**БАЗЫ ДАННЫХ ОТ MAIL.RU**

Крайне рекомендуемые книги:

* Заводны Дж, Шварц Б, Зайцев П, Ткаченко В, Ленц А, MySQL. Оптимизация производительности
* Стив Макконел. Совершенный код

База данных - структурированные данные, организованные специальным образом



Тип данных - число, строка, время

Домен - тип данных с дополнительными ограничениями (например номер - число от 1 до 4-х значного числа)

Атрибут - часть домена, который обладает ограничениями домена

Кортеж - конкретная строка, которая представлена атрибутом

Ключ - ID, отличие одного обьекта от другого

# РЕЛЯЦИОННАЯ АЛГЕБРА

Один к одному

Один к множеству

Множество к одному

Множество ко множеству

**Выборка -** выбираем атрибут, накладываем ограничение, смотрим результат

(if(x > 10 && x <20)

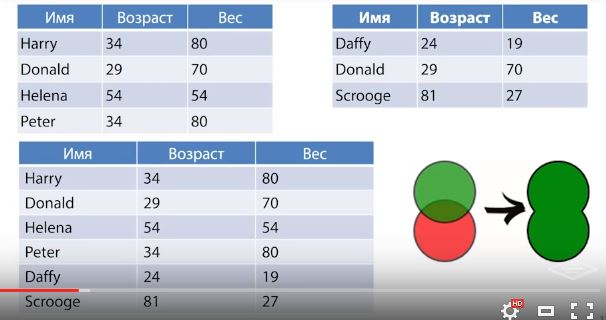
**Проекция -** выбирает указанный атрибут с уникальными кортежами. Если кортежи совпадают, выводит только последнию строку из всех повторяющихся

Полностью убираются из результирующего списка только те, которые полностью совпадают

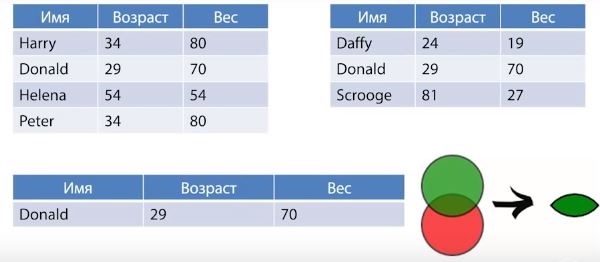
В данном случае, задано два условия - Возраст и Вес.

Первая строка совпадает с последней, потому первая не будет отображена

**Обьеденение**

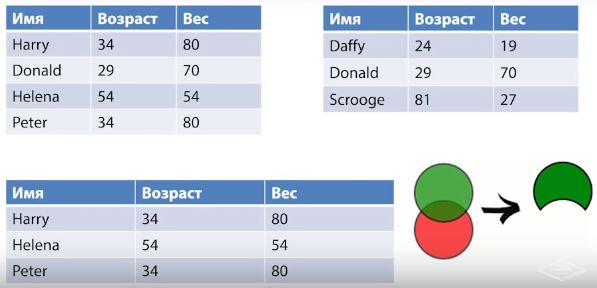
Обьеденяет два или несколько атрибутов таблиц

Принцип обьедениня такой же, как в проекции - повторяющиеся кортежи удаляются, остальные соединяются

**Пересечение**

В результирующей таблице появляются атрибуты, которые полностью идентичны между собой

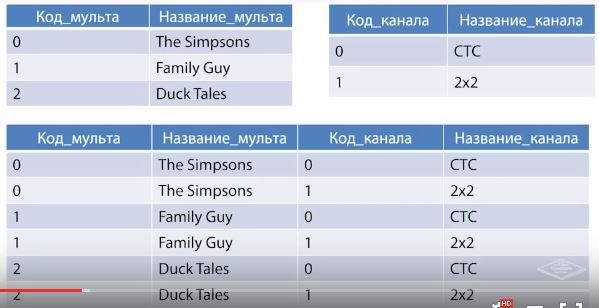
**Разность**

****Вычитание из одной таблицы другую.

Из первой удаляются удаляются кортежи, где информация идентична второй

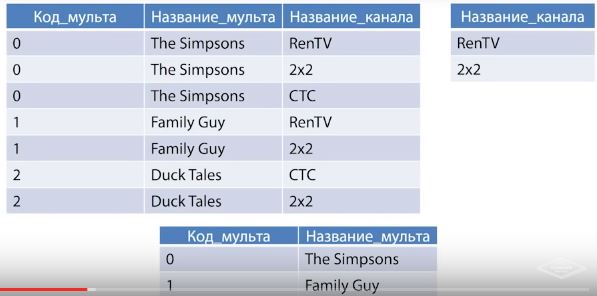
В результирующей таблице - первая таблица с удаленными котрежами

**Произведение**

****

Декартово обьеденение:

Каждому кортежу из одного обьединения сопоставляется каждый кортеж из другого

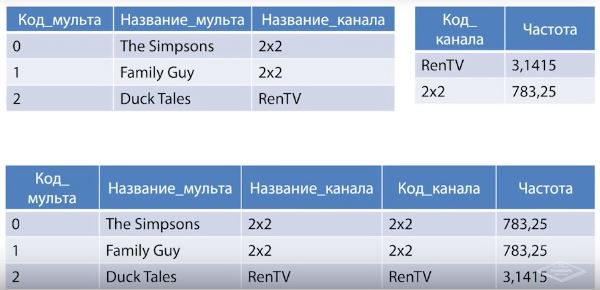
**Деление**

Получить таблицу, которая удовлетворяет запросам

В данном примере мульты (7 штук) поделили на выборку каналов (2 штуки) и получили в выборке те мульты, которые **одновременно есть** и на RenTV и на 2+2

Если бы в выборку с каналами вошел третий канал, CTC, то в результирующей таблице была бы только одна запись: The Simpsons, которые показываются по всем трем каналам

**Соединение**

Соединение информации из одного отношения и из другого

Если бы какой-то информации не хватало, то остались бы пустые столбцы

# ПРОЕКТИРОВАНИЕ БАЗ ДАННЫХ

Основные задачи

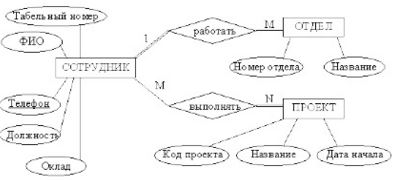
* Обеспечение хранения в БД всей необходимой информации
* Обеспечение возможности получения данных по всем необходимым запросам
* Сокращение избыточности и дублирования данных
* Обеспечение целостности данных (правильности их содержания): исключение противоречий в содержании данных, исключение их потери и т.д

Критерии оценки модели данных

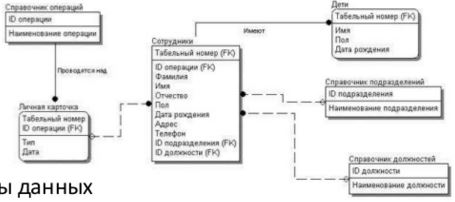
* Структурная достоверность: соответствует способу определения и организации информации
* Простота: удобство изучения модели как профи, так и обычными юзерами
* Выразительность: способность предоставлять различия между данными, связи между даными и ограничения
* Отсутствие избыточности: исключение излишней информации, т.е. любая часть данных должна быть представленна только один раз
* Способность к совместному использованию: отсутствие приналежности к какому-то особому положению или технологии, и следовательно, возможность использования модели во многих приложениях и технологиях
* Расширяемость: способность развиваться и включать новые требования с минимальным воздействием на работу уже существующих приложений
* Целостность: согласованность со способом использования и управления информацией внутри предприятия
* Схематическое представление: возможность представления модели с помощью наглядных схематических обозначений

**Этапы проектирования**

1. Внешнее представление - совокупность требований со стороны конкретной функции, выполняемой пользователем
2. Концептуальная схема - полная совокупность всех требований к данным, полученной из пользовательских представлений о реальном мире
3. Внутренняя схема - схема баз данных

Сначала рисуем **концептуальное проектирование** (инфологическое):

построение модели предметной области, в которой проработаны сущности, атрибуты и связи:

Потом рисуем **логическое проектирование**

Это создание базы данных на основе конкретной модели данных, например реляционной модели данных. На этом этапе прорабатываются записи, элементы данных и связи между записями

**Физическое проектирование** - создание схемы базы данных для конкретной СУБД (MySQL, Oracle, PostgreSQL и т.д.)

На этом этапе разрабатывается группирование данных, индексы и методы доступа

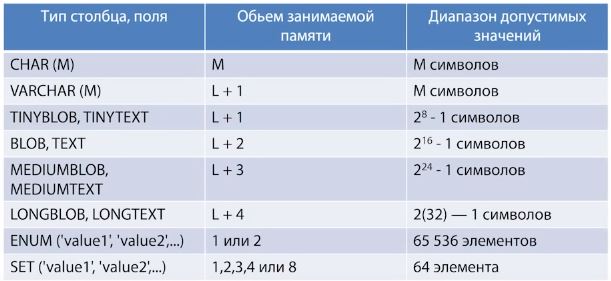
**ТИПЫ ДАННЫХ MYSQL**

**Бинарные числа**

Boolean (M) - алиас для INT, занимает 1 байт, диапазон значение - 0 или 1. Обычно используют TINYINT(1) - хотя можно хранить значение от 0 до 9, но хранят 0 или 1

BIT (M) - (M+7 / 8 байт) - никогда не используется

**Строки**

**CHAR -** сколько символов написали, столько и будем хранить в байтах

Hello - 5 байт

**VARCHAR** - количество символов + 1

Hello = 6 байт

**ENUM** - сохраняет номер и значение. Часто используется для хранения значения в буквах, например справочник полов - F и M

SET - используется для множества количества значений, которые были заранее определены

# СОЗДАНИЕ ТАБЛИЦЫ

* обозначения в скобках **[]** - необязательные аргументы
* круглые скобки **()** - писать нужно, часть синтаксиса
* фигурные скобки **{}** - создают условие **if**
* **|** - или

Есть 3 способа создать таблицу:

1. CREATE [TEMPORAY] TABLE [IF NOT EXISTS] tbl\_name

(create\_definition, ...)

[table\_options]  
[partition\_options]

Создание временной таблицы, что бы не делать выборки из одной большой. IF NOT EXIST - проверяет, существует ли такая таблица

1. CREATE [TEMPORAY] TABLE [IF NOT EXISTS] tbl\_name

[(create\_definition, ...)]

[table\_options]  
[partition\_options]

select\_statement <= отличие от первого варианта

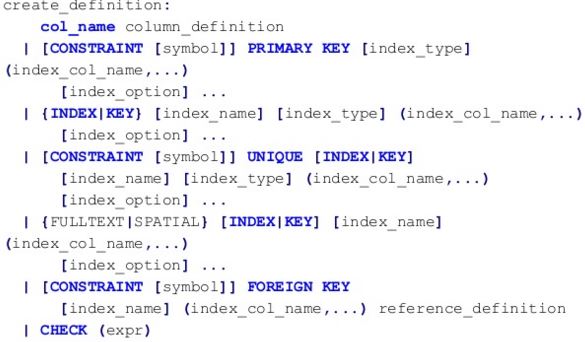
**select\_statement** - запрос на выборку. Создает таблицу на основании результатов выборки и заполняет таблицу выбранными значениями

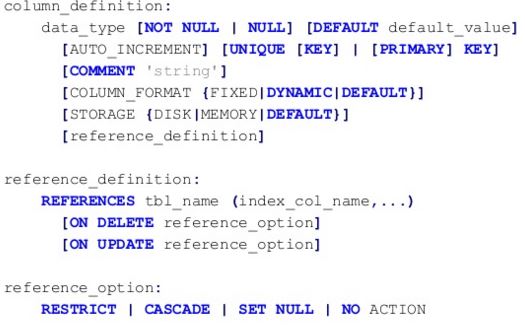
1. CREATE [TEMPORAY] TABLE [IF NOT EXISTS] tbl\_name

{ LIKE old\_tbl\_name | (LIKE old\_tbl\_name) }

Скопировать таблицу с индексами, но без значений

**create\_definition**





# КУРС ПО MYSQL ОТ СПЕЦИАЛИСТА (НИКИТИН)

**УСТАНОВКА MYSQL**

На официальном сайте есть бесплатная версия MySQL - <http://dev.mysql.com/downloads/mysql/>

* Скачиваем MSI installer. Разрядность не имеет значения
* Запускаем MSI file, выбираем Custom install
* Можно выбрать только Server. Включает в себя Shell
* После установки появится мастер настройки. Выбираем Developer Machine
* Окно Type and Networking. Оставляем TCP/IP port как 3306, оставляем галочку Open Firewall port.
* Окно Accounts and Roles - админ уже существует под логином root, потому просто устанавливаем ему пароль - root
* Окно Windows Service - оставляем все галочки по умолчанию. Тут можно указать название службы для быстрого запуска из консоли. По умолчанию стоит MySQL57 (с учетом регистра)
* Окно Apply Server Configuration - окончание установки. Рекомендуется временно отключить антивирус
* Дальше "ок" до конца
* По умолчанию сервер устанавливается по пути C:\Program Files (x86)\MySQL
* Основной файл настроек (**my.ini**) лежит в скрытой папке C:\ProgramData\MySQL\MySQL Server 5.7

# Установка кодировки

Кодировка базы данных не оказывает влияния на внешние кодировки, т.к. это лишь указание, в какой кодировке хранить внутренне таблицы. Отдавать же MySQL может в любой кодировке

После установки, в "Пуск" появляется вызов командной строки с уже переданными параметрами

"--defaults-file=C:\ProgramData\MySQL\MySQL Server 5.7\my.ini"

"-uroot" "-p" "--default-character-set=utf8"

Эти настройки он тянет из my.ini. Для начала, надо указать правильный SET NAMES для установки кодировки.

Таблицы кодировки лежат в папке с сервером по адресу

C:\Program Files\MySQL\MySQL Server 5.7\share\charsets

Папку, где лежат таблицы кодировок MySQL сам по себе не знает, но ее можно указать. В my.ini ищем поле [client] , там указываем кодировку и путь к папке с кодировками

[client]

default-character-set=utf8

character-sets-dir="C:/Program Files/MySQL\MySQL Server 5.7/share/charsets"

Что бы изменения вступили в силу, нужно перезапустить сервер:

net stop MySQL57

net start MySQL57

# ТЕОРИЯ БАЗ ДАННЫХ

**Виды и типы баз даннных**

* картотеки
* сетевые
* иерархические: файловая система компьютера - классический пример. Изначально оптимизированы на чтение. Нахождение конкретного файла происходит мнгновенно, поиск множества файлов (перебор) - медленно

Реестр windows тоже иерархическая база данных

* Реляционные
* Обьектно-ориентированные
* Многомерные

**Особенности** **реляционных баз данных:**

* данные хранятся в таблицах, состоящих из стобцов и строк
* на пересечении каждого столбца и строчки стоит в точности одно значение
* у каждого столбца есть свое имя, которое служит его названием, и все значения в одном столбце имеют один тип
* столбцы располагаются в определенном порядке, который определяется при создании таблицы, в отличии от строк, которые располагаются в произвольном порядке. В табилце может не быть ни одной строчки, но обязательно должен быть хотя бы один столбец
* запросы к базе данных возвращают результат в виде таблиц, которые тоже могут выступать как обьект запросов

Реляционная база данных имеет четко оговоренную структуру, потому Excel не является реляционной базой данных: его можно заполнять как по вертикали, так и по горизонтали

**OLTP или OLAP**

OLTP **-** это выполнение операций без задержки, легкая модификация.

OLAP- анализ. Просмотр всех записей. Реляционные базы данных не расчитаны на OLAP.

Oracle и Microsoft БД делают рассчеты заранее - "кубические расчет", когда сервак ночью считает сумму, среднюю и тд. по каждому пересечение горизонтали вертикали

MySQL считает все в реальном времени

# ТИПЫ ДАННЫХ В БАЗАХ ДАННЫХ

Типов очень много, в основном для обратной совместимости

Основные типы:

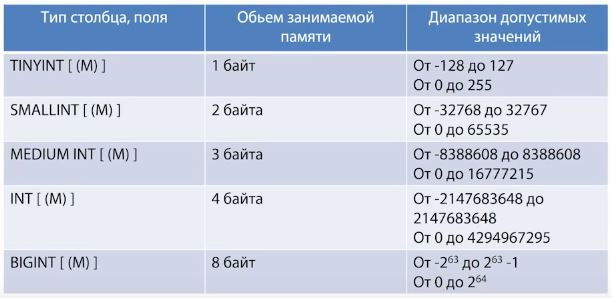
* числовые
* дата и время
* символьные

**NULL**

* NULL не равен NULL
* Конкатенация с участием NULL все приравниевает к NULL.
* Подсчет среднего числа 1, 2 и NULL = 1.5. NULL посчитался как 0 в данном случае
* С самого начала проектирования лучше приравнять пустые поля к 0, что бы потом не путаться - DEFAULT = 0
* При подсчете через команду COUNТ могут быть ошибки с пустыми строками: если указать пересчитать все COUNT(\*), будут подсчитаны строки, учитывая NULL значения

# Целочисленный тип

Операции с целочисленными значениями (типа INT) на порядок быстрее операций с дробными

(FLOAT, REAL).

Числа бывают двух типов:

SIGNED

UNSIGNED

Компьютеры хранят его как старший бит, в 1 бите 2 в 8 степени байт.

2 в 8 степени = 256, -1 бит на указатель SIGNED или UNSIGNED, потому получаем значение в 255

Выигрыш от правильного выбора типа целого числа для сохранения как можно большего кол-ва памяти очень заметен на больших таблицах, когда данные можно хранить в оперативной памяти

Многие системы по умолчанию сразу выставляют значения в BIGINT, потому лучше указывать конкретно

# Тип дробных чисел

В бородатые времена самым распространенным типом хранения данных был Comma Separated Values (CSV). Записи хранились в строках через запятую:

Вася Пупкин, 40 лет, занес 401123.87 рубля

**Плавающая точка** - смысл плавающей точки в том, что число представляется в двух частях:

* мантисса. Число от 0 до 10
* порядок - на сколько нужно умножить число, что бы получить целое число

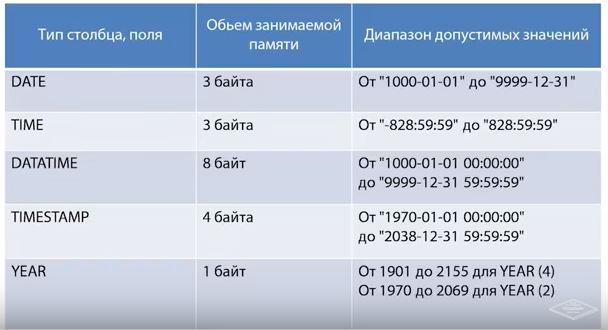
В настоящее время FLOAT = REAL = DECIMAL = FIXED

В современных базах данных число всегда хранится в **FLOAT**

**Дата и время**

Всегда записывается от большего к меньшему

dummy date - 0000-00-00 00:00:00 для DATE

**TIMESTAMP** - самый частоиспользуемый тип

Считает в секундах, потому такой небольшой промежуток времени.

При выборке инфы применяет **часовой пояс**

**Строковый тип данных**

**CHAR()** - тип хранения данных, в котором надо указывать, какая строка будет хранится. Если используется не все указанное пространство, остающееся пространтсво займет пробелами.

Максимальная длина - 255 символов.

Сейчас используется очень редко

CHAR(10) "Hello" //займет символами 7 байт, остальные 3 байта до 10 добьет пробелами

**VARCHAR()** - строка может быть переменной длины, строка занимает длина в байтах +1 байт для указания длины строки. Используется когда строка неизвестной длины.

Максимальная длина 65535 символов

Для хранения большего количества символов есть:

TEXT / BLOB - 2 в 16 (-1 байт) степени = 65535 чисволов

MEDIUMTEXT / MEDIUMBLOB - 2 в 24 степени (-1 байт) = 16777215 символов, 16 Мб

LONGBLOB - 2 в 32 степени (-1 байт) , 4 Gb

**ENUM(перечисление)** - столбец, который может принимать значение из списка допустимых значения, явно перечисленных в спецификации столбца в момент создания таблицы. Например дни недели: указываем пн, вт, ср, чт, пт, сб, вс. И ничего кроме этих указанных значений положить уже будет нельзя

Если делается вставка некорректного значения (т.е. вставка строки, не перечисленной в списке допустимых, то вставляется пустая строка, что является указанием на ошибочное значение.

ENUM очень компактен, следует стараться им пользоваться

Максимальное количество символов - 65535

**SET** - подразновидность ENUM. В отличие от ENUM, в SET может быть множество значение. Например рабочие дни: понедельник среда пятница - и все это в одной ячейке

Максимальное количество элементов, которые можно задать в условие - 64

**ПРОЕКТИРОВАНИЕ БАЗЫ ДАННЫХ**

Сущность - обьект, у которого есть свойства. Например - заказчик. У заказчика есть ФИО, это тоже сущность со своими свойствами, но простая.

Бритва Окамма - не создавай сущности без нужды

**Ключи и ключевые атрибуты**

* **Ключ отношения -** атрибут, или набор атрибутов, однозначно идентифицирующий обьект в БД. Однозначность можно проверить через два свойства: ключ должен быть уникален и ключ должен быть вечен. Например - фамилия Зосимчух. Уникальность ключа зависит от окружения, например в комнате один Зосимчух, а в мире - их много
* **Составной ключ -** ключ, использующий несколько атрибутов. Например Зосимчух Максим Валерьевич. Комбинация неуникальных полей может составлять уникальный составной ключ.
* **Первичный ключ -** ключ используемый системой для идентификации обьектов. Например ИНН, номер паспорта и пенсионный номер - все они являются ключами отношения, поскольку они иникальны для каждого человека и они вечны. Тот ключ из них, который выбирается как главный, и будет называться первичным
* **Ключ-кандидат** - ключ, по каким-либо причинам неиспользуемый, как первичный. Например, номер паспорта, если мы выбрали ключом отношения ИНН
* **Суррогатный ключ** - ключ, значения которого генерируются самой системой (лень искать\выбирать первичный ключ, присваиваем сгенерированный ID). Этот ключ не несет никакой информационной нагрузки, кроме идентификации

**НОРМАЛИЗАЦИЯ БД**

Это приведение структуры базы данных к виду, обеспечивающему минимальную избыточность

* Первая нормальная форма (1NF)
* Вторая нормальная форма (2NF)
* Третья нормальная форма (3NF)
* Нормальная форма Бойса-Кодда (BCNF)
* Четвертая нормальная форма (4NF)
* Пятая нормальная форма (5NF)

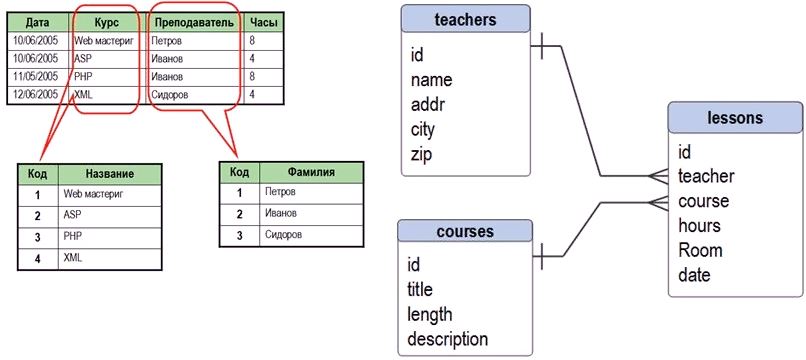
**Первая нормальная форма (1NF)**

Таблица находится в первой нормальной форме (1NF), если каждый ее атрибут атомарен, т.е. может содержать только одно значение. Это значит не писать несколько значений в одной ячейке. Например: преподаватель Иванов читал APS и PHP, но мы значения разделяем

**Вторая нормальная форма (2NF)**

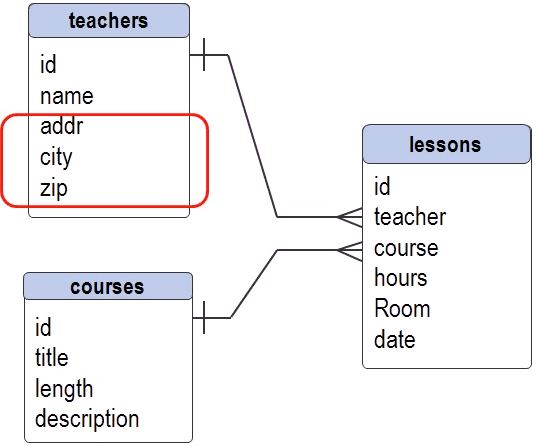
Таблица находится в 2NF, если она находится в 1NF и при этом ее атрибут, не входящий в состав первичного ключа, функционально полно зависит от первичного ключа

У любой таблицы есть ключи. Когда данные берутся из заранее определенного массива, т.е. по ссылкам, это и есть вторая нормальная форма. Это значит, что под каждую колонку у нас есть свой справочник, например справочник преподавателей, справочник курсов и т.д. И в результирующей таблице, например расписание курсов, появляется обьединение этих справочников. Преподаватели и курсы не записываются в оригинальном в виде, а передаются в виде из ID по ссылке из справочника



На картинке справа нарисована связь один ко многим

**Третья нормальная форма (3nf)**

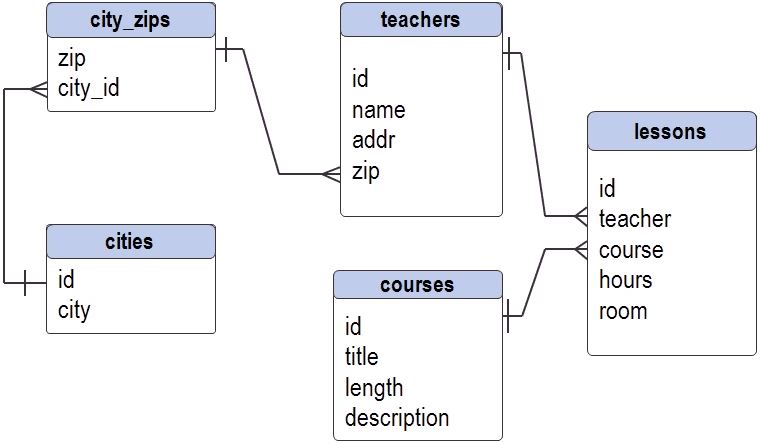
Таблица находится в 3NF, если она находится в 2nf и при этом любой ее неключевой атрибут зависит **только** от первичного ключа

*Транзитивная зависимость*

Не должно быть повторений двух очевидных вещей: в данном случае атрибут city и zip - идентичны. Если у города не киевский индекс, то город по умолчанию не может быть Киевом, и наоборот

Такая ситуация легко может привести к ошибке: если оператор ошибется и вобьет город - Москва, а индекс города - Киев, по какому атрибуту ориентироваться?

Задача 3NF - не допустить такого. Потому лишний атрибут делается ссылкой на справочник на основной ключ - ключ отношения:



**Workbench - средство для проектирования от MySQL**

* Открываем workbench. Создаем новую модель. Сразу сохраняем файл - File - save as
* Нажимаем сверху Add Diagram, открывается лист формата a4
* Выбираем справа корень - название модели
* Помещаем New text object (N) - это будет просто название таблицы. Name - название поля, Text - то, что будет отображаться.
* Помещаем таблицу Place a new table (T), два раза кликаем. Имя - выбираем логически правильное имя, например справочник teachers. Collation - будет учитываться регистр или нет, пока ставим Defalut. Engine - InnoDB. Comments - описываем, что это за таблица, для себя
* Берем за верхний край окна редактирования таблицы и тянем вверх. Становятся видны ключи
* Datatype - тип колонки
* PK - primary key - первичный ключ
* NN - not NULL - отсутствие значения
* UQ - ключ-кандидат
* B
* UN - USIGNED / SIGNED, используется для чисел
* ZF - zero field - добивай нулями числа. Например 0001, 0002
* AI - автоинкремент. Суррогатный ключ
* G
* первый атрибут - idteachers, workbench ставит сам. Тип - INT, оставляем. Поскольку ID у нас Primary Key, PK галочка ставится автоматически, как и NN. В конце отмечаем AI для суррогатного ключа
* Клацам внизу и создаем поле name. Даем ему тип - VARCHAR(45). Ставим параметр NN
* Клацам снизу и создаем поле adress - тоже самое
* Создаем новую таблицу, называем lessons
* В ней создаем:
* idlessons проставится сам, даем ему AI;
* room - номер аудитории, есди в номере есть буквы (345А), имеет смысл дать тип - CHAR(5) и NN;
* date - когда читаются курсы, даем ему тип DATETIME и NN
* length - продолжительность урока, ставим INT (в целочисленных часах или минутах)
* Теперь нужно создать связь между курсами и преподами. Задаем вопрос: много курсов может читать один препод? Если да, значит это связь один ко многим.
* Берем пунктирную линию, клацаем по дочернему элементу (lessons), потом по родительскому (teachers). Почему пунктирную: это линия отношения. Ставится в 99% случаев. Сплошная линия (идентифицирующий ключ) используется, если ключ составной - и вместо ID используется комбиниция значений разных таблиц
* На дочернем элементе должен появится значек растроенной вилки, что означает связь один ко многим
* внизу lessons должна появится новая строчка: teachers\_idteachers INT с красным ромбиком, что означает внешний ключ

**ЯЗЫК SQL**

SQL (Structured Query Language) - универсальный язык, применяемый для создания, модификации и управления данными в реляционных базах данных

SQL делится на три части:

* Операторы определения данных (Data Definition Language, DDL)
* Операторы манипуляции данными (Data Manipulation Language, DML)
* Операторы определения доступа к данным (Data Control Language, DCL)

Язык регистронечувствителен. Традиционно все стандартные ключевые слова пишутся заглавными буквами

**CИНТАКСИС SQL**

SELECT - выбрать

FROM - из

; - окончание строки, перевод строки роли не играет

\G вывести все в таблицу поле - значение

\с - отменить введенный текст команды

*Все встроенные команды MYSQL можно посмотреть, набрав help или \?*

**Создание и удаление базы данных**

CREATE DATABASE [IF NOT EXISTS] db\_name - создать базу данных

CREATE DATABASE module3; /\* создать БД с пробелами в названии \*/

CREATE DATABASE `module 3`; /\* создание БД с именем с пробелами через обратный апостроф \*/

CREATE DATABASE IF NOT EXISTS `module 3` /\*ошибки не будет, поскольку стоит if not exist\*/

DROP DATABASE `module 3` /\*удалить базу данных\*/

SELECT \* FROM module3.test /\*показать все записи из таблицы test в ДБ module3\*/

USE module3 /\* задать базу данных по умолчанию. Это не SQL-оператор, это только команда на клиенте \*/

SELECT \* FROM test /\* показать все записи из таблицы test в ДБ, заданной по умолчанию \*/

**Создание таблиц**

CREATE [TEMPRORARY] TABLE [IF NOT EXIST] tbl\_name

CREATE TABLE t (c CHAR(20) CHARACTER SET utf8 COLLATE utf8\_bin);

/\*

c - название колонки,

CHAR(20) - тип

CHARACTER SET utf8 - указание кодировки для колонки. У каждой колонки может быть разная

COLLATE utf8\_bin - сравнение с учетом регистра. bin - бинарно, побайтово Арарат !== АРАРАТ

COLLATE general\_ci - без учета регистра Арарат == АРАРАТ

\*/

Кодировка крайне редко задается для отдельных таблиц. Исключение - таблица паролей, там разность регистра нужна. Обычно кодировка определяется сразу для базы данных:  
CREATE DATABASE module3 COLLATE utf8\_general\_ci;

Создание таблицы с несколькими колонками

CREATE TABLE table1

(

id INT UNSIGNED NOT NULL DEFAULT 0 COMMENT 'Код товара',

name VARCHAR(50) NOT NULL DEFAULT '' COMMENT 'Название товара',

price FLOAT NOT NULL DEFAULT 0 COMMENT 'Цена товара'

) COMMENT 'Таблица товаров';

NOT NULL - не может быть пустой

DEFAULT 0 - если указан NOT NULL, то если нет данных, ставит 0

COMMENT - комментарий таблицы. Ставить их считается хорошим тоном

DESC table1; /\* DESCRIBE table1 - описание таблицы \*/

DROP TABLE table1 /\*удаление таблицы\*/

--первый способ создания таблицы

CREATE TABLE table2

(

id INT UNSIGNED NOT NULL AUTO\_INCREMENT COMMENT 'Код товара',

name VARCHAR(50) NOT NULL DEFAULT '' COMMENT 'Название товара',

price FLOAT NOT NULL DEFAULT 0 COMMENT 'Цена товара',

PRIMARY KEY (id, name, price)

) COMMENT 'Таблица товаров с первичным ключом';

AUTO\_INCREMENT - если колонка INT и она является первичным ключом, она может иметь свойство auto increment

PRIMARY KEY - составной ключ был составлен из трех. PRIMARY KEY - такое же свойство таблицы, как и колонка, потому принято указывать его в конце перечислений колонок

-- второй способ создания таблицы - по образцу

CREATE TABLE table3 LIKE table2; /\* создать таблицу по образцу table2. Копируется полностью структура, но не копируются данные \*/

--третий способ создания таблицы - на основе запроса

CREATE TABLE table4 COMMENT 'Таблица по запросу'

SELECT user, host, password

FROM mysql.user;

--создание ВРЕМЕННОЙ таблицы на основе запроса. Существует до конца этого соединения,

--другим пользователям не видна

CREATE TEMPORARY TABLE table5 COMMENT 'Результаты запроса'

SELECT user, host, password

FROM mysql.user;

**Удаление таблиц**

DROP [TEMPORARY] TABLE

[IF EXISTS] tbl\_name [, tbl\_name]

...

[RESTRICT | CASCADE]

Если таблица на удаление имеет связи с другими таблицами. Напр. две таблицы: преподаватели и уроки. В уроках используется ссылка на преподавателя - код из справочника преподавателей. При удалении таблицы преподавателей, как будет реагировать сервер?

RESTRICT - запретить удаление, пока не будут удалены связанные таблицы

CASCADE - сервер самостоятельно удаляет все связанные таблицы

**Модификация таблиц - используется редко**

ALTER TABLE table2 DROP COLUMN price, DROP COLUMN amount; /\* модифицировать таблицу table2 путем удаления в ней колонок price и amount \*/

ALTER TABLE table1 CHANGE column1 cloumn2 INTEGER; /\* переименуй колонку column1 в column2 и присвой целочисленный тип \*/

ALTER TABLE table1 MODIFY table2 BIGINT NOT NULL; /\* тоже самое, что сверху. CHANGE меняет имя и тип, а MODIFY - только тип\*/

ALTER TABLE tbl\_name CONVERT TO CHARACTER SET charset\_name; /\*изменить кодировку\*/

ALTER TABLE table2 ADD INDEX (d), ADD INDEX (a) /\*добавить индексы\*/

**INFORMATION\_SCHEMA - база метаданных сервера.** Только на чтение. Современный и продвинутый вариант SHOW, так же показывает комментарии

use INFORMATION\_SCHEMA; /\*переключаемся на использование информационной схемы\*/

-- список таблиц указанной БД

SELECT table\_name, table\_comment

FROM information\_schema.tables

WHERE table\_schema = 'module3';

-- спецификация колонок для таблицы table2 БД module3

SELECT column\_name, data\_type, column\_comment

FROM information\_schema.columns

WHERE table\_schema = 'module3'

AND table\_name = 'table2';

-- Список БД

SELECT \*

FROM information\_schema.schemata \G

**Индексы**

Индексы - попытка уйти от перебора, что бы ускорить поиск.

Поиск информации - "найти льва в пустыне". Массив должен быть отсортирован. Длинна массива например 100 символов. Делим его пополам, смотрим длину массива - если искомое число в текущей половине отсутствует, отбрасываем эту половину и делим пополам следующую, так же проверяем. До профита.

Строится массив (временная таблица) - где значение поля - ссылка на реальную запись. Поиск происходит по временной таблице, которая ведет на реальную запись.

"Сбалансированное дерево"

Есть проблема с нехваткой оперативной памяти. Вся информация делится на много мелких кусочков. Каждый кусочек индексируется и помещается в главный справочник. При обращении к записи, сервер обращается к индексу кусочков (индекс индексов), потом грузит соответствующий кусок, в нем ищет запись

Индексы ускоряют выборку в разы, но замедляют модификацию - при модификации серверу предстоит проверить все дерево, и внести изменения в индексные файлы при необходимости

CREATE [UNIQUE|FULLTEXT|SPATIAL]  
 INDEX index\_name

[index\_type]

ON tbl\_name (index\_col\_name, ...)

имя индекса (index\_name) может быть любым, но традицонно его начинют с "ix": ixName, ixPrice

CREATE INDEX ixName ON table2 (name) /\*создать индекс для колонки name в таблице table2\*/ CREATE INDEX ixName ON table2 (login, password) /\*создать быстрый поиск по комбинации имя + пароль\*/

После создания индекса сервер уже самостоятельно включает быстрый поиск по индексу.

Индексировать нужно не все подряд: размер индексных таблиц может быть аналогичным самой таблице + сильно замедляется модификация, особенно на массовых запросах

Индексируются только простые типы данных и максимум блоки - CHAR VARCHAR, никаких TEXT и там, где большой обьем инфы!

DROP INDEX ixPrice ON table2 /\*удалить индекс\*/

--создать индекс при создании таблицы

DROP TABLE IF EXISTS table5;

CREATE TABLE table5

(

id INT UNSIGNED NOT NULL AUTO\_INCREMENT COMMENT 'Код товара',

code CHAR(4) NOT NULL DEFAULT 'AAAA',

name VARCHAR(50) NOT NULL DEFAULT '' COMMENT 'Название товара',

price FLOAT NOT NULL DEFAULT 0 COMMENT 'Цена товара',

CONSTRAINT pkId PRIMARY KEY (id),

CONSTRAINT ixCode UNIQUE KEY (code),

INDEX ixName (name),

INDEX ixPrice (price)

) COMMENT 'Таблица товаров с ключом и индексами';

-- добавим товар

INSERT INTO table5 (code, name, price)

VALUES ('ZZZZ', 'Проба', 16.8);

--например ошиблись - добавляем товар с тем же артикулом

INSERT INTO table5 (code, name, price)

VALUES ('ZZZZ', 'Траляля', 20);

-- будет ошибка - повтор значения ключа кандидата (UNIQUE)

Для TEXT \ BLOB есть специальная версия индекса - **полнотекстовый индекс (FULLTEXT)**

Сервер проходит по записям, берет каждую запись, делит ее на слова, делает суммарную таблицу всех слов и ее индексирует. Поиск по отдельным словам, входящим в состав колонки.

CREATE TABLE ftTest (

id INT UNSIGNED NOT NULL AUTO\_INCREMENT,

title VARCHAR(5000) COMMENT 'заголовок новости',

description VARCHAR (5000) COMMENT 'Описание новости',

CONSTRAINT pkId PRIMARY KEY (id),

FULLTEXT INDEX ixFullText (title, description)

) ENGINE=INNODB;

INSERT INTO ftTest (title, description)

VALUES

('текст1 текст1 текст1'),

('текст2 текст2 текст2');

--по умолчанию ищет 4 символа и больше

SELECT \*

FROM ftTest

WHERE MATCH (title, description)

AGAINST ('текст2' IN NATURAL LANGUAGE MODE);

**Ограничения**

PRIMARY KEY - сервер делает по нему индекс автоматически. Часто сокращают как pk. Первичный ключ служит *ограничением* на входные данные, т.е. каждый раз при добавлении или модификации записи в таблице, сервер проверяет, что бы не было дублей первичного ключа.

CONSTRAINT - означает ограничение для сервера

FOREIGN KEY - проверка существования ключа во внешнем справочнике

CONSTRAINT pkId PRIMARY KEY (id) /\*пример выше. Ограничение и определение первичного ключа для значения id. Первичному ключу дали имя pkId\*/

CONSTRAINT ixCode UNIQUE KEY (code) /\*ключ кандидат. это задается словом UNIQUE. При добавлении записи сервер будет так же проверять на уникальность ключ-кандидат\*/

Для создания связи таблиц с ограничением есть несколько правил:

* Связь должна быть только по первичному ключу или ключу-кандидату, нельзя делать свзять по любой колонке (таблица уроки вытаскивает преподавателей из справочника по id преподавателя, который является PRIMARY KEY)
* Колонка в дочерней таблице и справочнике должны быть одного типа (напр. INT UNSIGNED)
* Когда мы ставим PRIMARY KEY ИЛИ UNIQUE KEY индексы ставятся автоматически. Когда присваиваем FOREIGN KEY индексы нужно указывать специально

CREATE TABLE parent (

id INT NOT NULL,

PRIMARY KEY (id)

) ENGINE=INNODB;

CREATE TABLE child (

zip INT, /\* любые данные в таблице \*/

parent\_id INT, /\*тут типы должны совпадать с родителем\*/

INDEX par\_ix (parent\_id), /\*ставим отдельно индекс на родителя\*/

FOREIGN KEY (parent\_id) REFERENCES parent(id) /\*делаем связь\*/

ON DELETE CASCADE /\*автоматически удаляй связанные записи\*/

) ENGINE=INNODB

/\* вставка в таблицу с FOREIGN KEY\*/

INSERT INTO child (zip, parent\_id)

VALUES ('1111', (SELECT id from parent WHERE cityName='Москва')),

('2222', (SELECT id from parent WHERE cityName='Пекин'));

**ОПЕРАТОРЫ МАНИПУЛИРОВАНИЯ ДАННЫМИ**

**INSERT -** вставка в таблицу

INSERT INTO table2 (name, price)

VALUES (10, 'Товар', 'Характеристика');

Если первый аргумент VALUES - id, который при создании таблиц был указан примерно так

id INT UNSIGNED NOT NULL AUTO\_INCREMENT COMMENT 'Код товара'

то, если при INSERT не указан первый аргумент, (id), значение соответсвенно будет автоинкрементироваться, либо принимать то значение, которое установлено в DEFAULT

**IGNORE** - Позволяет внести в БД все записи, игнорируя (не внося) ошибочные)

INSERT IGNORE INTO table2 VALUES

(21, 'Товар 21', 123),

(22, 'Товар 21', 123),

(21, 'Товар 21', 123), /\*повтор pkID - ошибка, не будет внесен\*/

Добавить в таблицу данные из запроса. Число колонок из запроса и их тип должны совпадать с указанными в INSERT:

INSERT INTO table2 (name, price) /\*вставить имя и цену в table2\*/

SELECT FROM user, 123 as price /\*выбрать из user, в user нет колонки price - указываем\*/

FROM mysql.user /\*выбрать из mysql.user\*/

**Ускорение INSERT**

**Способ 1**

При массированном INSERT можно предварительно ***отключить INDEX****,* что бы mysql не пересчитывал индексы для каждой записи, а включить уже сразу после INSERT

-- отключение индексов (для движка MyISAM, на INNODB не работает)

ALTER TABLE table DISSABLE KEYS;

-- INSERT 10.000 записей

INSERT IGNORE INTO table2 VALUES

(21, 'Товар 21', 123),

(22, 'Товар 21', 123),

(21, 'Товар 21', 123),

-- включение индексов (для движка MyISAM, на INNODB не работает)

ALTER TABLE table2 ENABLE KEYS;

**Способ 2**

Загрузка из внешнего файла. Предварительно нужно подготовить текстовый файл, разделить записи (по ум. по табуляция), убедиться, что все колонки присутствуют в правильной последовательности, убедиться, что кодировка файла соответствует кодировке в базе данных, убедиться, что перенос строк

CR|LF - винда - передвинь каретку и следующая строка (2 байта) \r \n

LF - UNIX - только новая строка (1 байт)

-- загрузка данных

LOAD DATA

INFILE 'D:/MySQL/Samples/module-3/data.txt'

INTO TABLE table2

FIELDS TERMINATED BY '\t' /\* через что разделяем колонки. Тут - табуляция\*/

LINES TERMINATED BY '\r\n' /\*окончание строки. Виндовая версия\*/

**DELETE**

**--** удаляет все записи из таблицы

DELETE FROM table2

-- удаление последней записи

DELETE FROM table2

ORDER BY id DESC

LIMIT 1;

**UPDATE**

**Модифицирует записи таблицы**

UPDATE table2

SET /\* заменить следующие значения на: \*/

name = CONCAT('name', ' \*'), /\*соединить строку с пробелом и звездочкой\*/

price = price \* 1.5 /\*текущую цену во всех рядаъ умножить на 1.5\*/

WHERE id BETWEEN 10 AND 30; /\*отобрать записи с ID между 10 и 30\*/

**SELECT**

SELECT отбирает все колонки, а WHERE - ряды

SELECT может работать не только с таблицами, но и с примитивными типами данных, а так же с переменными

SELECT 2 + 2 AS result /\* вернет в виде таблице result - 4\*/

Можно так же в создаваемой таблице переименовать выбираемые колонки:

SELECT name AS teacher phone AS tel /\*в результирующей таблице будут teacher и tel\*/

FROM teachers; /\*выбрать из таблицы teachers\*/

Предикат - уточняющий фильтр

SELECT name, phone

FROM teachers

ORDER BY name; /\* предикат - отсортируй по имени \*/

-- ORDER BY 1; /\* сортировка по первой колонке - name в данном случае \*/

**BETWEEN, LIKE, IN, IS NULL**

SELECT \* FROM tbl\_name

WHERE auto\_col IS NULL;

SELECT \* FROM tbl\_name

WHERE col1 BETWEEN 2 AND 5; /\*Поиск значения колонки\*/

SELECT \* FROM t1

WHERE name LIKE '%John%'; /\*поиск вхождения слова John\*/

SELECT \* val1 FROM tbl1

WHERE val1 IN ('1', '2', 'a'); /\*перечисление значений, как в клаузуле OR\*/

**СКАЛЯРНЫЕ ФУНКЦИИ**

Одной записи на входе соответствует одна запись на выходе

SELECT DISTINCT YEAR(lesson\_date), MONTH(lesson\_date) /\*скалярные функции\*/

FROM lessons

WHERE teacher IN (1,2,3)

**Агрегатные функции**

Множеству записей на входе соответствует только одна на выходе

AVG() - среднее значение

COUNT() - количество, число записей, в качестве аргумента - имя столбца.

GROUP\_CONCAT() - склеивание строк

MIN, MAX - мин и макс

SUM() - стандартное отклонение (статистика)

STD - статистическое отклонение

VARIANCE() - отклонение от среднего

SELECT COUNT(\*) FROM student; /\*количество студентов\*/

SELECT COUNT (DISTINCT results) FROM student; /\*количество неповторяющихся значений\*/

**GROUP BY**

Группировка по значению колонки. Часто используется с функциями-агрегатами

--сгруппирует записи по году и подсчитает сумму этой колонки (всех записей)

SELECT year, SUM(profit)

FROM sales

GROUP BY year

То, что написано в GROUP BY **должно быть указано на выходе - в SELECT-e:**

SELECT teacher, course, SUM(length)

FROM lessons

GROUP BY teacher, course; /\*множественная группировка\*/

WITH ROLLUP - группировка с подитогами и суммарным итогом

SELECT teacher, course, SUM(length)

FROM lessons

GROUP BY teacher, course WITH ROLLUP; /\*группировка с подитогами\*/

HAVING - второй отбор по результатам группировки. WHERE до группировки, HAVING - после. HAVING без GROUP BY использовать не рекомендуется

SELECT YEAR(lesson\_date), MONTH(lesson\_date), SUM(length)

FROM lessons

WHERE teacher IN (1,3,5) /\*фильтр до группировки\*/

GROUP BY YEAR(lesson\_date), MONTH(lesson\_date)

HAVING SUM(length) > 20; /\*фильтр после группировки\*/

**Порядок обработки предикатов сервером**

Рекомендуется писать запросы в заданном порядке:

WHERE

GROUP

HAVING

ORDER

LIMIT

**JOIN - объединение таблиц**

Объединения таблиц бывают разных типов. Формат записи - какую таблицу писать раньше или позже значения не играет. Использование JOIN считается хорошим тоном и упрощает чтение

* внутреннее объединение - копируется только то, что имеет соответствия

SELECT country.name, countryLanguage.language

FROM country

JOIN countryLanguage ON country.code = countryLanguage.countryCode

WHERE countryLanguage.language = 'Russian';

-- тот же запрос, только без INNER JOIN - полный аналог

SELECT country.name as country, countryLanguage.language as lang

FROM country, countryLanguage

WHERE countryLanguage.language = 'Ukrainian'

GROUP BY country;

* Левое \ правое внешнее объединение. Таблицы добавляются в порядке записи FROM tbl\_1 LEFT JOIN tbl\_2 - tbl\_1 слева, tbl\_2 справа. В отличии от внутреннего обьединения, копируется все, и по возможности ставится соответствие из левой
* Полное объединение - копируются все записи из всех таблиц, по возможности ставятся соответствия. Если соответствий нет по каким-либо записям, проставляется значение NULL
* Кросс-объединение (CROSS JOIN) каждой строчке левой таблицы просоединяются все записи правой. Произведение между первой и второй таблицей. Сейчас нет практического применения

SELECT albums.name, albums.year, artists.name FROM albums, artists

Когда нужен LEFT JOIN? Когда каждый препод читал курсы в последний раз? Собрать воедино даты для всех преподов и выбрать из них максималью

SELECT teachers.name, MAX(lessons.lesson\_date)

FROM teachers

INNER JOIN lessons ON teachers.id = lessons.teacher

GROUP BY teachers.name;

Но если препод только пришел на работу, для него данные по прочитанным курсам будет NULL (еще не успел начитать). Внутреннее объединение показывает только поля со всеми связями, потому нового препода в списке не будет. Для этого используем внешнее объединение:

SELECT teachers.name, MAX(lessons.lesson\_date)

FROM teachers

LEFT JOIN lessons ON teachers.id = lessons.teacher

GROUP BY teachers.name

ORDER BY 1; /\*отсортировать по первой колонке\*/

Если прямой связи между таблицами нет, используются промежуточные, такая как country между city и countryLanguage

**UNION - объединение запросов в одну таблицу**

(SELECT name, countryCode FROM city WHERE city.countryCode = 'FIN')

UNION

(SELECT name, countryCode FROM city WHERE city.countryCode = 'DNK');

**Операторы IN, ANY, ALL, SOME, EXISTS**

SOME - одному из значений

ALL - каждому из значений

ANY -

-- 2й запрос! Строится после первого

SELECT country.name

FROM country

WHERE country.code <> NOT IN --или (ALL) где код страны не равен любому из значений

(

-- 1й запрос! Список стран-миллионников.

SELECT DISTINCT city.countryCode

FROM city

WHERE city.population > 1000000

);

**ОБЪЕДИНЕНИЕ ЗАПРОСОВ И КОНСОЛИДАЦИЯ ДАННЫХ**

**Временные таблицы**

Не видна другим пользователям и живет только до конца сеанса

**Представление или VIEW**

Частые запросы можно сохранять в БД, что бы не писать их каждый раз. VIEW - это сохраненный в БД запрос, который себя ведет как обычная таблица

CREATE VIEW currCourses AS --создание запроса

SELECT DISTINCT course -- тело запроса

FROM lessons

WHERE MONTH(lesson\_date) = 9

AND YEAR(lesson\_date) = 2006

После этого VIEW сохранится в таблицах (можно увидеть, написав SHOW TABLES). Также для него работают запросы, как для обычной таблице, хотя это не таблица, а запрос. При обращении к ней, сервер делает запрос новый запрос

SELECT \* FROM currCourses -- будет вести себя как таблица

Соответственно таблица будет самообновляема при добавлении новых записей в родительскую таблицу

DELETE FROM currCourses WHERE id=5; --хотя запрос к VIEW, удалит из родительской таблицы

Часто используется для изменения структуры уже давно существующей БД, что бы не ломать то, что уже создано и работает.

Поскольку сервер MySQL кэширует запросы, скорость обращения к небольшой VIEW может быть в разы выше, чем к полной таблице.

Из VIEW так же можно удалять наполнение или модифицировать его, и это повиляет на родительскую таблицу НО ТОЛЬКО если в ней нету выражений:

* Aggregate functions (SUM(), MIN(), MAX(), COUNT(), and so forth)
* DISTINCT
* GROUP BY
* HAVING
* UNION or UNION ALL
* Subquery in the select list (fails for INSERT, okay for UPDATE, DELETE)
* Certain joins (see additional join discussion later in this section)
* Reference to nonupdatable view in the FROM clause
* Subquery in the WHERE clause that refers to a table in the FROM clause
* Refers only to literal values (in this case, there is no underlying table to update)
* ALGORITHM = TEMPTABLE (use of a temporary table always makes a view nonupdatable)
* Multiple references to any column of a base table (fails for INSERT, okay for UPDATE, DELETE)

**ХРАНИМЫЕ ПРОЦЕДУРЫ**

Хранимая процедура - такой же объект, как таблицы и VIEW. В отличии от VIEW, процедура - это действие.

Процедура - набор действий, которые выполняются по требованию программиста.

* В большинстве СУБД при первом запуске хранимой процедуры выполняется ее компиляция (выполняется синтаксический анализ и генерируется план доступа к данным). В дальнейшем ее обработка осуществляется быстрее.

В MySQL впрочем, сервер хранит процедуры как обычный текст, но благодаря кэшированию, ее производительность все равно выше обычного запроса.

* SQL-иньекция:

-- пример 1

SELECT id

FROM user

WHERE login = '' AND pwd = '' ;

admin'; -- то, что может ввести пользователь на сайте

-- в результате вторая часть запрос комментируется

-- пример 2

' OR 1=1 -- в поле ввода пароля или пусто, или 1=1, что сразу является true

Выход - **не менять запросы программно, не конкатенировать запросы**!

Хранимые процедуры вызываются по-другому, потому более безопасны.

* Вместо хранения часто используемого запроса, клиенты могу ссылаться на соответствующую процедуру. При ее вызове содержимое сразу же обрабатывается сервером, и позволяет избежать пересылки через сеть сотен команд

**Создание процедуры**

Традиционно процедурам дают имена, которые начинаются с префикса cp

DELIMITER // -- задаем синтаксис для окончания строки, со стандартного ";" на "//"

CREATE PROCEDURE simpleproc() /\*пишется через скобки, внутри могут быть аргументы\*/

BEGIN

SELECT COUNT(\*) INTO param1 FROM t; -- фигачим запросы, но тут есть ";"

SELECT COUNT(\*) INTO param1 FROM t; -- фигачим запросы, но тут есть ";"

END;

// -- даем серверу понять, что команда окончилась тут

DELIMITER ; -- меняем окончание строки на стандарт, сразу после создания процедуры

При разработке процедур надо стараться, что бы внутри нее не было два отдельных запроса

**Вызов процедуры**

CALL simpleproc(); -- будут выполнены все запросы внутри процедуры

**Аргументы процедуры**

Аргументами в процедуру могут передаваться переменные с ключевыми словами

**IN** - на вход, подставляем переменные

**OUT** - на выход, данные в переменную подставляются после выполнения процедуры

Переменная в MySQL указывается через @: @variable

Переменная в процедуре создается **без собачки**, и это означает локальную видимость переменной

--типичный пример использования процедур:

DELIMITER //

CREATE PROCEDURE sp\_courses\_by\_dates

--перечисление аргументов - два на вход, и последний возвращаемый

(IN date\_start DATE, IN date\_end DATE, OUT record\_count INT)

BEGIN

--создание временной таблицы с результатами запроса

CREATE TEMPORARY TABLE course\_by\_dates\_results

SELECT courses.id, courses.title

FROM courses

INNER JOIN lessons ON courses.id = lessons.course

--использование переданных аргументов - по ним сделали временную таблицу

WHERE lessons.lesson\_date BETWEEN date\_start AND date\_end;

--поместим результат подсчета в переменную record count

SELECT COUNT(\*) INTO record\_count

FROM course\_by\_dates\_results;

-- вернем результат

SELECT \* FROM course\_by\_dates\_results;

-- удалим временную таблицу

DROP TEMPORARY TABLE course\_by\_dates\_results;

END;

//

DELIMITER ;

-- использование процедуры - вводим параметры и указываем переменную для результата

CALL sp\_courses\_by\_dates ('2006-09-16', '2006-10-25', @count);

SELECT @count -- выведет подсчитанное количество

Внутри процедуры можно обьявить локальные переменные через DECLARE

CREATE PROCEDURE sp1 (x VARCHAR(5))

BEGIN

DECLARE xname VARCHAR(5) DEFAULT 'bob'; -- объявили переменную xname = 'bob'

SET xname = 'john'; -- дали переменной другое значение

END;

**Обработчики ошибок в процедурах SQL**SQL можно сказать является языком программирования, потому в нем так же есть обработчики ошибок:  
CONTINUE - несмотря на нестрандартную ситуацию, продолжай выполнение

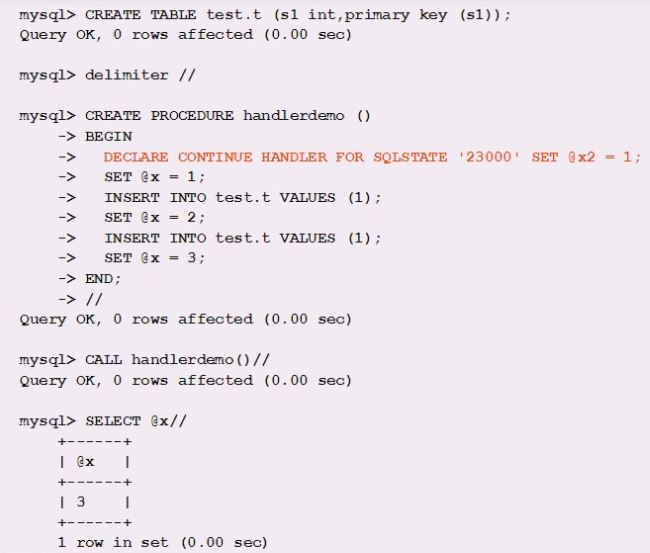
EXIT - немедленно выйти, завершив процедуру

UNDO - попытка отката назад

Справочник кода ошибок в SQL

<http://dev.mysql.com/doc/refman/5.7/en/error-messages-server.html>

Пример обработчика ошибок:

1. Навесили обработчик на ошибку 23000 - дублирование ключа с директивой CONTINUE

2.Назначаем глобальной переменной @x значения, что бы следить за ходом выполнения

3. Выводим результат.

Поскольку стоит CONTINUE, сервер игнорирует повторяющееся значение в колонке с PRIMARY KEY и продолжает выполнение команд

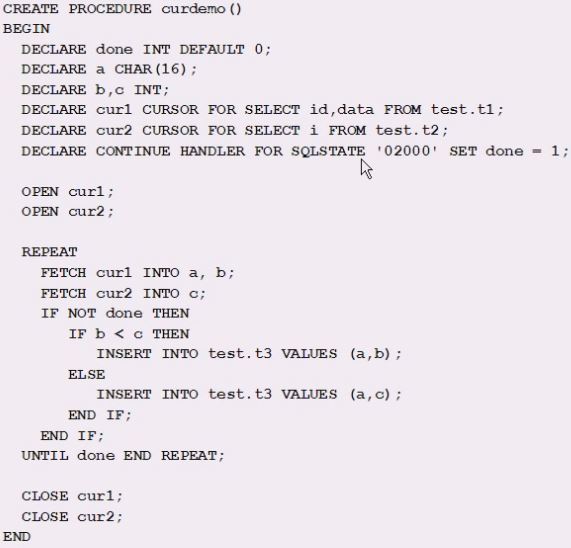
**Курсоры в SQL**

Поскольку в SQL все является таблицей, считывать результат можно только по колонкам, не по рядам. Считывать по рядам позволяет курсор

DECLARE - обьявление переменных и курсора

OPEN - инициализация курсора

Вешаем обработчик на ошибку 02000 - выборка закончена. Это будет означать конец операции, когда done == true, цикл завершится



**Управляющие конструкции в хранимых процедурах**

**IF**

-- например, есть переменная record\_count, которая хранит инфу о количестве записей

IF record\_count = 0 THEN

--данных нет. Вернем предупреждающую таблицу

SELECT 0 AS id, 'Данных нет' AS title

ELSE

-- вернем результат

SELECT \*

FROM course\_by\_dates\_results

END IF;

**CASE** - аналог switch

CASE case\_value

WHEN when\_value THEN statement\_list

WHEN when\_value THEN statement\_list

ELSE statement\_list

END CASE

**LOOP** - цикл без условия

LEAVE - прекращение цикла (аналог break в js)

ITERATE - переходит к следующему шагу цикла (аналог continue в С-языках)

CREATE PROCEDURE doiterate(p1 INT)

BEGIN

label1: LOOP -- запускаем цикл с меткой

SET p1 = p1 + 1; -- инкремент

IF p1 < 10 THEN ITERATE label1; END IF -- возвращается к метке label1 пока p1 < 10

LEAVE label1; -- поскольку IF не вернул к старту, выходит из цикла

END LOOP label1;

SET @x = p1;

END

**REPEAT** - классический цикл с проверкой постусловия (аналог do ... while ). Работает, пока условие == false

CREATE PROCEDURE dorepeat(p1 INT) -- определение repeat

BEGIN

SET @x = 0;

REPEAT SET @x = @x +1; UNTIL @x > p1 END REPEAT;

END

CALL dorepeat (1000) -- вызов

**WHILE** - цикл с предусловием. Работает, пока условие == true (аналог - оператор while)

CREATE PROCEDURE dowhile()

BEGIN

DECLARE v1 INT DEFAULT 5;

WHERE v1 > 0 DO

...

SET v1 = v1 - 1;

END WHILE;

END

**ХРАНИМЫЕ ФУНКЦИИ**

В отличии от процедур, которые могу вернуть таблицу, хранимая функция возвращает скаляр - т.е. одно значение.

Создается точно так же, как хранимая процедура, отличие - указать тип возвращаемых данных

Частое использование - что бы отсортировал по строке, например есть список аудиторий   
БК-1, БК-2, БК-12. Сортировка ORDER BY вернет в таком порядке: БК-1, БК-12, БК-2. Для сортировки по строке, пишем функцию:

-- обязательно указать тип данных

CREATE FUNCTION sp\_room\_number (room VARCHAR(5)) RETURNS INT

BEGIN

-- берет подстроку, начиная с 4 символа и переводит в число через умножение на 1

RETURN SUBSTR(room, 4) \* 1;

END;

-- вызов процедуры

SELECT sp\_room\_number('БК-10') -- вернет 10

--использование

SELECT room, lesson\_date

FROM lessons

ORDER BY sp\_room\_number(room);

**ТРИГГЕРЫ**

Это хранимые процедуры особого типа, которую пользователь не вызывает непосредственно, а исполнение которой обусловленно определенной модификацией данных в заданной таблице или столбце.

Триггер запускается автоматически при попытке изменения данных в таблице, с которой связан.

Момент записи триггера определяется с помощью ключевых слов BEFORE (триггер запускается до выполнения связанного с ним события, например до добавления записи) или AFTER (после события)

TRIGGER trigger\_name trigger\_time trigger\_event

ON tbl\_name FOR EACH ROW trigger\_stmt

**ТРАНЗАКЦИИ**

**Блокировки и их типы**

**Блокировки - устаревшие методы, лучше использовать транзакции**

Конкурентные запросы - когда например серевер делает группировки по записи, и в это же время приходит запрос на добавление записей в эту же таблицу. Сервер нормально разруливает единичные запросы, но когда приходит серия запросов, например необходимо сделать пару SELECT перед тем, как сделать INSERT в таблицу, и между SELECT и INSERT вклинивается конкурентный запрос, большая вероятность, что сервер его выполнит, чем нарушит работу INSERT.  
Блокировка таблиц - пока идет работа серии с какой-то таблицей, она заблокирована

-- блокировка на чтение. Пользователь с другим подключением нормально считывает

-- информацию из таблицы, но модифицировать данные он не может

LOCK TABLES teachers READ;

-- снять блокировку

UNLOCK TABLES;

-- блокировка на запись - пользователям запрещены даже SELECT

LOCK TABLES teachers WRITE;

Поскольку конкурирующий запрос будет ждать, пока таблица будет разблокирована, это потеря времени. Что бы этого избежать, на MyISAM есть отложенные запросы

-- отложенный запрос - северу дается команда на INSERT тогда, когда сможет,

-- последующий код продолжает выполнятся (типа асинхронности)

INSERT DELAYED INTO teachers (id, name) VALUES (10, 'Новый препод');

Именно потому, блокировки должны быть короткими. Редактируем таблицу - сохраняем данные в буфере - по нажатие кнопки "ОК" блокируется таблица, вносятся изменения и тут же разблокируется.

**Блокировки записей**

Аналог mutex для виндовс - гарантия того, что никакая другая программа не сможет что-то сделат. Например запуск программы только в одной копии. В MySQL тоже самое. На практике используется очень редко из-за ненадежности

-- первый аргумент - флажек, второй - время в секундах, до разблокировки записи

SELECT GET\_LOCK('abc', 10);

SELECT RELEASE\_LOCK('abc');

**ТРАНЗАКЦИИ**

Транзакция - процесс, которая может быть выполнена либо полностью успешно, соблюдая целостность данных и независимо от параллельно идущих других транзакций, либо не выполнена вообще и тогда не должна произвести никакого эффекта.

Транзакция - это не сущность БД, это логика. Транзакция может быть очень большой:

1. например есть сайт, БД 1С и факс.
2. Пользователь покупает товар с сайта, товар блокируется в 1С и пользователю отправляется факс.
3. В это время сайт глючит. Необходимо провести компенсацию: на сайте вывести ошибку, разблокировать товар в 1С и выслать пользователю второй факс с извенениями.

Это все будет единой транзакцией.

**ACID**

Что бы определить, что такое транзакция, нужно мысленно пропустить объект через ACID:

* **Automicity** (атомарность) - определяет, что транзакция является наименьшим, неделимым блоком шагов алгоритма. Другими словами, любые части (подоперации) транзакции либо выполняются все одновременно, либо не выполняется ни одной такой части. Поскольку в реальности все же возникает некоторая последовательность выполнения команд внутри транзакции, вводится понятие "отката" (rollback), при котором результаты всех уже произведенных действий возвращаются в исходное состояние.
* **Consistency** (согласованность): по окончанию транзакция оставляет данные в непротиворечивом состоянии. Например, если поле в базе данных описано как имеющее только уникальные значения строк, то ни при каком исходе транзакции дубликатов никакой строки появиться не может.
* **Isolation** (изоляция): конкурирующие, параллельно текущие во времени транзакции не могут пересекаться на одних и тех же ресурсах. Для обеспечения изоляции вводятся, к примеру, специальные замки на измененных ресурсах, запрещающие другим транзакциям эти ресурсы менять до окончания поменявшей транзакции.
* **Durability** (долговечность): независимо от проблем на нижних уровнях (к примеру, обесточивание системы или сбои в оборудовании) изменения, сделанные успешно завершенной транзакцией, останутся сохраненными после возвращения системы в работу.

**Синтаксис**

START TRANSACTION; -- определяем транзакцию

SELECT id, name FROM teachers;

INSERT INTO teachers(id, name) VALUES (11, 'Новый препод');

ROLLBACK; -- откатываем все изменения на старт транзакции

COMMIT; -- подтверждаем транзакцию

Пока идет транзакция, другие пользователи могут обращаться к таблице и даже изменять ее, транзакция никак не блокирует других пользователей. Изменения в таблицу вносятся только после COMMIT. При конкурирующих изменениях сервер все-таки блокирует записи.

InnoDB все действия делает в транзакциях изначально. На массированных операциях назначение AUTOCOMMIT = 0 ускорит выполнение операций. Причем при обрыве соединения, InnoDB самостоятельно сделает ROLLBACK

SET AUTOCOMMIT = 1; -- можно изменить на 0 для текущей сессии

**Изоляция**

Всего существует 4 уровня изоляции. Иногда изоляция вредит, а не помогает, и тогда ее можно изменить. Какие могут быть проблемы: записи-фантомы (запись исчезает и появляется, потому что какая-то другая транзакция уже что-то делает с этой же таблицей), повторяющиеся записи

*Уровень 0*. Никакой изоляции нет, все проблемы присутствуют в полном спектре

SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

*Уровень 1.* Изменения видны после COMMIT, но если были одновременно открыты две транзакции, и первая завершилась быстрее, внеся изменения в таблицу, эти изменения тут же отобразятся во второй транзакции, что создаст видимость добавления-отсутствия некоторых данных

SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;

*Уровень 2.* Изоляция по дефолту. Пока другая сессия не введет COMMIT, изменения не будут видны. При чем, вторая транзакция видит таблицу на момент своего открытия, потому если первая транзакция внесла изменения в таблицу и даже сделала COMMIT, все равно вторая транзакция будет видеть таблицу на момент открытия, пока сама не сделает COMMIT. Или ROLLBACK

SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;

*Уровень 3.* Транзакция не видит внесенных изменений даже сделанных самой собой. На серверах используется редко

SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;

**ТИПЫ ХРАНИЛИЩ И ДВИЖКИ**

**Движки** - это как плагины к БД. Каким образом хранятся данные на нижнем уровне.

SHOW ENGINES; -- показывает движки

CREATE TABLE t (i INT) ENGINE = INNODB; -- указать движок

ALTER TABLE t ENGINE = INNODB; -- сменить на существующей таблице

Некоторые движки отличаются между собой, потому просто так сменить не удастся. При попытке сменить InnoDB на MyISAM будет жаловаться, что предварительно нужно убрать внешние ключи, которые MyISAM не поддериживает.

* MyISAM **-** собственный движок MySQL. Самый быстрый на запись, медленный на чтение, нету транзакций. В данный момент не разрабатывается. Оптимизирован для добавления и изменения данных
* InnoDB **-** транзакционный движок. Используется всегда бляд. Особенность движка - физически хранит файлы отдельно от форматов, обычно в папке на уровеньт выше. Оптимизирован под чтение.
* Archive - не поддерживает PRIMARY KEY и FOREIGN KEY. Идеально для архивов, справочников и т.д.
* CSV - FOREIGN KEY нет, транзакций тоже нет. Хранит записи в текстовом виде, разделяя запятой и перевод строки для разделения строк. Идеально для загрузки в MySQL из текстовых записей, если есть огромные объемы информации
* MRG\_MYISAM - склеивает большие объемы данных в одну таблицу. Это тот же MyISAM
* MEMORY - хранит все данные в оперативной памяти. На диск ничего не записывается. Идеален для временных таблиц
* FEDEREATED - для распределенных БД. Несколько баз данных собираются в один, и сервер думает, что это его собственная база. Работает быстро и хорошо.
* BLACKHOLE - удаляет все записи внутри таблиц с этим движком

**РЕЗЕРВНОЕ КОПИРОВАНИЕ**

В MySQL есть встроенная утилита для бекапов - mysqldump. ЕЕ синтаксис в целом совпадает с синтаксисом MySQL. Что она делает: создает файл, в котором создает команды по созданию указанной БД. Файл создается с расширением sql. Из командной строки

-- создать бекап в файле backup.sql

mysqldump -uroot -ppaswword world > backup.sql

-- загрузить бекап из файла backup.sql в world (world должен быть создан)

mysqldump -uroot -ppaswword world < backup.sql

**Способ 2**

Для начала необходимо создать ПУСТУЮ папку, добавить ключ -Т,после указать путь к пустой папке, после указать имя БД

mysqldump -uroot -ppaswword world -T d:\MySQL\Samples\world\backup2 world

Данные выгрузятся в папку потаблично, разбитые на sql-текстовые команды и собственно сами таблицы, tab-separated. Однако такой бекап уже придется грузить вручную, потаблично, через DATA LOAD INFILE. Восстановление БД в этом случае предполагает написание скриптов для автоматического восстановления

**Способ 3**

Через Workbench. Открываем его, заходим в нужный сервер, слева появится колонка с разными опциями. Нужно найти Data Export. Выбираем нужную БД, снизу указываем путь, куда ее сохранить, выбираем, хотим сохранить в раздельные таблицы или в один монолитный файл, отмечаем галочками все пункты (что бы выгрузилось все), нажать Export. Выведет че-то типа

Running: mysqldump.exe --defaults-file="c:\users\alexandr\appdata\local\temp\tmpxuehng.cnf" --user=root --host=localhost --protocol=tcp --port=3306 --default-character-set=utf8 "world" "city"

**Вспомогательные команды**

CHECK TABLE city; -- проверка состояния таблицы, должно быть ОК

REPAIR TABLE city QUICK; -- восстановление таблицы (не работает на InnoDB)

OPTIMIZE TABLE city; -- дефрагментирует таблицу (не работает на InnoDB)

**УПРАВЛЕНИЕ ПОЛЬЗОВАТЕЛЯМИ**

MySQL идентифцирует пользователей не только по паре логин\пароль, а по тройке:  
логин, сетевой адрес машины пользователя и пароль:

CREATE USER 'fedya'@'%' -- логин федя, сетевой адрес - любой (любая машина)

SET PASSWORD FOR 'fedya'@'%' = PASSWORD('qwerty123') -- пароль

Информация о пользователях хранится в файле mysql.user:

SELECT \* FROM mysql.user \G

-- более удобный вариант через information\_schema

use information\_schema;

desc table\_privileges;

SELECT \* FROM user\_privileges;

Пароли хранятся в хэше, и вносить изменения можно прямо в таблицу MySQL

**Раздача привелегий для пользователей**

Дать привилегии

GRANT SELECT, INSERT -- разрешен доступ к селектам и инсертам

ON world.\* -- на любой таблицу в БД world

TO 'fedya'@'%' -- пользователю fedya на любой машине

Отобрать привилегии

--забрать все привилегии. Если заданы, как вверху - надо указывать конкретно

REVOKE ALL PRIVILEGES --или REVOKE SELECT, INSERT PRIVILEGES

ON world.product\_log --к таблице product\_log

FROM 'fedya'@'%' --у юзера федя

Более удобным способом является создание и раздача привилегий через Workbench, в разделе Users and privileges. Там можно перейти в таб schema privileges и настроить выборочные права, или выбрать из списка готовых шаблонов в табе Administative Roles

**Восстановить доступ к root**

Сброс пароля для root:

1. Нужен доступ к командной строке
2. Остановить MySQL-службу:

net stop mysql

1. Дадим команду на доступ со всеми правами:

mysqld --skip-grant-tables

1. Сервер уже работает. Не закрывая это окно, логинимся в mysql обычным способом, не вводя никаких логинов и паролей
2. Меняем пароль для нужного юзера:

UPDATE USER SET PASSWORD = PASSWORD('newpass') WHERE user = 'root'

1. Перезапустить сервер

**ОПТИМИЗАЦИЯ ЗАПРОСОВ**

Некоторые запросы могут тормозить. Посмотреть, что происходит в запросе можно:

EXPLAIN

SELECT COUNT(\*)  
 FROM city c1

WHERE EXISTS (SELECT \* FROM city c2 WHERE c1.name = c2.name)

\G

-- оптимизируем через добавление индекса

CREATE INDEX ixName on city (name);

-- теперь верхний запрос отработает в 300 раз быстрее, потому что проводится проверки не

-- перемножаются, а проводится только 1

**Расстановка индексов**

Лучше взять существующию модель БД, распечатать ее и началь расставлять точки.

1. Что индексируется обязательно? Связи. PRIMARY KEY уже является индексом, как и UNIQUE KEY. Обязательно стоит индексировать FOREIGN KEY
2. При разработке приложени, запросы стоит писать и сохранять в отдельный файл, после этого просмотривать их и расставлять индексы для:

* WHERE запросов - *WHERE =, WHERE <, WHERE >*

WHERE teacher > 3; -- надо проиндексировать teacher

* *GROUP BY*
* *ORDER BY*

1. Создаем индексы для каждого через CREATE INDEX, каждому индексу даем логическое имя
2. Проверяем запросы через EXPLAIN все запросы. Смотрим, что пишет possible\_keys: если там везде NULL, значит соотвутсвующий индекс можно удалить.
3. Получаем карту индексов, готово

Этот способ работает железно, кроме картографических таблиц - таблицы логов. Любые дневники и логи следует привести в первую нормальную форму и больше их не трогать - это ускоряет работу сервера.

Нормализация замедляет работу сервера. Потому оптимальной является 2-я или максимум 3я форма

Для сложных выборок на больших таблицах, для рассчета значений типа MAX(), MIN(), AVG(), которые очень дорогие, можно сделать предрасчитанную таблицу. Ночью, когда нагрузка на сервер минимальная, поставить в планировщике задач запуск хранимой процедуры с подсчетом этих значений. После подсчета, скорость обращения к ней будет в несколько десяток раз быстрее

На запросах типа

WHERE name LIKE 'm%' \G;

Индекс будет использоваться, т.к. начало имеет четко определенную букву (если конечно колонка *name* была предварительно проиндексирована), но на запросе

WHERE name LIKE '%m%' \G;

Индекс использоваться уже не будет

**Query cache**

SHOW VARIABLES LIKE 'query\_cache%' -- посмотреть все переменные, которые отвечают за кеш

*query\_cache\_limit* - максимальный размер кеша

*query\_cache\_min\_res\_unit*  - максимальный размер пакета для одного кеша (для исключения больших кешей

*query\_cache\_size -* размер текущего кеша

Настраивается в my.ini а так же в Workbench -> Option files -> Performance