МИНИСТЕРСТВО НАУКИ и высшего образования РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования «МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ»

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

филиал «РКТ» МАИ в г. Химки Московской области

**Специальность 09.02.03 «Программирование в компьютерных системах»**

**ОТЧЕТ ПО ПРОИЗВОДСТВЕННОЙ ПРАКТИКЕ**

**ПМ.02 «Разработка и администрирование баз данных**»

**Студент**

**Группы МП-32-17\_\_Гурушкин Виктор\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / (\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)**

**Руководитель**

**практики от организации \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / (\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)**

**Руководитель**

**практики от филиала Шумаев А.Ю. / ( \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ )**

**2020г.**

**ПРОГРАММА ПРОИЗВОДСТВЕННОЙ ПРАКТИКИ**

По специальности 09.02.03 «Программирование в компьютерных системах»

|  |  |  |
| --- | --- | --- |
| Дата | Наименование выполняемых работ | Количество дней практики |
| 11.06.20 | Ознакомление с программой практики. Прохождение инструктажа по технике безопасности и охране труда, изучение внутреннего распорядка организации и правил работы. Разработка задания на производственную практику. | 4 |
| 12.06.20 | Описание структуры сети, в которой может функционировать разработанная база данных. | 4 |
| 13.06.20 | Описание сетевого оборудования необходимого для работы с базой данных. | 4 |
| 15.06.20 | Описание программного обеспечения необходимого для работы с базой данных по сети. | 4 |
| 16.06.20-18.06.20 | Разработка клиентских интерфейсов для клиент-серверных приложений. | 16 |
| 20.06.20 | Создание инфокоммуникационной системы.  Выполнение сетевых настроек для взаимодействия с СУБД. | 4 |
| 22.06.20-24.06.20 | Создание концептуальной модели данных.  Создание логической модели данных.  Создание физической модели данных.  Описание механизмов обеспечения целостности базы данных. | 16 |
| 25.06.20-01.07.20 | Описание СУБД представленной базы данных и ее возможностей. Внесение различных данных в базу данных.  Структурирование запросов базы данных.  Определение методов создания хранимых процедур и триггеров. Создание хранимых процедур и триггеров базы данных. | 50 |
| 02.07.20-03.07.20 | Определение способов управления правами пользователей.  Описание распределения прав пользователей и управления ими в базе данных. | 12 |
| 03.07.20-04.07.20 | Определение методов создания и синхронизации реплик базы данных. Описание существующих механизмов репликации в базе данных. Определение методов создания резервных копий базы данных. Описание существующих механизмов резервного копирования в базе данных. | 12 |
| 06.07.20 | Составление отчёта по практике | 6 |
| 07.07.20 | Подготовка к защите отчёта по практике. | 6 |
| 08.07.20 | Итоговая аттестация по производственной практике - зачёт | 6 |

Руководитель практики от филиала «РКТ» МАИ преподаватель

Шумаев А.Ю.

Дата \_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(подпись)*

**АТТЕСТАЦИОННЫЙ ЛИСТ ПО ПРОИЗВОДСТВЕННОЙ ПРАКТИКЕ**

студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*ФИО*

обучающийся на 3-ем курсе по специальности СПО 09.02.03 «Программирование в компьютерных системах»

успешно прошел производственную практику по профессиональному модулю   
**ПМ.02 «Разработка и администрирование баз данных»**

в объеме 144 часа с 11.06.2020г. по 08.07.2020г.

в организации филиала «РКТ» МАИ

Виды и качество выполнения работ

|  |  |  |
| --- | --- | --- |
| Виды работ, выполненных обучающимся(ейся) во время практики | Объем работ | Качество выполнения работ в соответствии с технологией и (или) требованиями организации, в которой проходила практика |
| Ознакомление с программой практики. Прохождение инструктажа по технике безопасности и охране труда, изучение внутреннего распорядка организации и правил работы. Разработка задания на производственную практику. | 4 |  |
| Описание структуры сети, в которой может функционировать разработанная база данных. | 4 |  |
| Описание сетевого оборудования необходимого для работы с базой данных. | 4 |  |
| Описание программного обеспечения необходимого для работы с базой данных по сети. | 4 |  |
| Разработка клиентских интерфейсов для клиент-серверных приложений. | 16 |  |
| Создание инфокоммуникационной системы.  Выполнение сетевых настроек для взаимодействия с СУБД. | 4 |  |
| Создание концептуальной модели данных.  Создание логической модели данных.  Создание физической модели данных.  Описание механизмов обеспечения целостности базы данных. | 16 |  |
| Описание СУБД представленной базы данных и ее возможностей. Внесение различных данных в базу данных.  Структурирование запросов базы данных.  Определение методов создания хранимых процедур и триггеров. Создание хранимых процедур и триггеров базы данных. | 50 |  |
| Определение способов управления правами пользователей.  Описание распределения прав пользователей и управления ими в базе данных. | 12 |  |
| Определение методов создания и синхронизации реплик базы данных. Описание существующих механизмов репликации в базе данных. Определение методов создания резервных копий базы данных. Описание существующих механизмов резервного копирования в базе данных. | 12 |  |
| Составление отчёта по практике | 6 |  |
| Подготовка к защите отчёта по практике. | 6 |  |
| Итоговая аттестация по производственной практике - зачёт | 6 |  |

Руководитель практики от филиала «РКТ» МАИ Шумаев А.Ю.

|  |  |  |
| --- | --- | --- |
| Дата\_\_\_\_\_\_\_\_\_\_\_\_\_ | *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*  *подпись* | *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*  *Расшифровка подписи* |

**ДНЕВНИК** **ПРОХОЖДЕНИЯ ПРАКТИКИ**

|  |  |  |
| --- | --- | --- |
| Дата | Наименование выполняемых работ | Подпись  руководителя |
| 11.06.20 | Ознакомление с программой практики. Прохождение инструктажа по технике безопасности и охране труда, изучение внутреннего распорядка организации и правил работы. Разработка задания на производственную практику. |  |
| 12.06.20 | Описание структуры сети, в которой может функционировать разработанная база данных. |  |
| 13.06.20 | Описание сетевого оборудования необходимого для работы с базой данных. |  |
| 15.06.20 | Описание программного обеспечения необходимого для работы с базой данных по сети. |  |
| 16.06.20-18.06.20 | Разработка клиентских интерфейсов для клиент-серверных приложений. |  |
| 20.06.20 | Создание инфокоммуникационной системы.  Выполнение сетевых настроек для взаимодействия с СУБД. |  |
| 22.06.20-24.06.20 | Создание концептуальной модели данных.  Создание логической модели данных.  Создание физической модели данных.  Описание механизмов обеспечения целостности базы данных. |  |
| 25.06.20-01.07.20 | Описание СУБД представленной базы данных и ее возможностей. Внесение различных данных в базу данных.  Структурирование запросов базы данных.  Определение методов создания хранимых процедур и триггеров. Создание хранимых процедур и триггеров базы данных. |  |
| 02.07.20-03.07.20 | Определение способов управления правами пользователей.  Описание распределения прав пользователей и управления ими в базе данных. |  |
| 03.07.20-04.07.20 | Определение методов создания и синхронизации реплик базы данных. Описание существующих механизмов репликации в базе данных. Определение методов создания резервных копий базы данных. Описание существующих механизмов резервного копирования в базе данных. |  |
| 06.07.20 | Составление отчёта по практике |  |
| 07.07.20 | Подготовка к защите отчёта по практике. |  |
| 08.07.20 | Итоговая аттестация по производственной практике - зачёт |  |

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 7](#_Toc53870885)

[1. GITHUB 8](#_Toc53870886)

[1.1. О системе контроля версий 8](#_Toc53870887)

[1.2. Локальные системы контроля версий 8](#_Toc53870888)

[1.3. Централизованные системы контроля версий 9](#_Toc53870889)

[1.4. Распределённые системы контроля версий 10](#_Toc53870890)

[2. ОСНОВНЫЕ ОПЕРАЦИИ GITHUB 12](#_Toc53870891)

[1.1. Репозиторий 12](#_Toc53870892)

[1.2. Лес git’а 12](#_Toc53870893)

[1.3. Ветвление 14](#_Toc53870894)

[1.4. Создание новой ветки 14](#_Toc53870895)

[1.5. Переключение между ветками 14](#_Toc53870896)

[1.6. Слияние веток 14](#_Toc53870897)

[1.7. Фичи git’а 15](#_Toc53870898)

[3 SQLITE 16](#_Toc53870899)

[3.1 Определение 16](#_Toc53870900)

[3.2 Особенности 17](#_Toc53870901)

[3.3 Использование 23](#_Toc53870902)

[3.4 Инструменты для разработки 24](#_Toc53870903)

[ЗАКЛЮЧЕНИЕ 26](#_Toc53870904)

[СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ 27](#_Toc53870905)

# ВВЕДЕНИЕ

Практическая выполнена в соответствии с рабочей программой ПМ.02 «Разработка и администрирование баз данных». В работе представлены основные этапы анализа и разработки предметной области «Разработка и администрирование базы данных», навыки работы с GitHub и SQLite.

База данных играет важную роль в большинстве предметных областей. Благодаря динамической природе автоматизированных отраслей сейчас приложения требуют некоторых механизмов хранения, доступа и изменения данных. Поскольку важность баз данных стремительно растёт, реляционные системы управления базами данных набирают свою популярность.

В качестве программного обеспечения для создания интерфейса выбран редактор Notepad++. А для написания кода выбраны: язык гипертекстовой разметки HTML, язык таблиц стилей CSS и для работы с базами данных PHP. Использован веб-интерфейс phpmyadmin и СУБД SQLite.

В процессе выполнения проекта должен быть освоен ряд следующих профессиональных компетенций: ознакомление со структурой Git, реализация базы данных в конкретной системе управления базами данных. Получение знаний и опыта в области в технологии СУБД «SQLite» Также мною будут решаться вопросы администрирования базы данных и реализовываться методы и технологии её защиты.

# GITHUB

Эта глава о том, как начать работу с Git. Вначале изучим основы систем контроля версий, затем перейдём к тому, как запустить Git на вашей ОС и окончательно настроить для работы. В конце главы вы уже будете знать, что такое Git и почему им следует пользоваться, а также получите окончательно настроенную для работы систему.

* 1. О системе контроля версий

Что такое «система контроля версий» и почему это важно? Система контроля версий — это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии. Для контроля версий файлов в этой книге в качестве примера будет использоваться исходный код программного обеспечения, хотя на самом деле вы можете использовать контроль версий практически для любых типов файлов.

Если вы графический или web-дизайнер и хотите сохранить каждую версию изображения или макета (скорее всего, захотите), система контроля версий (далее СКВ) — как раз то, что нужно. Она позволяет вернуть файлы к состоянию, в котором они были до изменений, вернуть проект к исходному состоянию, увидеть изменения, увидеть, кто последний менял что-то и вызвал проблему, кто поставил задачу и когда и многое другое. Использование СКВ также значит в целом, что, если вы сломали что-то или потеряли файлы, вы спокойно можете всё исправить. В дополнение ко всему вы получите всё это без каких-либо дополнительных усилий.

* 1. Локальные системы контроля версий

Многие люди в качестве метода контроля версий применяют копирование файлов в отдельную директорию (возможно даже, директорию с отметкой по времени, если они достаточно сообразительны). Данный подход очень распространён из-за его простоты, однако он невероятно сильно подвержен появлению ошибок. Можно легко забыть, в какой директории вы находитесь, и случайно изменить не тот файл или скопировать не те файлы, которые вы хотели.

Для того, чтобы решить эту проблему, программисты давным-давно разработали локальные СКВ с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль ревизий.



Рисунок 1. Локальный контроль версий.

Одной из популярных СКВ была система RCS, которая и сегодня распространяется со многими компьютерами. RCS хранит на диске наборы патчей (различий между файлами) в специальном формате, применяя которые она может воссоздавать состояние каждого файла в заданный момент времени.

* 1. Централизованные системы контроля версий

Следующая серьёзная проблема, с которой сталкиваются люди, — это необходимость взаимодействовать с другими разработчиками. Для того, чтобы разобраться с ней, были разработаны централизованные системы контроля версий (ЦСКВ). Такие системы, как CVS, Subversion и Perforce, используют единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища. Применение ЦСКВ являлось стандартом на протяжении многих лет.



Рисунок 2. Централизованный контроль версий.

Такой подход имеет множество преимуществ, особенно перед локальными СКВ. Например, все разработчики проекта в определённой степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать ЦСКВ, чем оперировать локальными базами данных на каждом клиенте.

Несмотря на это, данный подход тоже имеет серьёзные минусы. Самый очевидный минус — это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всё — всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков. Локальные СКВ страдают от той же самой проблемы: когда вся история проекта хранится в одном месте, вы рискуете потерять всё.

* 1. Распределённые системы контроля версий

Здесь в игру вступают распределённые системы контроля версий (РСКВ). В РСКВ (таких как Git, Mercurial, Bazaar или Darcs) клиенты не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени) — они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, умрёт, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных.



Рисунок 3. Распределённый контроль версий.

Более того, многие РСКВ могут одновременно взаимодействовать с несколькими удалёнными репозиториями, благодаря этому вы можете работать с различными группами людей, применяя различные подходы единовременно в рамках одного проекта. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

## ОСНОВНЫЕ ОПЕРАЦИИ GITHUB

Начнем, естественно, с загрузки. Надеемся, какая у вас операционная система, вы знаете. И сразу предупредим новичков: не путайте git и GitHub — это разные вещи. Нас интересует именно git, а GitHub (или ему подобные сервисы вроде Bitbucket или GitLab) — это по сути хостинг для проектов, использующих git.

* 1. Репозиторий

Итак, вот у вас уже есть git. Теперь нужно создать хранилище версий для него. Запомните, это хранилище называется репозиторий (англ. repository) — при случае можете вставить где-нибудь это словечко. В зависимости от того, какая у вас оболочка, соответствующей командой создайте новую директорию, откройте ее (в командной строке, она же оболочка, а не проводником или чем-то подобным) и выполните:

git init

Все, локальный репозиторий в этой папке создан. То, что здесь сейчас хранится, будет бекапом, поэтому, чтобы его не испортить, создадим рабочую копию (англ. check out) локальной версии:

git clone [url]

Где [url] — это путь до клонируемого репозитория. Мы разбираем сейчас случай, когда вы создаете рабочую копию собственного репозитория, поэтому в качестве [url] здесь вам нужно указать путь до директории, для которой мы выполняли git init.

Но если вы крутой чувак и уже работаете с удаленным сервером, то вот такая команда будет для вас в самый раз:

git clone username@host:/path/to/repository

* 1. Лес git’а

Немного теории. Git в своей работе управляет тремя структурами, которые называются деревьями. Первое — это рабочая директория, в ней хранятся файлы, с которыми вы прямо сейчас работаете. Ну, она ж рабочая, логично. Второе — это Index, этакий чек-поинт, который позволяет вам вносить изменения и ничего не портить. А третье — это HEAD, который указывает на последний сделанный вами коммит. (Чтобы вы не запутались в терминологии: коммит (англ. commit) — это сохранение состояния проекта в репозиторий. Короче, считайте, новая версия.)

Так вот, чтобы вы не заблудились в этих трех соснах, запомните две крутые команды: add и commit. Они позволят вашей работе спокойно бродить по git’у, сохраняясь, куда надо. Если вы придумали что-то гениальное и тут же внесли изменение в рабочую копию проекта, то не спешите сразу коммитить! Сначала испытайте в Index’е, для этого выполните:

git add [имя\_файла]

если вы внесли изменение только в один файл, или

git add \*

если вы хорошо потрудились поменяли сразу кучу исходников. Изменения положительны? Хорошо потестили? Тогда скорее коммитить:

git commit -m "Commit message"

Вы, конечно же, плохой парень и не оставляете комментариев в коде. Но git — другое дело. Не ленитесь оставлять поясняющие сообщения: будьте уверены, вам хватит других проблем, кроме как разбираться, что же поменялось в этом коммите по сравнению с прошлой версией. Поясняющая картинка:

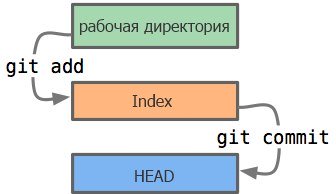
[](https://s3.tproger.ru/uploads/2016/06/EXcwUk86PWs.jpg)

Рисунок 4. Git add и commit

Теперь файл(-ы) прочно обосновались в HEAD вашей рабочей локальной копии. Оттуда их не выгнать, но в вашем удаленном репозитории их все еще нет. Давайте сунем их еще и туда! Используйте:

git push origin master

Только вместо master напишите название нужной ветки. Ах да, вы же еще не знаете, что такое ветки. Ну ладно, пока что запомните это место, а когда прочитаете про ветвление, вернетесь сюда.

Ах да, для крутых чуваков, работающих с серверами (как раз тут уместно говорить про GitHub, например), команда будет такой:

git remote add origin [сервер]

* 1. Ветвление

По-английски эта штука зовется branching — лучше как следует вникните в этот вопрос и почитайте про ветвление подробнее, я вас с ним только познакомлю. Ветвление используется для одновременной и независимой разработки разных фич (ну, или накопления большего количества багов, ведь исходного кода становится больше). Основной веткой является master — она появляется при создании репозитория. Другие ветки — это песочницы, когда достаточно в них наиграетесь, слейте в единое целое в master. Сейчас поясню, как это делается.

* 1. Создание новой ветки

Вот вы решили проработать какую-нибудь новую фичу. Создайте для нее новую ветку:

git checkout -b [новая\_ветка]

Ах да, фантазия-то у вас, наверное, работает на полную катушку, ну да поумерьте её в деле именования веток: назвать ветку можно только именем, допустимым для переменной в вашем любимом языке.

* 1. Переключение между ветками

Надо сделать перерыв в работе с этой фичей и переключиться на другую ветку? Используйте (если работаете с локальным репозиторием, то указывать его имя не обязательно):

git checkout [репозиторий]/[ветка]

Ну, а если вы уже совсем не хотите с ней работать, то удалите ее совсем:

git branch -d [ветка]

Со своей веткой вы можете творить любые непотребства: ее никто не увидит, пока вы сами ее не пропушите в удаленный репозиторий командой:

git push origin [ветка]

* 1. Слияние веток

Чтобы слить ветку в ту, с которой вы сейчас работаете, используйте:

git merge [ветка]

Но, понятное дело, это все приводит к конфликтам. И это реально проблема. Так что попробуйте исправлять все ручками прямо в директории с репозиторием. Только потом не забудьте пометить, что вы их «слили»:

git add [имя\_файла]

Кстати, ветки можно сравнить:

git diff [одна\_ветка] [другая\_ветка]

Так, теперь приступим к более решительным действиям. Будем обновлять свой репозиторий в соответствии с самым свежим коммитом. Сделать это очень просто (а вот вернуть обратно не очень, поэтому трижды подумайте, прежде чем совершать эту ужасную ошибку):

git pull

Я, конечно, понимаю, что вы слишком круты, чтобы оставлять какие-либо пометки на будущее — все держите в голове — но все-таки рекомендую вам оставлять тэги. И это не моя выдумка, так делают многие:

git tag [tag] [первые\_десять\_символов\_соответствующего\_коммита]

Вы не знаете, какие первые символы у имени нужного коммита? Не беда, смотрите в историю репозитория — его лог:

git log

Там есть куча разных параметров для использования этой полезной штуковины, ну да погуглите их сами. Ах да, кстати, мы уже писали как-то про то как сделать git log более информативным.

* 1. Фичи git’а

Если вы ленивый, и вам не охота по-трупрогерски все писать в оболочке своей ОСи, то можете использовать GUI git’а:

gitk

В источнике найдете еще кучу других GUI-шек.  
Если вам стандартный вывод git’а кажется скучным, раскрасьте его:

git config color.ui true

Ну, и есть еще такая штука — интерактивное индексирование. Когда у вас будет уже достаточно большой проект, то ужать представление index’а в log’е можно будет так:

git add -i

# 3 SQLITE

## 3.1 Определение

SQLite — это встраиваемая кроссплатформенная БД, которая поддерживает достаточно полный набор команд SQL и доступна в исходных кодах (на языке C). Позиция функциональности SQLite где-то между MySQL и PostgreSQL. Однако, на практике, SQLite нередко оказывается в 2-3 раза (и даже больше) быстрее. Такое возможно благодаря высокоупорядоченной внутренней архитектуре и устранению необходимости в соединениях типа «сервер-клиент» и «клиент-сервер».

Всё это, собранное в один пакет, лишь немногим больше по размеру клиентской части библиотеки MySQL, является впечатляющим достижением для полноценной базы данных. Используя высоко эффективную инфраструктуру, SQLite может работать в крошечном объёме выделяемой для неё памяти, гораздо меньшем, чем в любых других системах БД. Это делает SQLite очень удобным инструментом с возможностью использования практически в любых задачах, возлагаемых на базу данных.

Движок БД представляет библиотеку, с которой программа компонуется и SQLite становится составной частью программы. Вся БД хранится в единственном стандартном файле на машине, на которой исполняется программа. Несколько процессов или потоков могут одновременно без каких-либо проблем читать данные из одной базы. Запись в базу можно осуществить только в том случае, если никаких других запросов в данный момент не обслуживается; в противном случае попытка записи оканчивается неудачей, и в программу возвращается код ошибки. Другим вариантом развития событий является автоматическое повторение попыток записи в течение заданного интервала времени.

Поддерживаемые типы данных: NULL: NULL-значение; INTEGER: целое со знаком, хранящееся в 1, 2, 3, 4, 6, или 8 байтах; REAL: число с плавающей запятой, хранящееся в 8-байтовом формате IEEE; TEXT: текстовая строка с кодировкой UTF-8, UTF-16BE или UTF-16LE; BLOB: тип данных, хранящийся точно в таком же виде, в каком и был получен.

Сама библиотека SQLite написана на C; существует большое количество привязок к другим языкам программирования, в том числе Apple Swift, Delphi, C++, Java, C#, VB.NET, Python, Perl, Node.js, PHP, PureBasic, Tcl (средства для работы с Tcl включены в комплект поставки SQLite), Ruby, Haskell, Scheme, Smalltalk, Lua и Parser, а также ко многим другим.

Простота и удобство встраивания SQLite привели к тому, что библиотека используется в браузерах, музыкальных плеерах и многих других программах. В частности, SQLite используется в: Adobe Integrated Runtime — среда для запуска приложений (частично); Gears; Autoit; Фреймворк Qt; Платформа XUL на движке Gecko 1.9+, XULRunner 1.9+ и, потенциально, все приложения, основанные на этой платформе, в том числе: Mozilla Firefox (начиная с версии 3.0); SQLite Manager; Skype; Viber;

Многие программы поддерживают SQLite в качестве формата хранения данных (особенно в Mac OS и iOS, Android), в том числе: 1С Предприятие; Adobe Photoshop Lightroom; AIMP; Google Chrome; Opera (начиная с версии 10.50); Safari; XnView;

## 3.2 Особенности

Использование SQLite в многопоточных приложениях:

SQLite может быть собран в однопоточном варианте (параметр компиляции SQLITE\_THREADSAFE = 0). В этом варианте его нельзя одновременно использовать из нескольких потоков, поскольку полностью отсутствует код синхронизации. Проверить, есть ли многопоточность можно через вызов sqlite3\_threadsafe(): если вернула 0, то это однопоточный SQLite. По умолчанию, SQLite собран с поддержкой потоков (sqlite3.dll). Есть два способа использования многопоточного SQLite: serialized и multi-thread.

Serialized (надо указать флаг SQLITE\_OPEN\_FULLMUTEX при открытии соединения). В этом режиме потоки могут как угодно дергать вызовы SQLite, никаких ограничений. Но все вызовы блокируют друг друга и обрабатываются строго последовательно.

Multi-thread (SQLITE\_OPEN\_NOMUTEX). В этом режиме нельзя использовать одно и то же соединение одновременно из нескольких потоков (но допускается одновременное использование разных соединений разными потоками). Обычно используется именно этот режим.

Формат данных:

База данных SQLite может хранить (текстовые) данные в UTF-8 или UTF-16. Набор вызовов API состоит из вызовов, которые получают UTF-8 (sqlite3\_XXX) и вызовов, которые получают UTF-16 (sqlite3\_XXX16). Если тип данных интерфейса и соединения не совпадает, то выполняется конвертация «на лету».

Поддержка UNICODE:

По умолчанию — поддержки нет. Нужно создать свой collation (способ сравнения) через sqlite3\_create\_collation. И определить свои встроенные функции like(), upper(), lower() через www.sqlite.org/c3ref/create\_function.html. Есть проект «International Components for Unicode», ICU и некоторые собирают SQLite DLL уже с ним.

Немного про работу ICU и SQLite:

Значение внутри БД может принадлежать к одному из следующих типов хранения (storage class): NULL, INTEGER (занимает 1,2,3,4,6 или 8 байт), REAL (число с плавающей точкой, 8 байт в формате IEEE), TEXT (строка в формате данных базы, обычно UTF-8), BLOB (двоичные данные, хранятся «как есть»).

Порядок сортировки значений разных типов:

NULL меньше всего (включая другой NULL);

INTEGER и REAL меньше любого TEXT и BLOB, между собой сравниваются арифметически;

TEXT меньше любого BLOB, между собой сравниваются на базе своих collation;

BLOB сравниваются между собой через memcmp().

SQLite выполняет неявные преобразования типов «на лету» в нескольких местах:

при занесении значения в столбец (тип столбца задает рекомендацию по преобразованию);

при сравнении значений между собой.

Столбец может иметь следующие рекомендации приведения типа: TEXT, NUMERIC, INTEGER, REAL, NONE.

Значения BLOB и NULL всегда заносятся в любой столбец «как есть».

В столбец TEXT значения TEXT заносятся «как есть», значения INTEGER и REAL становятся строками. В столбец NUMERIC, INTEGER числа записываются «как есть», а строки становятся числами, если могут (то есть допустимо обратное преобразование «без потерь»). Для столбца REAL правила похожи на INTEGER(NUMERIC); отличие в том, что все числа представлены в формате с плавающей запятой. В столбец NONE значения заносятся «как есть» (этот тип используется по умолчанию, если не задан другой).

При сравнении значений разного типа между собой может выполняться дополнительное преобразование типов.

При сравнении числа со строкой, если строка может быть преобразована в число «без потерь», она становится числом.

В SQLite в уникальном индексе может быть сколько угодно NULL значений (с этим согласен Oracle и не согласен MS SQL).

Если в вызове sqlite3\_open() передать имя файла как ":memory:", то SQLite создаст соединение к новой (чистой) БД в памяти. Это соединение абсолютно неотличимо от соединения к БД в файле по логике использования: доступен тот же набор SQL команд. Сейчас это исправлено и можно открыть два соединения к одной БД в памяти.

rc = sqlite3\_open("file:memdb1?mode=memory&cache=shared", &db);

ATTACH DATABASE 'file:memdb1?mode=memory&cache=shared' AS aux1;

Чтобы открыть соединение к БД используется вызов sqlite3\_open(). В любой момент времени мы можем к открытому соединению присоединить еще до 10 баз данных через SQL команду ATTACH DATABASE.

sqlite3\_open('foo.sqlite3', &db); // откроем соединение к БД в файле "foo.sqlite3"

sqlite3\_exec(&db, "ATTACH 'bar.sqlite3' AS bar", ... ); // присоединим "bar.sqlite3"

Теперь все таблицы БД в файле db1.sqlite3 стали прозрачно доступны в нашем соединении. Для разрешения конфликтов имен следует использовать имя присоединения (основная база называется «main»):

SELECT \* FROM main.my\_table UNION SELECT \* FROM bar.my\_table

Ничего не мешает присоединить к БД новую базу в памяти и использовать ее для кэширования и пр.

sqlite3\_open('foo.sqlite3', &db); // откроем соединение к БД в файле "foo.sqlite3"

sqlite3\_exec(&db, "ATTACH ':memory:' AS mem", ... ); // присоединим новую БД в памяти

Передайте пустую строку вместо имени файла в sqlite3\_open() и будет создана временная БД в файле на диске. Причем, после закрытия соединения к БД, она будет удалена с диска.

SQL команда PRAGMA служит для задания всевозможных настроек у соединения или у самой БД:

PRAGMA name; // запросить текущее значение параметра name

PRAGMA name = value; // задать параметр name значением value

Настройку соединения (очевидно) следует проводить сразу после открытия и до его использования.

Некоторые параметры:

PRAGMA page\_size = bytes; // размер страницы БД; страница БД - это единица обмена между диском и кэшом, разумно сделать равным размеру кластера диска

PRAGMA cache\_size = -kibibytes; // задать размер кэша соединения в килобайтах, по умолчанию он равен 2000 страниц БД

PRAGMA encoding = "UTF-8"; // тип данных БД, всегда используйте UTF-8

PRAGMA foreign\_keys = 1; // включить поддержку foreign keys, по умолчанию - ОТКЛЮЧЕНА

PRAGMA journal\_mode = DELETE | TRUNCATE | PERSIST | MEMORY | WAL | OFF; // задать тип журнала

PRAGMA synchronous = 0 | OFF | 1 | NORMAL | 2 | FULL; // тип синхронизации транзакции

Журнал и фиксация транзакций

SQLite тщательно блюдет целостность данных в БД (ACID), реализуя механизм изменения данных через транзакции. Кратко о транзакциях: транзакция либо полностью накатывается, либо полностью откатывается. Промежуточных состояний быть не может. Если вы не используете транзакции явно (BEGIN; ...; COMMIT;), то всегда создается неявная транзакция. Она стартует перед выполнением команды и коммитится сразу после. Отсюда жалобы на «медленность» SQLite. SQLite может вставлять и до 50 тыс записей в секунду, но фиксировать транзакций он не может больше, чем ~ 50 в секунду. Именно поэтому, не получается вставлять записи быстро, используя неявную транзакцию.

При настройках по умолчанию SQLite гарантирует целостность БД даже при отключении питания в процессе работы. Достигается подобное изумительное поведение ведением журнала (специального файла) и хитроумным механизмом синхронизации изменений на диске. Обновление данных в БД работает так:

— до любой модификации БД SQLite сохраняет изменяемые страницы из БД в отдельном файле (журнале), то есть просто копирует их туда; — убедившись, что копия страниц создана, SQLite начинает менять БД; — убедившись, что все изменения в БД «дошли до диска» и БД стала целостной, SQLite стирает журнал.

Если SQLite открывает соединение к БД и видит, что журнал уже есть, он соображает, что БД находится в незавершенном состоянии и автоматически откатывает последнюю транзакцию. То есть механизм восстановления БД после сбоев, фактически, встроен в SQLite и работает незаметно для пользователя. По умолчанию журнал ведется в режиме DELETE.

PRAGMA journal\_mode = DELETE

Это означает, что файл журнала удаляется после завершения транзакции. Сам факт наличия файла с журналом в этом режиме означает для SQLite, что транзакция не была завершена, база нуждается в восстановлении. Файл журнала имеет имя файла БД, к которому добавлено "-journal".

В режиме TRUNCATE файл журнала обрезается до нуля (на некоторых системах это работает быстрее, чем удаление файла).

В режиме PERSIST начало файла журнала забивается нулями (при этом его размер не меняется, и он может занимать кучу места).

В режиме MEMORY файл журнала ведется в памяти и это работает быстро, но не гарантирует восстановление базы при сбоях (копии данных-то нету на диске).

А можно и совсем отключить журнал (PRAGMA journal\_mode = OFF). В этой ситуации перестает работать откат транзакций (команда ROLLBACK) и база, скорее всего, испортится, если программа будет завершена аварийно. Для базы данных в памяти режим журнала может быть только либо MEMORY, либо OFF.

Мы знаем, что современные системы используют хитроумное кэширование для повышения производительности и могут откладывать запись на диск. Допустим, SQLite завершил запись в БД и хочет стереть файл журнала, чтобы отметить факт фиксации транзакции. Если в этот промежуток времени отключится питание, то журнала уже не будет, а БД еще не будет целостной — потеря данных!

PRAGMA synchronous задает степень «паранойи» SQLite на это счет.

Режим OFF (или 0) означает: SQLite считает, что данные фиксированы на диске сразу после того как он передал их ОС (то есть сразу после вызова соот-го API ОС). Это означает, что целостность гарантирована при аварии приложения (поскольку ОС продолжает работать), но не при аварии ОС или отключении питания.

Режим синхронизации NORMAL (или 1) гарантирует целостность при авариях ОС и почти при всех отключениях питания. Существует ненулевой шанс, что при потере питания в самый неподходящий момент база испортится. Это некий средний, компромисный режим по производительности и надежности.

Режим FULL гарантирует целостность всегда и везде и при любых авариях. Но работает, разумеется, медленнее, поскольку в определенных местах делаются паузы ожидания. И это режим по умолчанию.

Режим журнала WAL

По умолчанию, режим журнала БД всегда «возвращается» в DELETE. Допустим, мы открыли соединение к БД и установили режим PERSIST. Изменили данные, закрыли соединение. На диске остался файл журнала (начало которого забито нулями). Открываем соединение к БД снова. Если не задать режим журнала в этом соединении, он опять будет работать в DELETE. Как только мы обновим данные, механизм фиксации транзакций сотрет файл журнала.

Режим журнала WAL работает иначе — он «постоянный». Как только мы перевели базу в режим WAL, она останется в этом режиме, пока ей явно не поменяют режим журнала на другой.

Изначально SQLite проектировалась как встроенная БД. Архитектура разделения одновременного доступа к данным была устроена примитивно: одновременно несколько соединений могут читать БД, а вот записывать в данный момент времени может только одно соединение. Это, как минимум, означает, что пишущее соединение ждет «освобождения» БД от читающих. При попытке записать в «занятую» БД приложение получает ошибку SQLITE\_BUSY (не путать с SQLITE\_LOCKED!). Достигается этот механизм разделения доступа через API блокировки файлов (которые плохо работают на сетевых дисках, поэтому там не рекомендуется использовать SQLite; узнать больше )

В режиме WAL (Write-Ahead Logging) «читатели» БД и «писатели» в БД уже не мешают друг другу, то есть допускается модификация данных при одновременном чтении. Короче говоря, это шаг в сторону больших и серьезных СУБД, в которых все так и есть. Утверждается также, что SQLite в WAL работает быстрее.

Но есть и недостатки: — требуется некоторые дополнительные ништяки от ОС (unix и Windows имеют эти ништяки); — БД занимает несколько файлов (файлы «XXX-wal» и «XXX-shm»); — плохо работает на больших транзакциях (условно, если транзакция больше 50 Мбайт); — нельзя открыть такую БД в режиме «только чтение»; — возникает дополнительная операция checkpoint.

Фактически, в режиме WAL данные БД разделяются между БД и файлом журнала. Операция checkpoint переносит данные в БД. По умолчанию, это делается автоматически, если журнал занял 1000 страниц БД. То есть, идут быстрые COMMIT-ы и вдруг какой-то COMMIT задумался и начал делать checkpoint. Если такое поведение нежелательно, можно делать checkpoint вручную (когда все спокойно), можно это делать и в отдельном процессе.

Пределы

Несмотря на миниатюрность, SQLite в реальности не накладывает серьезных ограничений на размеры полей, таблиц или БД.

По умолчанию, BLOB или строкое значение могут занимать 1 Гбайт и это же ограничение размера одной записи (можно поднять до 2^31 — 1, параметр SQLITE\_MAX\_LENGTH).

Количество столбцов: 2000 (можно поднять до 32767, SQLITE\_MAX\_COLUMN).

Размер SQL оператора: 1 МБайт (1073741824 байт, SQLITE\_MAX\_SQL\_LENGTH).

Одновременный join: 64 таблицы.

Присоединить баз к соединению: 10 (до 62, SQLITE\_MAX\_ATTACHED)

Максимальное количество страниц в БД: 1073741823 (до 2147483646, SQLITE\_MAX\_PAGE\_COUNT).

Если задать размер страницы 65636 байт, то максимальный размер БД будет примерно 14 Терабайт.

Максимальное число записей в таблице: 2^64 — 1, но на практике, конечно, ограничение размера вступит раньше

## 3.3 Использование

Процедурный интерфейс к SQLite почти такой же, как у MySQL и других расширений БД. По большей части переход к SQLite потребует только изменить mysql/pq/etc... префикс функции на sqlite.

<?php

// создаём новую базу (процедурный интерфейс)

$db = sqlite\_open("db.toy-store");

// создаём таблицу products

sqlite\_query($db, "CREATE TABLE products (id INTEGER PRIMARY KEY, name CHAR(255))");

// добавляем что-нибудь для примера

sqlite\_query($db, "INSERT INTO foo (name) VALUES (product1)");

sqlite\_query($db, "INSERT INTO foo (name) VALUES (product2)");

sqlite\_query($db, "INSERT INTO foo (name) VALUES (product3)");

// выполняем запрос

$result = sqlite\_query($db, "SELECT \* FROM products");

// проходим в цикле выборкой по ячейкам

while ($row = sqlite\_fetch\_array($result)) {

print\_r($row);

/\* каждый результат будет выглядеть примерно так

Array

(

[0] => 1

[id] => 1

[1] => product1

[name] => product1

)

\*/

}

// закрываем соединение с базой

sqlite\_close($db);

?>

## 3.4 Инструменты для разработки

При разработке базы данных, удобно использовать инструмент Adminer, в котором и будет редактироваться БД проекта. На рисунке 4 изображена категория «продукты» с ее компонентами.

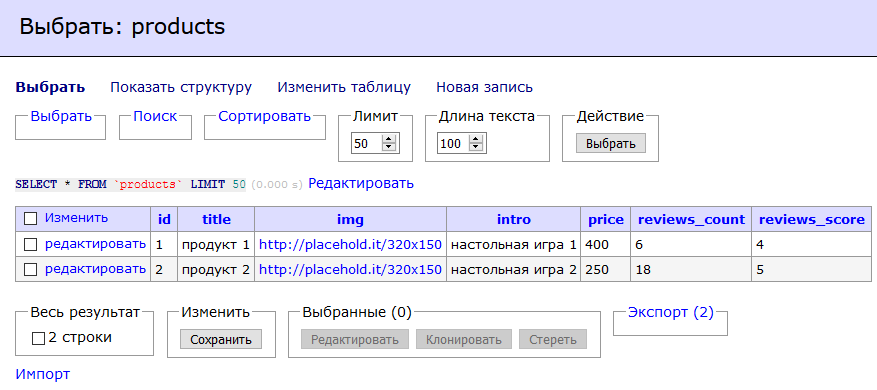


Рисунок 4. «Категория products»

Нажав на ссылку «редактировать», мы перейдем в режим редактирования значения. На рисунке 3 изображено окно редактирования одного из продуктов категории «products»

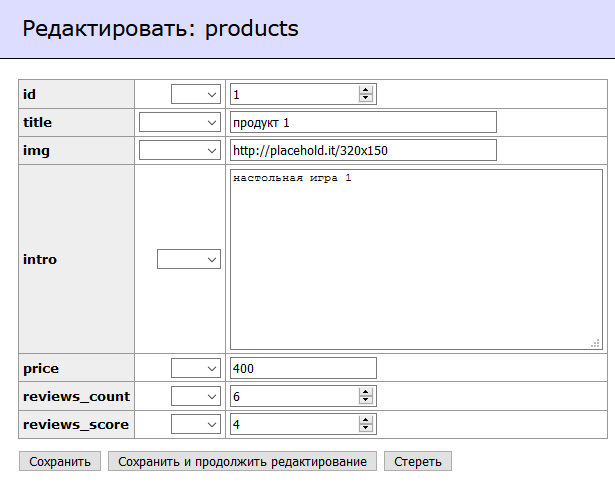


Рисунок 5. «Редактировать products»

# ЗАКЛЮЧЕНИЕ

Мы рассмотрели большинство основных способов настройки клиента и сервера Git с тем, чтобы он был максимально соответствовал вашим рабочим процессам и проектам.

Мы узнали о всевозможных настройках, атрибутах файлов и о перехватчиках событий, а также рассмотрели пример настройки сервера с соблюдением политики. Теперь вам должно быть по плечу заставить Git подстроиться под практически любой тип рабочего процесса, который только можно вообразить.

Были получены знания в SQLite3, что даёт множество преимуществ в отличии от других СУБД. Множество фрэймворков таких как Django, Ruby on Rails и web2py по умолчанию используют SQLite3. Многие браузеры используют данный инструмент для хранения локальных данных. Так же она используется в качестве хранилища данных таких ОС как Android и Windows Phone 8.

# СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Github: [Электронный ресурс]/ https://tproger.ru/translations/difference-between-git-and-github - статья в интернете.
2. SQLite: [Электронный ресурс]/ https://ru.bmstu.wiki/SQLite - статья в интернете.
3. Руководство SQLite: [Электронный ресурс]/ https://proglib.io/p/sqlite-tutorial - статья в интернете.