ПРОЕКТИРОВАНИЕ РЕЛЯЦИОННЫХ БАЗ ДАННЫХ

Процесс **проектирования** базы данных включает в себя следующие шаги:

1. Инфологическое проектирование.

– Определение задач, стоящих перед базой данных.

– Сбор и анализ документов, относящихся к исследуемой предметной области.

– Описание особенностей предметной области, которые позволяют установить зависимости и связи между объектами.

– Создание модели предметной области.

– Определение групп пользователей и перечня задач, стоящих перед каждой группой.

2. Выбор аппаратной и программной платформы.

3. Выбор системы управления базой данных (СУБД) и других инструментальных программных средств.

4. Логическое проектирование базы данных.

– Концептуальное проектирование (не зависит от типа базы данных):

• выделение сущностей и их атрибутов;

• определение связей между сущностями.

– Логическое проектирование – развитие схемы БД с учетом принимаемой модели (иерархической, сетевой, реляционной и т.д.).

– Создание схем отношений, определение типов данных атрибутов и ограничений целостности.

– Нормализация отношений.

– Определение прав доступа пользователей к объектам БД.

5. Физическое проектирование БД.

– Написание текста создания основных объектов базы данных на языке SQL (пользователи, таблицы и др.)

– Написание текста создания вспомогательных объектов базы данных (представления, индексы, триггеры, роли и т.д.)

**Пример проектирования базы данных**

Пусть нужно спроектировать базу данных ПРОДАЖИ, которая будет отображать:

– сведения о товарах, которые хранятся в каталоге;

– сведения о клиентах, которые когда-либо сделали заказ или в будущем собираются его совершить;

– сведения о заказанных товарах.

**1.** **Инфологическое проектирование**

Основными задачами инфологического проектирования является определение предметной области системы и формирование информационно-логической модели предметной области.

Инфологическая модель предметной области представляет собой описание структуры и динамики предметной области, характера информационных потребностей пользователей в терминах, понятных пользователю и не зависимых от реализации БД.

Инфологическую модель можно представить в виде *диаграммы вариантов использования* (**use case** диаграммы).

Продавец и заказчик взаимодействуют с системой продажи товаров, т. е. обращаются к сервису "Оформить заказ на покупку товара".

Этот сервис выступает в качестве варианта использования диаграммы, первоначальная структура которой может иметь вид:



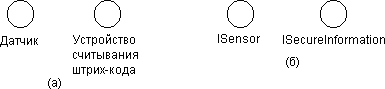
продавец

заказчик

К базовым элементам диаграммы относятся: вариант использования, актер и интерфейс.

**Вариант использования** применяется для обозначения особенностей поведения системы или другой сущности (например, оформление заказа на покупку товара, получение информации о кредитоспособности клиента и пр.)

Описание: Рис20**Актер** – это внешняя по отношению к проектируемой системе сущность, которая взаимодействует с системой и использует ее возможности для решения определенных задач.

**Интерфейс** служит для спецификации параметров модели. В диаграммах вариантов использования интерфейсы определяют совокупность операций, обеспечивающих необходимый набор сервисов или функциональности для актеров.

Один продавец может участвовать в оформлении нескольких заказов, в то же время каждый заказ может быть оформлен только одним продавцом, который несет ответственность за корректность его оформления.

С другой стороны, каждый заказчик может оформлять на себя несколько заказов, но, в то же время, каждый заказ должен быть оформлен на единственного заказчика, к которому переходят права собственности на товар после его оплаты.

На следующем этапе разработки диаграммы вариант использования "Оформить заказ на покупку товара" может быть уточнен на основе введения в рассмотрение дополнительных вариантов. Детальный анализ процесса продажи товаров позволяет выделить в качестве отдельных сервисов такие действия, как:

– обеспечить заказчика информацией о товаре;

– согласовать условия оплаты товара;

– заказать товар со склада.

Продажа товаров по каталогу предполагает наличие самостоятельного информационного объекта — каталога товаров, который не зависит от реализации сервиса по обслуживанию заказчиков.

Каталог товаров может запрашиваться заказчиком или продавцом при необходимости выбора товара и уточнения деталей его продажи.

Полученная уточненная диаграмма вариантов использования будет содержать 6 вариантов использования и 2 субъекта, между которыми установлены отношения включения и расширения.



На диаграммах для связывания элементов используются *соединительные линии*, которые называются отношениями.

**Виды отношений**: отношение ассоциации, отношение обобщения, отношение включения, отношение расширения.

***Отношение ассоциации*** служит для обозначения роли актера в отдельном варианте использования. Отношение устанавливает, какую конкретную роль играет актер при взаимодействии с экземпляром варианта использования. Графическое обозначение отношения ассоциации может включать дополнительные условные обозначения (имя и кратность).



продавец

***Отношение расширения*** между вариантами использования обозначается пунктирной линией со стрелкой, направленной от того варианта использования, который является расширением для исходного варианта использования. Данная линия со стрелкой помечается ключевым словом “extend” (“расширяет”).

Описание: Рис25

Отношение расширения отмечает тот факт, что один из вариантов использования может присоединять к себе дополнительное поведение.

***Отношение обобщения*** графически обозначается сплошной линией со стрелкой, которая указывает на родительский вариант использования.

Описание: Рис26

Отношение обобщения применяется в том случае, когда необходимо отметить, что дочерние варианты использования обладают всеми атрибутами и особенностями поведения родительских вариантов, участвуют во всех отношениях родительских вариантов. В свою очередь, дочерние варианты могут наделяться новыми свойствами поведения, которые отсутствуют у родительских вариантов использования.

***Отношение включения*** между двумя вариантами использования указывает, что поведение одного варианта использования включается в качестве составного компонента в последовательность поведения другого варианта использования. Графически данное отношение обозначается пунктирной линией со стрелкой, направленной от базового варианта использования к включаемому. Помечается ключевым словом “include” (“включает”).

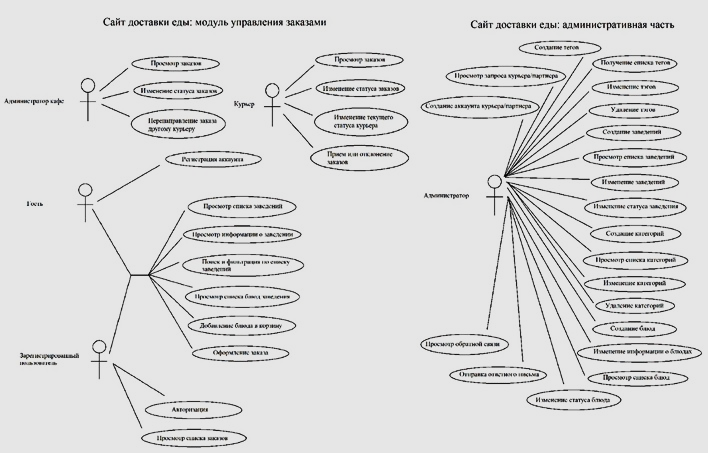
Описание: Рис28

***Кратность***. При моделировании возникает необходимость в указании количества объектов, связанных посредством одного экземпляра ассоциации. Это число называется кратностью роли ассоциации. Кратность указывает на то, сколько объектов должно соответствовать каждому объекту на противоположном конце. Кратность можно задать равной единице (1), указать диапазон: “ноль или единица” (0..1), “много” (0..\*), “единица или больше” (1..\*), число (например, 3).



Пример полной диаграммы вариантов использования

для интернет-магазина продажи еды



**2.** **Выбор аппаратной и программной платформы**.

Для выполнения этого этапа необходимо ориентировочно знать объём работы организации, а также иметь представление о характере и интенсивности запросов.

Объём внешней памяти, необходимый для функционирования системы, складывается из двух составляющих: память, занимаемая модулями СУБД (ядро, утилиты, вспомогательные программы), и память, отводимая под данные (МД). Для реальных баз данных обычно наиболее существенным является МД.

На основе результатов анализа предметной области можно приблизительно оценить объём памяти, требуемой для хранения данных.

Примем ориентировочно для базы данных ПРОДАЖИ что:

* одновременно осуществляется около пятидесяти заказов в день (по 1К на каждый заказ);
* выполнение каждого заказа состоит в среднем из четырёх этапов (по 0,5К на этап);
* в компании работают 10 сотрудников (по 0,5К на каждого сотрудника);
* устаревшие данные переводятся в архив (накапливаются в архиве БД).

Тогда объём памяти для хранения данных в год примерно составит:

Mд = 2(50\*1+50\*4\*0,5+10\*0,5) = 310 \* 365 ≈ 114Мб

Коэффициент 2 нужен для того, чтобы учесть необходимость выделения памяти под дополнительные структуры (например, индексы).

Требуемый объём оперативной памяти определяется также на основании анализа интенсивности запросов и объёма результирующих данных. Для нашей БД никаких специальных требований к объёму внешней и оперативной памяти компьютера не предъявляется.

**3.** **Выбор системы управления базой данных (СУБД) и других инструментальных программных средств**

Если заказы будут осуществляться через сайт, то для реализации требуемых функций лучше всего подходит MySQL. База данных поддерживает реляционную модель данных и предоставляет разнообразные возможности для работы с данными.

**4. Логическое проектирование базы данных**

***Концептуальное проектирование***.

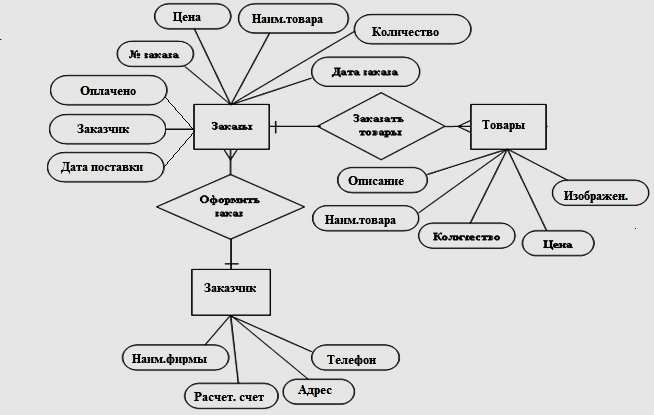
Информационные поля: *Наименование товара, Цена, Количество, Изображение, Описание товара, Наименование фирмы-заказчика, Адрес заказчика, Телефон заказчика, Расчетный счет заказчика, Номер заказа, Наименование заказанного товара, Цена продажи заказанного товара, Количество заказанного товара, Дата заказа, Дата поставки.*

Анализ позволяет выявить **сущности** (объекты предметной области) и **атрибуты** (свойства сущностей).

Модель предметной области может быть описана любым удобным для разработчика способом (словесное описание, набор формул, диаграмма потоков данных и т.п.).

Обычно при проектировании баз данных создается схема предметной области в виде **ER–диаграммы** (entity-relation diagram, диаграмма «сущность-связь»).

Пример **ER**–диаграммы



Для базы данных ПРОДАЖИ можно выделить следующие сущности предметной области и их атрибуты:

Сущность: **Товары**.

Атрибуты: Наименование товара, Цена, Количество, Изображение, Описание товара.

Сущность: **Заказчики**.

Атрибуты: Наименование фирмы, Адрес, Телефон, Расчетный счет.

Сущность: **Заказы**.

Атрибуты: Номер заказа, Наименование товара, Цена продажи, Количество, Общая стоимость, Дата заказа, Дата поставки, Заказчик, Телефон, Оплачено.

***Логическое проектирование***

– Создание схем отношений, определение типов данных атрибутов и ограничений целостности.

**Схема отношения «Товары»:**

|  |  |  |
| --- | --- | --- |
| **Имя поля** | **Тип** | **Примечания** |
| Наименование | Текстовый | PK, nvarchar(20), not null |
| Цена | Денежный | real |
| Количество | Числовой | int |
| Изображение | Поле объекта OLE | varbinary |
| Описание товара | Текстовый | nvarchar(120) |

**Схема отношения «Заказчики»:**

|  |  |  |
| --- | --- | --- |
| **Имя поля** | **Тип** | **Примечания** |
| Наименование фирмы | Текстовый | PK, nvarchar(20), not null |
| Адрес | Текстовый | nvarchar(20) |
| Телефон | Текстовый | nvarchar(50) |
| Расчетный счет | Текстовый | nvarchar(15) |

**Схема отношения «Заказы»:**

|  |  |  |
| --- | --- | --- |
| **Имя поля** | **Тип** | **Примечания** |
| Номер заказа | Числовой | PK, nvarchar(10) |
| Наименование товара | Текстовый | FK, nvarchar(20), not null |
| Цена продажи | Денежный | real |
| Количество | Числовой | int |
| Дата заказа | Дата / время | date |
| Дата поставки | Дата / время | date, больше даты заказа |
| Заказчик | Текстовый | FK, nvarchar(20) |
| Телефон | Текстовый | nvarchar(50) |
| Оплачено | Текстовый | nvarchar(3) |

***Нормализация отношений***

Проектирование схемы БД должно решать задачи минимизации дублирования данных и упрощения процедур их обработки и обновления. Для решения подобных проблем проводится **нормализация** отношений.

**Нормализация** — это процесс организации данных, включающий создание таблиц и установление отношений между ними в соответствии с правилами, которые обеспечивают защиту данных и делают **базу** **данных** более гибкой, устраняя избыточность и несогласованные зависимости.

Пример. Пусть исходная информация о продажах имеет следующие поля:

**Наименование товара, Цена, Количество на складе, Изображение, Описание товара, Наименование заказанного товара, Цена заказанного товара, Количество заказанного товара, Общая стоимость, Дата поставки, Заказчик, Адрес заказчика, Расчетный счет заказчика, Телефон заказчика, Оплачено**.

Чтобы таблица соответствовала **1-й нормальной форме**, необходимо, чтобы все значения ее полей были неделимыми, не вычисляемыми и не должно быть совпадающих строк.

Для уменьшения избыточности информации, нужно преобразовать первоначальную таблицу к двум таблицам:

**Наименование товара, Цена, Количество на складе, Изображение, Описание товара**

и

**Наименование заказанного товара, Цена заказанного товара, Количество заказанного товара, Общая стоимость, Дата поставки, Заказчик, Адрес заказчика, Расчетный счет заказчика, Телефон заказчика, Оплачено**.

Первая таблица соответствует первой нормальной форме, а во второй таблице имеется избыточность. Если один заказчик купит много товаров, то в таблице будут повторяться одни и те же исходные реквизиты заказчика. К тому же поле **Общая стоимость** может быть вычислено по полям **Цена заказанного товара** и **Количество заказанного товара**. Поэтому вторую таблицу надо разбить на две:

**Заказчик, Адрес заказчика, Расчетный счет заказчика**

и

**Наименование заказанного товара, Цена заказанного товара, Количество заказанного товара, Дата поставки, Заказчик, Телефон заказчика, Оплачено**.

Поле **Общая стоимость**, как вычисляемое, в структуру таблицы не включается.

Чтобы таблица соответствовала **2-й нормальной форме** (2NF), необходимо, чтобы она находилась в 1-й нормальной форме и все не ключевые поля полностью зависели от ключевого.

Из полученных трех таблиц первые две соответствуют второй нормальной форме, а в третьей таблице нет ключевого поля. Если в качестве такового взять поле **Наименование заказанного товара**, то оно может принимать одно и то же значение для различных заказчиков. Поэтому требуется ввести новое поле, которое являлось бы первичным ключом для всех остальных. Таким ключом может стать поле **Номер заказа**, которое надо добавить, т. е. третья таблица теперь должна содержать поля:

**Номер заказа, Наименование заказанного товара, Цена заказанного товара, Количество заказанного товара, Дата поставки, Заказчик, Телефон заказчика, Оплачено**.

Определим имена таблиц:

ТОВАРЫ (**Наименование товара, Цена, Количество на складе, Изображение, Описание товара**);

ЗАКАЗЧИКИ (**Заказчик, Адрес заказчика, Расчетный счет заказчика**);

ЗАКАЗЫ (**Номер заказа, Наименование заказанного товара, Цена заказанного товара, Количество заказанного товара, Дата поставки, Заказчик, Телефон заказчика, Оплачено)**.

Ключевые поля подчеркнуты.

Для перехода к **3-й нормальной форме** (3NF), необходимо обеспечить, чтобы все таблицы находились во 2-й нормальной форме и все не ключевые поля в таблицах не зависели друг от друга.

В таблице ЗАКАЗЫ поля **Заказчик** и **Телефон заказчика** взаимно зависимы. Чтобы привести таблицу к 3 нормальной форме надо поле **Телефон заказчика** переместить в таблицу ЗАКАЗЧИКИ.

Теперь полученные таблицы соответствуют всем требованиям.

ТОВАРЫ (**Наименование товара, Цена, Количество на складе, Изображение, Описание товара**);

ЗАКАЗЧИКИ (**Заказчик, Адрес заказчика, Расчетный счет заказчика, Телефон заказчика**);

ЗАКАЗЫ (**Номер заказа, Наименование заказанного товара, Цена заказанного товара, Количество заказанного товара, Дата поставки, Заказчик, Оплачено)**.

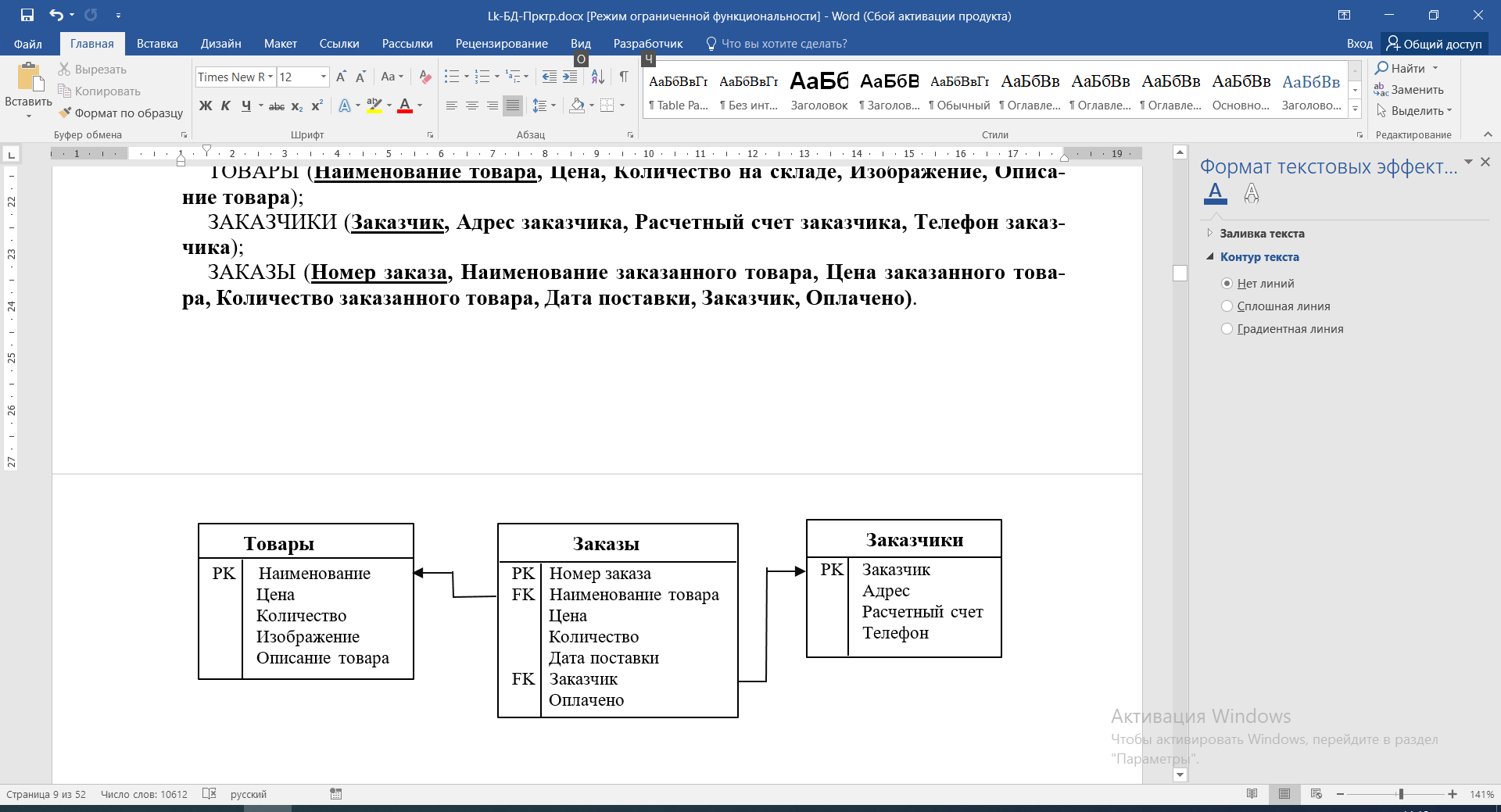


Схема базы данных

***Описание групп пользователей и прав доступа****.*

Для каждой группы пользователей надо определить права доступа к каждой таблице. Права доступа должны быть распределены так, чтобы для каждого объекта БД был хотя бы один пользователь, который имеет право добавлять и удалять данные из объекта.

Права доступа к таблицам для групп пользователей

|  |  |  |
| --- | --- | --- |
| Таблицы | Группы пользователей (роли) | |
| Продавцы | Заказчики |
| Товары | siud | s |
| Заказчики | s | suid |
| Заказы | s | s |

Используются сокращения:

s – чтение данных (select);

i – добавление данных (insert);

u – модификация данных (update);

d – удаление данных(delete).

**Система управления базами данных MySQL**

MySQL — это ***реляционная*** система управления базами данных с открытым исходным кодом.

В настоящее время является самой популярной в веб-приложениях. Из преимуществ стоит отметить простоту использования, гибкость, низкую стоимость владения (относительно платных СУБД), а также масштабируемость и производительность.

Разработчиком данной СУБД была шведская компания MySQL AB. В **1995** году она выпустила первый релиз MySQL (написана на С и С++).

В 2008 году компания MySQL AB была куплена компанией Sun Microsystems ($1 млрд), а в 2010 году уже компания **Oracle** поглотила Sun и тем самым приобрела права на торговую марку MySQL ($7,4 млрд).

Текущей актуальной версией СУДБ является версия **8.0**, которая вышла в январе 2018 года.

MySQL обладает кроссплатформенностью, имеются дистрибутивы под самые различные ОС, в том числе популярные версии Linux, Windows, MacOS.

Для **масштабных** корпоративных решений используется база MS SQL Server, которая нуждается в больших мощностях.

База данных MySQL — для **компактных** веб-приложений.

В MS SQL Server входят инструменты анализа данных, сервер отчетов и многое другое, для чего в MySQL требуется искать сторонние решения. Однако это же обстоятельство придает СУБД большую гибкость.

**MySQL 8 выпускается в нескольких редакциях**

**MySQL Standard Edition** – это редакция со стандартным функционалом, она позволяет реализовать высокопроизводительные, надежные и масштабируемые приложения.

**MySQL Enterprise Edition** – данная редакция включает в себя полный функционал и набор дополнительных инструментов, которые предназначены для достижения высокой масштабируемости, безопасности и надежности.

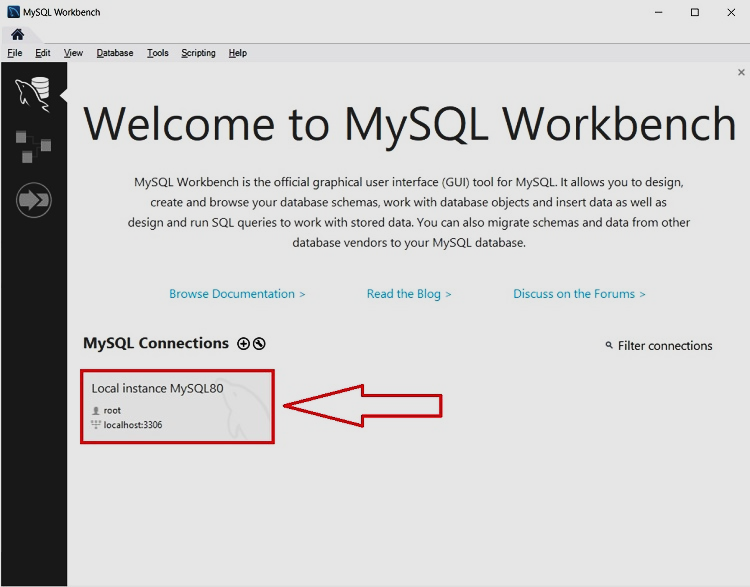
**MySQL Cluster** (CGE) – это распределенная база данных, сочетающая линейную масштабируемость и высокую доступность. Она предназначена для критически важных приложений.

**MySQL Community Edition** – это бесплатная редакция, которую можно скачать с официального сайта MySQL. Редакция поддерживает весь необходимый функционал для реализации полноценных интернет ресурсов.

**MySQL. Основные понятия**

MySQL активно используется при обработке данных на **сайтах**. База данных обладает кроссплатформенностью, имеются дистрибутивы под самые различные ОС, в том числе наиболее популярные версии Linux, Windows, MacOS.

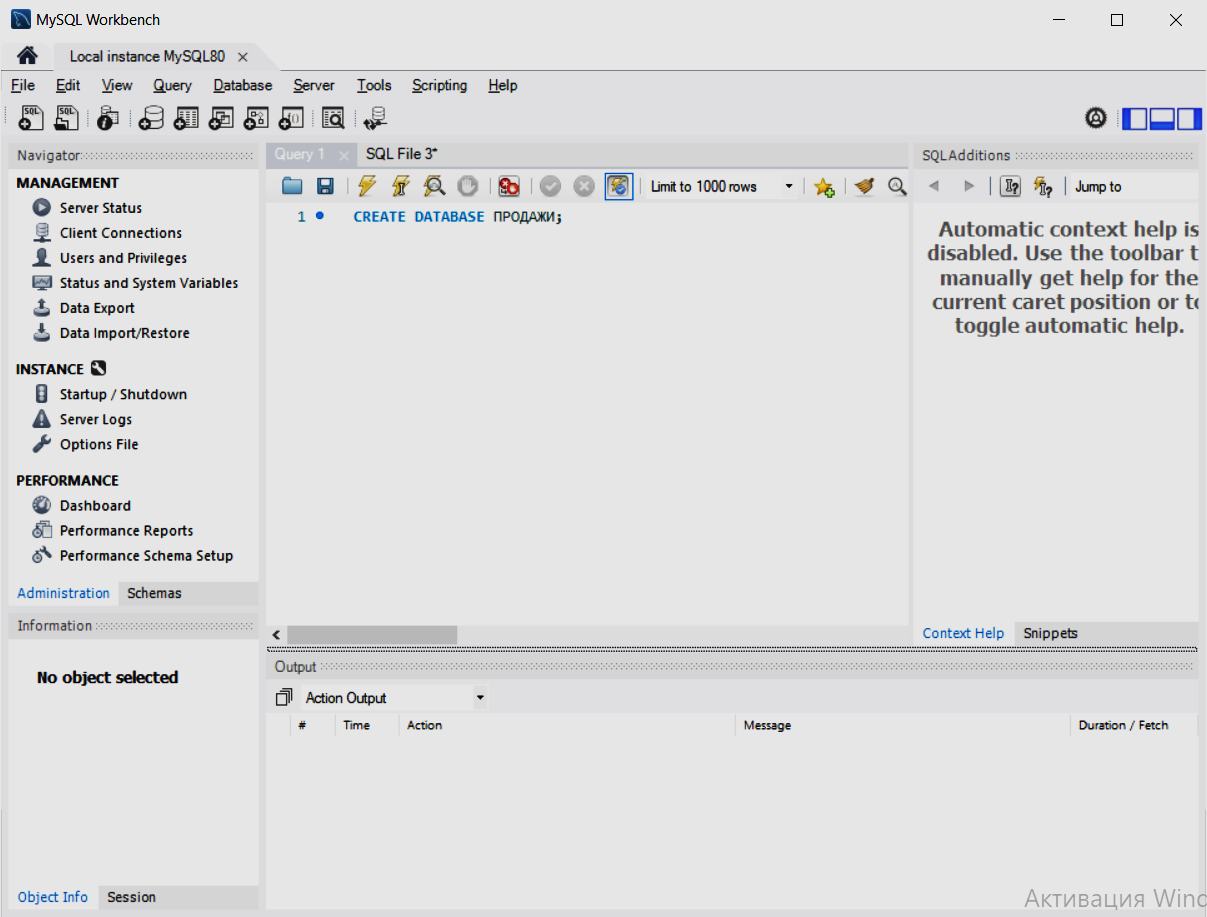
Скачать MySQL в редакции Community для Windows можно с сайта **https://dev.mysql.com/downloads/installer.**

При установке сервера MySQL также устанавливается консольный клиент **MySQL Workbench**, с помощью которого можно создавать, удалять, изменять базы данных и управлять ими.

При запуске MySQL Workbench на стартовой странице будет отражено подключение к локальному серверу, которое будет осуществляться от имени пользователя **root**. Следует нажать на прямоугольник.

Затем нужно ввести пароль, чтобы сохранить его можно поставить галочку у пункта «Save password in vault» и нажать ОК.

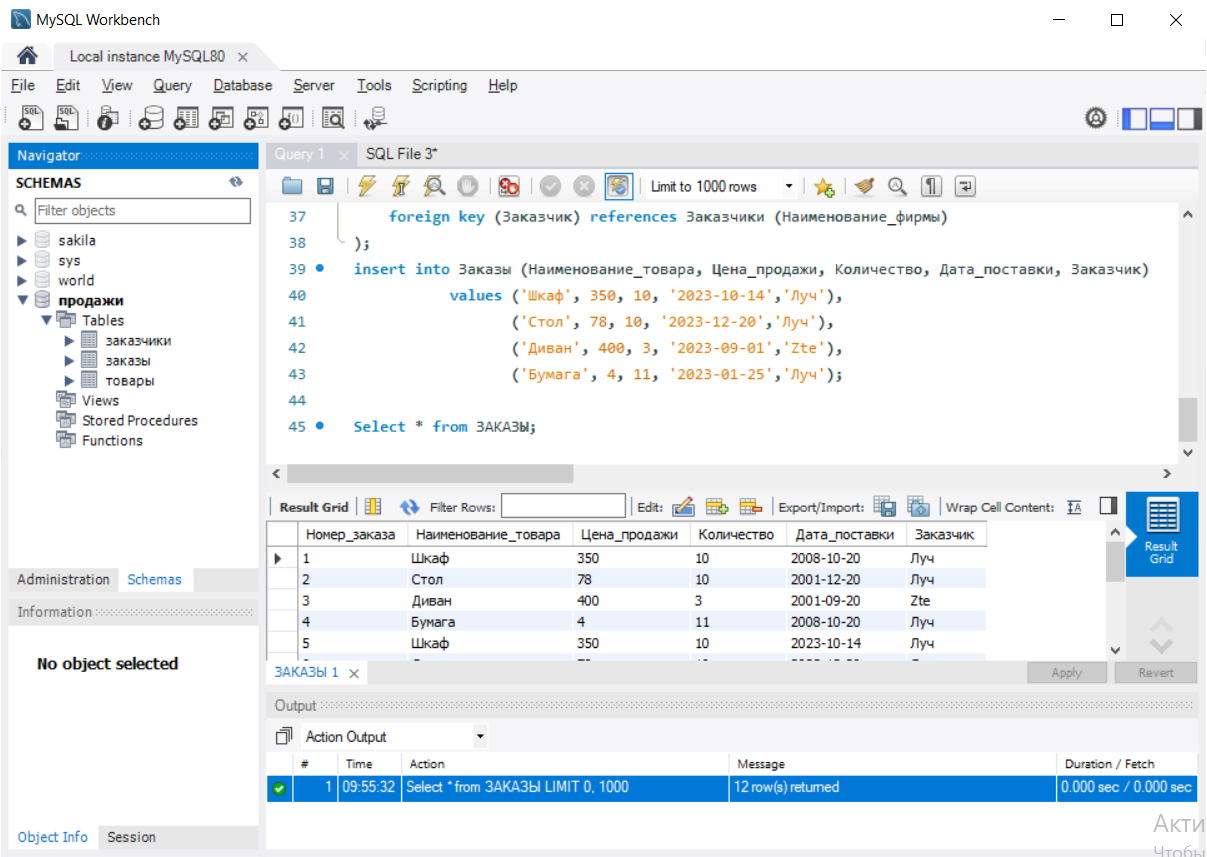
Произойдет подключение к серверу.



Чтобы скрыть ненужные окна можно использовать специальные значки в правой верхней части среды MySQL Workbench.

В окне навигатора можно перейти на вкладку **Schemas** и посмотреть на схему данных, где можно увидеть, какие объекты находятся в базе.

Для создания базы данных надо в верхней левой части экрана нажать на значок **SQL** с плюсом под пунктом меню **File**. В центральной части программы откроется окно для ввода скрипта SQL. Чтобы сценарий выполнился следует на панели инструментов нажать на значок молнии, который располагается над скриптом.

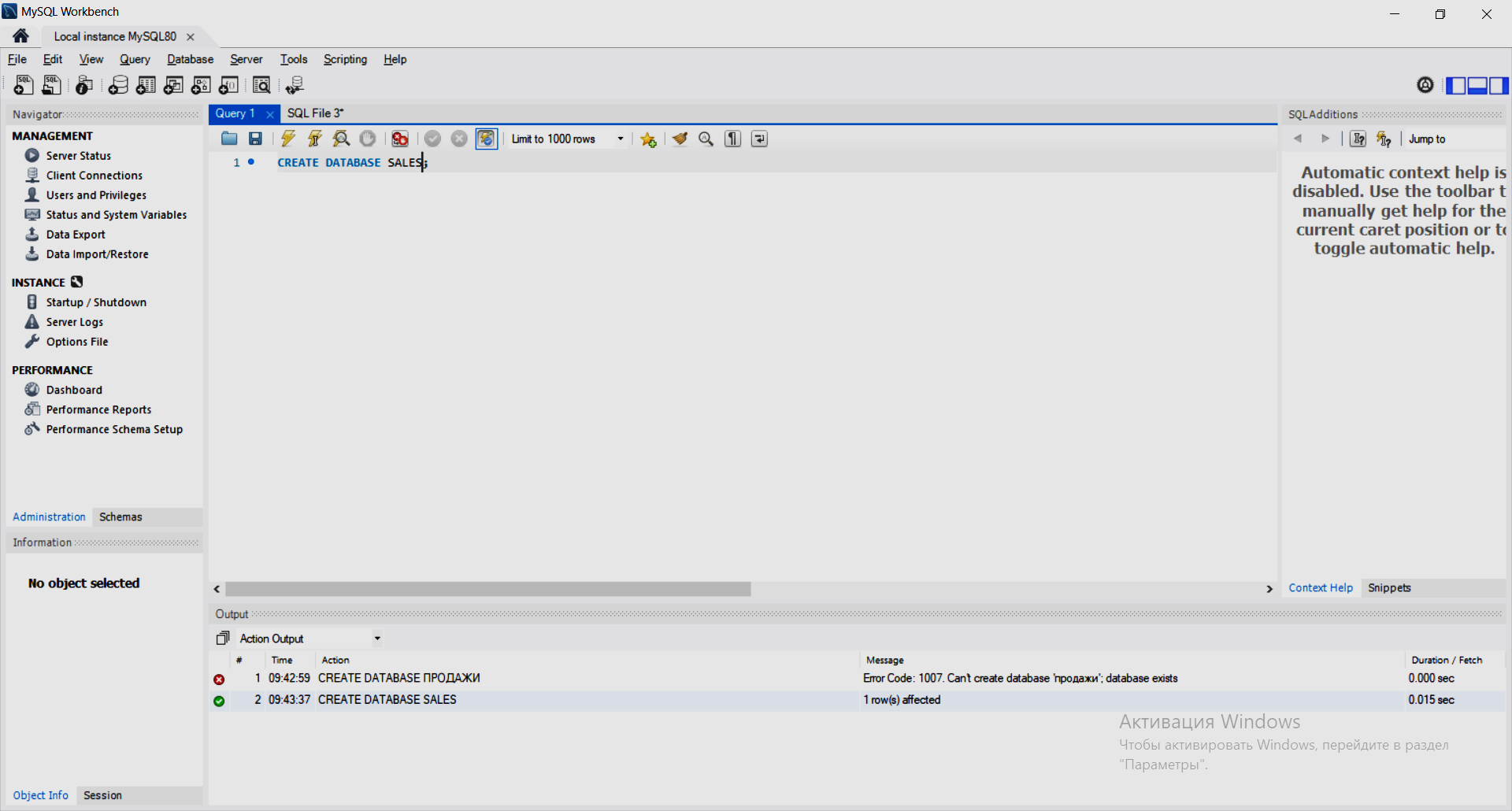


В окне представлен пример выполнения SELECT-запроса на выборку данных из таблицы **Заказы**.

Если нужно внести изменения в структуру таблиц, то после выполнения соответствующего сценария следует в контекстном меню имени базы выполнить команду **Refresh all**.

## Язык SQL

## Первый международный стандарт языка SQL был принят в 1989 г.

Для ввода команд языка SQL надо в верхней левой части экрана нажать на значок **SQL** с плюсом .

При использовании кодировки Windows-1251, перед выполнением SQL-инструкций необходимо настроить кодировку соединения при помощи операторов:

**SET** character\_set\_client='cp1251';

**SET** character\_set\_results='cp1251';

**SET** character\_set\_connection='cp1251';

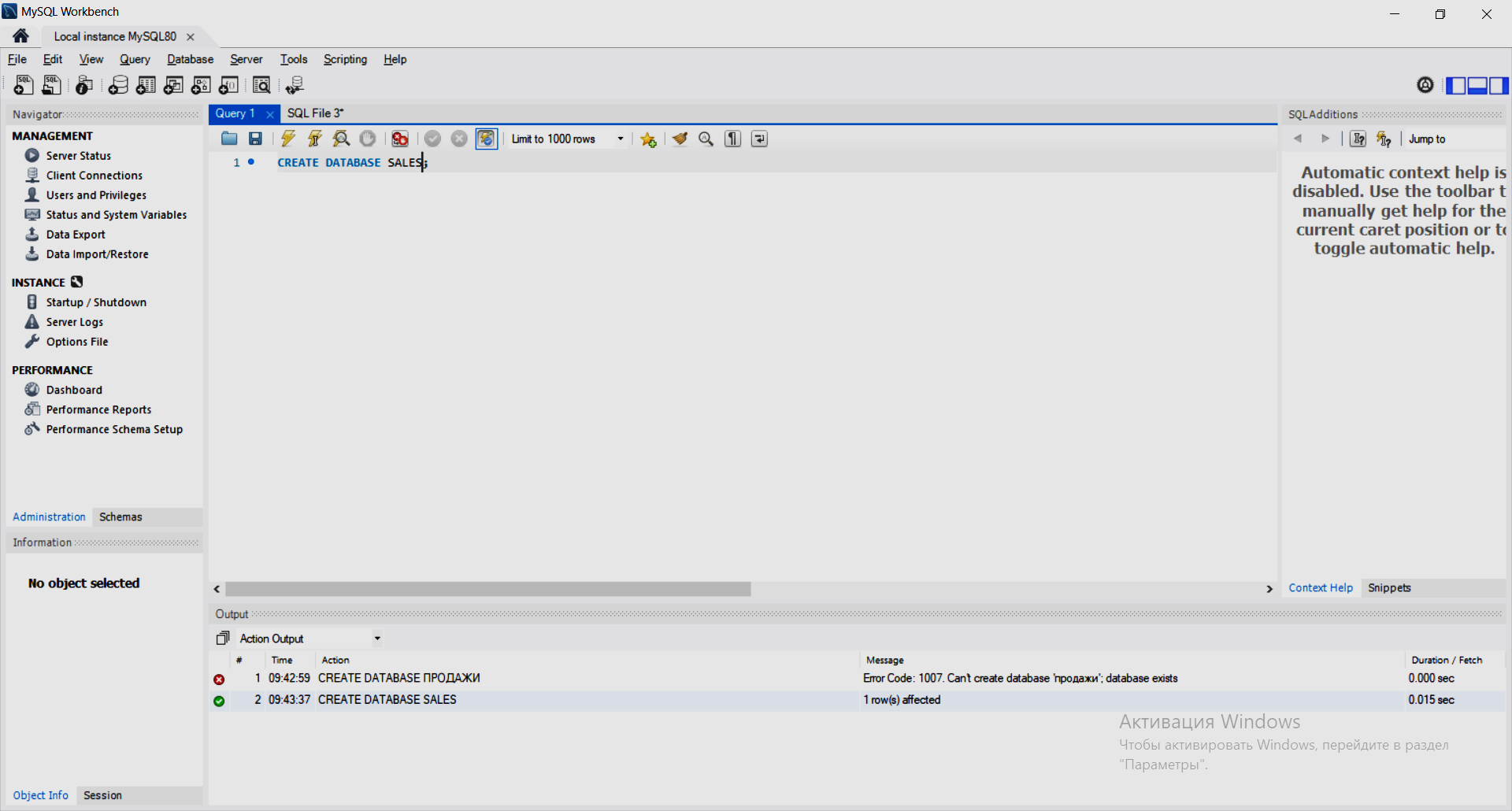
Эти три оператора эквивалентны одному: **SET** **NAMES** 'cp1251'

Переменная character\_set\_client устанавливает кодировку данных, отправляемых от клиента, переменная character\_set\_results устанавливает кодировку данных, отправляемых клиенту, переменная character\_set\_connection устанавливает кодировку, в которую преобразуется информация, пришедшая от клиента, перед выполнением запроса на сервере.

При использовании Юникода UTF-8 этот оператор выглядит следующим образом: **SET** **NAMES** 'utf8mb4'

С помощью команды SHOW можно просмотреть существующие базы данных, таблицы в базе данных и поля в таблице.

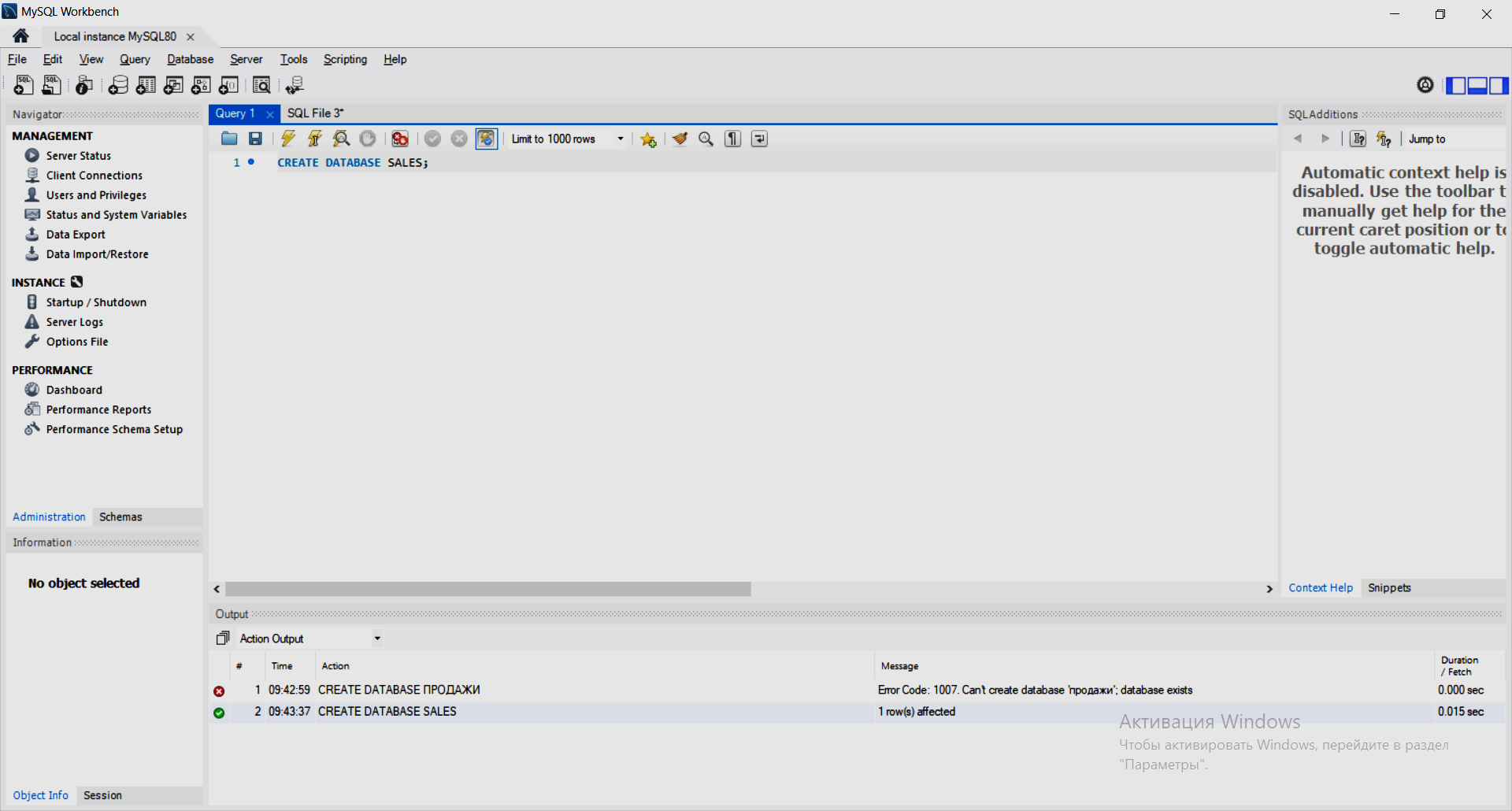
**SHOW databases;**

Для создания базы данных надо в верхней левой части экрана нажать на значок **SQL** с плюсом .

В центральной части программы откроется окно для ввода скрипта SQL.

Например, для создания базы данных ПРОДАЖИ следует написать следующий оператор:

**CREATE database ПРОДАЖИ;**

Чтобы скрипт выполнился на панели инструментов надо нажать на значок молнии .

Пример создания и заполнения таблицы **Товары**:

**USE ПРОДАЖИ;**

**#Создание и заполнение таблицы Товары**

**CREATE TABLE Товары**

**( Наименование varchar(20) primary key,**

**Цена real,**

**Количество int**

**);**

**INSERT into Товары (Наименование, Цена, Количество)**

**values ('Стол', 78, 10),**

**('Диван', 400, 3),**

**('Шкаф', 450, 10);**

Чтобы увидеть, как именно выглядит таблица **Товары**, надо использовать SQL-запрос, который выберет все записи из таблицы:

**select \* from Товары;**

Если нужно наращивать значение одного из полей, то можно использовать опцию **auto\_increment**,предназначенную для автоматического увеличения поля первичного ключа таблицы на единицу.

При создании таблиц используются **ограничения целостности**, приведенные в таблице.

**data type −** предотвращает появление значений, не соответствующих типу данных.

**not null −** не допускает появление в столбце значений null.

**primary key −** определяет первичный ключ.

**foreign key −** устанавливает связь между таблицей со столбцом, имеющим свойство **foreign key** и таблицей, имеющей столбец со свойством **primary key**.

**unique −** предотвращает появление пустых и повторяющихся значений.

**check −** не допускает появление значения, не удовлетворяющего логическому условию.

Для изменения строк таблицы используется оператор **UPDATE**. Например, содержимое столбца **Цена** увеличить на 1 для товара, наименование которого **Стол**, можно командой:

**UPDATE ТОВАРЫ set Цена = Цена+1 Where Наименование = 'Стол';**

Чтобы удалить одну или несколько строк используется оператор **DELETE**. В примере, приведенном ниже, удаляется строка из таблицы **Товары**, в которой в столбце **Наименование** записан текст **Стул**.

**DELETE from ТОВАРЫ Where Наименование = 'Стул';**

Для удаления таблицы служит оператор **DROP table if exists Товары;**

Показать все таблицы в базе можно с помощью операторов:

**use ПРОДАЖИ;  
 show tables;**

Показать все столбцы в таблице Товары:

**show columns from Товары;**

Чтобы вывести информацию о товарах, а также для каждого товара посчитать общую стоимость (произведение цены на количество) и вычисляемому столбцу дать имя **total**, следует написать следующий запрос:

**USE ПРОДАЖИ;**

**SELECT Наименование, Цена, Количество, Цена\*Количество**

**AS Общая\_стоимость FROM Товары;**

**Если нужно использовать пробел в псевдониме, то команда будет выглядеть следующим образом:**

**SELECT Наименование, Цена, Количество, Цена\*Количество**

**AS 'Общая стоимость' FROM Товары;**

**Пусть требуется определить размер скидки на товар в зависимости от его общей стоимости:**

**SELECT Наименование\_товара, Цена\_продажи, Количество, Заказчик, IF(Цена\_продажи\*Количество 500, 'скидка 50%',**

**IF(Цена\_продажи\*Количество 200, 'скидка 30%', 'скидка 10%'))**

**AS Скидка FROM Заказы;**

**Найти товары, наименования которых начинаются на букву «С» и цена в пределах от 30 до 100. Отсортировать результаты по цене в порядке убывания:**

**SELECT Наименование, Цена FROM Товары**

**WHERE Цена BETWEEN 30 and 100 and Наименование LIKE 'С%'**

**ORDER BY Цена DESC;**

**Соединение таблиц**

Оператор **внутреннего соединения** INNER JOIN выводит только те строки из таблиц, которые удовлетворяют записанному в запросе условию.

**Левое внешнее соединение** LEFT OUTER JOIN включает в результирующий набор строки, удовлетворяющие записанному в запросе условию и набор несоединенных строк таблицы, имя которой записано слева от ключевых слов LEFT OUTER JOIN.

Пример. На основании таблиц **Товары** и **Заказы** сформировать перечень товаров с ценой исходной и ценой продажи, при этом не выводить повторяющиеся строки:

**SELECT Distinct z.Наименование\_товара, t.Цена, z.Цена\_продажи**

**FROM Заказы z Inner Join Товары t**

**On z.Наименование\_товара = t.Наименование;**

В запросе использованы псевдонимы для имен таблиц.

**Подзапросы**

**Подзапрос** – это SELECT-запрос, который выполняется в рамках другого запроса.

Пусть требуется определить товары и даты их поставки для фирм, которые располагаются в Минске.

**SELECT z.Наименование\_товара, z.Дата\_поставки, zk.Наименование\_фирмы**

**FROM Заказы z Inner Join Заказчики zk**

**On z.Заказчик=zk.Наименование\_фирмы**

**Where Заказчик In (Select Наименование\_фирмы**

**FROM Заказчики Where Адрес LIKE 'Минск%');**

**Группировка**

Основное назначение **группировки** с помощью секции GROUP BY – разбиение множества строк на группы в соответствии со значениями в заданных столбцах, а также выполнение вычислений над группами строк с помощью наиболее часто используемых функций:

**− avg** (вычисление среднего значения);

**− count** (вычисление количества строк);

**− max** (вычисление максимального значения);

**− min** (вычисление минимального значения);

**− sum** (вычисление суммы значений).

При использовании секции **GROUP BY** в SELECT-списке допускается указывать только те столбцы, по которым осуществляется группировка.

Пример. Определить общую сумму и количество заказанных товаров для каждой фирмы.

**SELECT zk.Наименование\_фирмы,**

**sum(Цена\_продажи) Сумма,**

**count(\*) 'Количество заказанных товаров'**

**FROM Заказы z Inner Join Заказчики zk**

**On zk.Наименование\_фирмы = z.Заказчик**

**GROUP BY zk.Наименование\_фирмы;**

**Встроенные функции**

Для работы со ***строками*** в MySQL определен ряд **встроенных функций**:

CONCAT (объединяет строки),

LEFT, RIGHT, SUBSTRING (вырезание с начала, с конца или с некоторой позиции строки определенное количество символов),

REPLACE (замена одной подстроки на другую),

LOWER, UPPER (перевод строку в нижний или в верхний регистр),

REPEATE (повтор подстроки) и др.

Например, чтобы вывести информацию одновременно о заказчике и его адресе можно использовать запрос:

**SELECT CONCAT(Наименование\_фирмы, ' ', Адрес) As Фирма**

**FROM Заказчики;**

Для замены данных можно применить оператор:

**SELECT replace(Наименование\_фирмы, 'Луч', 'Марко') FROM Заказчики;**

Чтобы повторить текст служит команда:

**SELECT** **repeat**(**'xa'**, **6**);

Для работы с ***числовыми данными*** в MySQL имеются функции:

ROUND (определяет количество знаков после запятой),

CEILING, FLOOR (округление чисел в меньшую или большую сторону),

RAND (генерация случайного числа) и др.

К числовым функциям относятся функции возвращающие абсолютные значения, синусы и косинусы углов, квадратный корень числа и т. д.

Некоторые встроенные функции для работы с ***датами и временем***:

NOW (возвращает текущую дату и время в формате `YYYY-MM-DD hh:mm:ss`),

CURDATE (возвращает значение текущего года, месяца, дня),

DATE (значение даты в формате `YYYY-MM-DD`),

DATEDIFF (вычисляет разницу в днях между первым и вторым параметрами),

DATE\_ADD (добавляет одну дату к другой),

EXTRACT (извлекает часть даты) и др.

Вывести товары с ценой, превышающей 100, с двумя знаками после запятой:

**SELECT Наименование, round(Цена, 2) As 'Цена товара'**

**FROM Товары where Цена 100;**

Пусть надо найти заказы, доставка которых должна была осуществиться 5 дней тому назад:

**SELECT \* FROM Заказы**

**WHERE DATEDIFF(CURDATE(), Дата\_поставки) = 5;**

Определить заказы, доставка которых запланирована на десятый месяц года:

**SELECT \* FROM Заказы**

**WHERE EXTRACT(MONTH FROM Дата\_поставки) = 10;**

Найти товары и заказчиков, у которых дата поставки больше текущей:

**SELECT Наименование\_товара, Заказчик FROM Заказы**

**WHERE Дата\_поставки curdate();**

**Представления**

**Представление** (View) – это ***поименованный*** SELECT-запрос.

Представления дают возможность пользователям и приложениям получать данные без непосредственного доступа к таблицам.

Например, пусть разработано представление, в котором определяются наименования заказанных товаров, цены их продаж, заказчики и даты поставки:

**CREATE** **View** **Zakaz** **As**

**SELECT Наименование\_товара 'Товары',**

**Цена\_продажи,**

**Заказчик**,

**Дата\_поставки FROM Заказы;**

Тогда запрос, в котором выводится информация о товарах, заказанных конкретным заказчиком, можно записать кратко:

**SELECT \* from Zakaz** **where Заказчик = 'Луч';**

Представление обладает несколькими преимуществами.

Оно ограничивает доступ только теми данными, которые возвращает представление.

Приложения, обращающиеся к представлению, не могут увидеть структуру таблиц и другие данные, которые содержат таблицы, что повышает безопасность.

Представление также помогает упростить запросы. Например, разработчику приложений не потребуется создавать детализированные многотабличные соединения, вместо этого он может обратиться к представлению в базовом операторе SELECT.

При этом в MySQL допускается удаление таблицы, на которую ссылается представление, без возникновения ошибки. Когда пользователь или приложение попытается обратиться к такому представлению, MySQL подаст сигнал, который может серьезно повлиять на рабочую нагрузку.

**Хранимые функции и процедуры**

**Хранимая функция** – это объект БД, представляющий собой поименованный код T-SQL.

Чтобы добавить хранимую функцию в базу данных MySQL надо использовать оператор CREATE FUNCTION.

Хранимая функция может возвращать только **одно** значение.

Например, пусть хранимая функция **CenaZakazy** выводит текст, в котором определяется уровень цены товара.

**DELIMITER //**

**CREATE FUNCTION CenaZakazy( cena int )**

**RETURNS varchar(20)**

**DETERMINISTIC**

**BEGIN**

**DECLARE cost varchar(20);**

**Case**

**When cena < 50 Then set cost = 'цена < 50';**

**When cena between 50 and 100 Then set cost ='цена от 50 до 100';**

**else set cost = 'цена 100';**

**END CASE;**

**RETURN cost;**

**END; //**

**DELIMITER ;**

Первый оператор DELIMITER изменяет разделитель на двойной прямой слэш (//), а второй оператор DELIMITER меняет разделитель обратно на точку с запятой. Это позволяет передать на сервер все определение функции как **единый** оператор.

Характеристика DETERMINISTIC указывает, что функция будет возвращать одни и те же результаты для одних и тех же входных параметров при каждом вызове функции.

С использованием данной функции информацию об уровне цен товаров, заказанных конкретным заказчиком, можно найти с помощью запроса:

**SELECT Заказы.Наименование\_товара,**

**CenaZakazy(Заказы.Цена\_продажи)**

**FROM Заказы where Заказы.Заказчик = 'Луч';**

Пусть требуется подсчитать количество товаров, заказанных фирмой. Хранимая функция для подсчета может иметь следующий вид:

**DELIMITER //**

**CREATE FUNCTION CountZakazy(f varchar(20))**

**RETURNS int**

**DETERMINISTIC**

**BEGIN**

**SELECT count(Номер\_заказа)**

**FROM Заказы z Inner Join Заказчики zk**

**On zk.Наименование\_фирмы = z.Заказчик**

**where zk.Наименование\_фирмы = f INTO c;**

**RETURN c;**

**END; //**

**DELIMITER ;**

Определить количество заказов для всех фирм можно с помощью запроса:

**select Наименование\_фирмы,**

**CountZakazy(Наименование\_фирмы) from Заказчики;**

**Хранимая процедура** ̶ это набор инструкций SQL, которые могут храниться на сервере.

Вместо того, чтобы хранить часто используемый запрос, клиенты могут ссылаться на соответствующую хранимую процедуру.

Это обеспечивает лучшую производительность, поскольку данный запрос должен анализироваться только однажды и уменьшается трафик между сервером и клиентом. Также повышается уровень безопасности.

Процедура может вернуть **одно** значение, **несколько значений**, результирующий **набор**.

Пример простой процедуры подсчета количества всех заказов.

**DELIMITER //**

**CREATE PROCEDURE CountZakazy()**

**BEGIN**

**DECLARE K INT;**

**set k = (SELECT COUNT(Номер\_заказа) FROM Заказы);**

**SELECT k;**

**END//**

**DELIMITER ;**

Вызов процедуры: **call CountZakazy;**

В хранимой процедуре, которая осуществляет ввод данных в таблицу **Товары** используются ***три входных*** параметра:

**DELIMITER //**

**CREATE PROCEDURE TovaryInsert (t varchar(20), cn real, kl int)**

**BEGIN**

**INSERT into Товары (Наименование, Цена, Количество)**

**values (t, cn, kl);**

**END//**

**DELIMITER ;**

Вызов процедуры: **call TovaryInsert ('Тумба', 20, 30);**

В следующей хранимой процедуре используется один ***выходной*** параметр:

**delimiter //**

**CREATE PROCEDURE simpleproc (OUT param1 INT)**

**BEGIN**

**SELECT COUNT(\*) INTO param1 FROM Заказчики;**

**END;//**

**delimiter ;**

Вызов процедуры: **CALL simpleproc(@a);**

**SELECT @a;**

Символ **@** перед переменной означает *глобальную* переменную.

Хранимые процедуры полезны в различных ситуациях:

* Когда несколько клиентских приложений написаны на *разных языках* или работают на разных платформах, но необходимо выполнять одни и те же операции с базой данных.
* Когда *безопасность* имеет первостепенное значение. Банки, например, используют хранимые процедуры и функции для всех общих операций. Это обеспечивает согласованную и безопасную среду, а процедуры могут гарантировать, что каждая операция правильно регистрируется. При такой настройке приложения и пользователи не будут иметь прямого доступа к таблицам базы данных, а смогут выполнять только определенные хранимые процедуры.
* Хранимые процедуры могут обеспечить *повышенную производительность*, поскольку между сервером и клиентом требуется меньший обмен информации. Большая часть работы выполняется на стороне сервера и меньше выполняется на стороне клиента.
* Хранимые процедуры позволяют иметь *библиотеки* на сервере базы данных.

**Отличия** хранимых функций от хранимых процедур:

* Хранимая функция может возвращать только одно значение, в то время как хранимая процедура может вернуть несколько значений или целый результирующий набор.
* Хранимая функция поддерживает только входные параметры. Хранимая процедура поддерживает параметры IN, OUT и INOUT в любом сочетании.
* Хранимая функция может включать предложение RETURNS в определении перед телом функции. Это предложение указывает тип данного, возвращаемого функцией. Хранимые процедуры не имеют этого предложения.

**Индексы**

**Индекс** – это объект базы данных, позволяющий **ускорить поиск** информации.

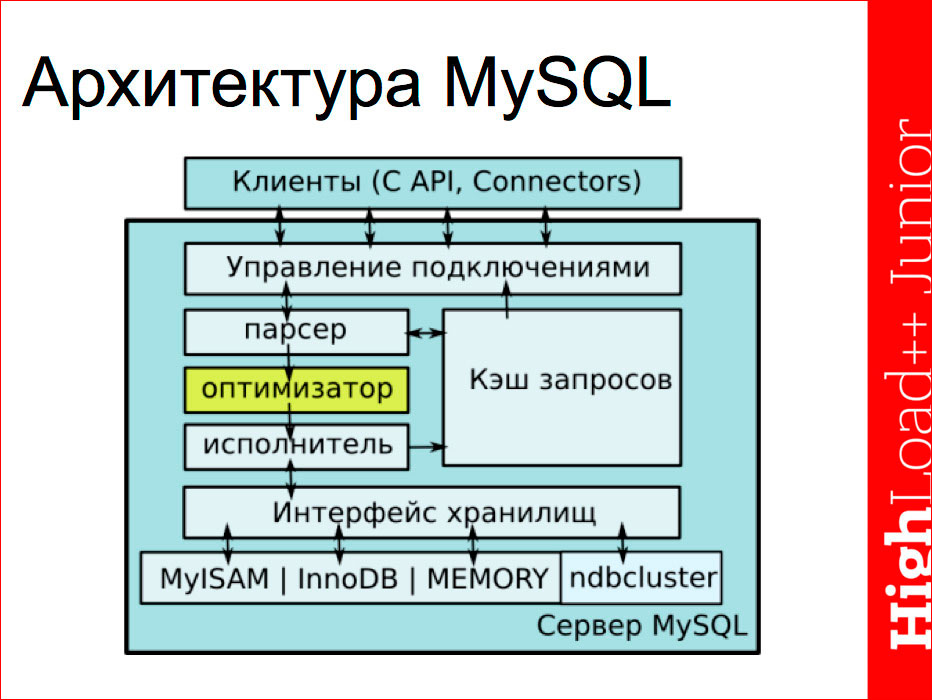
Без индекса поиск данных осуществляется по всей таблице, начиная с первой записи, пока не будут найдены соответствующие строки.

Если же таблица содержит индекс по рассматриваемым столбцам, то база данных может быстро определить позицию для поиска. Все индексы хранятся в памяти в виде B-деревьев.

Однако в случае, когда необходим доступ почти ко всем строкам, быстрее будет последовательное чтение, так как при этом не требуется операций поиска по диску. Так что иногда индексы бывают только помехой. Например, если копируется большой объем данных в таблицу, то лучше не иметь никаких индексов.

**Архитектура MySQL**

Схематически сервер можно представить следующим образом:



Первый блок — **клиенты**, которые обращаются к серверу

(через функции или API (Application programming interface) по протоколу TCP/IP либо UNIX Socket).

Они попадают на **блок управления** подключениями, где происходит авторизация клиента и запуск процесса исполнения. Каждый клиент работает в своем независимом потоке.

**Кэш запросов** представлен одним общим потоком для всех клиентов.

С блока управления подключениями запрос попадает на основной конвейер, который состоит из трех частей — парсер, оптимизатор и исполнитель. Эта часть превращает SQL-запрос в выборку данных:

* ***парсер*** проверяет синтаксис запроса, запрашивает у хранилищ наличие данных таблиц и полей, права доступа непосредственно к этим полям и проверяет наличие ответа в кэше запросов, после чего передает запрос оптимизатору;
* ***оптимизатор*** запрашивает в интерфейсе хранилищ статистику по индексам, на основании которой он строит план запроса, который передает исполнителю;
* ***исполнитель*** обращается по полученному плану непосредственно за данными в **хранилище**, передает ответ клиенту через интерфейс и, при необходимости, меняет в кэше обновления.

MyISAM - система хранения данных, подходит для использования в небольших интернет-проектах (до версии 5) – если много чтения;

InnoDB - транзакционная подсистема хранения, была создана для обработки большого количества краткосрочных транзакций (по умолчанию) – если много записи;

MEMORY - механизм хранения данных для быстрого доступа;

ndbcluster - система распределенных баз данных, лежащая в основе кластера MySQL (хранение в оперативной памяти).

**Виды индексов**

В MySql существует три вида индексов: PRIMARY, UNIQUE и INDEX, а слово KEY используется как синоним слова INDEX.

PRIMARY – уникальный индекс (ключ) с ограничением, все индексированные им поля not null. Таблица может иметь только один первичный индекс, но он может состоять из нескольких полей.

UNIQUE – ключ, задающий поля, которые могут иметь только уникальные значения.

INDEX – обычный индекс.

Для исследования индекса можно создать временную таблицу с именем, например, **EX**:

**CREATE DATABASE temporary;**

**USE temporary;**

**CREATE temporary table EX**

**( Ckey int auto\_increment,**

**Number int,**

**Words varchar(100),**

**PRIMARY KEY (Ckey)**

**);**

Заполнить эту таблицу можно с помощью процедуры и оператора ее вызова:

**DELIMITER //**

**CREATE PROCEDURE zapolnenie()**

**BEGIN**

**DECLARE i INT DEFAULT 0;**

**WHILE i < 1000 DO**

**INSERT EX(Number, Words)**

**values(floor(30000\*RAND()), repeat('строка ', 10));**

**set i = i + 1;**

**END WHILE;**

**END;//**

**DELIMITER ;**

Вызов процедуры: **CALL zapolnenie();**

Чтобы проверить, какие индексы есть в таблице, надо использовать команду:

**SHOW INDEXES from EX;**

***Время выполнения запроса*** для подсчета например количества строк в таблице определяется с помощью скрипта:

**set profiling = 1; #подключение профилирования (анализа производительности)**

**SELECT count(\*) from EX;**

**show profiles;**

Результат – время выполнения запроса **0.01576825** сек.

После создания индекса, например, по столбцу **Number**, время выполнения значительно уменьшится.

**CREATE index EXPLRE on EX(Number asc);**

Если в индексной таблице производились случайные вставки или удаления, индекс может стать **фрагментированным**.

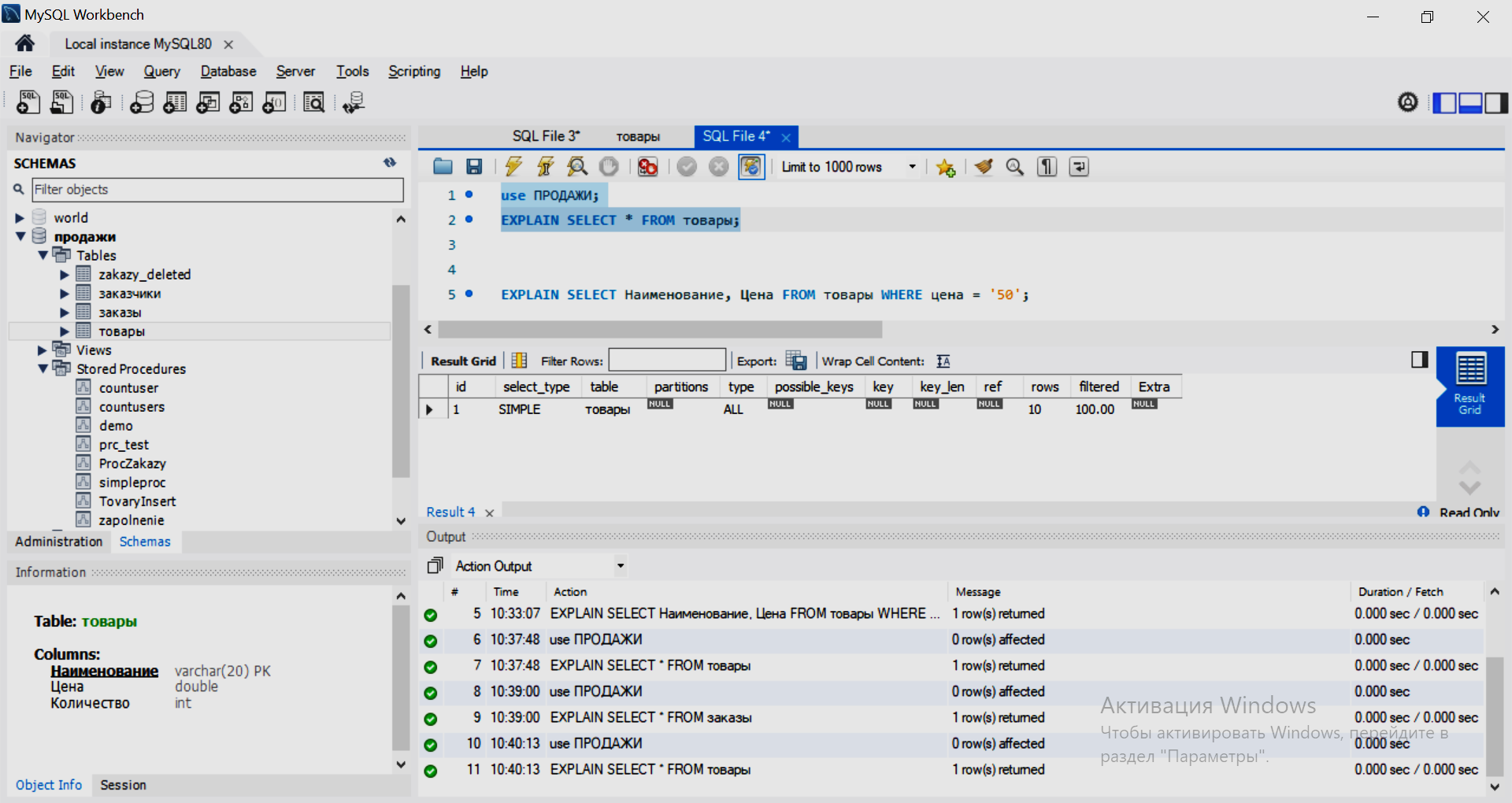
Уменьшить фрагментацию можно, если периодически использовать команду **mysqldump** для копирования дампа таблицы (файл с содержимым базы данных) в текстовый файл, записи на диск и повторного считывания таблицы из дампа.

**Анализ и оптимизация запросов**

Для анализа запросов обычно используется ключевое слово EXPLAIN

use ПРОДАЖИ;

EXPLAIN select \* FROM товары;



**id** - порядковый идентификатор каждого SELECT, находящегося внутри запроса (в случае использования вложенных подзапросов)

**select\_type**– тип SELECT запроса. Возможные значения:

SIMPLE – запрос содержит простую выборку без подзапросов и UNION'ов

PRIMARY – запрос является внешним запросов в JOIN

DERIVED – запрос SELECT является частью подзапроса внутри выражения FROM

SUBQUERY – первый SELECT в подзапросе

DEPENDENT SUBQUERY - первый SELECT, зависящий от внешнего подзапроса

UNCACHEABLE SUBQUERY – некешируемый подзапрос

UNION – SELECT является вторым или последующим в UNION

DEPENDENT UNION – SELECT является вторым или последующим запросом в UNIONи зависит от внешних запросов/li>

UNION RESULT – SELECT является результатом UNION'а

**table** – таблица, которой относится текущая строка

**type** – тип связывания таблиц. Это один из самых важных столбцов в результате, потому что по нему можно вычислить потерянные индексы или понять, как можно улучшить запрос.  
Возможные значения:

***system*** – таблица содержит только одну строку (системная таблица);

***const*** - таблица содержит не более одной соответствующей строки, которая будет считываться в начале запроса. Поскольку имеется только одна строка, оптимизатор в дальнейшем может расценивать значения этой строки в столбце как константы. Таблицы const являются очень быстрыми, поскольку они читаются только однажды;

***eq\_ref*** - для каждой комбинации строк из предыдущих таблиц будет cчитываться одна строка из этой таблицы. Это наилучший возможный тип связывания среди типов, отличных от const. Данный тип применяется, когда все части индекса используются для связывания, а сам индекс - UNIQUE или PRIMARY KEY;

***ref*** - из этой таблицы будут считываться все строки с совпадающими значениями индексов для каждой комбинации строк из предыдущих таблиц. Тип ref применяется, если для связывания используется только крайний левый префикс ключа, или если ключ не является UNIQUE или PRIMARY KEY (другими словами, если на основании значения ключа для связывания не может быть выбрана одна строка). Этот тип связывания хорошо работает, если используемый ключ соответствует только нескольким строкам;

***fulltext*** – объединение, использующее полнотекстовый (FULLTEXT) индекс таблиц;

***ref\_or\_null*** – то же самое, что и ref, только содержащее строки со значением NULL в полях;

***index\_merge*** – объединение, использующее список индексов для получения результата запроса;

***unique\_subquery*** – результат подзапроса в выражении IN возвращает одну строку, используемую в качестве первичного ключа;

***index\_subquery*** – то же самое, что и unique\_subquery, только в результате больше одной строки;

***range*** – в запросе происходит сравнение ключевого поля с диапазоном значений (используются операторы BETWEEN, IN, >, >=);

***index*** – в процессе выполнения запроса сканируется только дерево индексов;

***all*** – в процессе выполнения запроса сканируются все таблицы. Это наихудший тип объединения и обычно указывает на отсутствие надлежащих индексов в таблице;

**possible\_keys** – показаны возможные индексы, которые могут использоваться MySQL для поиска данных в таблице. На самом деле, значение этого столбца, очень часто помогает оптимизировать запросы. Если значение равно NULL, значит, никаких индексов не используется.

**key** – отображается текущий ключ, используемый MySQL в данный момент. В этом столбце может отображаться индекс, отсутствующий в possible\_keys. Оптимизатор запросов MySQL всегда пытается найти оптимальный ключ, который будет использоваться в запросе. При объединении нескольких таблиц, MySQL может использовать индексы, также не указанные в possible\_keys.

**key\_len** – содержит длину ключа, выбранного оптимизатором запросов MySQL. Если значение key равно NULL, то key\_len тоже NULL. По значению длины ключа можно определить, сколько частей составного ключа в действительности будет использовать MySQL.

**ref** – показаны поля или константы, которые используются совместно с ключом, указанным в столбце key.

**rows** – количество строк, которые анализируются MySQL в процессе запроса. Это еще один важный показатель, указывающий на необходимость оптимизации запросов, особенно тех, которые содержат JOIN и подзапросы.

**extra** – содержит дополнительную информацию о процессе выполнения запроса. Если значениями этого столбца являются ”Using temporary”, “Using filesort” и т.п, то это говорит о том, что это «проблемный» запрос, требующий оптимизации.

**Оптимизация производительности с помощью EXPLAIN**

Пример. Пусть имеется база данных, в которой нет ни индексов, ни первичных ключей.

**EXPLAIN** **SELECT count(\*) from EX;**

Запрос плохой, потому что в таблицах отсутствуют индексы.

Тип объединения равен ”ALL” (наихудший вариант).

Это значит, что MySQL не может найти ни одного ключа, который может участвовать в объединении, поэтому значение столбцов possible\_keys и key равно NULL. Хуже всего то, что в процессе запроса MySQL будет сканировать все записи во всех таблицах, об этом говорит значение столбцов rows. При выполнении запроса будут просмотрены 91.750.822.240 записей (7 × 110 × 122 × 326 × 2996), чтобы получить результат из 4 записей. Это действительно ужасно, и будет только хуже, когда количество записей в базе данных будет увеличиваться.

Надо добавить первичные ключи у всех таблиц и выполнить запрос еще раз.

Как правило, при создании индексов, обращают внимание на поля, по которым происходит объединение (JOIN), - это отличные кандидаты, для присвоения индексов, потому что MySQL всегда «просматривает» их при поиске связанных записей.

Теперь давайте выполним запрос еще раз после добавления индексов.

После добавления индексов, количество сканируемых записей снизилось до 4 (1 × 1 × 4 × 1 × 1).

Это говорит о том, что для каждой записи с ключом orderNumber из таблицы orderdetails MySQL сможет найти связанные записи во всех таблицах, используя индексы, а не сканируя все таблицы полностью.

В первой строке результата, который выводит EXPLAIN, вы можете видеть, что тип объединения равен ”const”, - это самым быстрый тип объединения таблиц, содержащих более одной записи. В данном случае MySQL будет использовать первичный ключ в качестве индекса.

**Курсоры**

**Курсор** является программной конструкцией, которая дает возможность пользователю обрабатывать строки результирующего набора запись за записью.

В MySql курсоры могут использоваться только в ***хранимых процедурах***.

Пример. Пусть требуется определить количество заказов конкретной фирмы и вывести результат в определенном виде: **Фирма …, количество заказов - …**

**DELIMITER //**

**CREATE PROCEDURE ProcCurcor(zk varchar(20))**

**BEGIN**

**declare done int default false;**

**declare c int default 0;**

**declare cur1 CURSOR for**

**SELECT Заказчик from Заказы where Заказчик = zk;**

**declare continue handler for not found set done = true;**

**OPEN cur1;**

**r\_loop: LOOP**

**FETCH cur1 into zk;**

**if done then leave r\_loop;**

**end if;**

**set c = c + 1; #определение количества заказов в цикле**

**END LOOP;**

**SELECT (concat('Фирма ', zk, ', ', 'количество заказов', ' - ', c));**

**CLOSE cur1;**

**END;//**

**DELIMITER ;**

Курсор объявляется в операторе DECLARE.

Обработчик **continue handler** устанавливает значение **done** в **true**, когда курсор не находит больше результатов.

Переменная **done** и оператор **If** используются для определения конца обрабатываемых данных.

Для выполнения курсора нужно выполнить команду:

**call ProcCurcor('Луч');**

**Транзакции**

**Транзакция** − это механизм базы данных, позволяющий таким образом объединять несколько операторов, изменяющих базу данных, чтобы при их выполнении они или все выполнились или все не выполнились.

Транзакция начинается со специального оператора START TRANSACTION. Чтобы закончить транзакцию, нужно либо зафиксировать изменения (COMMIT), либо откатить их (ROLLBACK).

Пример. Если закончился на складе товар с наименованием «Стол», то можно создать хранимую процедуру для удаления его из таблиц с помощью простой транзакции и выполнить процедуру с использованием оператора Call.

**DELIMITER //**

**CREATE PROCEDURE TovarDel(tv varchar(20))**

**BEGIN**

**DECLARE EXIT HANDLER FOR SQLEXCEPTION**

**BEGIN**

**ROLLBACK; #откат транзакции в случае ошибки**

**END;**

**START transaction;**

**DELETE from Заказы where Наименование\_товара = tv;**

**DELETE from Товары where Наименование = tv;**

**COMMIT;**

**END//**

**DELIMITER ;**

Вызов процедуры: **CALL TovarDel('Стол');**

**DECLARE EXIT HANDLER FOR SQLEXCEPTION** − обработчик, который означает, что в случае возникновении ошибки, производится откат предыдущей операции, выдается сообщение об ошибке и осуществляется выход из текущего блока кода.

В MySql не рекомендуется использовать хранимые процедуры для организации транзакций, так как в основном ошибки базы данных обрабатываются на стороне приложений, таких, например, как приложения на языке PHP. В примере транзакцию лучше записывать, начиная с оператора START и заканчивая COMMIT, обработку ошибок реализовать в приложении.

В MySQL транзакции поддерживаются только таблицами innoDB.

Не существует механизма вложенных транзакций.

Одно соединение с БД — одна транзакция.

У транзакций есть 4 уровня изолированности:

* **0 — Чтение неподтверждённых данных (грязное чтение)**
* (**Read Uncommitted**) — самый низкий уровень изолированности. При этом уровне возможно чтение незафиксированных изменений параллельных транзакций. Нет гарантии, что незафиксированная транзакция будет в любой момент откачена, поэтому такое чтение является потенциальным источником ошибок.
* **1 — Чтение подтверждённых данных**
* (**Read Committed**) — здесь возможно чтение данных только зафиксированных транзакций. В этом режиме строки, которые участвуют в выборке в рамках транзакции, для других параллельных транзакций не блокируются, из этого вытекает проблема: «Неповторяемое чтение» — это ситуация, когда в рамках транзакции происходит несколько выборок по одним и тем же критериям, и между этими выборками совершается параллельная транзакция, которая изменяет данные, участвующие в этих выборках. Так как параллельная транзакция изменила данные, результат при следующей выборке по тем же критериям в первой транзакции будет другой.  Вторая проблема — «Фантомное чтение».
* **2 — Повторяемое чтение**
* (**Repeatable Read, Snapshot**) —  на этом уровне изолированности возможно чтение данных только зафиксированных транзакций. Отсутствует проблема «Неповторяемого чтения», то есть строки, которые участвуют в выборке в рамках транзакции, блокируются и не могут быть изменены другими параллельными транзакциями. Но таблицы целиком не блокируются. Из-за этого остается проблема «фантомного чтения». Это означает, что параллельные транзакции могут вставлять строки в таблицу, в которой совершается выборка, поэтому два запроса могут дать разный результат в разное время при вставке данных параллельными транзакциями.
* **3 — Сериализуемый**
* (**Serializable**) — самый надежный уровень изолированности транзакций, но при этом самый медленный.

На этом уровне отсутствуют проблемы параллельных транзакций, но за это придется платить быстродействием системы.

По умолчанию в MySQL установлен уровень изоляции Repeatable Read.

Многое зависит не от того, насколько хорош уровень изолированности транзакций в БД, а от того, как спроектировано приложение. При грамотном программировании можно даже использовать самый низкий уровень изолированности транзакций.

Уровень изолированности устанавливает оператор SET TRANSACTION глобально либо только для текущего сеанса.

* SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL  
  { READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE }

Для выполнения этого оператора нужно иметь привилегию SUPER. Применение ключевого слова SESSION уста­навливает уровень изоляции по умолчанию всех будущих транзакций только для теку­щего сеанса.

**Триггеры**

**Триггер** – это особый вид хранимой процедуры, предназначенной для обработки событий UPDATE, DELETE и INSERT в базе данных.

Они могут быть выполнены либо до (Before-триггеры), либо после события (After-триггеры).

Предположим, что каждый раз при добавлении строк в таблицу **Товары** нужно определять общую стоимость добавленных товаров в глобальной переменной **@sum**. Предварительно нужно очистить эту переменную с помощью оператора:

**set @sum = 0;**

Триггер для подсчета суммы:

**delimiter //**

**CREATE TRIGGER Insert\_sum After INSERT**

**ON Товары FOR EACH ROW**

**begin**

**SET @sum = @sum + NEW.Цена;**

**end; //**

**delimiter ;**

Если выполнить оператор добавления товаров

**INSERT into Товары (Наименование, Цена, Количество)**

**values ('Зеркало', 60, 10),**

**('Ковер', 260, 20);**

то можно вывести полученную сумму, использовав команду:

**SELECT @sum;**

Пусть требуется при удалении строки из таблицы **Заказы** часть полей удаленной строки сохранить в таблице **Zakazy\_Deleted**.

Для создания таблицы **Zakazy\_Deleted** служит скрипт:

**CREATE TABLE Zakazy\_Deleted**

**( Наименование\_удал varchar(20),**

**Цена\_удал real,**

**Заказчик\_удал varchar(20)**

**);**

Before-триггер, который срабатывает при удалении строки из таблицы Заказы:

**delimiter //**

**CREATE TRIGGER Trig\_Del After Delete**

**ON Заказы FOR EACH ROW**

**begin**

**INSERT into Zakazy\_Deleted (Наименование\_удал, Цена\_удал, Заказчик\_удал)**

**values (Old.Наименование\_товара, Old.Цена\_продажи, Old.Заказчик);**

**end; //**

**deli miter ;**

Здесь **Old** означает, что берутся старые значения полей. При изменении полей для новых значений определено ключевое слово **New**.

Для проверки работы триггера надо выполнить команды:

**DELETE from Заказы Where Наименование\_товара = 'Диван';**

**SELECT \* from Zakazy\_Deleted;**

**Администрирование базы MySql**

Чтобы создать ***резервную копию*** базы данных надо в приложении Workbench перейти на вкладку **Administration** и выполнить команду **Data Export**.

На вкладке **Object Selection** выбрать объекты для копирования, отметить пункт **Export to Self-Contained File** и определить путь для размещения резервной копии.

Нажать кнопку **Start Export**. На вкладке **Export Progress** отображается результат.

Для ***восстановления*** базы данных на вкладке **Administration** надо выполнить команду **Data Import**, отметить пункт **Import from Self-Contained File**, найти нужные файлы и восстановить их нажав кнопку **Start Import**.

**Создание учетных записей**

Чтобы создать ***нового пользователя*** с помощью графического интерфейса нужно в приложении Workbench на панели **Administration** выполнить команду **Users and Privileges**, в появившемся окне нажать кнопку **Add Account**.

На вкладке **Login** указать имя пользователя, пароль и т. д. На вкладке **Administrative Roles** определить роли и разрешения текущего пользователя.

После того, как все настройки завершены, нажать кнопку **Apply**.

Для реализации вышеуказанных функций также можно использовать операторы языка SQL.

Например, для ***определения имени пользователя и пароля*** служит оператор, общий вид которого:

**CREATE USER 'имя пользователя'@'localhost' IDENTIFIED BY 'пароль';**

Например, создание пользователя с именем '**user1**' с паролем '**password1**', которому разрешено подключаться к серверу из подсети '**localhost**':

**CREATE USER** **'user1'@'localhost'** **IDENTIFIED BY** **'password1';**

Для ***смены пароля пользователя*** можно воспользоваться командой:

**ALTER USER 'имя пользователя'@'localhost' IDENTIFIED BY 'New\_Password';**

***Удалить пользователя*** можно, используя команду:

**DROP USER 'имя пользователя'@'localhost';**

**Назначение разрешений учетным записям**

**Разрешения** (привилегии), предоставленные учетной записи MySQL, определяют, какие операции может выполнять учетная запись.

Привилегии MySQL различаются в зависимости от контекста, в котором они применяются, и на разных уровнях работы:

* *Административные* привилегии позволяют пользователям управлять работой сервера MySQL. Эти привилегии являются глобальными, поскольку они не относятся к конкретной базе данных.
* Разрешения *базы данных* применяются к базе данных и ко всем объектам в ней. Эти привилегии могут быть предоставлены для конкретных баз данных или глобально, чтобы они применялись ко всем базам данных.
* Права *доступа к объектам* базы данных, таким как таблицы, индексы, представления и [хранимые процедуры/функции](https://docs-python.ru/packages/klient-bd-mysql/khranimye-protsedury-funktsii-bd-mysql/), могут быть предоставлены для определенных объектов в базе данных (например, для всех таблиц в базе данных).

Чтобы назначить вновь созданному пользователю ***неограниченные права доступа*** к базе данных, надо выполнить команду:

**GRANT ALL PRIVILEGES ON** **\* . \*** **TO** **'имя пользователя'@'localhost';**

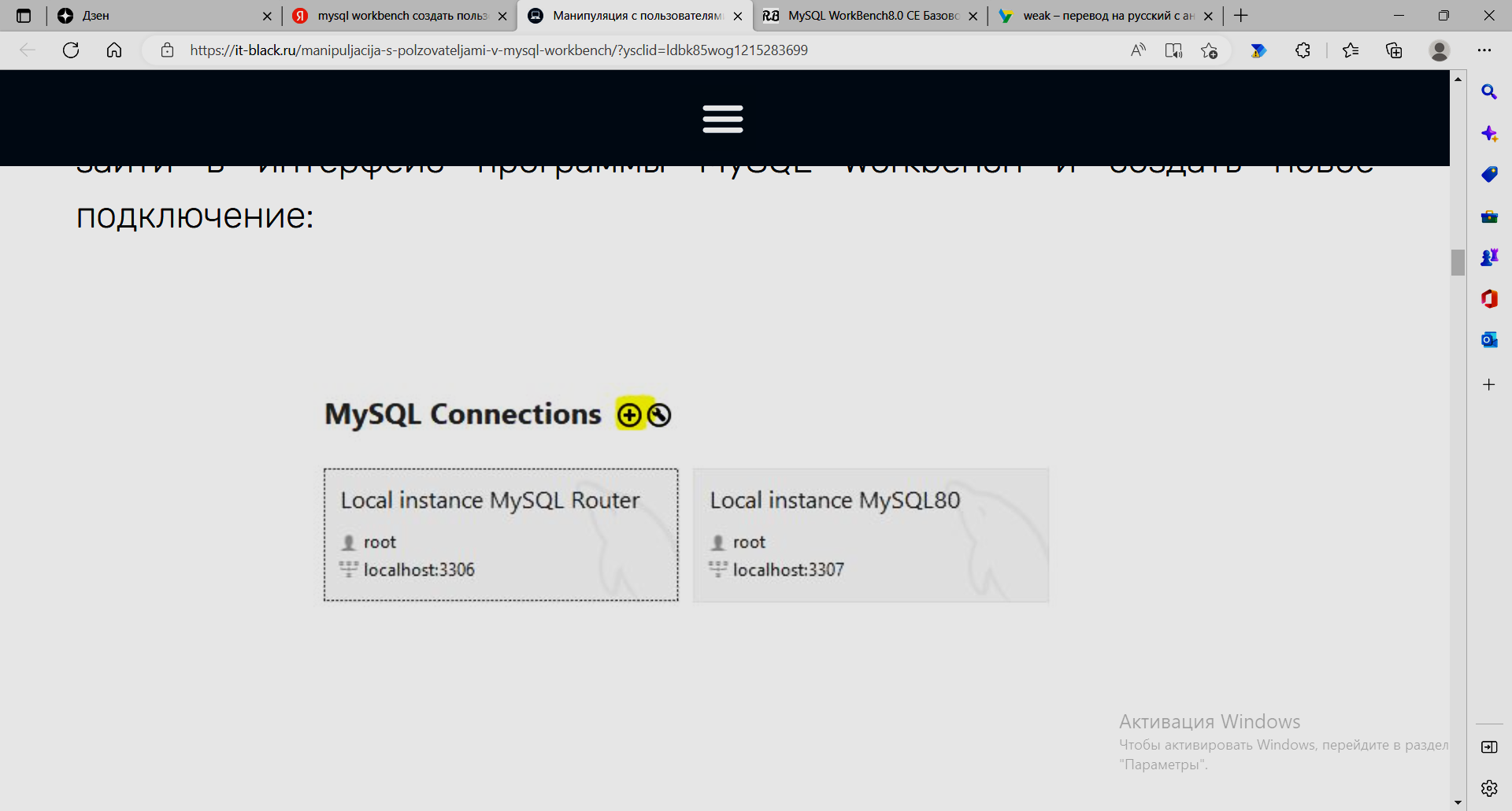
Чтобы изменения вошли в силу нужно запустить команду обновления:

**FLUSH PRIVILEGES;**

Новый пользователь получает те же права доступа в базе данных, что и пользователь **root**.

Все пользователи хранятся в базе данных в таблице **user**. Проверить наличие нового пользователя можно с помощью команды:

**SELECT user, host from mysql.user;** // вывод таблицы **user**

Чтобы войти в базу **под новым именем** необходимо на начальной странице MySQL Workbench создать новое подключение нажав на значок плюса.

В открывшемся окне ввести имя нового пользователя в поля **Connection Name** и **Username**.

После создания нового подключения нажать на него правой кнопкой мыши и в контекстном меню выполнить команду **Start Command Line Client**.

Для безопасности лучше предоставлять ***разные разрешения*** для различных пользователей с помощью команды:

**GRANT [тип прав] ON [имя базы данных].[имя таблицы] TO ‘имя пользователя’@'localhost’;**

Эти разрешения [**тип прав**] могут быть любой комбинацией SELECT, INSERT, UPDATE, DELETE, INDEX, CREATE, ALTER, DROP, GRANT OPTION или ALL.

Например, если надо предоставить привилегии SELECT, INSERT, UPDATE и DELETE в таблице **Заказы** пользователю с именем **user1**, то следует выполнить команду:

**GRANT SELECT, INSERT, UPDATE, DELETE ON ПРОДАЖИ.Заказы** **TO 'user1'@'localhost';**

Для отмены разрешений служит оператор:

**REVOKE [тип прав] ON [имя базы данных].[имя таблицы] FROM ‘имя пользователя’@'localhost’;**

### Предоставление ролей учетной записи пользователя

**Роль** – это именованный набор привилегий.

Подобно учетным записям пользователей, роли могут иметь привилегии, предоставленные им и отозванные у них.

Для создания роли необходимо использовать оператор **CREATE ROLE.**

Например,

**CREATE ROLE 'a\_developer', 'u\_read', 'u\_write';**

Для назначения привилегий *ролям*, необходимо выполнить инструкцию GRANT, используя тот же синтаксис, что и для назначения привилегий учетным записям пользователей:

**GRANT ALL ON Заказы TO 'a\_developer';**

**GRANT SELECT ON Заказы TO 'u\_read';**

**GRANT INSERT, UPDATE, DELETE ON Заказы TO ' u\_write';**

Синтаксис **GRANT** без предложения **ON** предоставляет **роли** *учетным записям*, а не отдельные привилегии.

Предположим, что требуется одна учетная запись *разработчика*, две учетные записи *пользователей*, которым требуется доступ *только для чтения*, и одна учетная запись *пользователя*, которой требуется доступ для *чтения/записи*:

**CREATE USER 'dev1'@'localhost' IDENTIFIED BY 'dev1pass';**

**CREATE USER 'read\_user1'@'localhost' IDENTIFIED BY 'read\_user1pass';**

**CREATE USER 'read\_user2'@'localhost' IDENTIFIED BY 'read\_user2pass';**

**CREATE USER 'rw\_user3'@'localhost' IDENTIFIED BY 'rw\_user3pass';**

Чтобы назначить каждой учетной записи пользователя необходимые привилегии, можно использовать альтернативный синтаксис оператора GRANT, который позволяет предоставлять роли, а не привилегии:

**GRANT 'a\_developer' TO 'dev1'@'localhost';**

**GRANT ' u\_read' TO 'read\_user1'@'localhost', 'read\_user2'@'localhost';**

**GRANT ' u\_read', 'app\_write' TO 'rw\_user3'@'localhost';**

Оператор GRANT для учетной записи **rw\_user3** предоставляет роли **'u\_read'** и **'u\_write'**, которые в совокупности обеспечивают необходимые права для чтения и изменения записей.

**Активация созданной роли MySQL**

Созданная роль изначально заблокирована и не может использоваться. Чтобы указать, какие роли должны становиться активными при подключении пользователя к серверу, необходимо вызвать инструкцию SET DEFAULT ROLE. Эта инструкция устанавливает по умолчанию все назначенные роли для каждой созданной ранее учетной записи:

**SET DEFAULT ROLE ALL TO 'dev1'@'localhost', 'read\_user1'@'localhost',**

**'read\_user2'@'localhost', 'rw\_user3'@'localhost';**

Теперь, если подключается пользователь **rw\_user3**, то начальное значение CURRENT\_ROLE() отражает новые назначения ролей по умолчанию:

**SELECT CURRENT\_ROLE();**

+--------------------------------+

**| CURRENT\_ROLE() |**

+--------------------------------+

**| `u\_read`@`%`,`u\_write`@`%` |**

+--------------------------------+

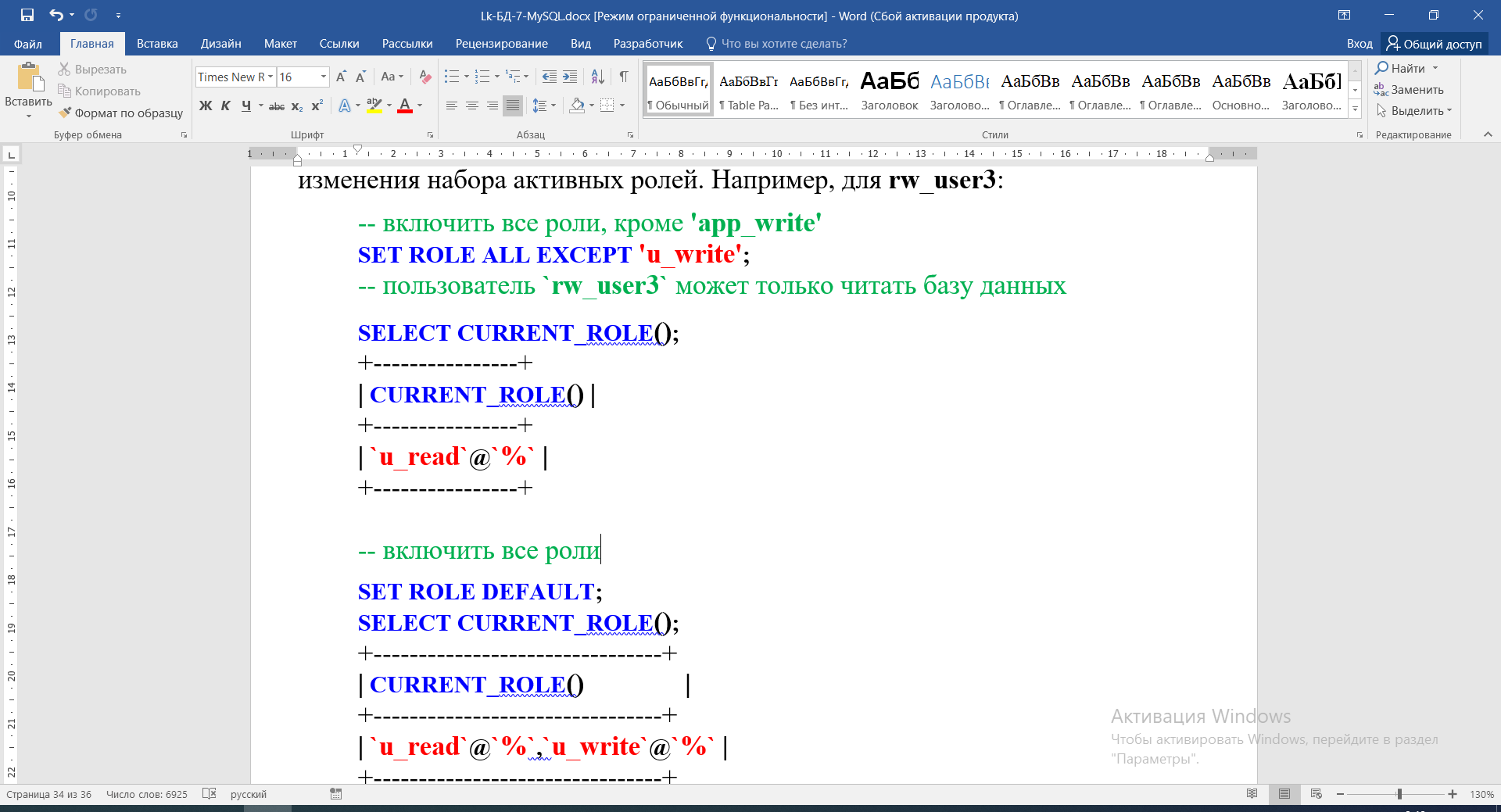
Чтобы все явно назначенные и обязательные роли автоматически активировались при подключении пользователей к серверу, необходимо включить системную переменную **active\_all\_roles\_on\_login**. По умолчанию автоматическая активация роли отключена.

В рамках сеанса, пользователь может выполнить команду SET ROLE для изменения набора активных ролей. Например, для **rw\_user3**:

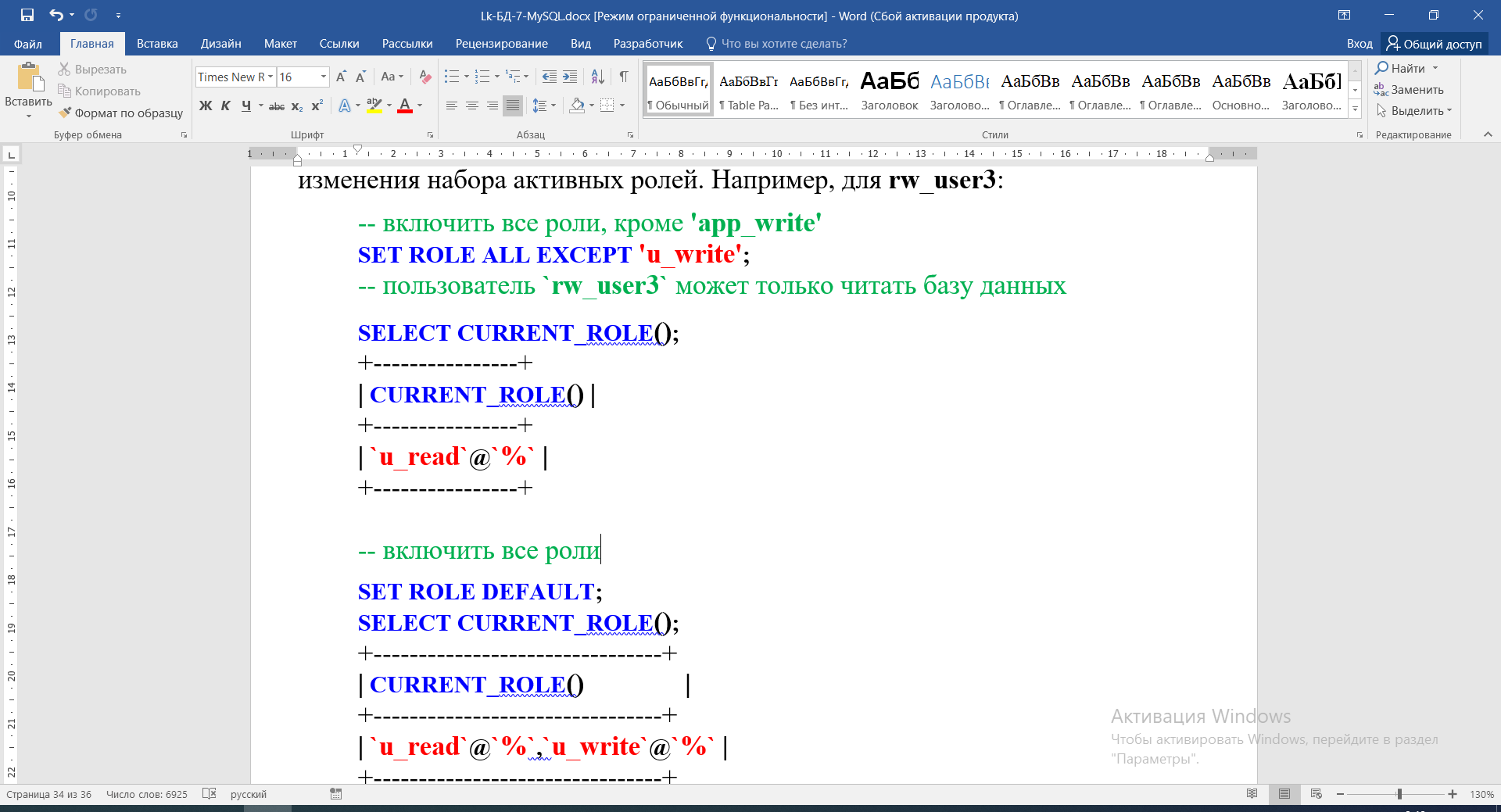
-- включить все роли, кроме **'app\_write'**

**SET ROLE ALL EXCEPT 'u\_write';**

-- пользователь **`rw\_user3`** может только читать базу данных



**SELECT CURRENT\_ROLE();**

-- включить все роли

**SET ROLE DEFAULT;**

**SELECT CURRENT\_ROLE();**

**Отзыв ролей или ролевых привилегий**

Точно так же, как роли могут быть предоставлены учетной записи, их можно отозвать из учетной записи:

**REVOKE role FROM user;**

REVOKE также можно применить к роли, чтобы изменить предоставленные ей привилегии. Это влияет не только на саму роль, но и на любую учетную запись, которой предоставлена ​​эта роль.

Предположим, необходимо временно сделать всех пользователей приложения доступными только для чтения. Чтобы отозвать привилегии у роли **u\_write** на изменение данных, используется инструкция REVOKE:

**REVOKE INSERT, UPDATE, DELETE ON Заказы FROM 'u\_write';**

По сути, пользователь **rw\_user3** стал пользователем только для чтения. Это также происходит для любых других учетных записей, которым предоставлена ​​роль **u\_write**. Хорошая иллюстрация, как использование ролей делает ненужным изменение привилегий для отдельных учетных записей.

Чтобы **восстановить** права измененной роли, надо просто повторно предоставить их:

**GRANT INSERT, UPDATE, DELETE ON Заказы TO 'u\_write';**

Для **удаления** роли необходимо использовать инструкцию DROP ROLE:

**DROP ROLE 'u\_read', 'u\_write';**

Удаление роли аннулирует ее для каждой учетной записи, которой она была назначена.

**Предоставление привилегии на функции/процедуры**

При работе с функциями и процедурами можно предоставить пользователям возможность исполнения этих функций и процедур в MySQL.

Синтаксис предоставления **привилегий** EXECUTE для функции / процедуры:

**GRANT EXECUTE ON [ PROCEDURE | FUNCTION ] object TO user;**

**object** – имя объекта базы данных, для которого предоставляются привилегии (имя функции или процедуры).

**user** – имя пользователя, которому будут предоставлены привилегии EXECUTE.

Например, если есть функция **GetBestPrice** и надо предоставить EXECUTE-доступ к ней пользователю с именем **user3**, то следует использовать оператор GRANT:

**GRANT EXECUTE ON FUNCTION GetBestPrice TO 'user4'@'localhost';**

Если надо предоставить всем пользователям возможность выполнить эту функцию, то используется оператор:

**GRANT EXECUTE ON FUNCTION GetBestPrice TO '\*'@'localhost';**

Аналогично для процедуры.

Для отмены привилегии на функцию или процедуру используется оператор:

**REVOKE EXECUTE ON [ PROCEDURE | FUNCTION ] object FROM user;**

**Использование процедуры для подсчета**

**количества пользователей**

Количество пользователей может быть возвращено как значение из процедуры.

**DELIMITER //**

**CREATE PROCEDURE countusers()**

**BEGIN**

**DECLARE usercount int;**

**SELECT count(\*) INTO usercount FROM mysql.user;**

**SELECT usercount;**

**END //**

**DELIMITER ;**

Вызов процедуры: **CALL countusers;**