. Если в реляционной модели содержимое строк хранится вместе, то в столбцовых базах данных вместе хранятся данные одного столбца. Каждый столбец таблицы зачастую хранится в отдельном файле.

Это позволяет более эффективно сжимать данные и сортировать столбцы независимо от других данных таблицы. К таким базам данных относятся Cassandra, HBase, Clickhouse.

Документоориентированные базы данных. В них единица хранения — целый документ, часто в формате JSON, XML или YML. Большинство документоориентированных БД не навязывают какой-либо схемы для данных в документах. В документ можно добавлять произвольные ключи и значения. Поэтому документоориентированные БД обычно называют бессхемными (**schemaless**) или неструктурированными. К таким база данных относятся CouchDB и MongoDB.

Графовые базы данных. В них данные представлены в виде графа со множеством вершин и соединяющих их ребер. Основной упор в таких базах данных делается на связях, они часто применяются для построения социальных сетей. К таким базам данных относятся FlockDB, Neo4j, Polyglot.

Объектные базы данных. В них данные хранятся не в виде отношений, состоящих из строк и столбцов, а в виде объектов, что упрощает работу с базой из объектно-ориентированного приложения. В результате с базой данных можно работать без хранимых процедур и средств объектно-реляционного отображения (ORM). К таким базам данных относятся db4o, InterSystems Cache.

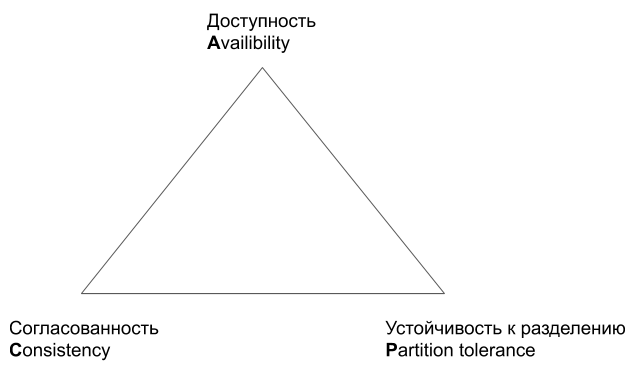
Базы данных полнотекстового поиска. В таких базах данных в качестве основной структурной единицы часто выбирается документ и используется неструктурированная система хранения. Основной упор в таких база[ данных делается на поисковый механизм и обеспечениt гибкости настройки релевантности поиска. К базам данных этого типа относятся Solr и ElasticSearch.

Таким образом, в современном мире больших данных и распределенных систем реляционную модель необходимо дополнять новыми решениями.

# CAP-теорема

Реляционные СУБД проектировались во времена, когда одна база данных убиралась на одном сервере, максимум в одном дата-центре. В реальности мы часто имеем дело с несколькими серверами, а зачастую и несколькими дата-центрами.

NoSQL-базы данных решают самые разные задачи, но одна из главных — поиск компромисса между согласованностью, доступностью и способностью к разделению. Проблему сформулировал Эрик Брюер в 2000 году, поэтому ее часто называют теоремой Брюера, но более известна она под названием CAP-теоремы: по первым трем буквам Согласованности, Доступности и способности к разделению.



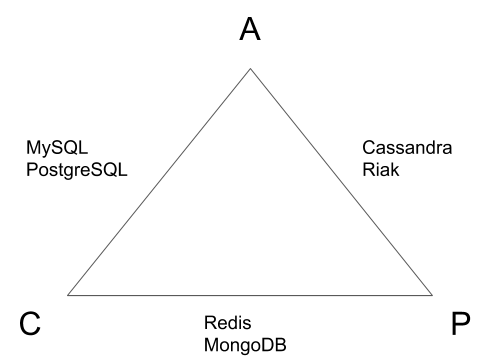
Согласованность означает, что любая операция чтения всегда возвращала самое последнее записанное значение. Например, когда мы настраиваем репликацию и слейв-сервер отстает от мастер-сервера, мы не обеспечиваем согласованность данных. Прочитав старые данные со слейв-сервера, мы можем получить не самый последний результат.

Доступность означает, что клиенты всегда имеют возможность читать и записывать данные.

Способность к разделению означает, что базу данных можно распределить по нескольким машинам, и она продолжит работу даже после отказа сегментов сети.

CAP-теорема утверждает, что в любой системе можно гарантировано обеспечивать выполнение только двух из этих трех требований. Теорема была доказана в 2002 году Сетом Гильбертом и Ненси Линч.

В современном мире у нас приложение почти всегда распределено в сети. При этом сетевые сбои весьма вероятны. Потери пакетов и сетевые задержки могут вызывать временное разделение сети, а значит, из компромиссов остаются либо согласованность, либо доступность.



У каждой базы данных есть своя специализация. Она может удовлетворять двум из трех условий CAP-теоремы. Обеспечение согласованности и доступности означает, что в случае разделения сети система блокируется, т. е., такие базы данных должны работать в одном дата-центре.

На нижней грани согласованности и способности к разделению, как правило, вводится сегментирование или шардирование данных. Однако в случае отказа часть данных может быть недоступна.

Если отдается предпочтение доступности и способности к разделению, могут возвращаться неточные данные. Зато база данных всегда доступна, даже в случае разделения сети. Не всегда базы данных можно однозначно отнести к одной из граней CAP-теоремы, многое зависит от режима, в котором они доступны и от характера сбоев.

# Redis

Redis — исключительно быстрый однопоточный сервер для хранения данных в оперативной памяти. Данные в нем хранятся по принципу «ключ-значение».

Очень часто Redis используется для кеширования. Однако он может использоваться и в качестве очереди, и для обмена Pub/Sub-сообщениями.

За счет того, что процессору не приходится переключаться между потоками, при большом количестве соединений достигается огромная производительность.

Причем Redis — это не просто хранилище «ключ-значение», значения могут быть как обычными строками, так и коллекциями (массивами, хешами, множествами), над которыми можно осуществлять операции.

Допускается сохранение данных на жесткий диск, что разрешает проблему холодного старта, когда после запуска должно пройти некоторое время, прежде чем заполнится кеш и страницы будут отдаваться клиенту быстро.

Ключам допускается назначение времени жизни, тем самым можно очищать старые данные в фоновом режиме.

Redis поддерживает транзакции, позволяя объединять несколько команд в блоки, результат выполнения которых сохраняется только в том случае, если все они завершились успешно.

Redis предоставляет механизм репликации. Более того, можно построить Sentinel-кластер из нескольких Redis-серверов, в котором участники будут следить за работоспособностью друг друга. В случае выхода из строя master-сервера, оставшиеся работоспособные ноды могут «проголосовать» и выбрать новый master-сервер. В результате выход из строя одного или нескольких участников кластера не приводит к отказу сервиса.

Благодаря встроенному механизму подписки (Pub/Sub) на основе Redis можно строить очереди заданий. Допустим, при загрузке изображения из него нужно нарезать десяток вариантов с различными размерами. Эта операция может занять длительное время и лучше ее выполнить в фоновом режиме, не задерживая пользователя. В этом случае задание можно поместить в очередь и завершить обслуживание запроса. В очереди, как правило, на отдельном сервере, задание будет рано или поздно выполнено.

Redis допускает создание скриптов на языке программирования Lua. На Lua также разрабатываются расширения для веб-сервера Nginx. В результате можно помещать данные непосредственно в Redis и забирать их из Nginx.

За счет того, что в обслуживании конечных клиентов участвуют исключительно быстрые сервера Nginx и Redis, можно получить исключительную производительность при обслуживании огромного количества одновременных запросов.

В небольшом ролике нам не удастся затронуть все возможности и варианты использования Redis. Однако введение его в проект предоставляет широкие возможности по масштабированию и построению отказоустойчивых сервисов.

Redis — одна из самых популярных NoSQL-баз данных в современных проектах.

## Клиент redis-cli

Для доступа к redis-серверу используется клиент **redis-cli**:

|  |
| --- |
| redis-cli |

По умолчанию клиент пытается соединиться с локальным сервером по порту 6379. Для выхода из клиента используется команда **QUIT** или **EXIT**.

Помимо **redis-cli**, в составе утилит установленного Redis можно найти **redis-benchmark**, которая измеряет  производительность ключевых команд на текущем сервере.

|  |
| --- |
| $ redis-benchmark -n 100000 |

Как видно, производительность сервера Redis может достигать свыше 100 000 RPS (операций в секунду), даже на запись. Столь впечатляющие результаты достигаются за счет EventLoop-механизма, когда соединения обрабатывает один поток в неблокирующем режиме.

## Справочная система

Клиент **redis-cli** содержит удобную справочную систему. Воспользоваться ею можно при помощи команды **HELP**, после которой следует указать название команды.

Например, запросить справочную информацию по команде **PING** можно следующим образом:

|  |
| --- |
| HELP PING |

Первое поле указывает название команды, в данном случае **PING**, и список возможных параметров. Поле **summary** кратко описывает назначение команды, а **since** — версию Redis, начиная с которой команда доступна.

Поле **group** указывает группу, к которой относится команда.

Воспользуемся символом **@**: если разместить после него название группы, можно получить список всех команд, входящих в группу.

|  |
| --- |
| HELP @connection |

В справочной системе работает функция автодополнения: для получения подсказки достаточно набрать команду **HELP @**, многократно нажимая клавишу **<TAB>**, можно перебирать доступные группы. Ниже приводится список доступных групп:

* **@generic** — команды общего назначения;
* **@string** — команды для работы со строками;
* **@list** — команды для работы со списками;
* **@set** — команды для работы с множествами;
* **@sorted\_set** — команды для работы с сортированными множествами;
* **@hash** — команды для работы с хешами;
* **@pubsub** — команды для организации подписчиков;
* **@transactions** — команды транзакционного механизма;
* **@connection** — команды управления соединением с сервером;
* **@server** — команды управления сервером;
* **@scripting** — автоматизация обработки данных;
* **@hyperloglog** — команды для работы с вероятностным алгоритмом подсчета уникальных элементов;
* **@cluster** — команды для обслуживания кластера redis-серверов;
* **@geo** — команды для работы с гео-координатами.

## Операции с ключами

Самый простой способ вставить новое значение в базу данных — воспользоваться командой **SET**:

|  |
| --- |
| SET key 'Hello world!' |

В качестве первого аргумента команда принимает ключ, а в качестве второго — значение. Извлечь вставленное командой значение можно при помощи команды **GET**:

|  |
| --- |
| GET key |

Команда принимает в качестве аргумента ключ и возвращает полученное значение.

Команда **MSET** позволяет вставить за один раз сразу несколько значений, при этом ключи и значения отделяются друг от друга пробелом:

|  |
| --- |
| MSET fst 1 snd 2 thd 3 fth 4 |

Давайте извлечем вставленные командой **GET** значения:

|  |
| --- |
| GET fst GET fth |

При помощи команды **MGET** можно извлекать сразу несколько значений:

|  |
| --- |
| MGET fst snd thd fth |

Так как структура хранимых данных предельно проста, полностью обновить запись «ключ-значение» можно при помощи команды создания нового значения — **SET**.

|  |
| --- |
| SET "key" "old" GET key SET "key" "new" GET key |

Однако Redis допускает и более сложные команды обновления значений. Так, с помощью команды **APPEND** можно добавить в конец существующей строки новое значение:

|  |
| --- |
| SET key "hello" APPEND key " world!" GET key SET count 0 |

При помощи команды **INCR** можно увеличить целочисленное значение на единицу:

|  |
| --- |
| INCR count GET count |

При помощи **INCRBY** можно увеличить целочисленное значение на произвольное целое значение:

|  |
| --- |
| INCRBY count 5 GET count |

Команде **INCRBY** можно передавать отрицательное значение. В этом случае будет осуществляться вычитание.

|  |
| --- |
| INCRBY count -3 GET count |

Впрочем, для вычитания существуют специальные команды **DECR** и **DECRBY**.

|  |
| --- |
| SET count 10 DECR count GET count DECRBY count 5 GET count |

Специальная команда **INCRBYFLOAT** позволяет прибавлять и удалять цифры с плавающей точкой:

|  |
| --- |
| GET count INCRBYFLOAT count 0.5 GET count INCRBYFLOAT count -1.3 |

Для удаления пары «ключ-значение» предназначена команда **DEL**, которая принимает в качестве параметра ключ удаляемой пары.

|  |
| --- |
| DEL key GET key |

Для извлечения данных из Redis всегда требуется ключ, поэтому важно знать список всех ключей, которые хранятся в базе данных.

Для получения такого списка используется команда **KEYS**, которая в качестве единственного аргумента принимает шаблон поиска. Если в качестве шаблона указать звездочку **\***, будет возвращен список всех доступных ключей:

|  |
| --- |
| KEYS \* |

Звездочку можно использовать в составе более сложных шаблонов, например в следующем шаблоне извлекаются все ключи, начинающиеся с символа **f**:

|  |
| --- |
| KEYS f\* |

Для переименования ключа предназначена команда **RENAME**, которая принимает в качестве первого аргумента названия ключа переименовываемой пары, а в качестве второго — новое имя, которое ему назначается:

|  |
| --- |
| RENAME fst snd GET snd GET fst |

## Время жизни ключа

Одна из основных специализаций Redis — быстрый кеш, расположенный в оперативной памяти. В связи с этим особую роль получает актуальность кеша, срок жизни которого обычно не очень велик.

|  |
| --- |
| SET timer "one minute" EXPIRE timer 60 |

Для задания срока хранения ключей предназначена команда **EXPIRE**, в качестве первого параметра которой передается имя ключа, а в качестве второго — время его жизни в секундах. Если срок жизни ключа не истек, то команда **EXISTS** возвращает значение 1, в противном случае возвращается 0.

|  |
| --- |
| TTL timer |

Чтобы выяснить, сколько секунд осталось до истечения срока жизни ключа, можно воспользоваться командой **TTL**. Ограничение на срок хранения можно отменить, воспользовавшись командой **PERSIST**:

|  |
| --- |
| PERSIST timer TTL timer |

## Типы ключей

Redis поддерживает несколько типов данных:

* строки — последовательность символов, заключенных в кавычки;
* числа — целые и с плавающей точкой, позволяющей прибавлять и вычитать значения;
* список — фактически массивы данных;
* хеш — хранит пары «ключ-значение», то есть помимо ключа к самому хешу, каждый элемент, который входит в его состав, также снабжается своим ключом;
* множество — неупорядоченная коллекция уникальных элементов, дублирующие значения в которой отбрасываются автоматически;
* отсортированное множество — точно так же, как в хешах, этот тип коллекции хранит пару ключ-значение, только в качестве ключа выступает числовое значение, задающее порядок следования элементов, что роднит коллекцию со списком. Как и традиционные множества, отсортированные множества сохраняют только уникальные значения.

## Коллекционные типы

Вложение коллекций не допускается, то есть коллекции не могут выступать в качестве элементов других коллекций.

В любой момент можно узнать тип значения при помощи специальной команды **TYPE**.

|  |
| --- |
| SET key "hello world!" TYPE key |

Для создания хэша можно воспользоваться командой **HSET**, которая принимает в качестве первого параметра ключ хеша, в качестве второго параметра ключ пары, а в качестве третьего — значение. Ниже создается хеш **admin** с регистрационными данными пользователя.

|  |
| --- |
| HSET admin login "root" HSET admin pass "password" HSET admin register\_at "2017-09-01" |

Создать представленный выше хэш можно при помощи одной команды **HMSET**, которая позволяет задать сразу все пары «ключ-значение»:

|  |
| --- |
| HMSET admin login "root" pass "password" register\_at "2017-09-01" |

Для чтения элементов хэша можно воспользоваться командой **HGET**:

|  |
| --- |
| HGET admin login |

Или извлечь все содержимое хэша при помощи команды **HVALS**:

|  |
| --- |
| HVALS admin |

Проверить существование поля с заданным именем можно с помощью команды **HEXISTS**:

|  |
| --- |
| HEXISTS admin login HEXISTS admin none |

Кроме того, в любой момент можно запросить все ключи хэша при помощи команды **HKEYS**:

|  |
| --- |
| HKEYS admin |

С помощью команды HGETALL можно извлечь все содержимое хеша, включая ключи и значения:

|  |
| --- |
| HGETALL admin |

Выяснить количество элементов в хэше можно с помощью команды **HLEN**:

|  |
| --- |
| HLEN admin |

Для вставки значений в множество можно воспользоваться командой **SADD**, первый параметр которой обозначает имя коллекции, а второй — вставляемое значение:

|  |
| --- |
| SADD email support@softtime.info SADD email igor@softtime.info SADD email support@softtime.info |

Команда **SADD** позволяет вставлять в коллекцию сразу несколько значений:

|  |
| --- |
| SADD email igorsimdyanov@gmail.com igor@simdyanov.ru igor@softtime.ru i.simdyanov@rambler-co.ru |

Сколько бы повторяющихся значений не было вставлено в коллекцию **email**, содержать она будет только уникальные значения, в чем можно убедиться, воспользовавшись командой **SMEMBERS**:

|  |
| --- |
| SMEMBERS email |

Выяснить количество элементов в множестве позволяет команда **SCARD**:

|  |
| --- |
| SCARD email |

Для удаления элемента из коллекции предназначена команда **SREM**:

|  |
| --- |
| SREM email igor@softtime.info |

Для извлечения случайного значения из множества можно воспользоваться командой **SPOP**:

|  |
| --- |
| SPOP email |

Сильная сторона множеств — возможность поиска объединения и пересечения нескольких множеств. Для демонстрации этих возможностей создадим дополнительную коллекцию **subscribers**, также содержащую список электронных адресов:

|  |
| --- |
| SADD subscribers igor@simdyanov.ru igor@softtime.ru SMEMBERS subscribers |

Для поиска общих электронных адресов коллекции **email** и **subscribers** можно воспользоваться командой **SINTER**:

|  |
| --- |
| SINTER email subscribers |

Для поиска по множеству email-адресов, не входящих во множество **subscribers**, можно воспользоваться командой **SDIFF**:

|  |
| --- |
| SDIFF email subscribers |

При помощи команды **SUNION** можно объединить оба множества **email** и **subscribers** в одно (дубликаты автоматически отбрасываются).

|  |
| --- |
| SUNION subscribers email |

## Базы данных

Как и в СУБД, в Redis имеются базы данных, однако у них нет названия и они нумерованные. По умолчанию утилита **redis-cli** открывает базу данных 0, для того, чтобы переключиться на другую базу данных, например 1, следует воспользоваться командой **SELECT**:

|  |
| --- |
| localhost:6379> SET key value localhost:6379> GET key localhost:6379> SELECT 1 localhost:6379[1]> GET key localhost:6379[1]> SET key value1 localhost:6379[1]> SELECT 0 localhost:6379> GET key |

Количество баз данных по умолчанию ограничено 16, однако это значение можно поменять в конфигурационном файле сервера, изменив значение директивы **databases**.

# MongoDB

MongoDB — это документоориентированная база данных с открытым кодом. Данные в ней хранятся в виде JSON-документов, языком запроса выступает JavaScript. В результате, например для отображения страницы сайта, нет необходимости собирать данные из нескольких таблиц: JSON-документ уже сразу может содержать всю необходимую информацию для формирования страницы. Это позволяет добиваться значительной производительности.

С другой стороны, структура документа никак не регламентируется. Можно записывать данные в одном формате, а при необходимости — изменить его и начать записывать дополнительные поля или убрать те ключи, которые больше не требуются.

Структуру документа диктует приложение, а не база данных. В этом и преимущество MongoDB — вы можете обрабатывать данные неизвестной заранее структуры, — и ее недостаток — в любой момент структура документа может быть изменена, а в выборке документы одной природы могут иметь разную организацию.

MongoDB поддерживает репликацию и шардирование, поэтому прекрасно масштабируется на чтение.

JSON — это сокращение от JavaScript Object Notation. Сами JSON-документы состоят из ключей и значений и допускают произвольную глубину вложения, поэтому MongoDB чрезвычайно популярна при разработке на JavaScript и Node.js.

Документ — это набор, состоящий из ключей и значений. Значение может быть представлено простым типом: строкой, числом, датой, а также массивом и другим документом.

На рисунке выше ключ **tags** принимает в качестве значения массив, а **comments** — массив JSON-документов. Такой подход сильно отличается от принятого в реляционной модели, когда мы добиваемся нормализации и под каждый из представленных типов данных (тегов, комментариев, пользователей) создаем отдельную таблицу.

Преимущества документа заключается в том, что вся необходимая информация хранится вместе и доступна сразу, не требуя операции соединения таблиц. В этом же заключается недостаток: данные денормализованы и часто дублируются.

Для работы с MongoDB необходим запущенный сервер MongoDB. Убедиться, что он запущен, можно, поискав среди процессов операционной системы **mongod**.

|  |
| --- |
| ps aux | grep mongod |

После этого получить доступ к базе данных можно при помощи консольного клиента **mongo**, который входит в состав дистрибутива:

|  |
| --- |
| mongo |

Давайте запросим текущую версию сервера

|  |
| --- |
| db.version() |

Запросы к базе данных выглядят как вызовы методов объектов языка JavaScript. Обратите внимание, что, если вы вызываете метод без указания круглых скобок, вы получаете его исходный код:

|  |
| --- |
| db.version function () {     return this.serverBuildInfo().version;  } |

Переключить текущую базу данных можно при помощи команды **use**:

|  |
| --- |
| use shop |

База данных автоматически появляется, как только в нее будет вставлен хотя бы один документ. Запросить список баз данных можно при помощи команды:

|  |
| --- |
| show dbs |

Как видим в этом списке базы данных shop нет, так в ней пока нет ни одного документа. Давайте что-нибудь вставим:

|  |
| --- |
| db.shop.insert({name: 'Ольга'}) |

Повторно вызываем команду **show dbs** и видим, что база данных **shop** появилась в списке:

|  |
| --- |
| show dbs |

Чтобы найти только что вставленное значение, следует воспользоваться методом **find()**:

|  |
| --- |
| db.shop.find() { "\_id" : ObjectId("5b49de717aa5dc224acb5ff8"), "name" : "Ольга" } |

Метод **find()** возвращает вставленный документ вместе с добавленным идентификатором объекта. В любом документе должен быть первичный ключ, который хранится в поле **\_id**. Его можно вставлять самостоятельно при условии, что он имеет уникальное значение. Однако, как правило, его не указывают при вставке и MongoDB назначает уникальное значение этому полю самостоятельно.

Давайте вставим еще одно значение:

|  |
| --- |
| db.shop.insert({name: 'Александр'}) |

Теперь в коллекции должно быть два документа, давайте проверим это, выполнив команду **count**:

|  |
| --- |
| db.shop.count() |

Да, так и есть.

|  |
| --- |
| db.shop.find() |

Как и раньше, можно запросить все документы при помощи метода **find()**. Однако **find()** может принимать селектор запроса, JSON-объект, с которым сравниваются все документы коллекции:

|  |
| --- |
| db.shop.find({name: 'Ольга'}) |

Если соответствие находится, запись попадает в результирующий ответ, если же документ не соответствует шаблону — он отбрасывается.

Для обновления используется метод **update**, которому задается минимум два аргумента. Первый определяет, какие документы обновлять, второй — как следует модифицировать отобранные документы.

Давайте обновим документ пользователя Ольги, добавив ей электронный адрес:

|  |
| --- |
| db.shop.update({name: 'Ольга'}, {$set: { email: 'olga@gmail.com' }}) |

Обратите внимание на ключ **$set**, который начинается с символа доллара: фактически это обозначение команды по вставке нового значения.

Если мы заходим удалить только что вставленное значение, мы можем воспользоваться ключом **unset**:

|  |
| --- |
| db.shop.update({name: 'Ольга'}, {$unset: { email: '' }}) db.shop.find() |

Вставлять можно не только простые значения, но и коллекции:

|  |
| --- |
| db.shop.update({name: 'Ольга'}, {$set: { contacts: { email: ['olga@gmail.com', 'olga@mail.ru'], skype: 'olgashop' }}}) db.shop.update({name: 'Александр'}, {$set: { contacts: { email: ['alex@gmail.com'], skype: 'alexander' }}}) db.shop.find() |

Если мы захотим искать по вложенным документам, используем специальную вложенную нотацию. Давайте найдем пользователей со Skype **alexander**:

|  |
| --- |
| db.shop.find({ 'contacts.skype': 'alexander' }) |

Как видим, мы можем обратиться к ключу вложенной структуры. Точка между **contacts** и **skype** означает, что нужно найти ключ **contacts**, который указывает на вложенный объект **skype**, а затем сравнивать значение этого вложенного ключа с указанным в запросе.

Допустим мы ходим добавить Александру новый электронный адрес [alex@mail.ru](mailto:alex@mail.ru). Можно было бы снова воспользоваться оператором **$set**, но это означало бы, что надо переписать и отправить на сервер весь массив электронных адресов. Так как нам нужно добавить в массив только один элемент, мы можем использовать оператор **$push**:

|  |
| --- |
| db.shop.update({name: 'Александр'}, {$push: { 'contacts.email': 'alex@mail.ru' }}) |

Для удаления данных из коллекции предназначен метод **remove()**:

|  |
| --- |
| db.shop.remove({name: 'Ольга'}) db.shop.find() |

Если мы не будем задавать методу **remove** селектор, то получим сообщение об ошибке:

|  |
| --- |
| db.shop.remove() |

Для удаления всей коллекции можно воспользоваться методом **drop()**:

|  |
| --- |
| db.shop.drop() db.shop.find() show dbs |

Обратите внимание, так как в коллекции **shop** были удалены все записи, такая коллекция больше не выводится в отчете команды **show dbs**.

Документы могут быть довольно объемные, поэтому в консоли с ними может быть неудобно работать. В случае MongoDB следует обратить внимание на клиенты с графическим интерфейсом, например Robomongo.

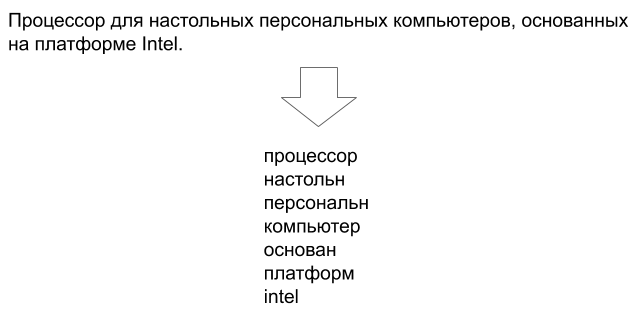
# СУБД полнотекстового поиска ElasticSearch

Пользователям все чаще приходится работать с большими объемами информации. Поэтому поле поиска — зачастую первое место, куда обращаются пользователи, чтобы приступить к изучению проблемы и найти ответы.

Поисковые машины, такие как Google или Яндекс, являются точками входа почти для всех пользователей Интернет. Пользователи ожидают от приложений такого же быстрого и точного ответа, как в больших поисковых системах. Поэтому релевантность поиска в современных приложениях приобретает большую важность.

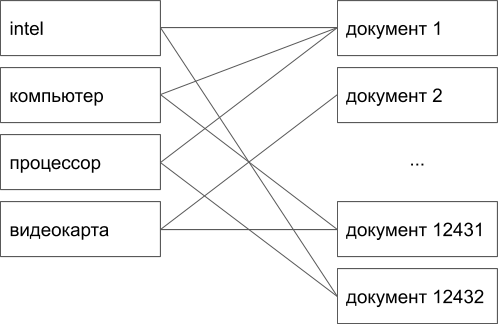
**Полнотекстовый поиск**

Базы данных полнотекстового поиска, как правило, документоориентированные, т. е., документ — это минимальная единица содержимого для хранения, поиска и возврата. Именно документ — главная цель поиска.



Поля документа подвергаются анализу, в ходе которого текстовые поля преобразуются в лексемы или токены. Как правило, они переводятся в нижний регистр, суффиксы и приставки отбрасываются, как и слишком часто встречающиеся слова, например предлоги. Часто встречающиеся слова нарушают релевантность поиска. Даже если слово не общеупотребительное, но слишком часто встречается в документах базы данных, ему назначается низкий вес, чтобы снизить его влияние на результат запроса. Если слово встречается слишком часто, например, более чем в 50 % документов — оно отбрасывается.

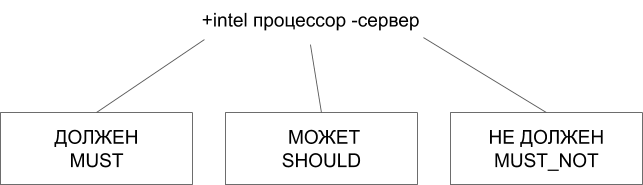
Поэтому чем больше ваша база данных, тем лучше работает полнотекстовый поиск. Если у вас в базе данных всего два документа, слова буду встречаться как раз в 50 % документов и полнотекстовый поиск будет работать из рук вон плохо.



По завершению анализа документы индексируются. Лексемы, извлеченные на этапе анализа, сохраняются отдельно и используются в дальнейшем для организации поиска по документу, также сохраняется исходный непреобразованный текст, чтобы иметь возможность вернуть его пользователю в результатах поиска. Лексемы и документы связываются обратным индексом, в котором по термину всегда можно найти документ.

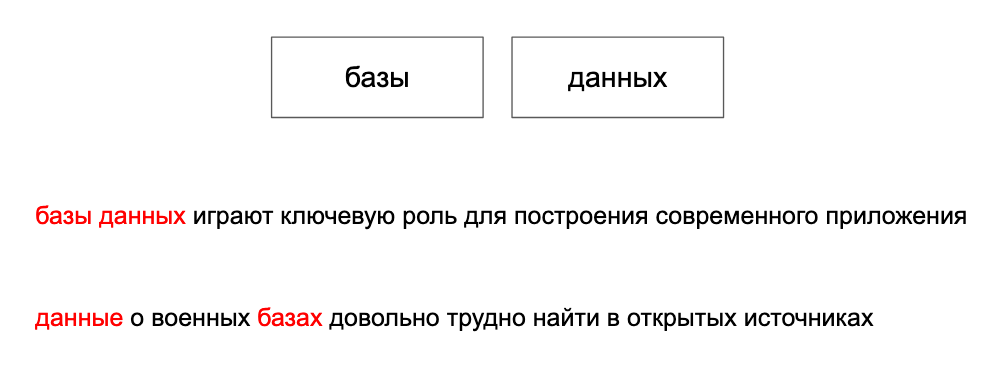
Когда вы ищете одно слово, все довольно просто: достаточно найти в обратном индексе слово и вернуть все документы, на которые ссылается элемент индекса. Если у вас несколько слов, вам потребуется использовать логический поиск, объединяя слова логическим И или ИЛИ.

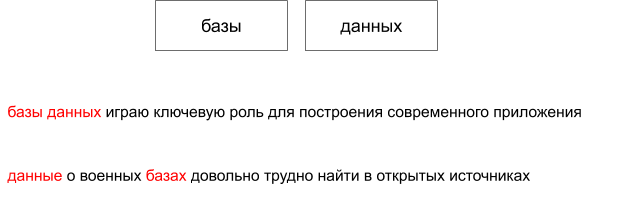
В случае И, все слова обязательно должны присутствовать в документе. В случае ИЛИ в документе должно присутствовать одно из слов. Может применяться логическое НЕ, если в результат не должны попадать документы с выбранными лексемами.



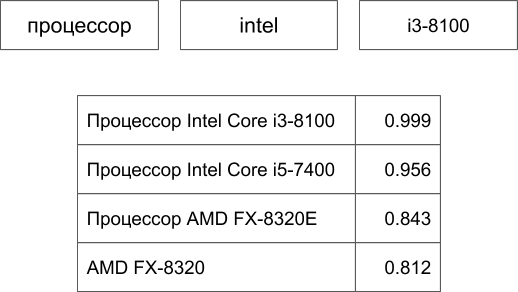
В ElasticSearch вместо И, ИЛИ и НЕ используются похожие операции — МОЖЕТ, ДОЛЖЕН и НЕ ДОЛЖЕН.

* ДОЛЖЕН — требует обязательного присутствия совпадения с термином внутри документа, иначе документ не будет считаться соответствующим запросу;
* МОЖЕТ указывает, что документ может содержать или не содержать совпадения с искомым термином, но при наличии совпадения документ получает более высокий ранг;
* НЕ\_ДОЛЖЕН требует, чтобы содержащий совпадения с термином документ не считался соответствующим запросу, даже если в нем имеются совпадения с операторами ДОЛЖЕН и НЕ\_ДОЛЖЕН.





Относительное расположение слов в запросе часто имеет большое значение. Например, поиск по фразе «База данных» должен возвращать документы с описанием SQL- или NoSQL-баз данных. Если механизм поиска не будет учитывать взаимное расположение терминов, вместо документов, посвященных программному обеспечению, он вернет множество документов, посвященных статистическим данным, военным базам и т. д. Это не совсем то, то ожидает пользователь, поэтому полнотекстовый поиск не только предоставляет возможность поиска по фразе, но и учитывает расстояние между словами — чем оно меньше, тем документ считается более релевантным.



Чтобы наиболее подходящие документы оказывались в начале поиска, документы сортируются по релевантности. Релевантность, в свою очередь, определяется функцией ранжирования. Для каждого документа функция ранжирования вычисляет оценку, определяющую, насколько полно документ соответствует запросу.

Чем чаще термин встречается в документе, тем более полно он соответствует запросу. Однако чем чаще термин встречается во всех документах, тем меньше у него вес в оценке.

## СУБД ElasticSearch

Одна из самых популярных баз данных для полнотекстового поиска — ElasticSearch, которая использует Java-библиотеку Lucene и считается промышленным сервером.

Сервер поддерживает кластеризацию и шардирование, обеспечивает высокую производительность и горизонтальное масштабирование. Для доступа к нему используется REST-подобный интерфейс, т. е., в основе запросов лежит протокол HTTP, поэтому обращаться к серверу можно непосредственно из браузера.

Чтобы воспользоваться ElasticSearch, нам потребуется сервер, который должен быть запущен на вашем компьютере:

|  |
| --- |
| ps aux | grep elastic |

По умолчанию сервер ожидает запросов по порту 9200. Проверить работоспособность сервера мы можем, отправив HTTP-запрос на по адресу **localhost:9200**.

|  |
| --- |
| curl 'http://localhost:9200' |

Итак, мы получаем информацию о сервере и его версии. Здесь мы воспользовались утилитой **curl** для отправки HTTP-запросов из консоли. Для этого можно воспользоваться и браузером. Когда мы обращаемся к ресурсу, используя адресную строку, мы отправляем ему HTTP-запрос методом **GET**.

Однако многие запросы к ElasticSearch требуют не только GET-запросов: например, заполнение индекса идет с использованием HTTP-запросов типа **POST**. Поэтому большую часть запросов мы будем осуществлять через консоль с использованием утилиты **curl**.

Для начала давайте вставим что-нибудь в базу данных ElasticSearch:

|  |
| --- |
| curl -H 'Content-Type: application/json' -X PUT 'http://localhost:9200/shop/products/1?pretty' -d' {   "name" : "Intel Core i5-7400",   "description" : "Процессор для настольных персональных компьютеров, основанных на платформе Intel.",   "price" : "12700.0",   "tags" : [  "комплектующие",  "процессоры",  "Intel"   ],   "created\_at" : "2018-10-10T20:35:12+00:00" }' |

Итак, мы вставили первый документ в индекс **shop** и тип **productions**. Если проводить аналогию с реляционными базами данных: индекс соответствует понятию базы данных, а тип — таблице.

|  |
| --- |
| curl 'http://localhost:9200/shop/products/1?pretty' |

Чтобы извлечь содержимое документа, мы можем воспользоваться следующим GET-запросом. Мы получаем JSON-документ, в поле **\_source** которого мы видим оригинальный документ.

Все ключи, начинающиеся с символа подчеркивания, относятся к служебным. Если нам не нужна мета-информация, мы можем запросить конкретное поле, например, **\_source**:

|  |
| --- |
| curl 'http://localhost:9200/shop/products/1/\_source?pretty' |

Можем детализировать запрос еще больше и запросить конкретное поле документа, например, **description**:

|  |
| --- |
| curl 'http://localhost:9200/shop/products/1/?\_source=description&pretty' |

Давайте вставим еще несколько записей, соответствующих процессорам:

|  |
| --- |
| curl -H 'Content-Type: application/json' -X PUT 'http://localhost:9200/shop/products/2' -d' {   "name" : "Intel Core i3-8100",   "description" : "Процессор для настольных персональных компьютеров, основанных на платформе Intel.",   "price" : "7890.00",   "tags" : [  "комплектующие",  "процессоры",  "Intel"   ],   "created\_at" : "2018-10-10T20:40:23+00:00" }'  curl -H 'Content-Type: application/json' -X PUT 'http://localhost:9200/shop/products/3' -d' {   "name" : "AMD FX-8320E",   "description" : "Процессор AMD.",   "price" : "4780.00",   "tags" : [  "комплектующие",  "процессоры",  "AMD"   ],   "created\_at" : "2018-10-10T20:42:06+00:00" }' |

И материнским платам:

|  |
| --- |
| curl -H 'Content-Type: application/json' -X PUT 'http://localhost:9200/shop/products/4' -d' {   "name" : "ASUS ROG MAXIMUS X HERO",   "description" : "Материнская плата Z370, Socket 1151-V2, DDR4, ATX",   "price" : "19310.00",   "tags" : [  "комплектующие",  "материнские платы",  "ASUS"   ],   "created\_at" : "2018-10-10T20:40:23+00:00" }'  curl -H 'Content-Type: application/json' -X PUT 'http://localhost:9200/shop/products/5' -d' {   "name" : "Gigabyte H310M S2H",   "description" : "Материнская плата H310, Socket 1151-V2, DDR4, mATX",   "price" : "4790.00",   "tags" : [  "комплектующие",  "материнские платы",  "Gigabyte"   ],   "created\_at" : "2018-10-10T20:44:36+00:00" }' |

Для извлечения документов можно использовать search-запрос. При помощи параметра **size** мы можем ограничить выборку только двумя элементами:

|  |
| --- |
| curl 'http://localhost:9200/shop/products/\_search?pretty&size=2' |

Можем отправлять и более сложные запросы, например ориентируясь на тэги. Так, мы можем извлечь все документы, у которых есть тэг «процессоры»:

|  |
| --- |
| curl -H 'Content-Type: application/json' 'http://localhost:9200/shop/products/\_search?pretty' -d' {   "query": {  "bool": {   "filter": {     "term": {       "tags": "процессоры"     }   }  }   } }' |

Однако стоит нам ошибиться в названии тэга и мы получим нулевой результат. Давайте заменим «процессоры» на «процессор»:

|  |
| --- |
| curl -H 'Content-Type: application/json' 'http://localhost:9200/shop/products/\_search?pretty' -d' {   "query": {  "bool": {   "filter": {     "term": {       "tags": "процессор"     }   }  }   } }' |

Ничего не найдено.

Когда мы используем ключевое слово **term** — мы ищем точное совпадение, для включения механизма полнотекстового поиска нам необходимо задействовать match-поиск. Каждый тип имеет свою схему — **mapping**, которая генерируется автоматически при индексации документа. Посмотреть текущий маппинг можно при помощи запроса **\_mapping**.

|  |
| --- |
| curl 'http://localhost:9200/shop/products/\_mapping?pretty' |

Сейчас он не поддерживает русский язык, давайте включим такую поддержку. Для начала удалим существующий индекс:

|  |
| --- |
| curl -X DELETE 'http://localhost:9200/shop?pretty' |

И явно создадим новый **mapping** с поддержкой русского языка:

|  |
| --- |
| curl  -H 'Content-Type: application/json' -X PUT 'http://localhost:9200/shop?pretty' -d' {   "settings": {  "analysis": {   "filter": {     "ru\_stop": {       "type": "stop",       "stopwords": "\_russian\_"     },     "ru\_stemmer": {       "type": "stemmer",       "language": "russian"     }   },   "analyzer": {     "default": {       "char\_filter": [         "html\_strip"       ],       "tokenizer": "standard",       "filter": [         "lowercase",         "ru\_stop",         "ru\_stemmer"       ]     }   }  }   },   "mappings": {  "products": {   "properties": {     "description": {       "type": "text"     },     "name": {       "type": "text"     },     "price": {       "type": "double"     },     "tags": {       "type": "text"     },     "created\_at": {       "type": "date"     }   }  }   } }' |

Давайте воспользуемся полнотекстовым поиском по всему документу:

|  |
| --- |
| curl -H 'Content-Type: application/json' 'http://localhost:9200/shop/products/\_search?pretty' -d' {   "query": {  "query\_string": {   "query": "процессор"  }   } }' |

Итак, у нас были найдены процессоры Intel и AMD.

Возможности ElasticSearch велики и допускают настройку всех аспектов полнотекстового поиска. Они заведомо превосходят аналогичные решения в реляционных СУБД. Поэтому ElasticSearch будет довольно часто встречаться вам в современных проектах.

# Колоночная СУБД ClickHouse

С момента появления базы данных традиционно используются для обработки коммерческих операций или коммерческих транзакций. Например, продажи товара или услуги, размещения заказа у поставщика, выплаты зарплаты и т. п. Хотя базы данных теперь используются в том числе и областях, в которых не происходит перехода денег из рук в руки, термин «транзакция» до сих пор используется для обозначения единой операции чтения и записи.

Сейчас БД применяются для множества самых разных областей: комментариев в сообщениях блогов, действий в играх, контактов в адресной книге и т. д. Порядок работы с ними во многом остался сходным с обработкой коммерческих операций, коммерческих транзакций.

Приложение обычно ищет небольшое количество записей по какому-либо ключу. На основе вводимых пользователем данных вставляются или обновляются записи. Такие операции принято называть «обработка транзакций в реальном времени» (OLTP).

Однако БД все шире используются для аналитической обработки данных. Поведение таких запросов совершенно другое. Обычно аналитический запрос должен просматривать огромное количество записей, в каждой из которой читается лишь несколько столбцов. Результатом обычно являются какие-то агрегированные показатели (например, количество, сумма или среднее значение):

* Какова общая выручка за октябрь?
* Каков средний чек покупателя за прошлый год?
* Какую марку товара чаще покупают с другой маркой товара?

Такие операции принято называть «аналитической обработкой данных в реальном времени» OLAP

Сначала для обработки обоих типов запросов использовались одни и те же базы данных, тем более SQL предоставляет гибкие возможности для составления агрегационных запросов. Тем не менее, в конце 1980-х — начале 1990-х годов возникла такая тенденция: компании стали разделять базы данных для транзакций и для анализа. Последние стали называть складами данных.

Обычно от баз данных, обрабатывающих транзакции, ожидают высокой доступности и скорости обработки транзакций, поэтому их стали оберегать от тяжелых аналитических запросов, которые зачастую могут выполняться часами, блокируя широкий диапазон данных.

Склад данных представляет собой отдельную БД, которую аналитики могут опрашивать так, как им заблагорассудится, не влияя при этом на транзакционную базу данных. Как правило, он содержит копию основной базы данных, зачастую в режиме read only. Чем больше компания и чем больше у нее данных, тем более вероятно, что у нее появится склад данных, в который может стекаться информация из одной, а зачастую и нескольких транзакционных баз данных.

Изначально для транзакционной базы данных и склада данных использовались одни и те же реляционные решения. Однако постепенно внутренее устройство баз данных стало различаться, так как запросы, на которые они ориентированы, совершенно разные.

Поэтому в настоящий момент эти две роли выделятся в специализированные базы данных. Как правило, в качестве складов данных выступают колоночные базы данных.

Коммерческие решения:

* Vertica.
* ParAccel.
* Teradata.
* SAP HANA.

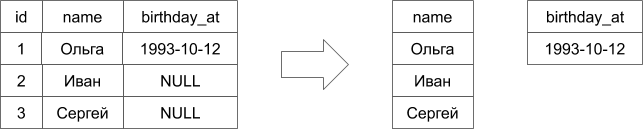
Свободные склады данных:

* HBase.
* Cassandra.
* Hypertable.
* Amazon SimpleDB.
* ClickHouse.

В реляционных таблицах можно хранить большие объемы данных, они прекрасно обрабатывают миллионы строк и гигабайтные таблицы. Однако для гигантских таблиц с триллионами строк и петабайт данных эффективное хранение и выполнение запросов становится непростой задачей.

В большинстве реляционных баз данные располагаются построчно. Все значения из одной строки таблицы хранятся рядом друг с другом.

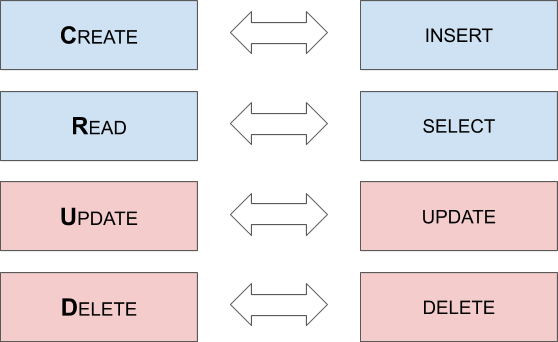
Документоориентированные БД устроены аналогично: весь документ обычно хранится в виде непрерывной последовательности байт. Чтобы выполнить аналитический запрос, часто нужны только несколько столбцов таблицы. Однако построчно-организованная СУБД будет извлекать все столбцы с диска в оперативную память, те столбцы которые не нужны, будут отброшены, усилия и память, потраченные на их извлечение, пропадут. Тут только два столбца, но когда их в таблице сотни, это существенно отражается на производительности. Идея столбцовых хранилищ проста: нужно хранить рядом значения не из одной строки, а из одного столбца.



Если каждый столбец хранится в отдельном файле, то запросу требуется только прочитать и выполнить синтаксический разбор необходимых столбцов, что может сэкономить массу усилий.

За счет раздельного хранения столбцов, данные прекрасно подвергаются сжатию. Зачастую количество различных значений в столбце невелико по сравнению с числом строк. У интернет-магазина могут быть миллионы торговых операций, но только сотни различных товаров.

Большая часть нагрузки в больших аналитических запросах относится к операциям чтения. Хранение сжатых данных в виде отдельных столбцов ускоряет выполнения этих запросов. Однако операции записи усложняются.



Поэтому в столбцовых базах данных операции обновления и удаления данных зачастую не предусмотрены. Мы можем выполнять только вставку и чтение.

В качестве колоночной СУБД мы рассмотрим Clickhouse, который был разработан компанией Яндекс для использования в проекте «Яндекс.Метрика».

Clickhouse — классическая колоночная СУБД, поддерживающая сжатие данных. Многие колоночные СУБД полностью располагаются в оперативной памяти. Clickhouse позволяет держать данные на жестком диске, что более разумно для склада данных.

Запросы выполняются в параллельном режиме на многоядерных процессорах. Пользователям предоставляется язык запросов, следующий традициям SQL. Конечно, речи о поддержке стандарта SQL не идет, все-таки мы имеем дело не с реляционной базой данных. Однако поддерживаются соединения и подзапросы. Допускается индексирование данных и приближенные вычисления. Clickhouse подавляет репликацию и шардирование.

База данных разработана российской компанией Яндекс, и на официальном сайте [clickhouse.yandex](https://clickhouse.yandex/docs/ru/) есть детальная документация на русском языке.

Для соединения с сервером ClickHouse используется клиент **clickhouse-client**:

|  |
| --- |
| SELECT 1 |

ClickHouse поддерживает разделение данных по отдельным базам данных. Получить текущий список баз данных можно при помощи команды **SHOW DATABASES**.

|  |
| --- |
| SHOW DATABASES; |

Создать базу данных можно при помощи команды:

|  |
| --- |
| CREATE DATABASE shop; |

Давайте убедимся, что база данных успешно создана:

|  |
| --- |
| SHOW DATABASES; |

Мы видим новую базу данных **shop**, выбрать ее в качестве текущей можно так же, как в случае с MySQL — командой **USE**:

|  |
| --- |
| USE shop |

Давайте создадим таблицу с посещениями **visits** с посещениями по дням:

|  |
| --- |
| CREATE TABLE `visits` (   `VisitDate` Date,   `Hits` UInt32 ) ENGINE = MergeTree(VisitDate, (VisitDate), 8192); |

Синтаксис ClickHouse очень похож на SQL, для создания таблицы мы используем команду **CREATE TABLE**. Название таблицы и столбцов заключаем в обратные кавычки. Для задания разных типов целочисленного столбца мы используем тип **Int**, суффиксы 8, 16, 32 и 64. по количеству бит в числе, префикс **U** сообщает, что число должно быть беззнаковым. Мы можем использовать и календарные типы, например **Date** для столбца **VisitDate**. Точку с запятой в конце запроса можно не указывать.

 Давайте вставим данные за октябрь 2018 года:

|  |
| --- |
| INSERT INTO visits VALUES ('2018-10-01', 45324), ('2018-10-02', 72241), ('2018-10-03', 69572), ('2018-10-04', 93242), ('2018-10-05', 92201), ('2018-10-06', 12579), ('2018-10-07', 17242), ('2018-10-08', 45298), ('2018-10-09', 67932), ('2018-10-10', 74590), ('2018-10-11', 67275), ('2018-10-12', 45475), ('2018-10-13', 10792), ('2018-10-14', 11243), ('2018-10-15', 78056), ('2018-10-16', 65345), ('2018-10-17', 94335), ('2018-10-18', 89345), ('2018-10-19', 52532), ('2018-10-20', 11437), ('2018-10-21', 13673), ('2018-10-22', 78234), ('2018-10-23', 67233), ('2018-10-24', 92345), ('2018-10-25', 82564), ('2018-10-26', 60234), ('2018-10-27', 14234), ('2018-10-28', 15865), ('2018-10-29', 72784), ('2018-10-30', 76335), ('2018-10-31', 89483); |

Для подсчета количества записей в таблице можно воспользоваться функцией **COUNT()**:

|  |
| --- |
| SELECT COUNT() FROM visits; |

Для извлечения первых 5 строк таблицы можно использовать следующий запрос:

|  |
| --- |
| SELECT \* FROM visits LIMIT 5; |

Как видим, синтаксис SELECT-запроса полностью эквивалентен SQL

|  |
| --- |
| SELECT \* FROM visits LIMIT 2\G |

Поддерживается даже вертикальный режим вывода при помощи модификатора **\G**, который был заимствован из MySQL.

Можно использовать и более сложные запросы. Например, давайте подсчитаем количество посещений по неделям, для этого воспользуемся функцией **toRelativeWeekNum** для преобразования даты **VisitDate** в номер недели:

|  |
| --- |
| SELECT toRelativeWeekNum(VisitDate) AS week, Hits FROM visits LIMIT 2; |

Обратите внимание, так же как и в SQL мы можем использовать ключевое слово **AS** для назначения псевдонима столбца. Теперь мы можем сгруппировать данные по полю **week** и воспользоваться функцией **SUM** для подсчета количества посещений в неделю.

|  |
| --- |
| SELECT toRelativeWeekNum(VisitDate) AS week, sum(Hits) AS hits  FROM visits  GROUP BY week; |

Давайте преобразуем полученные результаты в диаграмму при помощи функции **bar**:

|  |
| --- |
| SELECT   toRelativeWeekNum(VisitDate) AS week,   sum(Hits) AS hits,   bar(hits, 0, 500000, 20) AS bar FROM   visits GROUP BY   week; |

Функция получает в качестве первого аргумента текущее числовое значение. В данном случае это поле **hits**, полученное при помощи функции **sum** как сумма посещений за неделю. Два следующих аргумента задают минимальное и максимальное значения диаграммы, а последнее — ширину столбца.

Посмотреть список текущих таблиц можно при помощи запроса **SHOW TABLES**:

|  |
| --- |
| SHOW TABLES; |

Как видим, у нас пока только одна единственная таблица **visits**. Удалить таблицу можно при помощи команды **DROP TABLE**.

|  |
| --- |
| DROP TABLE visits; |

# Используемые источники

1. Клеппман М. Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб.: Питер, 2018. — 640 с.: ил.
2. Редмонд Эрик , Уилсон Джим Р. Семь баз данных за семь недель. Введение в современные базы данных и идеологию NoSQL. — М: ДМК Пресс — 384с.
3. Фаулер, Мартин, Садаладж, Прамодкумар Дж. NoSQL: новая методология разработки нереляционных баз данных. — Пер. с англ. — М.: ООО "И.Д. Вильямс", 2013. — 192 с.
4. 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.
5. Карпентер Д., Хьюитт Э. Cassandra. Полное руководство. 2-е изд. — М.: ДМК Пресс, 2017. — 400 с.
6. Бэнкер Кайл. MongoDB в действии. — М.: ДМК Пресс, 2017. — 394с.
7. <https://redis.io/documentation>
8. <https://clickhouse.yandex/docs/ru/>