Создание Web-сервисов на Python

Московский физико-технический институт и Mail.Ru Group

Неделя 3

Содержание

1 Хранение данных **3**

1.1 Хранение данных . . . 3

1.2 Реляционные базы данных . . . 5

1.3 Транзакции . . . 6

1.4 Индексы . . . 9

1.5 Нереляционные базы данных . . . 10

1.5.1 BASE . . . 11

1.6 Основные виды NoSQL баз данных . . . 12

1.6.1 Хранилище вида ”ключ-значение” . . . 12

1.6.2 Документоориентированные базы данных. . . . 12

1.6.3 Bigtable-подобные базы данных . . . 13

1.6.4 Графовые базы данных . . . 13

2 MySQL **13**

2.1 Создание баз и таблиц . . . 13

2.1.1 Удаление БД . . . 13

2.1.2 Создание БД . . . 14

2.1.3 Создание таблиц внутри БД . . . 14

2.1.4 create_definition . . . 15

2.2 Создание баз и таблиц. Практика . . . 17

2.3 Редактирование таблиц в MySQL Workbench . . . 19

2.4 Изменение баз и таблиц . . . 23

2.5 Типы данных столбцов . . . 25

2.5.1 Числовые типы данных . . . 25

2.6 Дата и время . . . 27

2.6.1 Символьные типы данных . . . 28

2.6.2 JSON 29

3 Работа с данными **30**

3.1 Сложные запросы 33

3.2 Примеры запросов к MySQL 37

4 Redis **45**

4.1 Обзор Redis с примерами 45

4.2 Сложные типы данных в Redis 48

1. Хранение данных

1.1. Хранение данных

Зачем хранить данные локально?

- сайт, откуда мы получали их по API, может оказатся недоступным в конкретный момент времени;
- у API, которым мы пользуемся, могут быть какие-то ограничения по числу, по частоте его использования;
- данные могут устареть на сайте, а мы хотим хранить историю;
- сайт не стоит на соседнем сервере, а находится где-то далеко, и обращаться к нему по сети довольно-таки долго;
- данные могут быть предоставлены самим пользователем, тогда их придется сохранить. Больше нам их получить неоткуда;
- нужно для удобства манипуляции с ними, поиска и так далее.

Вообще, существует два основных варианта хранения данных — это **кэширование** и **долгосрочное хранение**.
Кэширование подразумевает, что мы берем очень часто используемые данные и кладем их в кэш, но данные в кэше могут потеряться: то есть какие-то долго получаемые данные можем сложить в кэш и быстро их оттуда отдавать, но, если в какой-то момент этих данных в кэше не окажется по какой-то причине, мы можем их заново получить.
А долгосрочное хранение, подразумевает, что данные действительно хранятся и никуда не теряются.
Если данных мало, а памяти много, а программа у нас никогда не завершается, почему бы их не хранить там? Делать это можно, но ровно до того момента, когда программа прекратит работать. Это может случиться из-за того, что сервер перегрузился, из-за того, что в программе возникла ошибка. Поэтому можно кэшировать какие-то небольшие порции данных в памяти программы, но это не надежно. Памяти может не хватить. Если вы попробуете создать очень-очень большой массив, то это может закончиться тем, что система просто убьет ваше приложение из-за того, что оно сожрало слишком много памяти.
А что у нас будет с масштабированием? Если у вас ваше приложение запущено на двух компьютерах, у каждого — своя память, и общих данных у них нет. Данные могут устаревать, и вам придется реализовать какой-то механизм, чтобы те данные, которые вы храните у себя, были обновлены.
Если мы хотим хранить в памяти, но не в программе, существует уже готовое решение. Напрмер, Memcache, Redis или Tarantool. Это сервера. Они устанавливаются отдельно, и они умеют хранить данные в памяти. Redis и Tarantool помимо этого еще могут записывать данные на диск. Memcache, если будет сбой, данные потеряет. Redis мощнее, чем Memcache. В нем есть не только возможность хранить данные по ключу, какие-то значения, но и могут быть различные типы

данных, которые он может сохранить по ключу, словари, списки. Поддерживает различные операции над данными. Tarantool предоставляет еще больше возможностей.
Такие системы используются для хранения пар ключ-значений, как словарь в Python, только в стороннем сервисе. Через какое-то API вы обращаетесь к нему, сохраняете свои данные, потом можете их получить. Также они поддерживают механизм устаревания, то есть вы можете сказать, что данные по этому ключу должны быть годными в течение какого-то времени. Если вы их попробуете получить после этого, то вам эти данные не будут возвращены. Все эти системы очень быстрые, и у них есть модули для Python.

Также может возникнуть идея писать данные в файлы. Но, если речь идет о каких-то сложных данных, то все не так просто. Если мы получили данные из какого-то API, то нам пришла строка (JSON/XML) и мы можем ее сохранить в файл. В следующий раз, когда нам нужны будут те же самые данные, мы не пойдем по сети, а прямо возьмем сразу их из файла. Но данные у нас могут записаться не полностью, или, если мы перетираем одни данные другими, то будет еще хуже. Может возникнуть неконсистентность данных или вообще их потеря.
А что если у нас данных очень много? Например, там все население России, а нам нужен какой-то один человек. Если мы прочтем все это в память, памяти может не хватить.
Решением в нашем случае будет использовать базу данных. База данных, или система управления базами данных, обеспечивает как раз работу с большими объемами данных. Она обеспечивает управление данными на диске или в памяти, целостность этих данных, резервирование данных, и, как правило, у нее есть какой-то язык для доступа и определения этих данных.

Типы баз данных:

- клиент-серверные или встроенные базы данных. Клиент-серверные подразумевают, что у баз данных есть отдельно выделенный сервер, а ваше приложение является клиентом, оно обращается к серверу, сохраняет там данные, получает по какому-то протоколу. А встроенные подразумевают, что нет выделенного сервера баз данных, система управления базы данных встраивается прямо в приложение и поставляется вместе с ним. Не требует никакой установки и работает вместе с приложением.

- локальные и распределенные базы данных. Локальные базы данных — это когда все, что есть в этой базе данных, хранится на одном компьютере. А распределенные — это когда база данных работает на нескольких серверах одновременно.

- реляционные и нереляционные базы данных. Реляционные базы данных — состоящие из таблиц и отношений между ними, а нереляционные — это, как правило, хранилище «ключ-значение», там нет связей между таблицами. Это Redis, Tarantool.

Базы данных могут хранить огромное количество записей, очень быстро выбирать из этих записей нужные по каким-то параметрам, не перебирая все записи полностью, могут работать по сети, могут организовываться в кластера, и многое другое.

1.2. Реляционные базы данных

Реляционная база данных – это набор таблиц, чем-то похожих на таблицы Excel. Каждая таблица – это набор строк и столбцов, и она содержит информацию об объектах определенного типа. То есть, в одной таблице информация об одинаковых по типу объектах. И может иметь связи с другими таблицами. Связи – это ключевой момент.
В каждой строке таблицы находятся данные одного объекта. А столбцы описывают свойства этого объекта, например, логин, имя, пароль, если речь идет о пользователе. Все строки одной таблицы имеют одинаковую структуру, не может быть двух строк с разным набором полей в одной и той же таблице. А каждое поле, в свою очередь, имеет строго определенный тип данных.
Структура базы данных описывается схемой базы данных, которая включает в себя описание содержания, структуры и ограничения целостности. Описывается эта структура на языке СУБД, как правило, это SQL.

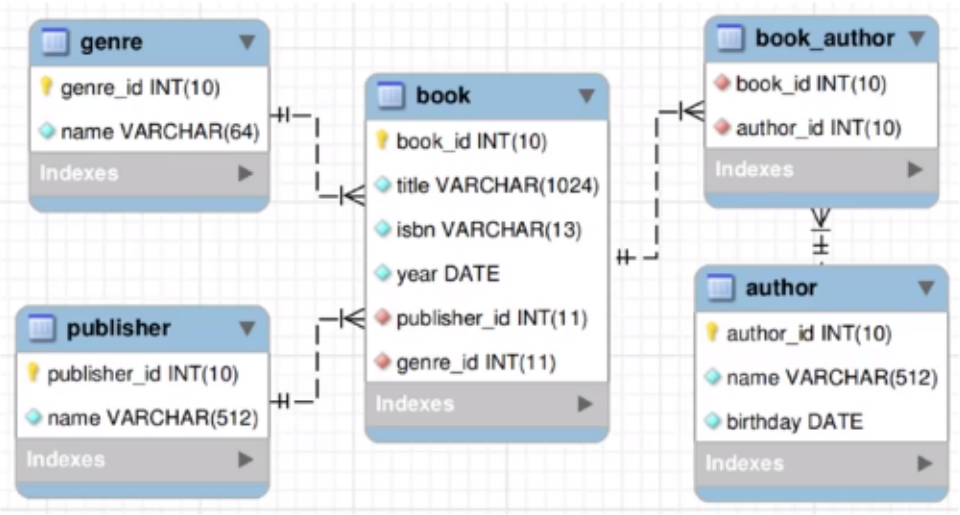

Рис. 1

Но удобно эту структуру представить графически. Здесь пять таблиц, в них есть поля, они представлены, типы полей указаны: INT, VARCHAR, DATE. VARCHAR – это строка переменной длины, но не больше, например, 1024 символов для поля title.
Кстати, типы полей тоже помогают контролировать целостность данных, то есть нельзя записать в поле с типом число строку. Соответственно, ошибочные данные записать нельзя.
Пунктир – связи между таблицами.
Один из принципов реляционных баз данных – это нормализация данных, то есть данные не дублируются, а если есть одинаковый набор данных у разных объектов, их проще вынести в отдельную таблицу и ссылаться на них. Это повышает целостность данных.
Таким образом, в реляционных базах данных есть связи и ограничения. Связи бывают один к одному, один ко многим и многие ко многим. Вот, например, связь между издателем и книгой – это один ко многим. У одной книги может быть только один издатель, но у одного издателя может быть много книг, изданных им. То же самое с жанрами. А вот, например, авторов книги может быть несколько. Книга может быть написана в соавторстве. С другой стороны, у одного

автора может быть тоже много книг. Таким образом, это – связь многие ко многим.
Связь один к одному в теории считается чем-то плохим, потому что если у вас одна строка одной таблицы жестко связана с одной строкой другой таблицы, то в принципе поля можно было бы сразу добавить в первую таблицу и не нужна вторая. С другой стороны, какие-то данные могут использоваться чаще, а какие-то реже, причем этих редко используемых данных может быть много. И тогда для оптимизации скорости можно разделить эти данные на несколько таблиц, в одной таблице хранить только самые нужные данные, а в другой какие-то уже дополнительные и иметь связь один к одному.
Ограничения: связь между таблицами реализуется с помощью внешних ключей, когда один id ссылается на другой id в другой таблице. Ограничение помогает нам сохранить целостность. Например, книга ссылается внешним ключом на издателя. Если мы удалим издателя, книга будет ссылаться на отсутствующие строки. Это – нарушение целостности данных. На связи между книгой и издателем можно наложить ограничения, например, в случае удаления издателя сказать базе данных, чтобы она удаляла все книги этого издателя. Тогда целостность данных не будет нарушена. Также мы можем запретить удалять издателей, на которых ссылается хотя бы одна книга. Это тоже поможет нам контролировать целостность данных.
Пунктирами показаны отношения. Жанр с книгой – один ко многим, издатель с книгой – один ко многим, а автор с книгой через промежуточную таблицу – многие ко многим. Почему через промежуточную таблицу? Таблицы имеют жесткую структуру, то есть количество колонок в каждой строке одинаковое. Соответственно, мы не могли бы сделать так, чтобы у одной книги был один автор, как эта одна колоночка для автора, а у другой две колонки для автора. Поэтому мы используем промежуточную таблицу-блок автор.
Пусть мы хотим выбрать все книги какого-то конкретного автора, мы знаем его author id. Например, мы получили его из таблицы author заранее, и хотим получить их упорядоченными по году. В качестве данных мы хотим получить заголовок книги, год издания, имя издателя и имя автора. Мы можем написать такой запрос:

```
SELECT title, year, p.name AS pub, a.name
AS author FROM
book JOIN publisher AS p USING (publisher_id) JOIN
book_author using (book_id)
JOIN author AS a USING (author_id) WHERE
a.author_id=1 ORDER BY year;
```

1.3. Транзакции

Транзакция — это последовательность операций с базой данных, которая может быть выполнена либо целиком, полностью и успешно независимо от других транзакций, либо полностью не выполнена. И тогда она не должна произвести никакого эффекта так, как будто бы ее вообще не было.
Мы переводим деньги в банке со счета A на счет B. Мы читаем баланс счета A, списываем с него сумму перевода, записываем новое значение баланса A, считываем баланс счета B, прибавляем к нему сумму перевода, записываем новое значение баланса счета B. Если все эти операции обернуть в транзакцию, то система будет гарантировать, что либо у вас все операции пройдут

успешно и деньги переведутся со счета A на счет B, либо вся транзакция целиком будет отменена, и состояние будет такое, как будто бы она никогда и не проводилась.

Хорошая транзакционная система соответствует требованиям ACID. Это акроним от слов

- Atomicity — атомарность,
- Consistency — согласованность,
- Isolation — изолированность,
- Durability — устойчивость.

Atomicity, атомарность, либо все операции внутри транзакции выполняются полностью, либо они все полностью не выполняются.
Consistency, согласованность,это даже немного шире, чем целостность. Если целостность подразумевает, что в базе данных у вас отсутствуют какие-то строки, которые ссылаются на другие строки в других таблицах, которых реально не существует, например, вы их удалили, а строки по-прежнему ссылаются, то Consistency подразумевает еще и какие-то требования бизнес-логики. Вот как в предыдущем примере с переводом денег. Ведь то, что мы, списав деньги с одного счета, должны положить их на другой — это наша бизнес-логика. В базе данных о ней ничего не известно, но тем не менее, она нам может помогать сохранить эту целостность в частности посредством транзакций.
Isolation, изолированность, подразумевает то, что если транзакции выполняются параллельно, несколько, то они выполняются так, как будто бы других транзакций одновременно не выполняется, то есть они не видят изменения друг друга.
Durability, устойчивость, подразумевает, что если система вам сообщила, что транзакция завершена, то она действительно завершена. Даже если в этот момент произойдет какой-то сбой после получения вами подтверждения, то этот сбой никак не повлияет на транзакцию.

Пока транзакции выполняются друг за другом, никаких проблем нет. Но в реальных системах у вас много процессоров, много процессов, которые одновременно пишут в базу данных, и это может приводить к различным артефактам, например, потерянному обновлению.
Допустим, две транзакции у вас пытаются изменить одно и то же поле: прибавить к нему 10, а другая пытается прибавить к нему 15. Если бы эти транзакции выполнялись друг за другом, то в итоге первая бы прибавила 10, сохранила это значение, вторая бы прибавила 15, сохранила это значение, и в итоге мы прибавили бы 25. Но если эти транзакции выполняются одновременно, то может получиться так, что первая прочтет старое значение, прибавит к нему 10, в это время вторая прочтет старое значение, поскольку первая еще не записала, прибавит к нему 15. Затем первая запишет новое значение, а затем вторая запишет новое значение. И получится, что мы не прибавим 25, а мы запишем старое значение + 10, а потом перетрем его старым значением + 15, то есть прибавим 15, а не 25.
Также один из артефактов называется «грязное» чтение. Это когда одна транзакция изменяет какие-то данные, затем другая транзакция их читает уже измененные данные. Но затем первая транзакция по каким-то причинам не может завершиться и полностью откатывает изменения. Но вторая-то уже прочла эти изменения, и у нас в итоге прочитаны данные, которых реально уже в базе данных нет.
Также один из артефактов — неповторяющееся чтение. Это когда одна транзакция прочитала какие-то данные, затем вторая их изменила, подтвердила эти данные, действительно внеслись изменения в базу данных. И затем первая транзакция опять читает те же самые строки и видит,

что значения полей уже другие. То есть два последовательных чтения в одной транзакции видят разные данные.
Также иногда имеет место фантомное чтение. Транзакция выбирает данные по какому-то условию и получает набор строк. Затем другая транзакция добавляет или изменяет строки, подходящие под это условие. И затем первая транзакция опять читает набор строк по тем же условиям, но получает уже совершенно другой набор строк.
Для решения этих проблем, которые возникают только в случае одновременных транзакций, в базах данных придумали **уровни изоляции**. Чем выше уровень изоляции, тем больше проблем на нем решается. Но это происходит за счет уменьшения количества параллельно выполняющихся транзакций.
Как решает проблему изоляции Oracle? Когда транзакция изменяет какие-то строки, они не пишутся в базу данных непосредственно, а создаются их копии, видные только этой транзакции. Но если две транзакции изменяют одни и те же строки, им обеим создаются свои копии, затем, когда первая из транзакций завершается, она записывает эти данные в базу, но вторая транзакция при этом уже завершиться не может из-за того, что она бы перетерла своим обновлением обновление первой транзакции. И поэтому эта транзакция полностью отменяется.

1.4. Индексы

Индекс используется для повышения скорости поиска данных.
Представим, что у нас есть база данных пользователей, в которой находятся миллионы записей, и мы хотим найти всех пользователей с возрастом 65 лет. Чтобы совершить эту операцию поиска, БД нужно перебрать все строки и для каждой строки проверить, является ли возраст пользователя цифрой 65 или нет.
Если по полю age построить индекс, то система управления базой данных создаст определенную структуру, оптимизированную под поиск. Как правило, это сбалансированное двоичное дерево поиска. В этом дереве будут перечислены все встречающиеся в базе данных значения поля age, они будут упорядоченны определенным образом, и для каждого значения будут ссылки на те строки, где оно встречается. Эта структура позволит очень быстро искать всех пользователей с определенным возрастом, с возрастом больше какого-то числа или меньше. Например, чтобы найти всех пользователей с возрастом 65 лет в таблице из миллиарда строк, нам придется сделать миллиард сравнений. Но если по этому полю есть индекс, то число сравнений будет менее 30.
Также индекс может быть составным, по нескольким колонкам, двум и более. В случае составного индекса важен порядок, в котором указаны колонки.
У книги есть автор и год издания. Допустим, мы сделаем составной индекс по автору и году. По этому индексу будет очень быстро найти все книги определенного автора за определенный год, но при этом индекс не будет использоваться, если мы просто захотим найти все книги за какой-то год. Внутри индекса по автору и году книги будут упорядочены сначала по автору, а внутри одного и того же автора — по году. Соответственно, действительно легко выбрать блок книг одного автора, или блок книг одного автора за определенный год. Но вот книги одного и того же года разбросаны по разным авторам. И системе управления базой данных пришлось бы либо брать блоки всех авторов и смотреть в нем все книги за определенный год, либо просто не использовать этот индекс, и как правило, базы данных так и делают.

Вообще индексов у таблицы может быть сколько угодно, но у индексов есть и минусы:

- Индекс ускоряет поиск, но не делает это бесплатно. Он замедляет модификацию данных, потому что требует время на свое обслуживание. То есть если вы добавите строки, удалите строки или измените какие-то строки, данные которых используются в этом индексе, индекс будет вынужден перестроиться.
- Это займет какое-то время, хотя и не очень большое. К тому же индекс занимает место на диске и в памяти. Соответственно, когда вы создаете индексы, вам нужно подумать, действительно ли он вам нужен, потому что он ускоряет поиск, но не задаром.

Также индекс может быть уникальным. Такой индекс отличается от обычного тем, что он не только упорядочивает данные, но еще и запрещает два одинаковых значения в том поле, по которому построен индекс. Это может помочь нам контролем целостности.

1.5. Нереляционные базы данных

Тогда как у всех реляционных баз данных есть общие признаки, то термин NoSQL обозначает по сути все, что не является реляционной базой данных.
С 2000 годов объем данных начал лавинообразно расти. Появляется термин Big Data, данные очень разнородные, их много, для их обработки нужны много компьютеров, кластера, параллельные вычисления.

- Реляционные БД очень сложны по своему устройству и поэтому плохо масштабируются на разные сервера.
- Все таблицы в реляционных базах данных имеют жесткую структуру. Это значит, что все записи в них одинаковой структуры. И когда появляется необходимость в разнородных данных, их не очень удобно хранить в таких таблицах.

Реляционная БД	**NoSQL типа «ключ-значение»**
Состоит из таблиц, таблицы состоят из строк и колонок, строки одной таблицы одинаковы по структуре.	Содержат коллекции, которые содержат записи, записи могут иметь разную структуру.
Схема данных определена заранее. Она строго типизирована, существуют отношения и ограничения.	Нет схемы данных, ничего заранее не определено, нет ограничений и отношений, из-за этого нет контроля целостности. У любой записи могут быть абсолютно любые поля.
Используется нормализация, чтобы избежать дублирования одних и тех же данных. Нормализация порождает отношения между таблицами. Данные одной записи хранятся по разным таблицам.	Между коллекциями или значениями в одной коллекции связей и отношений нет, данные денормализованы
Работа с данными структуры выполняется через SQL запросы, и SQL почти одинаковый для любой реляционной БД.	Работа с данными через API, у каждой свое.
SQL-запрос извлекает данные из одной или из нескольких таблиц, чтобы получить финальный результат, используя объединение этих таблиц или оператор join. Сложные условия, сложная фильтрация, агрегация.	Простые условия типа key=const, иногда агрегация и вторичные индексы.
Данные разбиты по плоским таблицам и хранятся не совсем так, как они хранятся в приложении. Необходим мэппинг из таблиц в данные приложений.	Данные более эффективно отображаются в структуры приложения.

1.5.1. BASE

В NoSQL рассматривается набор свойств BASE:

- базовая доступность (basic availability);
- гибкое состояние (soft state);
- согласованность в конечном счете (eventual consistency).

Базовая доступность подразумевает, что каждый запрос гарантированно завершается, удачно или неудачно, но по крайней мере он действительно завершается с каким-то ответом.
Гибкое состояние подразумевает, что состояние системы с течением времени может изменяться само по себе, даже без ввода каких-то новых данных со стороны. Это происходит для достижения согласования данных.
Согласованность в конечном счете подразумевает, что данные в какой-то момент времени в разных узлах могут быть рассогласованы, но спустя время они все-таки приходят к согласованному состоянию.
Системы на основе BASE не могут использоваться в биржевых или банковских системах, потому что там нужны транзакции. Но достичь всех свойств ACID в больших распределенных системах с многомиллионной аудиторией практически невозможно, и поэтому NoSQL системы жертвуют согласованностью данных ради масштабируемости и увеличения доступности.

1.6. Основные виды NoSQL баз данных

1.6.1. Хранилище вида ”ключ-значение”

Типичными представителями являются MemcacheDB, Redix, Amazon DynamoDB и Berkeley DB. Самый простой вид нереляционной базы данных.
По сути, это просто словарь или ассоциативный массив, который позволяет по уникальному ключу сохранить какое-то значение, сохранить и получить. Значение в некоторых базах данных может быть только строкой, а в некоторых базах данных бывает разных типов. Из-за простоты, эти базы данных очень хорошо масштабируются на много серверов. Ключ можно поделить на какие-то составные части и решить, что все ключи, начинающиеся условно на А, мы храним на одном сервере, и так далее. Благодаря этому можно очень легко распределить ключи между группой серверов. Каждый сервер будет обслуживать свой набор ключей.
У этих баз данных нет схемы базы данных, нет связей между значениями, количество ключей ограничено только суммарной памятью у всех серверов в кластере.
Минусы:

- невозможно делать какие-то операции, кроме поиска по точному значению ключа;
- невозможно быстро анализировать информацию, находящуюся в базе данных: только полный перебор всех ключей с соответствующей скоростью;
- нет связей между ячейками.

Не могут быть заменой традиционных реляционных баз данных, но зато очень хороши в качестве кэшей.

1.6.2. Документоориентированные базы данных.

Типичные представители — MongoDB, CouchDB и Exist.
Более сложная версия хранилищ «ключ-значение». Данные (документы) представлены в виде дерева или массива деревьев, есть иерархия. Одна запись называется документом. Есть корневой узел, внутри него есть внутренние узлы и листовые узлы, которые содержат данные. Каждый документ может содержать произвольное количество произвольных полей, то есть записи тоже разнородны. Но у документоориентированных баз данных уже есть индексы, благодаря которым можно осуществлять быстрый поиск даже при достаточно сложной структуре хранилища.
Не являются заменой реляционных баз данных, но тем не менее уже позволяют делать какие-то выборки без полного перебора всех документов и хорошо подходят, когда требуется какое-то упорядоченное хранение информации, но тем не менее нет множества связей между данными, и не нужна какая-то статистика по этим данным. По-прежнему не требует определения какой-то схемы, каждый документ может состоять из любого количества уникальных полей.

1.6.3. Bigtable-подобные базы данных

Типичные представители: HBase, Cassandra, Hypertable и SimpleDB. У них есть общий прародитель, это Google Bigtable — эта проприетарная база данных используется внутри корпорации Google.
Эти базы данных хранят данные в виде разреженной матрицы. Столбцы и строки этой матрицы используются в качестве ключей. Матрицы бывают многомерными. Разреженная матрица: несмотря на большие размеры этой матрицы, большинство ее ячеек пусто, то есть данных в ней относительно общих ее размеров не так много.
Такие базы данных имеют много общего с документоориентированными базами данных и применяются для целей веб-индексирования или схожих задач, предполагающих огромные базы данных и большое число серверов. Эти базы данных хорошо масштабируются, достаточно быстрые и способны работать с действительно огромными базами данных, что и требуется при веб-индексировании, в частности.

1.6.4. Графовые базы данных

Типичные представители: FlockDB, Giraph, OrientDB, Neo4j.
Лучше всего подходят для тех задач, в которых данные уже по своей структуре представляют граф.
Данные хранятся в виде узлов и связей между ними. Например, в соцсетях пользователи могут быть вершинами графа, а дружеские отношения между ними — ребрами этого графа.
Быстрее, нагляднее, удобнее, чем реляционные базы данных. Графовые базы данных могут работать при этом с очень большими графами, которые целиком в память не помещаются.

2. MySQL

2.1. Создание баз и таблиц

2.1.1. Удаление БД

```
DROP DATABASE [IF EXISTS] db_name
```

[IF EXISTS] просто предотвращает ошибку, если мы пытаемся удалить несуществующую базу данных.

2.1.2. Создание БД

```
CREATE DATABASE db_name
```

Также можно указать дополнительные опции при создании базы данных — это кодировка по умолчанию, обычно это UTF-8 или UTF-8MB4, и способ сравнения строковых данных. Как правило, это UTF-8 или UTF-8MB general CI, то есть сравнение без учета регистра. CI — это case insensitive. Эти будут значения установлены по умолчанию. Но вы можете выбрать другие кодировки, например CP-1251 и другие способы сравнения, например, с учетом регистра. Точно так же, если написать CREATE TAPE DATABASE [IF NOT EXISTS], то при попытке выполнить этот оператор с именем уже существующей базы данных, ошибки не будет. Если этого не написать — то будет.

2.1.3. Создание таблиц внутри БД

Для этого используется оператор CREATE TABLE. Существует несколько способов использования CREATE TABLE.

```
CREATE TABLE
tbl_name (create_definition,...)
[table_options]
[partition_options]
```

Способ 1 Это создание таблицы с нуля, но с перечислением всех ее настроек.

```
CREATE [TEMPORARY] TABLE
tbl_name (create_definition,...)
[table_options]
[partition_options]
```

Способ 2 Временная таблица отличается от обычной тем, что она создается внутри сессии, из других сессий ее не видно, а при завершении сессии она удаляется вместе со всеми данными. (create_definiton,...) — это названия колонок, индексов и так далее;
table_options – движок таблицы, способы кодировок по умолчанию и так далее. partition_options – разбиение таблиц на партиции, это нужно только для уже каких-то больших таблиц. Эти знания вы уже сможете почерпнуть в документации.

2.1.4. create_definition

create_definition — это либо перечисление колонок, либо перечисление индексов или ключей, либо перечисление ограничений. Все это может находиться друг за другом в произвольном порядке.
column_definition – определение колонки.

```
create_definition = col_name column_definition
```

Определение колонки состоит из типа данных, то есть перечисляя колонки таблицы, вы перечисляете их имена и column_definition, то есть описание колонки. Описание колонки состоит из описания типа данных этой колонки, может ли оно содержать пустые значения, или не может (NULL или NOT NULL), значение по умолчанию, которое присвоится этому полю, если вставляя новый ряд, вы не укажете значения для конкретно этой колонки, также является ли эта колонка автоинкрементной, то есть увеличивает ли она автоматически свое значение, если не передать конкретное, является ли она уникальной, является ли она первичным ключом, также можно добавить комментарий для колонки, что полезно для последующих, когда вы будете читать свою базу данных, смотреть в дальнейшем, поможет вам вспомнить, что же это за колонка.

```
column_definition = data_type [NOT NULL | NULL]
[DEFAULT default_value][AUTO_INCREMENT]
[UNIQUE[KEY][[PRIMARY]KEY]
[COMMENT'string'][referene_definition]
```

Типы данных у колонок:

- числовые типы данных, прежде всего это целочисленные данные TINYINT, SMALLINT, MEDIUMINT, INT и BIGINT: сответственно 1 байт, 2 байта, 3 байта, 4 байта или 8 байт. Также они бывают SIGNED и UNSIGNED, то есть знаковые и беззнаковые. Существует также тип данных с фиксированной точкой — это DECIMAL, или NUMERIC, он используется, как правило, в операциях с валютами. Также существуют типы данных с плавающей точкой — это FLOAT и DOUBLE, соответственно FLOAT — 4 байта, DOUBLE — 8 байт.
- Существуют много типов данных для хранения различных дат — это DATE, DATETIME, TIMESTAMP, TIME, YEAR.
- Также существуют строковые типы данных. Это, во-первых, CHAR и VARCHAR, CHAR имеет фиксированную длину, независимо от того, сколько фактически символов вы храните, остальные будут просто забиваться пустыми значениями, но длина зарезервированная будет всегда одинаковая. VARCHAR — она изменяется в зависимости от того, сколько символов вы сохраните. Также есть BINARY и VARBINARY, отличаются только тем, что там хранятся двоичные данные в отличие от CHAR, где хранят текстовые. Также существуют BLOB для хранения больших двоичных данных, TEXT для хранения больших текстовых данных. И тоже есть вариации BLOB'а и TEXT'а, на самом деле, различной длины.

- Существует ENUM — это перечисление, в этой колонке может храниться одно из перечисленных заранее значений.
- Существует SET, который похож на перечисление, но только может хранить одновременно несколько из перечисленных значений, а также существует JSON и различные геотипы.

В create_definition могут быть индексы и ограничения.

```
[CONSTRAINT [symbol]] PRIMARY KEY
[index_type] (index_col_type,...)
[index_option] ...

{INDEX|KEY}[index_name][index_type]
(index_col_type,...) [index_option] ...
```

Например, мы можем указать первичный ключ с различными типами и опциями для первичного ключа, также можем указать обычный ключ, то есть index, также у этих индексов бывают типы: он может быть b3 — это b дерево, или hash, различные типы индексирования, и различные опции, например, комментарии к индексу можно добавить.
Также можно указать уникальный ключ и ограничения связи с другими таблицами, например, внешние ключи. Все это же можно указать прямо при определении колонки, то есть можно указать при определении колонки сразу, является ли она, например, уникальным ключом или нет, а можно отдельно сначала перечислить просто все колонки, а потом указать, какие из них уникальные, какие из них индексы.
Также у ключей бывает reference_definition, когда это внешний ключ, он может ссылаться на таблицу — это мы указываем в reference_definition.

```
reference_definition = REFERENCES tbl_name
(index_col_name,...) [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]
[ON DELETE reference_option][ON UPDATE reference_option]

reference_option = RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT
```

Указываем, на какую таблицу он ссылается, по каким колонкам сопоставляются данные у двух таблиц и что делать в случае удаления связанных данных. Это указывается в reference_option. Можно запретить удалять данные связанных таблиц, если данные одной таблицы используются в другой, можно запретить удалять эти данные. Или можно также удалять данные из связанных таблиц при удалении данных из исходных и так далее.
При создании таблицы можно указать table_options.
Основные — это AUTO_INCREMENT, можно указать начальные значения для AUTO_INCREMENT'а, можно указать кодировку по умолчанию, сравнение по умолчанию, точно так же, как это делается для всей базы данных, можно у отдельных таблиц поменять кодировку. Ну кодировка бывает UTF, UTF-8MB4, CP1251, не используйте LATIN1, потому что русский

язык там не поддерживается.
Также бывают способы сопоставления различных строковых данных между собой, также можно указать комментарии ко всей таблице, пояснить, что это за таблица, зачем она нужна, и движок таблицы — это с помощью какого движка сохраняются данные в этой таблице. Есть InnoDB который поддерживает транзакции, внешние ключи, есть устаревший MyISAM, раньше он часто использовался, сейчас уже вряд ли вы его встретите..

2.2. Создание баз и таблиц. Практика

У пользователя, под которым мы будем создавать БД, должно быть право CREATE для них. Пользователь uroot в MeSQL вполне подойдет. Создадим БД.

```
create database books;
```

Можем удалить эту базу данных. Для этого достаточно написать drop database books.
Укажем кодировку по умолчанию не utf8, а utf8mb4.

```
create database books default character set utf8mb4;
```

Перейдем в эту базу данных. Это позволит нам писать имена всех таблиц, не указывая базу данных дополнительно.

```
use books
```

Создадим таблицу издателей.

```
create table `publisher` (
    `publisher_id` int(10) unsigned not null auto_increment primary key,
    `name` varchar(512) not null,
    'site' varchar(128) null
    ) engine = InnoDB;
```

Здесь у нас в скобочках: create definition, то есть перечисление всех столбцов и ограничение этой таблицы. Создадим первый столбец. Это будет первичный ключ: целочисленный, без знака, и в нем не может быть пустых значений. Также мы скажем, что это поле увеличивает свое значение при вставке данных сюда. Если не указывать это поле, то все строчки будут нумероваться по

порядку. А также скажем, что это первичный ключ.
Дальше у издателя будет название. Это будет строка переменной длины, максимум 512 символов, и она тоже не может быть пустой. Больше никаких дополнительных ограничений мы не наложим.
Сайт у нас тоже будет строка, varchar, 128 символов максимум, и она может быть пустая.
Укажем движок InnoDB.
Создадим таблицу жанров.

```
create table `genre` (
    `id` int(10) unsigned not null auto_increment,
    `name` varchar(64) not null comment 'название жанра',
    primary key ('genre_id'),
    unique_key 'uq_name' ('name'),
    ) auto_increment = 10;
```

Если в таблице издателей мы указали, что колонка является первичным ключом прямо в описании колонки, давайте в таблице издательств укажем это отдельной конструкцией, а также добавим уникальный ключ и комментарии.
Добавим колонку id. Она целочисленная, десять символов, без знака, не может быть пуста, автоинкремент, но мы не напишем, что она primary key.
Дальше у нас будет имя, varchar. Допустим, имя издательства — это 64 символа. И оно пусть тоже не может быть пустым, все-таки у издательства должно быть имя, но мы еще добавим комментарий. Просто достаточно написать слово comment, и это будет 'название жанра', чтобы всем было понятно, что же у нас одначает поле ‘name‘ в таблице жанров.
Добавим отдельно указание, что есть первичный ключ и уникальный ключ. Для этого напишем primary key и укажем, какая колонка у нас является первичным ключом. Имя у нас является уникальным ключом. Также можем указать какие-нибудь дополнительные опции, например, указать, что, когда мы вставим первую строчку в эту табличку, у нее айдишник будет десять, а не один, как по умолчанию.
Создадим теперь таблицу авторов и опустим все, что мы ранее экранировали.

```
create table author (
    author_id int unsigned not null auto_increment primary key,
    name varchar(512) not null,
    gender enum('male', 'female') default 'male',
    birthday date,
    );
```

author_id у нас будет int, не указываем, сколько длина, без знака, not null, иначе по умолчанию будет null.
Дальше мы оставим то же самое имя name, varchar нельзя опустить, как и максимальную длину. Также добавим пол в форме перечисления. Добавим день рождения, birthday, и это будет просто дата. Она может быть null, то есть мы не знаем когда автор родился.
Таблица книг очень большая. Создадим ее на основе другой таблицы.

```
create table book like course.book
```

У нас создалась на основе таблицы из другой базы данных таблица в нашей. Но при создании копии с таблицы копии ограничений не создадутся.
Создадим табличку связи между книгами и авторами.

```
create table book_author (
    book_id int(10) unsigned not null,
    author_id int(10) unsigned not null,
    constraint 'fk_book' foreign key ('book_id') references 'book' ('book_id') on
    delete cascade on update cascade,
    constraint 'fk_book' foreign key ('author_id') references 'author' ('author_id')
    on delete cascade on update cascade,
    );
```

У нас в ней не будет первичного ключа. Она будет только ссылаться на авторов и на книги. Аллотируем и добавим ограничения. Напишем constraint, назовём его, это у нас будет внешний ключ, fk_book мы его назовём, и он, напишем, что это foreign key, собственно, внешний ключ. И дальше мы указываем, с какого поля ссылается наша таблица. Это поле book_id нашей таблицы. И укажем, куда оно ссылается. Для этого мы используем ключевое слово references и указываем, на какую таблицу и на какое поле в ней наше поле book_id ссылается. Ссылается оно на таблицу book и поле book_id уже в той таблице. Заодно мы укажем, что случается, если мы удалим книгу, на которую эта таблица ссылается. Мы удалим записи в нашей таблице, если удалится книга, на которую эти записи ссылаются. И в случае обновления мы тоже запишем on update тоже cascade. И то же самое сделаем для автора.

2.3. Редактирование таблиц в MySQL Workbench

Для этого мы будем использовать SQL Workbench. Для начала мы должны создать соединение с MySQL сервером.

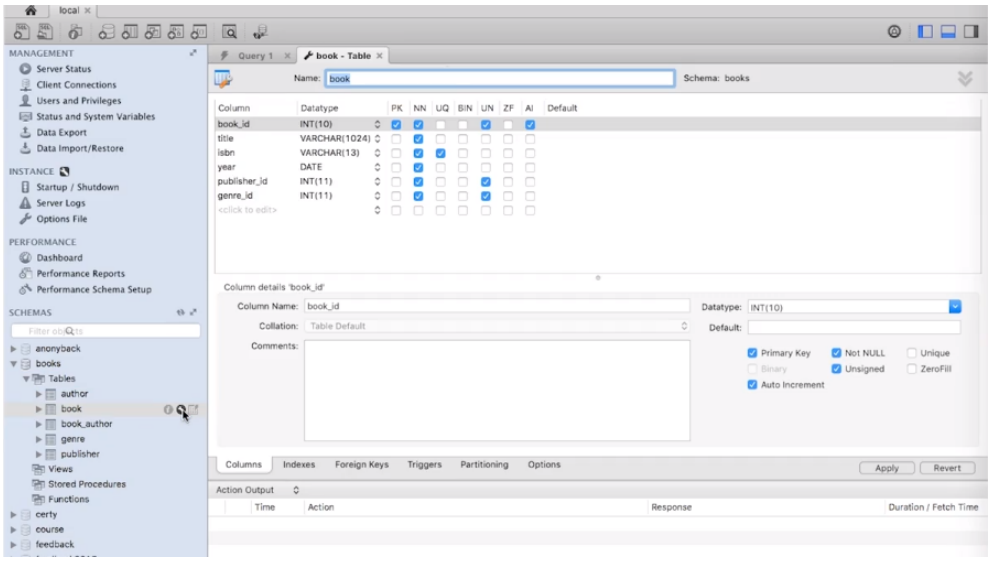

Рис. 2

Выберем таблицу book, у которой потерялись связи с другими таблицами. Нажмём на кнопочку гаечного ключа, рядом с ней находящуюся, и откроем режим редактирования.

Здесь мы видим прежде всего все колонки этой таблицы с их типами данных. И тут можем выбирать их, изменять эти типы данных, а также галочки напротив различных значений. Например, PK означает первичный ключ, NN — Not Null, UK — Unique, и так далее, UN — Unsigned, AI — Auto Increment. Вот здесь набор галочек позволяет полностью нам понять, что же из себя представляет каждая из этих колонок. Также здесь можно дописать к ней комментарии, и так далее.

Также есть вкладка индексов, где показаны все индексы этой таблицы с их типами, с колонками, по которым они построены, и внешние ключи. Внешних ключей у нас у этой таблицы нет, а должны быть.

Давайте добавим. Во-первых, у нас должен быть внешний ключ на таблицу издательств за таблицей книг. Назовём его fk_publisher и скажем, что ссылается он на таблицу books, publisher. Дальше мы укажем, по какой колонке связь идёт. Вот мы указываем, что наша колонка publisher_id связывается с колонкой publisher_id, но уже в таблице publisher. Дальше мы укажем, что случается при обновлении и удалении. Вот здесь вот достаточно проставить, у нас будет cascade.

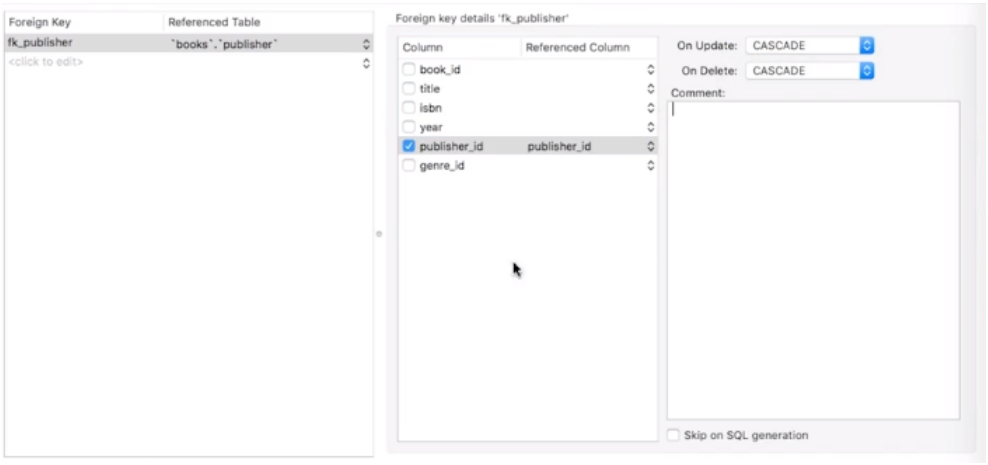

Рис. 3

Также мы добавим ещё ключ на жанры, это будет fk_genre, укажем, что ссылается он на таблицу жанров, и связь происходит по genre_id. Точно так же добавим, что cascade для update и delete. Нажав на кнопку Apply, мы увидим SQL, который собирается выполнять SQL Workbench, то есть наши действия в графическом интерфейсе она перевела в SQL, с помощью которого и будет обновлена база данных. Вот мы видим, что мы делаем ALTER TABLE books book. ALTER TABLE значит изменить таблицу. И добавим ADD CONSTRAINT. И дальше указываем, какие, собственно, ограничения. По сути запись вот этого CONSTRAINT самого выглядит так же, как она выглядела бы в создании таблицы, только здесь вот добавляется ключевое слово ADD. То есть примерно так же вы бы написали в CREATE, но в ALTER мы добавляем ключевое слово ADD и понимаем, что мы добавляем какие-то новые значения. Мы можем начать Apply и этот SQL будет выполнен.

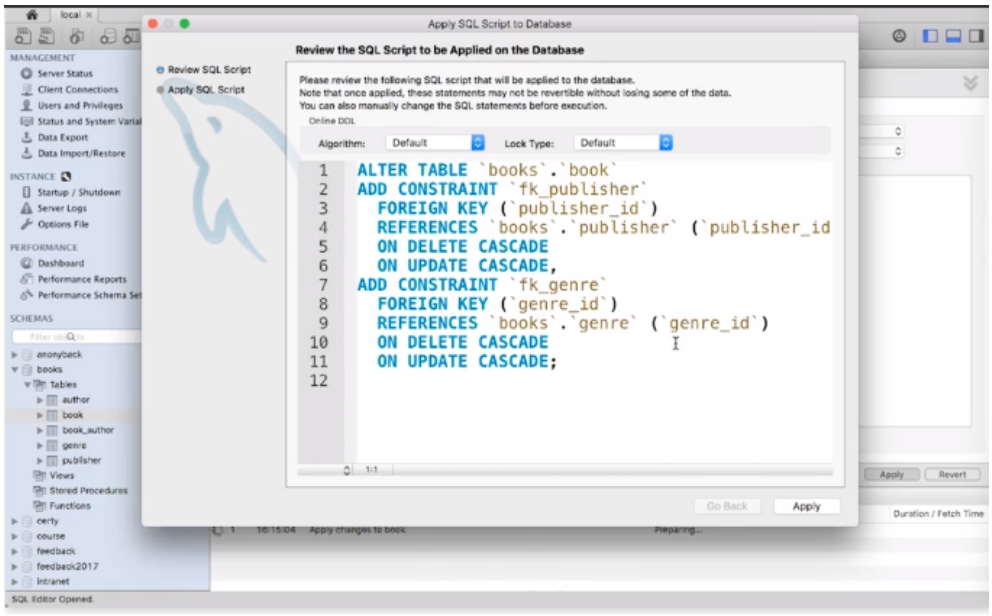

Рис. 4

Давайте отредактируем какую-то таблицу. Например, таблицу авторов, попробуем к ней добавить какие-то колонки дополнительные. Для этого мы точно так же нажимаем на редактировать таблицу авторов. Вот у нас есть имя, пол, дата рождения. Давайте, например, добавим к автору его рейтинг. Добавим сюда поле rate, выберем тип. Рейтинг не может быть, допустим, с пустым значением, а по умолчанию у него будет ноль. И добавим для рейтинга индекс.

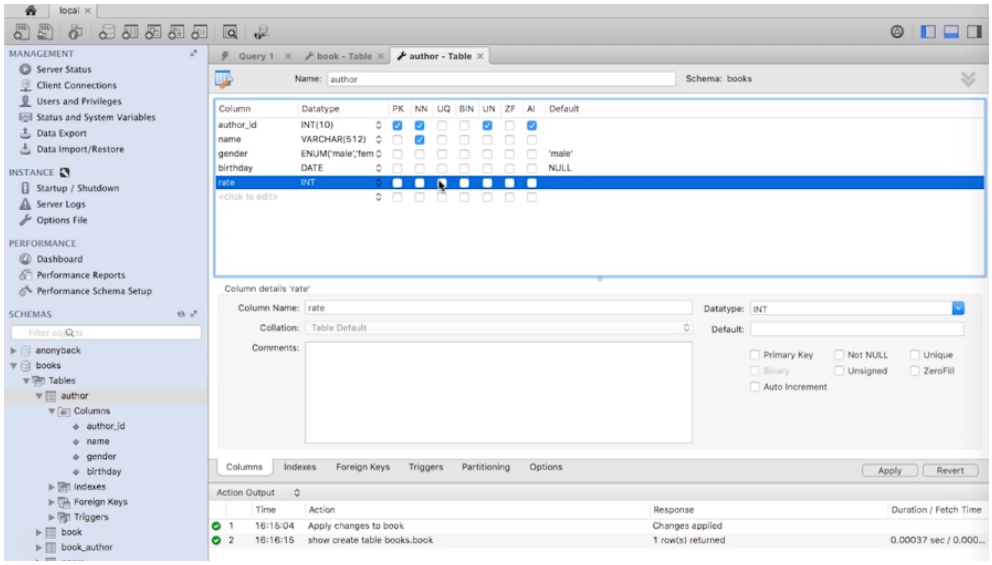

Рис. 5

Теперь нажимаем apply — и мы видим изменения, которые хочет опять внести Workbench в нашу таблицу. И здесь мы добавляем, в какое место она добавляется, поскольку это новая колонка, мы можем добавить её не только в конце, но и после какой-то любой другой колонки. Например, мы её добавляем после колонки birthday. Также мы добавляем индекс, указываем название индекса и по каким полям этот индекс строится. Применяем — и в нашей таблице теперь появляется новое поле rate. То есть мы перейдём к колонке, теперь мы видим, что оно всё есть.

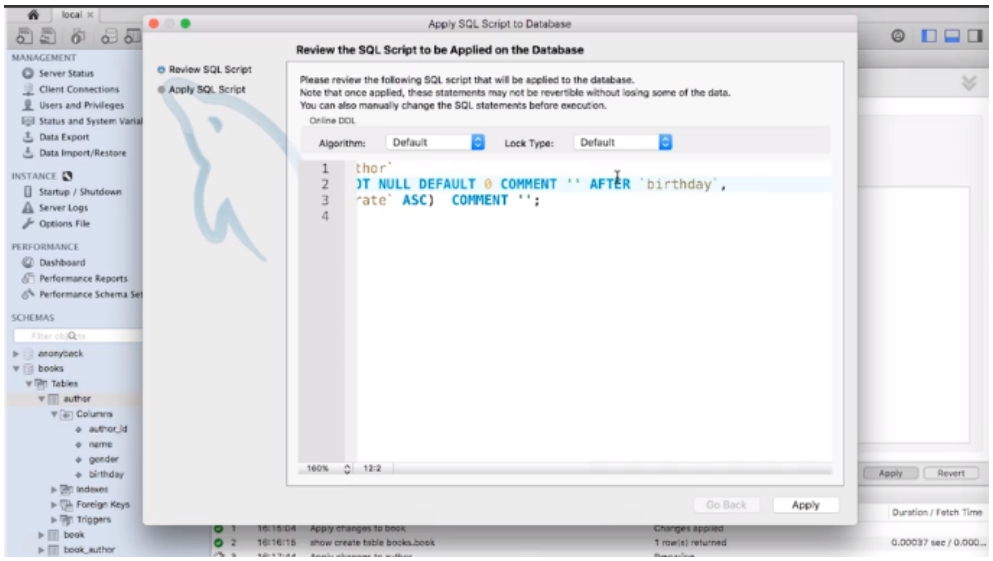

Рис. 6

2.4. Изменение баз и таблиц

Начнем с alter database.

```
ALTER DATABASE db_name
[alter_specication] ...

alter_specication:
[DEFAULT] CHARACTER SET [=] charset_name
| [DEFAULT] COLLATE [=] collation_name
```

alter table позволяет изменить таблицу уже существующую.

```
ALTER TABLE tbl_name
[alter_specication [, alter_specication] ...]

alter_specication = ADD, CHANGE, DROP
```

Что такое alter specification? Это любые действия с таблицей: добавление, удаление, изменение колонок, индексов и параметров таблицы.
Давайте рассмотрим один из вариантов — это add — добавить, с помощью которого можно добавить колонку, индекс или ограничение. Для начала add column.

```
ADD [COLUMN] col_name column_denition
[FIRST | AFTER col_name]
```

Добавляется имя колонки, column definition, такое же, как в create, и указывается, куда она добавляется: либо first — в начало таблицы, перед всеми остальными колонками, либо after и имя колонки, после которой добавляется новая колонка.

```
ADD {INDEX|KEY} [index_name] [index_type]
(index_col_name,...) [index_option]
```

Также мы можем добавить ключ или индекс с помощью add index, добавить ограничение: например, первичный ключ с помощью add primary key, добавить уникальный индекс, добавить внешний ключ с помощью add foreign key и так далее.

```
ADD [CONSTRAINT [symbol]] PRIMARY KEY
[index_type] (index_col_name,...) [index_option]
```

Также мы можем изменить уже существующую колонку с помощью change в alter specification.

```
CHANGE [COLUMN] old_col_name new_col_name
column_denition [FIRST|AFTER col_name]
```

Также в alter specification может быть drop. С помощью него мы можем удалить колонку, индекс, ключ, или внешний ключ, или вообще любые другие ограничения.

```
DROP [COLUMN] col_name

DROP {INDEX|KEY} index_name

DROP PRIMARY KEY

DROP FOREIGN KEY fk_symbol
```

Также в alter specification могут быть какие-то опции таблицы. С помощью него мы можем изменить значения автоинкремента, изменить кодировку по умолчанию в таблице, добавить или изменить комментарий, поменять даже движок таблицы.

Дальше давайте рассмотрим действия над таблицами: drop, rename и truncate. С помощью drop мы можем удалить таблицу или временную таблицу.

```
DROP [TEMPORARY] TABLE [IF EXISTS]
tbl_name [, tbl_name] ...;
```

С помощью rename мы можем переименовать одну таблицу и сохранить ее под другим именем. Также мы можем с помощью rename, например, поменять две таблицы местами. Но это можно сделать только через промежуточную таблицу.

```
RENAME TABLE tbl_name TO new_tbl_name
[, tbl_name2 TO new_tbl_name2] ...
```

Также можно очистить таблицу от данных с помощью truncate. Но на самом деле truncate не удаляет данные, а внутри себя он полностью удаляет таблицу и пересоздает ее, поэтому это достаточно быстро, но не работает для indb с внешними ключами.

```
TRUNCATE [TABLE] tbl_name
```

2.5. Типы данных столбцов

Есть три наиболее часто используемые их группы типов данных, которые могут храниться в столбцах MySQL. Это числовые типы, строковые типы и типы данных для хранения даты и времени. Также есть тип JSON, который появился относительно недавно, и есть достаточно специфичные типы данных для хранения геоданных.

2.5.1. Числовые типы данных

Начнём с числовых и с их подраздела — целочисленных типов данных. Это точные типы данных, они позволяют хранить число точно, и их существует пять видов:

- TINYINT
- SMALLINT
- MEDIUMINT
- INT

- BIGINT

Отличаются диапазоном хранимых значений и, соотвественно, занимаемым местом в базе данных: от одного байта для TINYINT'а до восьми байт для BIGINT'а. Также эти числа могут быть со знаком или без знака. Если это знаковое число, то диапазон допустимых значений распределяется равномерно вокруг нуля, а если это число без знаков, то весь этот диапазон сдвигается в положительную плоскость, и вы не можете хранить отрицательные, зато можете хранить больше положительных чисел.

Тип	Байт	Signed	Unsigned
TINYINT	1	-128 ... 127	0 ... 255
SMALLINT	2	-32768 ... 32767	0 ... 65535
MEDIUMINT	3	-8388608 ... 8388607	0 ... 16777215
INT	4	-2147483648 ... 2147483647	0 ... 4294967295
BIGINT	8	-9223372036854775808 ... 9223372036854775807	0 ... 18446744073709551615

Рис. 7

Типы с фиксированной точкой – это числовой тип данных, который позволяет хранить дробные числа. В MySQL существуют DECIMAL и NUMERIC, и они, на самом деле, одно и то же. Хранятся эти данные в двоичном виде в базе и условно занимают по четыре байта на каждые девять цифр, причём отдельно четыре байта для каждых девяти цифр целой части и по четыре байта для каждой из девяти цифр дробной части.
Также у этих DECIMAL указывается два параметра при создании колонки этого типа: M и D. M — это сколько цифр всего в этом числе, а D — сколько из них после запятой. Так, например, число DECIMAL (5,2) может хранить числа от −999.99 до +999.99.
Дальше поговорим о числах с плавающей точкой. Они похожи на предыдущие. Их существует два типа: FLOAT и DOUBLE. FLOAT — одинарная точность, занимает четыре байта в базе; DOUBLE — двойная точность, восемь байт. Соответственно, тип FLOAT точен где-то до седьмого знака после запятой, тип DOUBLE где-то до 15-го. Поскольку эти числа неточные, могут возникнуть проблемы при сравнении. Такие числа лучше не сравнивать прямым равенством, а смотреть их попадание в какой-то диапазон.
Также существует тип BIT. Это тип для хранения двоичных значений. Можно указать BIT(M), M — какое-то число, он будет позволять хранить M-битные значения. При этом M может быть от 1 до 64, то есть максимум можно хранить 64-битные значения. Также существует литерал, начинающийся с буквы b и дальше там единицы и нули, который позволяет записать двоичное число в двоичном виде с помощью единиц и нулей.

2.6. Дата и время

Типы для хранения даты и времени:

- DATE
- DATETIME
- TIMESTAMP
- TIME
- YEAR.

DATETIME занимает в базе пять байт и может занимать дополнительные от нуля до трех байт, если у него есть дробная часть секунд. Допустимый диапазон от 1000-го года до 9999-го года, плюс время.
Также существует тип DATE, не хранит время. Диапазон там такой же, и занимает он в базе три байта.
Также есть тип TIMESTAMP, он очень похож на DATETIME в том смысле, что он хранит тоже дату и время, занимает четыре байта, плюс от нуля до трех байт для дробной части секунд, если они есть. У него немножко другой диапазон. Он может хранить от 1970-го года до 2038-го года. И внутри себя он хранит количество секунд, прошедших с 1 января 1970-го, и хранит он их в часовой зоне UTC. Если у вас в MySQL установлена ваша таймзона другая, то при записи вашего времени, оно конвертируется в UTC, а при чтении конвертируется обратно. Поэтому если записать какое-то значение, а потом изменить настройки MySQL, можно прочитать совершенно другое.
Формат, в котором может быть записана дата в MySQL, может представлять год из четырех или из двух цифр, месяц и день, а также часы, минуты, секунды, и между ними могут быть любые разделители. Также дату от времени можно разделить буквой T, а вот дробную часть секунд от основных секунд можно отделить только точкой. Также можно указывать дату и время цифрами без каких бы то ни было разделителей. Если вы укажете дату, которой не существует и попытаетесь её сохранить в MySQL, то, в зависимости от настроек, у вас получится либо ошибочная дата, либо, если включён строгий режим, то будет ошибка.
Если вы захотите сохранять год, передавая только две цифры года, то если это будут цифры от 0 до 69, то будут иметься в виду года от 2000-го до 2069-го. Если вы будете передавать цифры от 70 до 99, то это будет означать год от 1970-го до 1999-го.
Также типы TIMESTAMP и DATETIME, которые, как мы знаем, могут хранить дату и одновременно время, могут автоматически инициализироваться или обновляться текущим временем. Это достаточно удобно для хранения каких-то значений, которые должны быть, например, даты последнего изменения записи.
Также существует тип TIME для хранения, как вы понимаете, только времени. А точнее, не времени в пределах дня, а какого-то временного интервала. Занимает в памяти три байта плюс дробная часть и может хранить от -838 часов до $+838$ часов. Соответственно, может записываться тоже в разных форматах, как с указанием трёх цифр для часов, минут и секунд, также с указанием дней через пробел перед часами для большего удобства. Ну и допускаются также какие-то сокращённые форматы, где можно опускать либо дни, либо часы, минуты, и оставлять,

например, можно даже просто указать количество секунд. Также допускается вариант записи без каких бы то ни было разделителей, как и во всех таких типах.
Также существует тип данных YEAR, который позволяет хранить только год от 1901-го до 2155-го, и занимает в базе этот тип один байт.

2.6.1. Символьные типы данных

Символьные типы данных:

- CHAR
- VARCHAR
- BINARY
- VARBINARY
- несколько видов BLOB
- несколько видов TEXT
- ENUM
- SET.

Типы CHAR и VARCHAR очень похожи друг на друга, но хранятся по-разному.
CHAR, в скобочках можно указать максимальное количество символов, – это строка фиксированной длины. Если вы сохраняете значение меньшей длины, чем указали при создании, то оно дополняется справа пробелами. При получении эти пробелы отрезаются. При этом M – максимальная длина строки, может быть от 0 до 255 символов, и занимать в базе такая строка будет M*W байт, где W – это размер одного символа в кодировке. Если это кодировка типа windows1251 или latin1, то это один байт, в UTF-8 это три байта на символ, И UTF-8mb4 – четыре байта на символ.
VARCHAR очень похож на CHAR, но это строка переменной длины. Она практически такая же, только не дополняется пробелами, то есть сохраняет ровно столько символов, сколько вы туда запишите. При этом указывается максимальная длина строки, и нельзя сохранить строку больше этого. Занимает L+1 или L+2 байта, в зависимости от M: максимальной длины. Количество байт, которые занимает строка, зависит от кодировки, и несмотря на то, что нет явного ограничения на количество символов в строке VARCHAR, есть ограничения на максимальное количество байт в одном ряду в MySQL, это 65535 байт. Причем это для всех колонок в ряду, поэтому это накладывает ограничения на суммарную длину всех колонок. Соответственно, если вы сохраняете в VARCHAR строку длиннее чем M – максимальная длина, которую вы указали при создании, то строка либо обрезается, либо происходит ошибка, в зависимости от настроек MySQL.
Также есть типы BINARY и VARBINARY, они очень похожи на CHAR и VARCHAR, отличаются тем, что хранят не символы, а байты.
BINARY точно так же, как и CHAR, дополняется справа, но нулями, а не пробелами, эти нули

при извлечении не обрезаются. Это может породить сложности.
Также существуют типы данных BLOB и TEXT. BLOB – это binary large object, предназначен для сохранения больших двоичных данных. Есть в четырех вариантах: TINYBLOB, BLOB, MEDIUMBLOB и LONGBLOB. Отличаются они максимально допустимым размером сохраняемых данных – от 2 в восьмой степени байт до 2 в тридцать второй степени байт.
Также и с TEXT. Есть TINYTEXT, TEXT, MEDUIMTEXT и LONGTEXT. Этот тип служит для сохранения больших текстовых данных. Поскольку данные в этих полях могут быть действительно большими, то несмотря на размер самого поля, который может сохранить много данных, у вас может не получиться их туда сохранить, потому что вы можете упереться в размер памяти, доступной на сервере, размер коммуникационных буферов этого сервера.

Есть другие нюансы с такими большими полями. Например, при сортировке учитывается не все поле, а только первые несколько его символов, но определенные, на самом деле, настройками max_sort_length, настройками MySQL.
Также есть поле ENUM – перечисление. В этом поле при создании можно перечислить строковые значения, которые в нем допускаются. То есть, вы присваиваете колонке одно из этих значений, а внутри оно преобразуется в индекс этого значения и сохранится в базе в виде одного или двух байт, в зависимости от того, сколько всего значений есть в этом перечислении, сколько вы их указали при создании, если до 255, то это будет один байт, если больше, то 2 байта. Всего максимум можно указать 65535 значений. Если вы начнете сортировать по этой колонке, то будет сортироваться она не по строковому представлению, а по индексу. Поэтому если вы сделаете ENUM, например (b;a;c), из трех значений b,a,c, именно им в таком порядке, то значение b у вас при сортировке окажется выше, чем значение a, и в свою очередь они все выше, чем значение c, ровно потому что не по символьному представлению, а по числовому индексу, по порядку внутри перечисления. И также лучше не делать никогда значениями ENUM цифры, потому что, когда вы сохраняете и передаете это значение ENUM в MySQL, вы можете передать как и его строковое представление, так, в принципе, и индекс. А когда у вас строковым представлением тоже является цифра, MySQL может перепутать, и вы можете получить не то, что сохраняли.
Также есть тип данных SET – множество. Он похож на ENUM в том смысле, что это – набор строковых значений, но только может хранится не одно из них, а несколько одновременно выбранных в каждой ячейке. По сути, за каждым значением закрепляется какой-то бит в последовательности битов и записываются номера битов этих значений, и получается одно число, тоже число, и оно сохраняется в базе данных. Соответственно, может занимать один, два, три, четыре или восемь байт в зависимости от того, сколько всего разных значений вы перечислили при создании SETа. Максимум может быть 64 значения. Каждый элемент представлен каким-то своим битом.

2.6.2. JSON

Типе данных JSON не так давно появился в MySQL, хотя, в принципе, и до этого вы спокойно могли сохранять JSON, например, в поле с типом TEXT. JSON – это, по сути, строка, но тип JSON всё-таки отличается от просто сохранения строки в текст, так как, если вы сохраните какую-то строку в JSON, то, во-первых, она будет проверена на соответствие формату перед тем, как попасть в базу данных. Также поля типа JSON внутри хранятся в специальном компактном формате, который быстрее читается, и удобнее с ним работать базе данных. И к тому же для

работы с полями типа JSON есть специальные функции, которые позволяют манипулировать подэлементами, находящимися внутри поля JSON. По полям типа JSON нельзя сделать индексы, но можно сделать автогенерируемые колонки. И вот уже по автогенерируемым колонкам, которые будут генерироваться на основе каких-то полей изнутри JSON, можно сделать индексы.

3. Работа с данными

Есть термин CRUD – это сокращение от английских слов create, read, update и delete, создавать, читать, обновлять и удалять. Это – акроним, обозначающий четыре базовых функции, используемых при работе с хранилищами данных. В MySQL CRUD реализуется с помощью четырех видов запросов: создание – insert, чтение – select, редактирование – update и удаление – это запрос delete.
Начнём с вставки новых данных с помощью insert.

```
INSERT [IGNORE] [INTO] tbl_name
[(col_name [, col_name] ...)]
{VALUES | VALUE} (value_list) [, (value_list)] ...
[ON DUPLICATE KEY UPDATE assignment_list]

value: {expr | DEFAULT}

value_list: value [, value] ...
```

Значение для каждой колонки представляется либо выражением, это может быть число, строка, формулы или словом default, что означает вставить значение колонки, которое прописано в ней по умолчанию.
Также существует другой синтаксис.

```
INSERT [IGNORE] [INTO] tbl_name
SET assignment_list
[ON DUPLICATE KEY UPDATE assignment_list]

value: {expr | DEFAULT}assignment:

col_name = valueassignment_list:

assignment [, assignment] ...
```

Существует также третий вид INSERT, когда данные в таблицу вставляются на основе данных, выбранных из другой таблицы или таблиц. Также можно SELECTe указывать какие-то формулы и константы. На основе всех совмещенных данных можно заполнить новую таблицу.

```
INSERT [IGNORE] [INTO] tbl_name
[(col_name [, col_name] ...)]
SELECT ...
[ON DUPLICATE KEY UPDATE assignment_list]
```

Также у INSERTа есть еще опция. INSERT [IGNORE] – одна из опций, ее можно указывать, можно не указывать. Есть опция low priority, high priority, delete. Но, на самом деле, low priority и high priority определяют порядок, в котором будут вставляться данные относительно запроса в SELECT. Ни low priority, ни high priority не работают в таблицах InnoDB. А мы рассматриваем именно движок InnoDB как основной, а delete вообще в последних версиях MySQL игнорируется. INSERT вставляет новые ряды в таблицу. INSERT values и INSERT SET вставляет на основе каких-то конкретных данных, которые вы указываете. А INSERT SELECT – на основе данных из других таблиц. Если указать в конце, я уже упоминал, [ON DUPLICATE KEY UPDATE], то можно обновить данные в рядах, которые при попытке вставить данные вызвали конфликты дублирующих со значением. Вы знаете, если у вас есть первичный ключ или уникальный ключ, и вы оставляете новые данные с таким же ключом, то будет конфликт данных, и ряд этот невозможно вставить, поскольку будут нарушены ограничения целостности. Вот в этом случае можно этот ряд не вставить, а найти этот дублирующийся ряд исходный и обновить его какими-то значениями, или же, как мы посмотрим дальше, IGNORE как раз означает проигнорировать такой INSERT, то есть просто не вставлять ряд. Итак, IGNORE игнорирует ошибки в случае конфликтов каких-то с дублирующимися данными. Если какие-то вы вставляете значения, не подходящие по типам, то эти значения тоже могут быть как-то проигнорированы или приведены к значениям по умолчанию. IGNORE найдет любые ошибки при INSERTе.
Также есть вариант сделать вместо INSERTа REPLACE, по синтаксису он очень похож на INSERT, но, как можно понять из его названия, он не только вставляет данные, но и заменяет их, то есть в случае конфликтов, там, где INSERT бы отвалился с ошибкой, REPLACE заменяет эту строку на новые данные, старые данные на новые. Он очень похож на INSERT [ON DUPLICATE KEY UPDATE], но в отличие от UPDATE, который действительно обновляет данные в случае конфликтов, REPLACE внутри себя в случае конфликта удаляет старую строку и потом вставляет новую. Если вы сделаете INSERT [ON DUPLICATE KEY UPDATE] и обновите данные, все будет хорошо, но если вы сделаете REPLACE, да, данное в исходной таблице обновятся, но данные в связанной таблице при этом удалятся, потому что сначала строка будет удалена, данные связанной таблицы удалятся, потом будет вставлена новая строка, но данные в связанной таблице уже не появятся. Также когда вы вставляете данные, вы можете не указывать какие-то колонки, и, например, если в настройках таблицы колонка указана как авто инкремент, ей автоматически будет присвоен следующий номер. Автоматически будет проставлено значение больше чем самое последнее в предыдущих рядах, и этот номер, который автоматически SQL присвоит, можно узнать с помощью встроенной функции LAST_INSERT_ID, то есть, вы вставляете строчку, а потом вам надо понять, какой же ID получился, его можно тоже получить.
Перейдем к чтению данных с помощью SELECT.

```
SELECT [DISTINCT] [SQL_CALC_FOUND_ROWS]
select_expr [, select_expr ...]
[FROM table_references
[WHERE where_condition]
[ORDER BY {col_name | expr} [ASC | DESC], ...]
[LIMIT {[oset,] count | count OFFSET oset}]
[FOR UPDATE | LOCK IN SHARE MODE]]
```

FROM – таблица или таблицы, WHERE – условия выбора, ORDER BY – условия сортировки, LIMIT – это ограничение, то есть можно выбрать не все ряды, которые подпадают под условия, и дополнительные ключевые слова [FOR UPDATE | LOCK IN SHARE MODE].
Селектом нужно извлечь хотя бы одну колонку, не может быть select вообще без указания колонок. Также можно извлекать колонку не под своим именем, т.е. автоматически при извлечении присваивать ей alias, например, вы можете извлечь name и переименовать ее и при этом в user name, то есть у вас в запросе на самом деле колонка name будет фигурировать под именем user name. И также можно написать select*, это значит выбрать вообще все колонки из таблицы.
Table_references указывает, из какой таблицы или таблиц нужно извлекать ряды данных, и таблицы, точно также им можно присваивать алиасы в этом селекте. Например, у вас таблица имеет длинное название, вы не хотите его везде повторять дальше в условиях, вы можете назвать ее, например, просто A.
Если у вас нет условия WHERE, то извлекаются все ряды из таблицы. Если же условие есть, то там присутствует where condition, это какое-то выражение, использующее внутри себя колонки, и если результатом этого выражения для какого-то ряда является TRUE, то этот ряд будет отобран в результирующую выборку.
ORDER BY позволяет сортировать ряды по какой-то колонке или колонкам, а также по их алиасам, можно даже по порядковому номеру колонки или по каким-то выражениям. Сортировать можно как по возрастанию, так и по убыванию.
С помощью лимита можно получить только часть выборки, то есть, например, можно написать LIMIT 10, и если бы у вас так выбралось, сто рядов, то выберутся только первые 10. А также можно указать два числа, то есть десять – это с первого по десятый выбирается, а можно как два числа, с какого и сколько, то есть, например, можно выбрать не первые 10, а начиная с пятого, и от него 10 рядов.
DISTINCT позволяет выкинуть одинаковые ряды из запроса, то есть, например, у вас есть таблица пользователей, у них есть разные поля, и, допустим, у них есть имя. И у многих пользователей имя совпадает. Вот вы делаете SELECT name из таблицы, и тогда вы получите список всех имен, в том числе будут дубли. Если вы напишите SELECT DISTINCT name, то вы получите только различные имена.
SQL_CALC_FOUND_ROWS позволяет посчитать, сколько рядов отобрала выборка, не учитывая лимит.
Опция FOR UPDATE позволяет заблокировать на чтение и запись все ряды, которые были просмотрены при селекте. То есть, вы можете какие-то ряды отобрать и сразу заблокировать, чтобы кто-то другой не начал их изменять.
LOG AND SHARE MODE блокирует просмотренные записи от удаления, но разрешает другим транзакциям читать эти данные.
Рассмотрим пример. Есть какая-то таблица, скажем, пользователей, в ней есть id, имена, есть

логин. Первые пять пользователей у нас тестовые. Они у нас не должны участвовать в выборке, поэтому мы выбираем, мы пишем WHERE 'id'>5, тех пользователей, у которых id больше 5, кроме первых. И мы хотим, например, выбрать пользователей, у которых возраст 10, 20 или 30 лет. И хотим выбрать их id и длину их логина. Мы можем упорядочить эти данные по имени. Само имя у нас в селекте не присутствует, но тем не менее данные будут отсортированы в том порядке, в котором шли их имена, и ограничить эту выборку первыми десятью пользователями.

```
SELECT `id`, LENGTH(`login`) AS len, 1 AS rate
FROM `user`
WHERE `id` > 5 AND `age` IN (10, 20, 30)
ORDER BY `name` DESC
LIMIT 10;
```

Поговорим про UPDATE. UPDATE по синтаксису похож на что-то среднее между INSERT и SELECT. Он отбирает записи, как это делает SELECT, и обновляет в них данные.

```
UPDATE [IGNORE] table_reference[s]
SET col_name = value [, col_name = value]...
[WHERE where_condition]
[ORDER BY ...]
[LIMIT row_count]value:

{expr | DEFAULT}
```

Соответственно, есть условие, по которому мы обновляем ряды. И если мы объявляем только одну таблицу, то ещё допускается ORDER BY и LIMIT. Если в UPDATE мы указываем несколько таблиц, это более сложно, там ORDER BY и LIMIT не допускаются.
Рассмотрим удаление. Удаление по синтаксису похоже на SELECT, потому что оно отбирает ряды, но не возвращает нам их, а удаляет. У него тоже есть WHERE, ORDER BY, LIMIT, просто эти ряды будут удалены.

```
DELETE [IGNORE] FROM tbl_name
[WHERE where_condition]
[ORDER BY ...]
[LIMIT row_count]
```

3.1. Сложные запросы

Посмотрим на более расширенный синтаксис оператора SELECT.

```
SELECT [DISTINCT] ...FROM
table_references
[WHERE where_condition]
[GROUP BY {col_name | expr | position}
[ASC | DESC], ... [WITH ROLLUP]]
[HAVING where_condition]
ORDER BY ...
```

Что такое table_references? Это table_reference, и еще table_reference, и еще и так далее. То есть их может быть несколько. Что такое table_reference? Это либо table_factor, либо join_table. table_factor – это таблица, возможно, с alias, или таблица, созданная под запросом, или несколько table_reference. join_table – это table_reference, после которой идет либо JOIN с другой таблицей, причем здесь в скобочках указано INNER и CROSS, в квадратных скобочках JOIN — это одно и то же.
Также может быть table_reference LEFT или RIGHT OUTER JOIN, слово OUTER можно опускать, и бывает еще table_reference NATURAL JOIN. Есть еще STRAIGHT_JOIN, в принципе, он похож на обычный JOIN, единственное, что если обычный JOIN может читать таблицы не в том порядке, в котором вы указали их в JOIN, то STRAIGHT_JOIN действительно будет читать таблицы слева направо.
Что такое join_condition, который у нас фигурирует здесь в JOIN? Это либо ON и список каких-то условий, очень похожих на where, либо USING и список колонок.
Посмотрим примеры. Допустим, у нас есть таблица books, которую мы рассматривали раньше. Если нам надо просто получить все заголовки книг, мы можем сделать SELECT title FROM books. Но если мы захотим получить заголовок книги и издателя, который ее издал, нам придется соединять две таблицы.

```
SELECT b.title,p.name FROM book AS b
INNER JOIN publisher AS p
ON b.publisher_id = p.publisher_id;
```

И мы напишем SELECT b.title, p.name FROM books AS b, то есть переименовываем таблицу books в b и выбираем из нее title, а из таблицы publisher мы выбираем name. Но эти две таблицы надо соединить. И для этого мы пишем INNER JOIN ON b.publisher_id = p.publisher_id, то есть мы соединяем по колонке publisher. Причем INNER JOIN соединяет эти строки, и если у какой-то левой строки не найдется соответствия, то есть не будет вдруг publisher_id такого в publisher, то эта строка не попадет в запрос вообще. Если не найдется соответствия, строка не попадет. Но у нас такого быть не может, потому что у нас publisher FOREIGN KEY, и мы бы не смогли проставить в таблицу books такой publisher_id, которого нет в publisher.

Мы можем даже упростить этот запрос. В USING можно перечислить названия колонок, по которым соединяете. То есть на равенство, как в ON, а какой-то более упрощенный синтаксис.

```
SELECT b.title,p.name FROM  book  b
JOIN  publisher  p  USING  (publisher_id)
```

Также мы можем написать это вот так. Это то же самое, NATURAL JOIN — то же самое, что JOIN с использованием USING, в котором перечислены все колонки, которые одновременно есть и в левой, и в правой таблице.

```
SELECT b.title,p.name FROM  book  b
NATURAL  JOIN  publisher  p;
```

А вот если мы соединяем таблицу через LEFT JOIN, то, если в правой таблице есть ряд, а в левой по условию ON или USING ряда нет, то ряд из левой таблицы попадет в выборку, а так как справа к нему нечего присоединить, туда попадет просто строка null'ов по числу колонок в другой таблице. Таким образом, этот факт можно даже использовать так, чтобы отобрать ряды из одной таблицы, которым нет соответствий в другой таблице. Мы можем написать вот такой запрос.

```
SELECT  left_tbl.*
FROM  left_tbl  LEFT  JOIN  right_tbl
ON left_tbl.id = right_tbl.id
WHERE right_tbl.id IS  NULL;
```

То есть мы соединяем две таблицы, и если у нас у какого-то ряда присоединился ряд из null'ов, мы можем отфильтровать эти ряды, и таким образом найти все ряды в левой таблицы, у которых нет соответствия в правой.
Поговорим о GROUP BY и о HAVING. Например, мы хотим получить id тех авторов, у которых есть более десяти книг. Написав HAVING books, COUNT мы обзываем словом books, мы как раз и выберем тех авторов, у которых больше десяти книг. Мы не можем здесь использовать WHERE, потому что GROUP BY, который мы написали, он считается уже после выборки. То есть сначала выбираются записи, они отфильтровываются с помощью WHERE. Потом с помощью GROUP BY они группируются, считаются агрегатные функции, COUNT — это агрегатная функция. И WHERE уже выполнился раньше, он не может на своем этапе посчитать агрегатные функции. Поэтому есть HAVING.

```
SELECT  author_id,  COUNT(*)  books  FROM    book
GROUP  BY  author_id  HAVING  books  >  10;
```

Давайте еще один пример. Допустим, мы хотим отобрать трех авторов, у которых самый большой средний рейтинг. Добавим еще к книге поле rating. Можем выбрать author_id, сгруппировать

по нему, посчитать (есть агрегатная функция AVG — average) средний рейтинг и отсортировать по этому полю и взять первые три записи.

```
SELECT author_id, AVG(rate) rating FROM book
GROUP BY author_id
ORDER BY rating DESC LIMIT 3;
```

Какие есть агрегатные функции? Их много, но основные:

- AVG() — average, среднее значение,
- COUNT() — подсчитывает количество рядов, которые были сжаты в одну при агрегации, есть COUNT(DISTINCT), он считает количество разных рядов,
- GROUP_CONCAT(), который позволяет какую-то колонку, которая была в нескольких сжатых рядах, в результате все эти значения поместить в одну строку через запятую. Все ряды сжимаются в один с помощью GROUP BY, а все значения в какой-то колонке в этой нашей результирующей выборке записывают друг за другом через запятую.
- MAX(), MIN(),
- SUM(), который считает сумму всех колонок внутри группы.

Рассмотрим запрос UNION, который позволяет объединить несколько SELECT'ов в один. Во всех этих SELECT'ах должно быть одинаковое количество колонок, поскольку формируется общая результирующая табличка из них. В качестве имен колонок в результате будут взяты названия колонок из первого SELECT'а. И на одних и тех же позициях в каждом SELECT'е должны иметь одинаковый тип данных данные, которые находятся.

```
SELECT ...
UNION [ALL | DISTINCT] SELECT ...
[UNION [ALL | DISTINCT] SELECT ...]
```

Когда мы пишем union без какой-то спецификации, без all или без distinct, то имеется в виду distinct. Это означает, что одинаковые ряды будут удалены из выборки. Если вы пишете all, то дубликаты сохранятся. При этом любой distinct, который встречается, он отменяет все all'ы, находящиеся левее него.
Рассмотрим пример.

```
select 1 union all select 1 union all select 1;
```

То есть вы сделали три раза select, который выбирает просто цифру 1, и по union'у их объединили. Вы получите три ряда, в каждом из которых будет 1.

```
select 1 union all select 1 union select 1;
```

Здесь будет всего один ряд. Хотя первый, второй select написаны через union all, третий union отменил это, и получился в итоге один ряд.
Каждый select внутри union может иметь свой ORDER BY и LIMIT. Но ORDER BY внутри select'ов, которые объединены по union'у, не имеет значения, потому что union вернет в своем порядке. Но если вы просто укажете ORDER BY во вложенных select'ах в union'е, то он будет выкинут оптимизатором и не будет применяться. Но если вы укажете ORDER BY с LIMIT'ом внутри вложенного select'а — это уже будет иметь смысл. Также вы можете использовать ORDER BY и LIMIT для всего запроса, который получен с помощью нескольких union'ов, то есть для всего результата. При этом обязательно указывать круглые скобки внутри, то есть SELECT'ы, которые объединяют через UNION, нужно обрамлять круглыми скобками, если вы используете там ORDER BY и LIMIT, иначе будет ошибка. Если вы используете ORDER BY и LIMIT в самом конце для всего запроса, как бы общей результирующей выборки, то круглые скобки в принципе не обязательны.

```
(SELECT a FROM t1 WHERE a=10 AND B=1)
UNION
(SELECT a FROM t2 WHERE a=11 AND B=2 ORDER BY a LIMIT 10);
```

3.2. Примеры запросов к MySQL

Есть две таблички - products и sales. Products - это продукты, а sales - это продажи. Давайте посмотрим, что в них находится.

```
mysql> select * from products;
+------------+--------------+
| product_id | product_name |
+------------+--------------+
|          1 | яблоки       |
|          2 | роботы       |
+------------+--------------+
2 rows in set (0,00 sec)

mysql> select * from sales;
+------+------------+--------+
| year | product_id | income |
+------+------------+--------+
| 2000 |          1 |    100 |
| 2000 |          2 |    123 |
| 2000 |          1 |     12 |
| 2001 |          1 |    234 |
| 2002 |          2 |    543 |
| 2003 |          1 |   7654 |
| 2003 |          2 |    876 |
| 2003 |          2 |   1233 |
| 2004 |          3 |    123 |
+------+------------+--------+
9 rows in set (0,00 sec)

mysql>
```

Рис. 8

В табличке products у нас есть два продукта: роботы и яблоки, а в табличке sales, если вы посмотрите самую последнюю строчку, то у нас есть продукт id третий, то есть какой-то несуществующий продукт. Я специально не стал делать ограничения на эту таблицу, то есть здесь нет FOREIGN KEY.

Посмотрим, чт получится, если мы захотим создать выборку, в которой будет год, product id и income, как есть и имя продукта, который мы продавали. Для этого нам придется соединить две таблички.

```
| 2003 |          1 |   7654 |
| 2003 |          2 |    876 |
| 2003 |          2 |   1233 |
| 2004 |          3 |    123 |
+------+------------+--------+
9 rows in set (0,00 sec)

mysql> select * from sales join products on sales.product_id = products.
product_id;
+------+------------+--------+------------+--------------+
| year | product_id | income | product_id | product_name |
+------+------------+--------+------------+--------------+
| 2000 |          1 |    100 |          1 | яблоки       |
| 2000 |          2 |    123 |          2 | роботы       |
| 2000 |          1 |     12 |          1 | яблоки       |
| 2001 |          1 |    234 |          1 | яблоки       |
| 2002 |          2 |    543 |          2 | роботы       |
| 2003 |          1 |   7654 |          1 | яблоки       |
| 2003 |          2 |    876 |          2 | роботы       |
| 2003 |          2 |   1233 |          2 | роботы       |
+------+------------+--------+------------+--------------+
8 rows in set (0,00 sec)

mysql>
```

Рис. 9

Как я говорил, join, если это полный join inner, а если мы не указываем никакого другого, то имеется ввиду именно inner join, в результирующую выборку попадают только те ряды, которые есть в обеих таблицах. Смотрите, у нас нет последнего ряда из таблицы sales, потому что для него не нашлось product в таблице products.
Мы написали избыточный запрос. Как мы могли его упростить? Поскольку у нас колонка product id и там и там называется, мы могли вместо on использовать using.

```
mysql> select * from sales join products using(product_id);
+------------+------+--------+--------------+
| product_id | year | income | product_name |
+------------+------+--------+--------------+
|          1 | 2000 |    100 | яблоки       |
|          2 | 2000 |    123 | роботы       |
|          1 | 2000 |     12 | яблоки       |
|          1 | 2001 |    234 | яблоки       |
|          2 | 2002 |    543 | роботы       |
|          1 | 2003 |   7654 | яблоки       |
|          2 | 2003 |    876 | роботы       |
|          2 | 2003 |   1233 | роботы       |
+------------+------+--------+--------------+
8 rows in set (0,00 sec)

mysql>
```

Рис. 10

Более того, поскольку у нас только одна одинаковая колонка есть в обеих таблицах, это product id, мы могли бы еще упростить.

```
mysql> select * from sales natural join products;
+------------+------+--------+--------------+
| product_id | year | income | product_name |
+------------+------+--------+--------------+
|          1 | 2000 |    100 | яблоки       |
|          2 | 2000 |    123 | роботы       |
|          1 | 2000 |     12 | яблоки       |
|          1 | 2001 |    234 | яблоки       |
|          2 | 2002 |    543 | роботы       |
|          1 | 2003 |   7654 | яблоки       |
|          2 | 2003 |    876 | роботы       |
|          2 | 2003 |   1233 | роботы       |
+------------+------+--------+--------------+
8 rows in set (0,00 sec)

mysql>
```

Рис. 11

Мы можем дописать сюда слово left, и у нас будет левое соединение. В этом запросе появился тот ряд, которого не было в предыдущем. Для всех рядов, которые удалось состыковать, состыковалось название продукта, а для того ряда с id = 3, которого просто не было, такого id, в таблице products, вместо product name мы получили null.

```
mysql> select * from sales natural left join products;
+------------+------+--------+--------------+
| product_id | year | income | product_name |
+------------+------+--------+--------------+
|          1 | 2000 |    100 | яблоки       |
|          1 | 2000 |     12 | яблоки       |
|          1 | 2001 |    234 | яблоки       |
|          1 | 2003 |   7654 | яблоки       |
|          2 | 2000 |    123 | роботы       |
|          2 | 2002 |    543 | роботы       |
|          2 | 2003 |    876 | роботы       |
|          2 | 2003 |   1233 | роботы       |
|          3 | 2004 |    123 | NULL         |
+------------+------+--------+--------------+
9 rows in set (0,00 sec)

mysql>
```

Рис. 12

А если бы у нас стояла задача найти те продажи, которые ошибочны, мы могли бы как раз

использовать для этого условие WHERE.

```
mysql> select * from sales natural left join products where products.pro
duct_id is null;
+------------+------+--------+--------------+
| product_id | year | income | product_name |
+------------+------+--------+--------------+
|          3 | 2004 |    123 | NULL         |
+------------+------+--------+--------------+
1 row in set (0,00 sec)

mysql>
```

Рис. 13

Давайте сделаем группировки и посчитаем агрегатные функции. Допустим, мы хотим получить, сколько у нас всего продалось яблок, и сколько у нас всего продалось роботов за все года. Можем сгруппировать по году и узнать, на какую сумму мы продали продуктов всего в каждом году.

```
mysql> select year, sum(income) from sales group by year;
+------+-------------+
| year | sum(income) |
+------+-------------+
| 2000 |         235 |
| 2001 |         234 |
| 2002 |         543 |
| 2003 |        9763 |
| 2004 |         123 |
+------+-------------+
5 rows in set (0,00 sec)

mysql>
```

Рис. 14

Мы можем пойти дальше, мы можем сгруппировать одновременно и по году, и по продукту и получить разбивку, сколько мы продали продукта в какой год. Поскольку, если мы посмотрим исходную таблицу, у нас один и тот же продукт продавался несколько раз в каждом году. А мы хотим получить по годам только уникальные строчки. Мы можем выбрать год, product id и сумму нашего дохода и сгруппировать сначала по году, а потом по product id.

```
mysql> select year, product_id, sum(income) from sales group by year,pro
duct_id;
+------+------------+-------------+
| year | product_id | sum(income) |
+------+------------+-------------+
| 2000 |          1 |         112 |
| 2000 |          2 |         123 |
| 2001 |          1 |         234 |
| 2002 |          2 |         543 |
| 2003 |          1 |        7654 |
| 2003 |          2 |        2109 |
| 2004 |          3 |         123 |
+------+------------+-------------+
7 rows in set (0,01 sec)

mysql>
```

Рис. 15

Можем добавить with rollup. Сначала мы получим сумму по первому продукту и второму продукту за 2000 год. Потом сумму по всем продуктам за 2000 год. with rollup добавляет промежуточные итоги по группировкам.

```
mysql> select year, product_id, sum(income) from sales group by year,pro
duct_id with rollup;
+------+------------+-------------+
| year | product_id | sum(income) |
+------+------------+-------------+
| 2000 |          1 |         112 |
| 2000 |          2 |         123 |
| 2000 |       NULL |         235 |
| 2001 |          1 |         234 |
| 2001 |       NULL |         234 |
| 2002 |          2 |         543 |
| 2002 |       NULL |         543 |
| 2003 |          1 |        7654 |
| 2003 |          2 |        2109 |
| 2003 |       NULL |        9763 |
| 2004 |          3 |         123 |
| 2004 |       NULL |         123 |
| NULL |       NULL |       10898 |
+------+------------+-------------+
13 rows in set (0,00 sec)
```

Рис. 16

Попробуем использовать having. Допустим, по годам мы получили какие-то продажи, и хотим отобрать только те года, в которых продажи были плохими. А плохие, допустим, у нас будут продажи, когда мы продали меньше чем нам, допустим, 200 наших условных единиц.

```
mysql> select year, sum(income) from sales group by year having sum(inco
me)<200;
+------+-------------+
| year | sum(income) |
+------+-------------+
| 2004 |         123 |
+------+-------------+
1 row in set (0,01 sec)
```

Рис. 17

Поговорим про union. Выбираем просто цифру единица. Получаем табличку, в которой есть одна колонка, можем дать алиас ей as one.

```
mysql> select 1;
+---+
| 1 |
+---+
| 1 |
+---+
1 row in set (0,00 sec)

mysql> select 1 as one;
+-----+
| one |
+-----+
|   1 |
+-----+
1 row in set (0,00 sec)
```

Рис. 18

Можем написать union select 2 as two. Мы получим один и два, но колонка по-прежнему называется one, в качестве названия колонки будет взято название из первого из встречающихся селектов.

```
mysql> select 1 as one union select 2 as two;
+-----+
| one |
+-----+
|   1 |
|   2 |
+-----+
2 rows in set (0,00 sec)

mysql> select 1 as one union select 1 as two;
+-----+
| one |
+-----+
|   1 |
+-----+
1 row in set (0,01 sec)
```

Рис. 19

Дальше, смотрите, мы поменяем здесь двойку на единицу и получим не 2 ряда, а только один. Потому что по умолчанию union distinct. Если мы здесь напишем union all, то мы получим два ряда. Если мы после этого пишем union select 1, то мы получим опять одну колонку. Заметьте, потому что union, мы не указали что он all, и он перебил предыдущий all.

```
mysql> select 1 as one union all select 1 as two;
+-----+
| one |
+-----+
|   1 |
|   1 |
+-----+
2 rows in set (0,00 sec)

mysql> select 1 as one union all select 1 as two union select 1;
+-----+
| one |
+-----+
|   1 |
+-----+
1 row in set (0,00 sec)
```

Рис. 20

4. Redis

4.1. Обзор Redis с примерами

Redis – это NOSQL in memory база данных вида ключ-значение, с расширенной поддержкой структурированных форматов данных, таких, как списки, хэши, множества и т.д., а также с возможностью выполнения на стороне сервера скриптов обработчиков на языке Lua.
Несмотря на то, что Redis держит данные в памяти, он обеспечивает постоянное сохранение этих данных на диск и гарантирует сохранность базы данных даже в случае аварийного завершения работы.
Redis также поддерживат транзакции, позволяющие выполнить за один шаг группу команд, и команда других запросов не смогут в нее вклиниться, при этом он гарантирует не противоречивость и последовательность выполнения набор команд, а в случае каких-то проблем позволяет и откатить эти изменения.
Для управления данными в Redis есть команды, например, инкримент, дикримент, которые позволяют увеличить, уменьшать значение как какого-то счетчика, операции над списками и множествами, например, можно сделать объединение и пересечение множеств, а также множественные выборки и функций сортировки, в общем-то тоже достаточно большой набор команд.
Для сохранения данных на диск есть два режима сохранения: периодическая синхронизация всех данных, которые находятся в памяти, т.е. целиком памяти на диск или ведения на диске лога изменений, причем во втором случае гарантируется полная сохранность всех изменений в любой момент. Возможна также организация кластера с master-slave репликации данных на несколько серверов. Также есть режим обмена сообщениями.

Что входит в состав Redis?

- redis-server — собственно сам redis;
- redis-sentinel — мониторинг и обеспечение отказоустойчивости кластера;
- redis-cli — консольный клиент;
- redis-benchmark — тест производительности;
- redis-check-aof и redis-check-dump — могут проверить и исправить файлы дампов БД.

Ключом может быть любая последовательность байт. Это может быть и текстовая строка, а может быть и содержимое jpg файла, причём максимальный размер ключа составляет 512 мегабайт. А что такое тогда значение? Значение в Redis бывает разных типов. Самый простой и очевидный – это строка. Например, если мы захотим кэшировать HTML код страницы, мы можем сохранить это каким-то ключом, вот это будет строка.
Давайте посмотрим на примерах: установим ключ. Допустим, мы кэшируем главную страницу, Команды get и set соответственно позволяют сохранить и получить строковые значения. Причем, если мы сохраним поверх уже существующего, оно будет просто переписан.

```
msk-wifi-17fap2-t_abramov-noname:~ t.abramov$ redis-cli
127.0.0.1:6379> set news:main "lorem ipsum"
OK
127.0.0.1:6379> get news:main
"lorem ipsum"
127.0.0.1:6379>
```

Рис. 21

А еще у set могут быть разные опции, например, если мы хотим установить ключ только если его еще не существовало, то есть опция NX.

Смотрите, у нас ключ установился, а если мs попробуем повторить эту команду, ключ у нас уже создан, она вернёт ошибку.

```
127.0.0.1:6379> set a 1234 nx
OK
127.0.0.1:6379> set a 1234 nx
(nil)
127.0.0.1:6379>
```

Рис. 22

Если мы сохраним в каком-то ключе целое число, для работы с ними специальная команда, которая позволяет его менять. Давайте сделаем set counter и давайте его увеличим с помощью команды incr. Также мы можем и увеличить не на единицу, а на произвольное значение, для этого есть команда incr by. 112 стало. Также есть команды decr, decr by, которая, соответственно, уменьшает счетчик. Важно, что все эти операции атомарны, то есть если два клиента одновременно начнут изменять какой-то один и тот же счетчик, не возникнет состояние гонки и значение всегда будет корректно изменено по сумме всех взаимодействий, скажем так.

```
127.0.0.1:6379> set counter 100
OK
127.0.0.1:6379> incr counter
(integer) 101
127.0.0.1:6379> incr counter
(integer) 102
127.0.0.1:6379>
127.0.0.1:6379> incrby counter 10
(integer) 112
127.0.0.1:6379>
```

Рис. 23

Также, например, есть команда get set, которая позволяет одновременно установить новое значение и вернуть старое, и она тоже атомарна.

Также может быть удобно записать и прочитать сразу несколько значений. Для этого есть команды m set и m get, M – это multiple.

```
127.0.0.1:6379> getset counter 10
"112"
127.0.0.1:6379> mset a 1 b 2 c 3
OK
127.0.0.1:6379> mget a c
1) "1"
2) "3"
127.0.0.1:6379>
```

Рис. 24

Также есть команда exists, котоое позволяет проверить существование какого-то ключа. И так же есть команда del, которая позволяет удалить какой-то ключ. Как видите если ключ есть, она удаляет и возвращает что успешно, а если нет, то просто возвращает, что не удалился.

```
127.0.0.1:6379> exists c
(integer) 1
127.0.0.1:6379> exists csdf
(integer) 0
127.0.0.1:6379> del c
(integer) 1
127.0.0.1:6379> del c
(integer) 0
127.0.0.1:6379>
```

Рис. 25

Также есть команда keys, которая позволяет искать какие ключи у нас есть.

```
127.0.0.1:6379> keys *
1) "b"
2) "a"
3) "counter"
4) "news:main"
127.0.0.1:6379>
```

Рис. 26

Есть команда expire, которая позволяет назначит для ключа срок хранения. Это бывает удобно в кэшировании.

```
127.0.0.1:6379> expire c 5
(integer) 1
127.0.0.1:6379> get c
"100"
127.0.0.1:6379> get c
"100"
127.0.0.1:6379> get c
"100"
127.0.0.1:6379> get c
"100"
127.0.0.1:6379> get c
"100"
127.0.0.1:6379> get c
(nil)
127.0.0.1:6379> get c
(nil)
127.0.0.1:6379>
```

Рис. 27

Также мы с помощью ttl команды можем узнать сколько осталось жить ключу.

```
127.0.0.1:6379> expire c 5
(integer) 1
127.0.0.1:6379> ttl c
(integer) 3
127.0.0.1:6379> ttl c
(integer) 2
```

Рис. 28

4.2. Сложные типы данных в Redis

soph, [07.09.18 15:39] Списки в Redis реализованы как linked lists. Добавление элемента в начало или конец списка происходит очень быстро и не зависит от общего количества элементов, уже имеющихся. Обращение же по индексу элемента может оказаться не столь быстрым.
Давайте попробуем создать список. Для этого есть команда rpush, которая добавляет в список элемент.

```
127.0.0.1:6379> rpush list 1
(integer) 1
127.0.0.1:6379> rpush list 2
(integer) 2
127.0.0.1:6379> rpush list 3 4 5
(integer) 5
127.0.0.1:6379>
```

Рис. 29

Одновременно можно добавить несколько значений.
Получить весь список можно с помощью команды lrange.

```
127.0.0.1:6379> lrange list 0 -1
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
127.0.0.1:6379>
```

Рис. 30

Можно извлекать элементы из списка, из начала или из конца.

```
127.0.0.1:6379> lpop list
"1"
127.0.0.1:6379> rpop list
"5"
127.0.0.1:6379> rpop list
"4"
127.0.0.1:6379> rpop list
"3"
127.0.0.1:6379> rpop list
"2"
127.0.0.1:6379> rpop list
(nil)
127.0.0.1:6379> rpop list
(nil)
127.0.0.1:6379> rpop list
(nil)
```

Рис. 31

Есть также блокирующие варианты команд lpop и rpop: plpop и prpop. Они работают точно также, только у них есть ещё второй параметр, который указывает, сколько времени ждать: если список пустой, они не возвращают сразу (nil), а будут ждать пока туда что-нибудь не запишут или пока не выйдет тот тайм-аут, который вы указали .
Redis автоматически создает ключ с привязанным к нему пустым списком при первой операции с этим списком.
Следующим типом данных в Redis являются хэши. Они подзволяют под одним ключом сохранить несколько значений, каждое со своим ключом.
Команда hmset позволяет установить сразу несколько ключей внутри хэша.

```
127.0.0.1:6379> hmset book title "the little prince" year 2015
OK
127.0.0.1:6379>
```

Рис. 32

Также мы можем получить все значения с помощью команды hgetall.

```
127.0.0.1:6379> hgetall book
1) "title"
2) "the little prince"
3) "year"
4) "2015"
```

Рис. 33

Можно добавить только один элемент (команда hset) или можно работать со значением внутри хэша с помощью команды hincrby.

```
127.0.0.1:6379> hset book rate 100
(integer) 1
127.0.0.1:6379> hincrby book rate 10
(integer) 110
127.0.0.1:6379>
```

Рис. 34

Следующий тип данных – это множество. Множество в Redis – это неупорядоченная коллекция строк. Множество может содержать только уникальные значения. Рассмотрим команду sadd, которая добавляет элементы в множество. С помощь команды smembers мы можем получить все элементы множества.

```
127.0.0.1:6379> sadd myset 1 2 3
(integer) 3
127.0.0.1:6379> sadd myset 2 3 4
(integer) 1
127.0.0.1:6379> smembers myset
1) "1"
2) "2"
3) "3"
4) "4"
127.0.0.1:6379>
```

Рис. 35

Чтобы проверить вхождение элемента, есть команда sismember.

```
127.0.0.1:6379> sismember myset 3
(integer) 1
127.0.0.1:6379> sismember myset 30
(integer) 0
127.0.0.1:6379>
```

Рис. 36

Команда spop извлекает случайный элемент из множества.

```
127.0.0.1:6379> spop myset
"1"
127.0.0.1:6379> spop myset
"2"
127.0.0.1:6379> spop myset
"4"
127.0.0.1:6379> spop myset
"3"
127.0.0.1:6379>
```

Рис. 37

Комнада sinter находит пересечение двух множеств.

```
127.0.0.1:6379> sadd myset 1 2 3 4
(integer) 4
127.0.0.1:6379> sadd myset2  3 4 5 6
(integer) 4
127.0.0.1:6379> sinter myset myset2
1) "3"
2) "4"
127.0.0.1:6379>
```

Рис. 38

Последний тип данных – упорядоченные множества. Тоже хранят уникальные строки, но помимо этого вместе со строкой они сохраняют параметр score, по которыму эти строки будут отсортированы.
Команда zadd добавляет элементы в упорядоченное множество (сначала вводим score, затем, собственно, значение), команда zrange выводит его элементы.

```
127.0.0.1:6379> zadd user 1 Ivan
(integer) 1
127.0.0.1:6379> zadd user 5 Pert
(integer) 1
127.0.0.1:6379> zadd user 5 Petr
(integer) 1
127.0.0.1:6379> zrange user 0 -1
1) "Ivan"
2) "Pert"
3) "Petr"
127.0.0.1:6379>
```

Рис. 39

Команда zrangebyscore позволяет получить из множества строки, у которых score лежит в каком-то диапазоне. Можно добавить лимит: с какого индекса и сколько записей нам нужно.

```
127.0.0.1:6379> zrangebyscore user 2 6 withscores limit 0 1
1) "Pavel"
2) "2"
127.0.0.1:6379> zrangebyscore user 2 6 withscores limit 1 1
1) "Pert"
2) "5"
127.0.0.1:6379>
```

Рис. 40