

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего
образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ И ПРОГРАММНОЙ ИНЖЕНЕРИИ

Проектирование баз данных

Методические указания к выполнению лабораторных работ (draft)

Санкт-Петербург

2022

Составитель: Н.В. Путилова

Рецензент:

Методические указания предназначены для студентов специальностей 02.03.03 и 09.03.04, изучающих дисциплину «Проектирование баз данных». В методические указания включены краткие теоретические сведения, необходимые для выполнения лабораторных работ, требования к содержанию отчетов и порядку выполнения работ, а также варианты индивидуальных заданий.

Методические указания подготовлены кафедрой компьютерных технологий и программной инженерии.

Оглавление

Оглавление	3
Лабораторная работа № 11 Разработка документной базы данных	3
1. Теоретическая часть	4
2. Выполнение лабораторной работы	10
3. Содержание отчета	11
4. Варианты заданий	11
Лабораторная работа № 12 Манипулирование данными в документной базе данных	11
1. Теоретическая часть	11
2. Выполнение лабораторной работы	24
3. Содержание отчета	24
4. Варианты заданий	24
Лабораторная работа № 9 Объектно-реляционные базы данных. Проектирование и создание	24
1. Теоретическая часть	24
2. Выполнение лабораторной работы	32
3. Содержание отчета	33
4. Варианты заданий	33
Лабораторная работа № 10 Объектно-реляционные базы данных. Манипуляция данными и пользовательские операторы	33
1. Теоретическая часть	33
Пользовательские агрегатные функции	37
CREATE AGGREGATE	39
Синтаксис	39
Описание	40
2. Выполнение лабораторной работы	42
3. Содержание отчета	42
4. Варианты заданий	42
Приложение 1 Распределение баллов	44
Семестр 6	44
Приложение 3 Варианты заданий лабораторных 9-12	44
Лабораторная работа № 11 Разработка документной базы данных	

1. Теоретическая часть

Создание БД

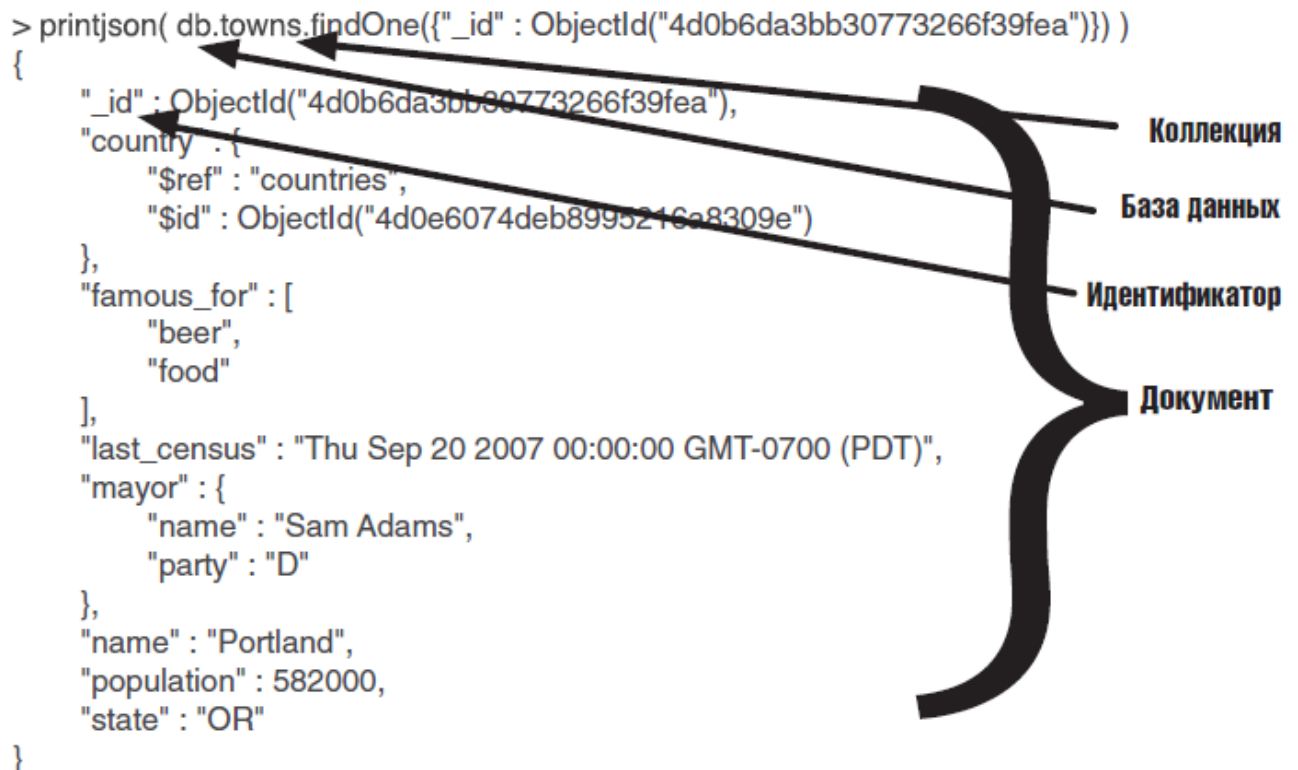
MongoDB реализует новый подход к построению баз данных, где нет таблиц, схем, запросов SQL, внешних ключей и многих других вещей, которые присущи объектно-реляционным базам данных.

В отличие от реляционных баз данных MongoDB предлагает документно-ориентированную(документную) модель данных, благодаря чему MongoDB работает быстрее, обладает лучшей масштабируемостью, ее легче использовать.

Отсутствие жесткой схемы базы данных и в связи с этим потребности при малейшем изменении концепции хранения данных пересоздавать эту схему значительно облегчают работу с базами данных MongoDB и дальнейшим их масштабированием. Кроме того, экономится время разработчиков. Им больше не надо думать о пересоздании базы данных и тратить время на построение сложных запросов.

Mongo – хранилище JSON-документов (строго говоря, данные хранятся в двоичном варианте JSON, который называется BSON).

Документ Mongo можно уподобить строке реляционной таблицы без схемы, в которой допускается произвольная глубина вложенности значений. Рисунок ниже поможет понять, как выглядит JSON-документ.



Документы вместо строк

Реляционные базы данных хранят строки, MongoDB хранит документы. В отличие от строк документы могут хранить сложную по структуре информацию. Документ можно представить как хранилище ключей и значений.

Ключ представляет простую метку, с которым ассоциировано определенный кусок данных.

Однако при всех различиях есть одна особенность, которая сближает MongoDB и реляционные базы данных. В реляционных СУБД встречается такое понятие как первичный ключ. Это понятие описывает некий столбец, который имеет уникальные значения. В MongoDB для каждого документа имеется уникальный идентификатор, который называется `_id`. И если явным образом не указать его значение, то MongoDB автоматически сгенерирует для него значение.

Значение `_id`

Коллекции

Если в традиционном мире SQL есть таблицы, то в мире MongoDB есть коллекции. И если в реляционных БД таблицы хранят однотипные жестко структурированные объекты, то в коллекции могут содержать самые разные объекты, имеющие различную структуру и различный набор свойств.

JavaScript

Родным языком Mongo является JavaScript, который применяется и в сложных MapReduce-запросах, и для простейшего получения справки:

```
> db.help()
> db.towns.help()
```

Эти команды выводят список доступных для данного объекта функций. `db` представляет собой JavaScript-объект, который содержит информацию о текущей базе данных. `db.x` – JavaScript-объект, представляющий коллекцию с именем `x`. Сами команды – обычные JavaScript-функции.

```
> typeof db
> typeof db.name_collection
> typeof db.tname_collection.insert
```

Чтобы ознакомиться с исходным кодом функции, вызовите ее без параметров и без скобок. Давайте добавим еще несколько документов в коллекцию `towns`, для чего напишем собственную JavaScript-функцию.

GridFS

Одной из проблем при работе с любыми системами баз данных является сохранение данных большого размера. Можно сохранять данные в файлах, используя различные языки программирования. Некоторые СУБД предлагают специальные типы данных для хранения бинарных данных в БД (например, BLOB в MySQL).

В отличие от реляционных СУБД MongoDB позволяет сохранять различные документы с различным набором данных, однако при этом размер документа ограничивается 16 мб. Но MongoDB предлагает решение - специальную технологию **GridFS**, которая позволяет хранить данные по размеру больше, чем 16 мб.

Система GridFS состоит из двух коллекций. В первой коллекции, которая называется `files`, хранятся имена файлов, а также их метаданные, например, размер. А в другой коллекции, которая называется `chunks`, в виде небольших сегментов хранятся данные файлов, обычно сегментами по 256 кб.

Для тестирования GridFS можно использовать специальную утилиту **mongofiles**, которая идет в пакете `mongodb`.

- Ручные ссылки, когда вы сохраняете поле `_id` одного документа в другом документе в качестве ссылки. Затем ваше приложение может выполнить второй запрос, чтобы вернуть соответствующие данные.
- DBRefs — это ссылки из одного документа в другой, использующие значение поля `_id` первого документа, имя коллекции и, необязательно, имя его базы данных. Устаревшее. проблема подгрузка всего документа в оперативну.

Проектирование базы данных

К проектированию документных баз данных вообще, и MongoDB в частности, нельзя походить как к проектированию реляционных БД.

Это разные парадигмы. Основные отличия и особенности, которые необходимо учитывать при проектировании:

1. В Коллекцию в отличие от таблицы можно вставить документы разной структуры.
2. Документ может содержать вложенные структуры.
3. Документ обязательно имеет ключ `_id`.
4. Излишнее количество коллекций сделает более сложным написание запросов.
5. Излишний уровень вложенности сделает более сложным написание запросов.
6. Модель БД – компромисс между количеством коллекций и уровнями вложенности
7. Модель базы данных не универсальна, а сильно зависима от задач(запросов).
8. То, что чаще всего необходимо найти в запросах целесообразно выносить в коллекции (верхний уровень)

Если в модель есть связь 1:1. её лучше всего реализовать как вложенность одной из сущностей в документ.

Если это связи 1:M можно применить 2 подхода:

Использовать ручную ссылку(как внешний ключ). Например, для концептуальной модели с группами и студентами это будет выглядеть как на рисунке ниже.

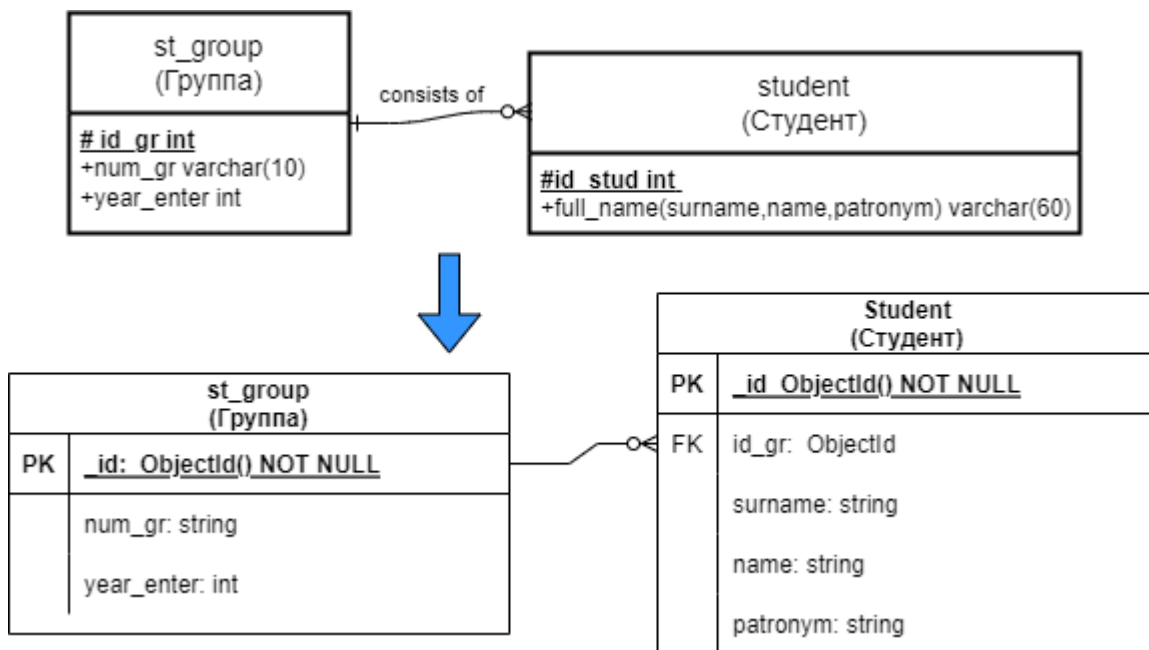


Рисунок. Преобразование связи 1:M концептуальной модели к логической . Вариант 1.
А можно вложить объекты сущности со стороны многие в документ на стороне 1. Как на рисунке ниже:

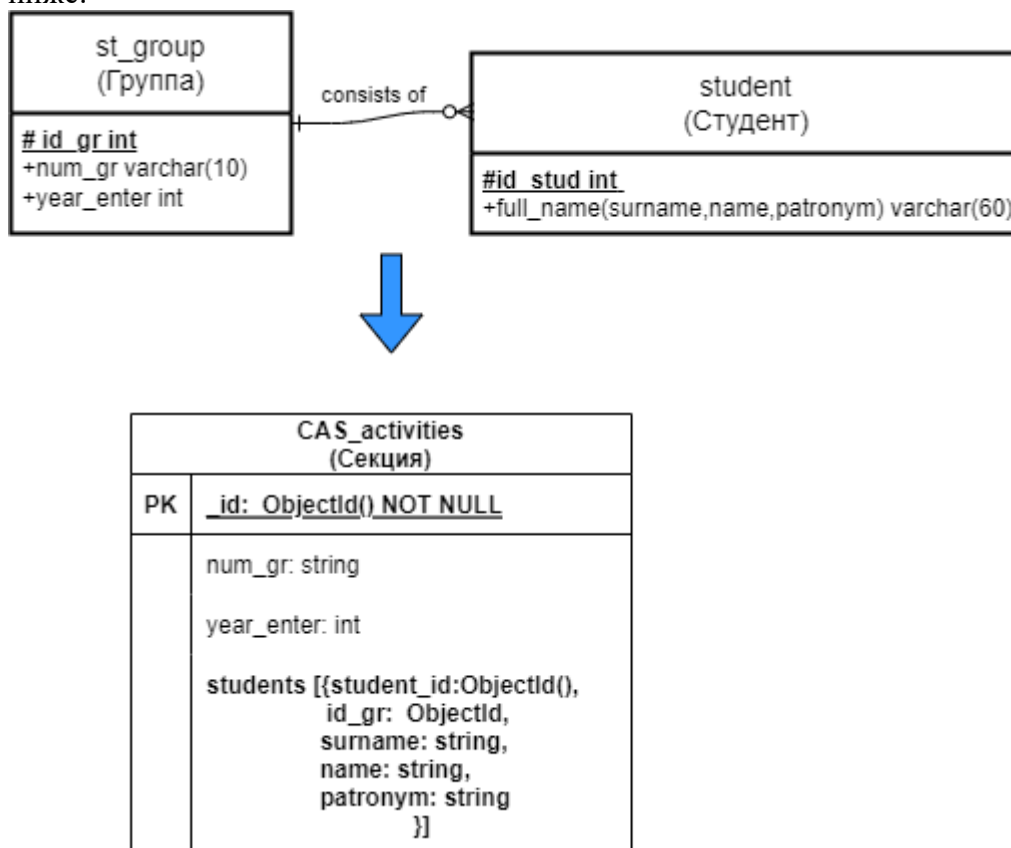


Рисунок. Преобразование связи 1:M концептуальной модели к логической . Вариант 2.
Связь многие ко многим можно реализовать 3 способами, которые рассмотрим на примере связи секций и студентов :

1. Как вложенность внешних ключей в один из документов(сущностей) связи как на рисунке ниже.

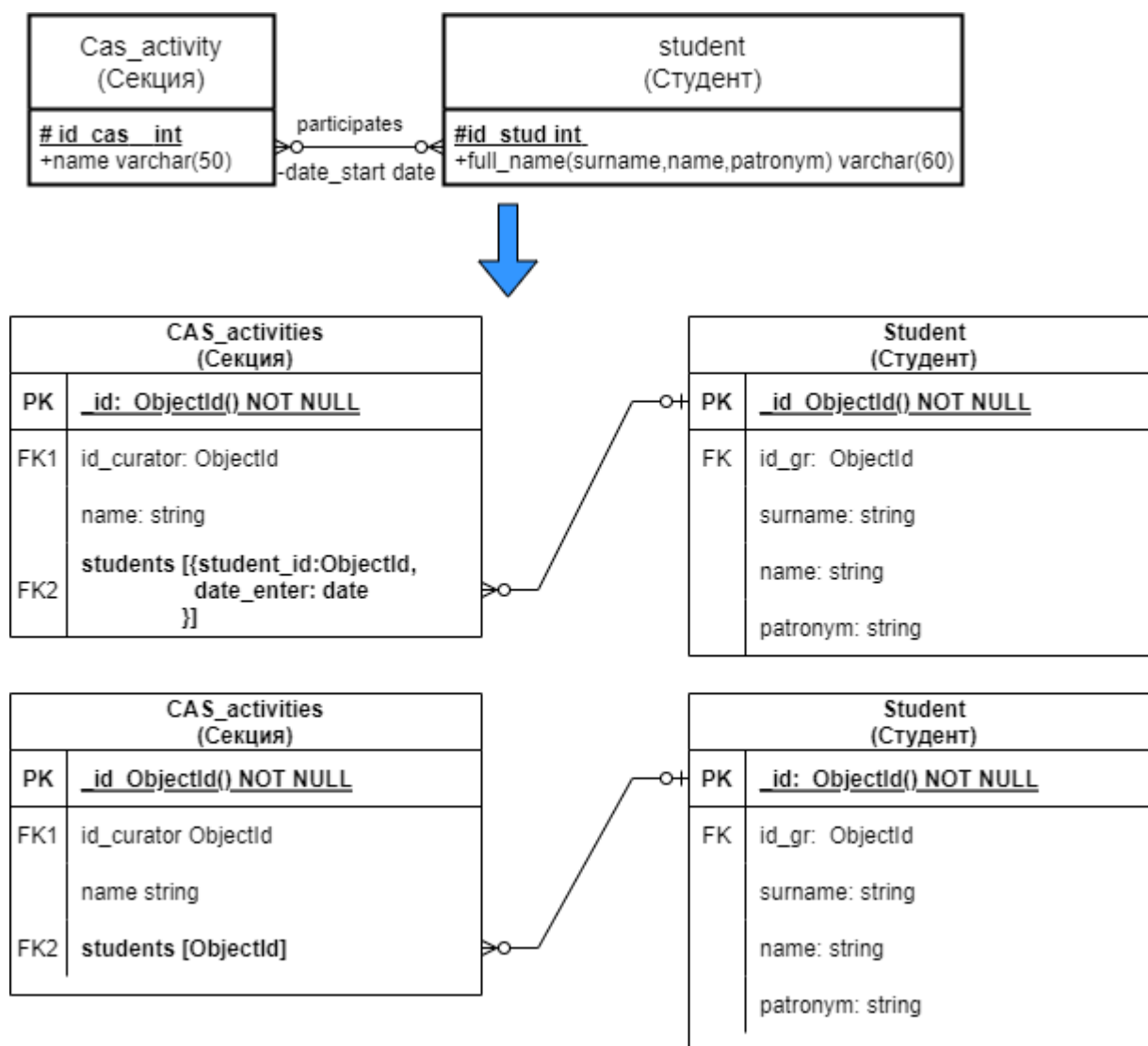


Рисунок. Преобразование связи М:М концептуальной модели к логической . Вариант 1
Обратите внимание: приведены примеры как для связи со свойством (датой вступления в кружок) на рисунке наверху, так и без свойства (на рисунке внизу)

2. Как вложенность объектов целиком со своей структурой в один из документов(сущностей) связи как на рисунке ниже.

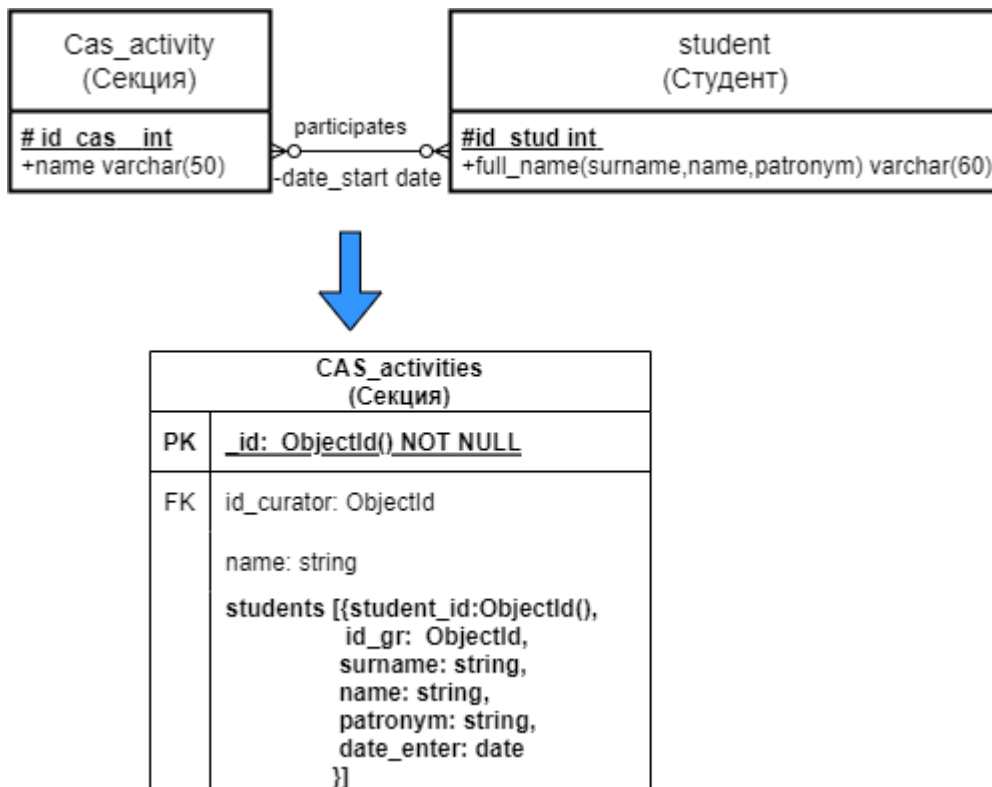


Рисунок. Преобразование связи М:М концептуальной модели к логической . Вариант 2

1. Как связь через внешние ключи в дополнительной таблице аналогично реляционной модели

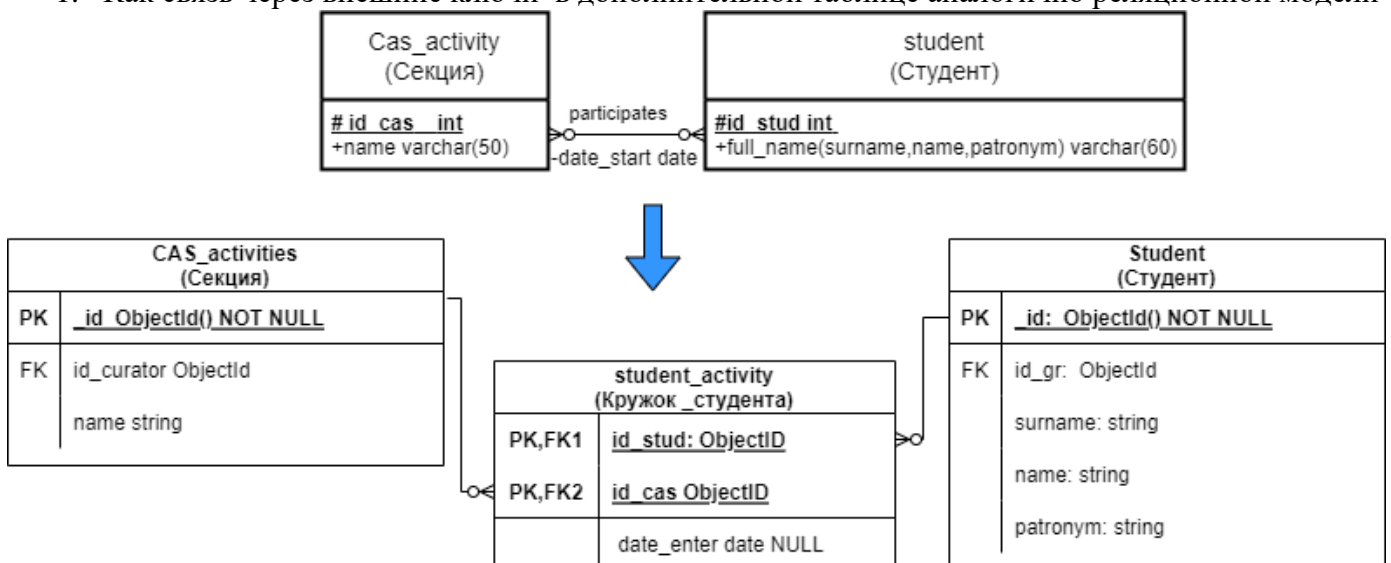


Рисунок. Преобразование связи М:М концептуальной модели к логической . Вариант 3

Работа с MongoDB

Работа с MongoDB может идти 2 способами: через оболочку MongoDB Compass и через интерфейс командной строки.

В MongoDB Compass

Вставка данных.

Не существует пустых моделей/схем БД без данных.

Для создания коллекции достаточно вставить в неё данные

Оболочка MongoDB предоставляет следующие методы для вставки документов в коллекцию:

- Для вставки одного документа используется команда [db.collection.insertOne\(\)](#).
- Для вставки нескольких документов используется команда [db.collection.insertMany\(\)](#).

Например вставка проекта и задачи

```
db.projects.insertOne(
{
  "name": "my_project_1",
  "created": "2022-02-24T9:00",
  "started": "2022-02-24T9:00",
  "ended": null,
  "status": "создан",
  "tasks": [
    {
      "_id": ObjectId(),
      "task_name": "анализтребований",
      "created": "2022-02-24T9:10",
      "started": "2022-02-24T9:20",
      "ended": "2022-03-24T18:20",
      "is_done": false,
      "executors": []
    }
  ]
})
```

Для вложенного документа-задания должен быть в явном виде прописан идентификатор. Автоматически они создаются только для документов верхнего уровня коллекции.

2. Выполнение лабораторной работы

Несмотря на то что считается что NoSQL базы данных не имеет схемы на самом деле схема есть Просто она написано неявно или не описано вообще тем не менее когда планируется использовать базу данных необходимо продумать формат представления данных, каким бы он ни был и задокументировать его. для хранения данных в базе данных mongodb используется формат JSON , точнее его бинарное представление BSON.

Необходимо спроектировать структуру json файла, соответствующую предметной области по варианту задания . Хотя бы в одной коллекции документы должны иметь вложенные.

Для выполнения работы необходимо

1)Спроектировать структуру json файла, соответствующую предметной области по варианту задания.

Хотя бы в одной коллекции документы должны иметь вложенные.

2) Создать базу MongoDB с тестовыми данными для запросов из варианта задания.

3. Содержание отчета

Содержание отчета:

- Текст задания.
- физическую модель БД.
- тестовые данные для запросов по варианту задания в формате таблицы аналогичной таблице тестовых данных в прошлом семестре.

4. Варианты заданий

Варианты заданий приведены в Приложении 3.

Лабораторная работа № 12 Манипулирование данными в документной базе данных

1. Теоретическая часть

Обновление данных

Метод `save`

Как и другие системы управления базами данных MongoDB предоставляет возможность обновления данных. Наиболее простым для использования является метод `save`. В качестве параметра этот метод принимает документ.

В этот документ в качестве поля можно передать параметр `_id`. Если метод находит документ с таким значением `_id`, то документ обновляется. Если же с подобным `_id` нет документов, то документ вставляется.

Если параметр `_id` не указан, то документ вставляется, а параметр `_id` генерируется автоматически как при обычном добавлении через функцию `insert`:

```
> db.users.save({name: "Eugene", age : 29, languages: ["english", "german",  
"spanish"]})
```

В качестве результата функция возвращает объект `WriteResult`. Например, при успешном сохранении мы получим:

```
WriteResult({"nInserted":
1 })
```

updateOne/updateMany

Более детальную настройку при обновлении предлагает функция `update`. Она принимает три параметра:

- `query`: принимает запрос на выборку документа, который надо обновить
- `objNew`: представляет документ с новой информацией, который заместит старый при обновлении
- `options`: определяет дополнительные параметры при обновлении документов. Может принимать два аргумента: `upsert` и `multi`.

Если параметр `upsert` имеет значение `true`, то `mongodb` будет обновлять документ, если он найден, и создавать новый, если такого документа нет. Если же он имеет значение `false`, то `mongodb` не будет создавать новый документ, если запрос на выборку не найдет ни одного документа.

Параметр `multi` указывает, должен ли обновляться первый элемент в выборке (используется по умолчанию, если данный параметр не указан) или же должны обновляться все документы в выборке.

Например:

```
> db.users.updateOne({name : "Tom"}, {name: "Tom", age : 25}, {upsert: true})
```

Теперь документ, найденный запросом `{name : "Tom"}`, будет перезаписан документом `{"name": "Tom", "age" : "25"}`.

Функция `update()` также возвращает объект `WriteResult`. Например:

```
WriteResult({"nMatched": 1, "nUpserted": 0,
"nModified": 1})
```

В данном случае результат говорит нам о том, что найден один документ, удовлетворяющий условию, и один документ был обновлен.

Обновление отдельного поля

Часто не требуется обновлять весь документ, а только значение одного из его ключей. Для этого применяется оператор `$set`. Если документ не содержит обновляемое поле, то оно создается.

```
> db.users.update({name : "Tom", age: 29}, {$set: {age :
30}})
```

Если обновляемого поля в документе нет, до оно добавляется:

```
> db.users.update({name : "Tom", age: 29}, {$set: {salary : 300}})
```

В данном случае обновлялся только один документ, первый в выборке. Указав значение `multi:true`, мы можем обновить все документы выборки:

```
> db.users.update({name : "Tom"}, {$set: {name: "Tom", age : 25}}, {multi:true})
```

Для простого увеличения значения числового поля на определенное количество единиц применяется оператор `$inc`. Если документ не содержит обновляемое поле, то оно создается. Данный оператор применим только к числовым значениям.

```
> db.users.update({name : "Tom"}, {$inc: {age:2}})
```

Удаление поля

Для удаления отдельного ключа используется оператор `$unset`:

```
> db.users.update({name : "Tom"}, {$unset: {salary: 1}})
```

Если вдруг подобного ключа в документе не существует, то оператор не оказывает никакого влияния. Также можно удалять сразу несколько полей:

```
> db.users.update({name : "Tom"}, {$unset: {salary: 1, age: 1}})
```

updateOne и updateMany

Метод `updateOne` похож на метод `update` за тем исключением, что он обновляет только один документ.

```
> db.users.updateOne({name : "Tom", age: 29}, {$set: {salary : 360}})
```

Если необходимо обновить все документы, соответствующие некоторому критерию, то применяется метод `updateMany()`:

```
> db.users.updateMany({name : "Tom"}, {$set: {salary : 560}})
```

Обновление массивов

Оператор `$push`

Оператор `$push` позволяет добавить еще одно значение к уже существующему. Например, если ключ в качестве значения хранит массив:

```
> db.users.updateOne({name : "Tom"}, {$push: {languages: "russian"}})
```

Если ключ, для которого мы хотим добавить значение, не представляет массив, то мы получим ошибку `Cannot apply $push/$pushAll modifier to non-array`.

Используя оператор `$each`, можно добавить сразу несколько значений:

```
> db.users.update({name : "Tom"}, {$push: {languages: {$each: ["russian", "spanish", "italian"]}}})
```

Еще пара операторов позволяет настроить вставку. Оператор `$position` задает позицию в массиве для вставки элементов, а оператор `$slice` указывает, сколько элементов оставить в массиве после вставки.

```
> db.users.update({name : "Tom"}, {$push: {languages: {$each: ["german", "spanish", "italian"], $position:1, $slice:5}}})
```

В данном случае элементы `["german", "spanish", "italian"]` будут вставляться в массив `languages` с 1-го индекса, и после вставки, в массиве останутся только 5 первых элементов.

Оператор `$addToSet`

Оператор `$addToSet` подобно оператору `$push` добавляет объекты в массив. Отличие состоит в том, что `$addToSet` добавляет данные, если их еще нет в массиве:

```
> db.users.update({name : "Tom"}, {$addToSet: {languages: "russian"}})
```

Удаление элемента из массива

Оператор `$pop` позволяет удалять элемент из массива:

```
> db.users.update({name : "Tom"}, {$pop: {languages: 1}})
```

Указывая для ключа `languages` значение 1, мы удаляем первый элемент с конца. Чтобы удалить первый элемент сначала массива, надо передать отрицательное значение:

```
> db.users.update({name : "Tom"}, {$pop: {languages: -1}})
```

Несколько иное действие предполагает оператор `$pull`. Он удаляет каждое вхождение элемента в массив. Например, через оператор `$push` мы можем добавить одно и то же значение в массив несколько раз. И теперь с помощью `$pull` удалим его:

```
> db.users.update({name : "Tom"}, {$pull: {languages: "english"}})
```

А если мы хотим удалить не одно значение, а сразу несколько, тогда мы можем применить оператор `$pullAll`:

```
> db.users.update({name : "Tom"}, {$pullAll: {languages: ["english", "german", "french"]}})
```

Удаление данных

Для удаления документов в MongoDB предусмотрен методы `DeleteOne`, `DeleteMany`:

```
db.collection.deleteOne(  
  <фильтр>,  
  {  
    writeConcern: <document>,  
    collation: <document>,  
    hint: <document|string>  
  }  
)
```

В итоге все найденные документы с `name=Tom` будут удалены. Причем, как и в случае с `find`, мы можем задавать условия выборки для удаления различными способами (в виде регулярных выражений, в виде условных конструкций и т.д.):

Метод `remove` также может принимать второй необязательный параметр булевого типа, который указывает, надо удалять один элемент или все элементы, соответствующие условию. Если этот параметр равен `true`, то удаляется только один элемент. По умолчанию он равен `false`:

Чтобы удалить разом все документы из коллекции, надо оставить пустым параметр запроса:

Удаление коллекций и баз данных

Мы можем удалять не только документы, но и коллекции и базы данных. Для удаления коллекций используется функция `drop`:

```
>  
db.users.drop()
```

И если удаление коллекции пройдет успешно, то консоль выведет:

```
true
```

Чтобы удалить всю базу данных, надо воспользоваться функцией `dropDatabase()`:

```
>
```

```
db.dropDatabase()
```

Выборка из БД

Наиболее простой способ получения содержимого БД представляет использование функции `find`. Действие этой функции во многом аналогично обычному запросу `SELECT * FROM Table`, который извлекает все строки. Например, чтобы извлечь все документы из коллекции `users`, созданной в прошлой теме, мы можем использовать команду `db.users.find()`.

Для вывода документов в более удобном наглядном представлении мы можем добавить вызов метода `pretty()`:

```
>
```

```
db.users.find().pretty()
```

Однако что, если нам надо получить не все документы, а только те, которые удовлетворяют определенному требованию. Например, мы ранее в базу добавили следующие документы:

```
db.users.insertOne({"name": "Tom", "age": 28, languages: ["english", "spanish"]})
db.users.insertOne({"name": "Bill", "age": 32, languages: ["english", "french"]})

db.users.insertOne({"name": "Tom", "age": 32, languages: ["english", "german"]})
```

Выведем все документы, в которых `name=Tom`:

```
> db.users.find({name:
```

```
"Tom"})
```

Такой запрос выведет нам два документа, в которых `name=Tom`.

Теперь более сложный запрос: нам надо вывести те объекты, у которых `name=Tom` и одновременно `age=32`. То есть на языке SQL это могло бы выглядеть так: `SELECT * FROM Table WHERE Name='Tom' AND Age=32`. Данному критерию у нас соответствует последний добавленный объект. Тогда мы можем написать следующий запрос:

```
> db.users.find({name: "Tom", age:
```

```
32})
```


Также несложно отыскать по элементу в массиве. Например, следующий запрос выводит все документы, у которых в массиве `languages` есть `english`:

```
> db.users.find({languages:
"english"})
```

Усложним запрос и получим те документы, у которых в массиве `languages` одновременно два языка: `"english"` и `"german"`:

```
> db.users.find({languages: ["english",
"german"]})
```

Теперь выведем все документы, в которых `"english"` в массиве `languages` находится на первом месте:

```
> db.users.find({"languages.0":
"english"})
```

Соответственно если нам надо вывести документы, где `english` на втором месте (например, `["german", "english"]`), то вместо нуля ставим единицу: `languages.1`.

Проекция

Документ может иметь множество полей, но не все эти поля нам могут быть нужны и важны при запросе. И в этом случае мы можем включить в выборку только нужные поля, используя проекцию. Например, выведем только порцию информации, например, значения полей `"age"` у все документов, в которых `name=Tom`:

```
> db.users.find({name: "Tom"},
{age: 1})
```

Использование единицы в качестве параметра `{age: 1}` указывает, что запрос должен вернуть только содержание свойства `age`.

И обратная ситуация: мы хотим найти все поля документа кроме свойства `age`. В этом случае в качестве параметра указываем `0`:

```
> db.persons.find({name: "Tom"},
{age: 0})
```

При этом надо учитывать, что даже если мы отметим, что мы хотим получить только поле `name`, поле `_id` также будет включено в результирующую выборку. Поэтому, если мы не хотим видеть данное поле в выборке, то надо явным образом указать: `{"_id": 0}`

Альтернативно вместо `1` и `0` можно использовать `true` и `false`:

```
> db.users.find({name: "Tom"}, {age: true, _id: false})
```

Если мы не хотим при этом конкретизировать выборку, а хотим вывести все документы, то можно оставить первые фигурные скобки пустыми:

```
> db.users.find({}, {age: 1, _id: 0})
```

Запрос к вложенным объектам

Предыдущие запросы применялись к простым объектам. Но документы могут быть очень сложными по структуре. Например, добавим в коллекцию `persons` следующий документ:

```
> db.users.insert({"name": "Alex", "age": 28, company: {"name": "microsoft", "country": "USA"}})
```

Здесь определяется вложенный объект с ключом `company`. И чтобы найти все документы, у которых в ключе `company` вложенное свойство `name=microsoft`, нам надо использовать оператор точку:

```
> db.users.find({"company.name": "microsoft"})
```

Использование JavaScript

MongoDB предоставляет замечательную возможность, создавать запросы, используя язык JavaScript. Например, создадим запрос, возвращающий те документы, в которых `name=Tom`. Для этого сначала объявляется функция:

```
fn = function() { return this.name=="Tom"; }  
  
db.users.find(fn)
```

Этот запрос эквивалентен следующему:

```
>  
db.users.find("this.name=='Tom'")
```

Собственно только запросами область применения JavaScript в консоли `mongo` не ограничена. Например, мы можем создать какую-нибудь функцию и применять ее:

```
> function sqrt(n) {  
return n*n; }  
  
> sqrt(5)  
25
```

Настройка запросов и сортировка

MongoDB представляет ряд функций, которые помогают управлять выборкой из бд. Одна из них - функция `limit`. Она задает максимально допустимое количество получаемых документов. Количество передается в виде числового параметра. Например, ограничим выборку тремя документами:

```
>  
db.users.find().limit(3)
```

В данном случае мы получим первые три документа (если в коллекции 3 и больше документов). Но что, если мы хотим произвести выборку не сначала, а пропустив какое-то количество документов? В этом нам поможет функция `skip`. Например, пропустим первые три записи:

```
>  
db.users.find().skip(3)
```

MongoDB предоставляет возможности отсортировать полученный из бд набор данных с помощью функции `sort`. Передавая в эту функцию значения 1 или -1, мы можем указать в каком порядке сортировать: по возрастанию (1) или по убыванию (-1). Во многом эта функция по действию аналогична выражению `ORDER BY` в SQL. Например, сортировка по возрастанию по полю `name`:

```
>  
db.users.find().sort({name: 1})
```

Ну и в конце надо отметить, что мы можем совмещать все эти функции в одной цепочке:

```
> db.users.find().sort({name:  
1}).skip(3).limit(3)
```

Поиск одиночного документа

Если все документы извлекаются функцией `find`, то одиночный документ извлекается функцией `findOne`. Ее действие аналогично тому, как если бы мы использовали функцию `limit(1)`, которая также извлекает первый документ коллекции. А комбинация функций `skip` и `limit` извлечет документ по нужному местоположению.

Параметр `$natural`

Если вдруг нам надо отсортировать ограниченную коллекцию, то мы можем воспользоваться параметром `$natural`. Этот параметр позволяет задать сортировку: документы передаются в том порядке, в каком они были добавлены в коллекцию, либо в обратном порядке.

Например, отберем последние пять документов:

```
> db.users.find().sort({ $natural: -1
```

```
}).limit(5)
```

Оператор \$slice

\$slice является в некотором роде комбинацией функций `limit` и `skip`. Но в отличие от них \$slice может работать с массивами.

Оператор `$slice` принимает два параметра. Первый параметр указывает на общее количество возвращаемых документов. Второй параметр необязательный, но если он используется, тогда первый параметр указывает на смещение относительно начала (как функция `skip`), а второй - на ограничение количества извлекаемых документов.

Например, в каждом документе определен массив `languages` для хранения языков, на которых говорит человек. Их может быть и 1, и 2, и 3 и более. И допустим, ранее мы добавили следующий объект:

```
> db.users.insert({"name": "Tom", "age": "32", "languages": ["english",  
"german"]})
```

И мы хотим при выводе документов сделать так, чтобы в выборку попадал только один язык из массива `languages`, а не весь массив:

```
> db.users.find ({name: "Tom"}, {languages: {$slice :  
1}})
```

Данный запрос при извлечении документа оставит в результате только первый язык из массива `languages`, то есть в данном случае `english`.

Обратная ситуация: нам надо оставить в массиве также один элемент, но не с начала, а с конца. В этом случае необходимо передать в параметр отрицательное значение:

```
> db.users.find ({name: "Tom"}, {languages: {$slice : -  
1}});
```

Теперь в массиве окажется `german`, так как он первый с конца в добавленном элементе.

Используем сразу два параметра:

```
> db.users.find ({name: "Tom"}, {languages: {$slice : [-1,  
1]}});
```

Первый параметр говорит начать выборку элементов с конца (так как отрицательное значение), а второй параметр указывает на количество возвращаемых элементов массива. В итоге в массиве `language` окажется `"german"`

Lookup

Для связи между 2 таблицами используется оператор `lookup`

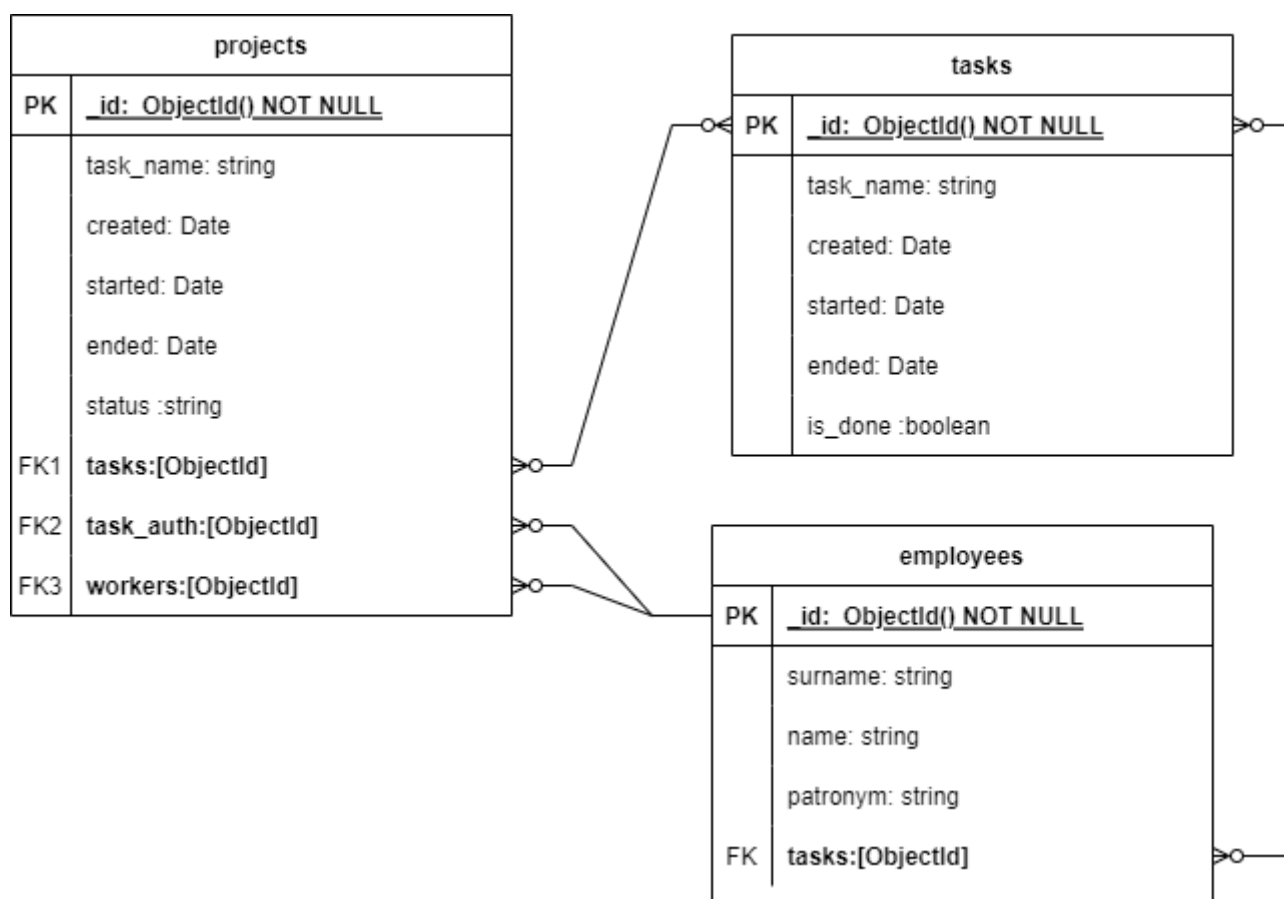
- {
- \$lookup:
- {
- from: <collection to join>,
- localField: <field from the input documents>,
- foreignField: <field from the documents of the "from" collection>,
- as: <output array field>
- }
- }

Для нахождения среднего используйте оператор \$avg в агрегации

<https://www.mongodb.com/docs/manual/reference/operator/aggregation/avg/>

//-----

Примеры выполнения задач



1. Задачи, в названии которых содержится слово «интеграция», но оно не последнее
2. Задача, относящаяся к 2 различным проектам
3. Задача с самым поздним сроком окончания
4. Проект с самым большим количеством задач
5. Человек, у которого нет незавершенных задач

Задачи, в названии которых содержится слово «интеграция», но оно не последнее

```
db.tasks.find({$and:[{"task_name":/интеграция/},{ "task_name" : {
"$not" : /интеграция$/ } }]}))
```

Задача, относящаяся к 2 различным проектам

- По шагам

1. Основа- Задача
2. Присоединить проекты
3. Создать поле с количеством проектов
4. Выбрать те задачи, у которых поле с количеством проектов ≥ 2

```
db.tasks.aggregate([
  {
    $lookup:
    {
      from: "projects",
      localField: "_id",
      foreignField: "pr_tasks",
      as: "project_tasks"
    }
  },
  {
    $project:{"_id":1,"task_name":1,"num_proj":{"$size:"$project_tasks"}}
  },
  {
    $match:{"num_proj":{"$gte: 2 }}
  }
])
```

Задача с самым поздним сроком окончания

1. Основа- Задача
2. Максимальный срок
3. Добавляем возможность сравнения, уравнивая уровни вложенности
4. Добавляем поле равенства
5. Ищем строки где значение поля равенства- истина

```
db.tasks.aggregate([
  {
    $project:{"_id":1,"task_name":1,"ended":1}
  },
  {
    $group:{
      _id:null,
      max_end: { $max: "$ended" },
      "task_data": { "$push": "$$ROOT" }
    }
  },
  {
    $unwind : "$task_data"
  },
  {
    $project:{"_id":"$task_data._id","task_nm":"$task_data.task_name","num":"$task_data.num_proj",max_end:1,
    "is_eq":{"$eq":["$task_data.ended","$max_end"]}}
  },
  {
    $match:{"is_eq":true}
  }
])
```

Проект с самым большим количеством задач

1. Основа- проект
2. Получаем количество задач
3. Ищем максимальное количество
4. Приводим к одному уровню
5. Создаем поле с равенством
6. Ищем строки где значение поля равенства- истина

```
db.projects.aggregate([
  { $project: { "_id": 1, "name": 1, "num_task": { $size: "$pr_tasks" } } },
  { $group: {
    _id: null,
    max_task_c: { $max: "$num_task" },
    "task_data": { "$push": "$$ROOT" }
  } },
  { $unwind: "$task_data" },
  {
    $project: {
      "_id": "$task_data._id",
      "project_nm": "$task_data.name",
      "num_task1": "$task_data.num_task",
      "max_task_c": 1,
      "is_eq": { $eq: [ "$task_data.num_task", "$max_task_c" ] }
    },
    { $match: { "is_eq": true } }
  ] )
```

Человек, у которого нет незавершенных задач

1. Основа- человек(сотрудник)
2. Присоединить задачи
3. Оставить в объекте, полученном присоединением только незавершенные задачи
4. Создать поле с количеством незавершенных задач
5. Условие на поле с количеством незавершенных задач на равенство 0

```
db.employees.aggregate([
  { $lookup: {
    from: "tasks",
    localField: "tasks",
    foreignField: "_id",
    as: "unended_tasks"
  } },
  { $project: {
    "_id": 1, "surname": 1, "name": 1, "patronym": 1, "unended_tasks": {
      $filter: { input: "$unended_tasks", as: "tsk", cond: { $eq: [ "$tsk.is_done", false ] } }
    }
  } },
  { $addFields: { "cnt_not_done": { $size: "$unended_tasks" } } },
  { $match: { "cnt_not_done": 0 } } ] )
```

2. Выполнение лабораторной работы

Привести пример обновления и удаления данных из базы

Выполнить запросы на выборку по варианту задания без использования функций `javaScript`

3. Содержание отчета

— Текст задания.

— модель БД.

— текст запросов;

— наборы данных, возвращаемые запросами в виде скриншотов.

4. Варианты заданий

Варианты заданий приведены в Приложении 3.

Лабораторная работа № 9 Объектно-реляционные базы данных.

Проектирование и создание

1. Теоретическая часть

Наследование

PostgreSQL реализует наследование таблиц, которое может оказаться полезным инструментом для разработчиков базы данных. (Возможность наследования определяется стандартом SQL:1999 и более поздними стандартами и во многих отношениях отличается от возможностей PostgreSQL)

Начнём с примера: предположим мы пытаемся создать модель данных для городов. В каждом штате есть несколько городов, но только одна столица. Мы хотим предоставить возможность быстрого нахождения города-столицы для любого отдельного штата. Всё это можно сделать, создав две таблицы, одну для столиц штата и другую для городов, которые не являются столицами. Однако, что произойдёт, когда мы захотим получить данные о каком-либо городе, в не зависимости от того, является он столицей или нет? Возможность наследования может помочь решить эту проблему. Мы создаём таблиц `capitals`, которая наследует от таблицы `cities`:


```
CREATE TABLE cities (
name          text,
population    float,
altitude      int      -- (in ft)
);
```

```
CREATE TABLE capitals (
state         char(2)
) INHERITS (cities);
```

В этом случае, таблица `capitals` наследует все колонки из родительской таблицы `cities`. В таблице столиц штатов также есть дополнительная колонка с аббревиатурой названия штата — `state`.

В PostgreSQL любая таблица может наследовать от нуля или более других таблиц, а запрос может ссылаться либо на все строки в таблице либо на все строки в таблице вместе с её потомками. Последнее является поведением по умолчанию. Например, следующий запрос ищет имена всех городов, включая столицы штатов, которые расположены на высоте свыше 500 футов:

```
SELECT name, altitude
FROM cities
WHERE altitude > 500;
```

С данными для примера, взятыми из учебного руководства PostgreSQL, этот запрос возвратит:

name	altitude
Las Vegas	2174
Mariposa	1953
Madison	845

С другой стороны, следующий запрос находит все города, которые не являются столицами штатов и которые также расположены на высоте свыше 500ft:

```
SELECT name, altitude
FROM ONLY cities
WHERE altitude > 500;
```

name	altitude
Las Vegas	2174
Mariposa	1953

Здесь слово "ONLY" перед таблицей `cities` говорит, что запрос должен выполняться только для таблицы `cities`, а не для таблиц, расположенных ниже `cities`, в иерархии наследования. Многие из команд, которые мы использовали ранее — `SELECT`, `UPDATE` и `DELETE` — поддерживают нотацию "ONLY".

В некоторых случаях, вы можете захотеть узнать из какой таблицы получена отдельная строка результата. В каждой таблице существует системная колонка с именем `tableoid`, которая расскажет о таблице:

```
SELECT c.tableoid, c.name, c.altitude
FROM cities c
WHERE c.altitude > 500;
```

запрос возвратит:

tableoid	name	altitude
139793	Las Vegas	2174
139793	Mariposa	1953
139798	Madison	845

(Если вы попытаетесь воспроизвести этот пример, то предположительно вы получите другие значения OID.) Выполнив объединение с таблицей `pg_class` вы можете увидеть сами имена таблиц:

```
SELECT p.relname, c.name, c.altitude
FROM cities c, pg_class p
WHERE c.altitude > 500 and c.tableoid = p.oid;
```

запросвозвратит:

relname	name	altitude
cities	Las Vegas	2174
cities	Mariposa	1953
capitals	Madison	845

Наследование не распространяется автоматически на данные из команд `INSERT` или `COPY` на другие таблицы в иерархии наследования. Вследующем примере, выполнение `INSERT` вызовет ошибку:

```
INSERT INTO cities (name, population, altitude, state)
VALUES ('New York', NULL, NULL, 'NY');
```

Мы могли бы надеяться, что данные как-нибудь будут перенаправлены в таблицу `capitals`, но этого не произойдёт: `INSERT` всегда вставляет данные точно в указанную таблицу. Однако, это не поможет в данном выше случае, потому что таблица `cities` не содержит колонки `state` и таким образом команда будет отвергнута, перед тем как к ней можно будет применить правило.

Для таблиц внутри иерархии наследования, могут быть заданы ограничения целостности `check`. Все ограничения `check` для родительской таблицы автоматически наследуются всеми её потомками. Однако, другие типы ограничений целостности не наследуются.

Таблица может наследовать более чем от одной родительской таблицы и в этом случае она будет содержать суммарный список колонок из родительских таблиц. К этому списку добавляются любые колонки, задаваемые в самой таблице-потомке. Если в нескольких родительских таблицах, встречаются колонки с одинаковым именем или такая колонка есть в родительской таблице и в определении таблицы-потомка, то эти колонки "сливаются" таким образом, что только одна такая колонка будет в таблице-потомке. При слиянии, колонки должны иметь одинаковый тип, иначе будет выдано сообщение об ошибке. Колонка после слияния получает копию всех ограничений целостности `check` от каждой слитой колонки.

Наследование таблицы задаётся, используя команду `CREATE TABLE`, с ключевым словом `INHERITS`. Однако, похожая команда `CREATE TABLE AS` не позволяет указывать наследование.

В качестве альтернативы, для уже созданной таблицы можно задать новую родительскую таблицу с помощью `ALTER TABLE`, используя подформу `INHERITS`. Чтобы выполнить эту команду, новая таблица-потомок уже должна включать колонки с тем же именем и типом, что и родительская таблица. Она также должна включать ограничения целостности `check` с тем же именем и

выражением `check` как и в родительской таблице. Похожим образом, связь наследования может быть удалена из таблицы-потомка, используя `ALTER TABLE` с подформой `NO INHERIT`.

Подходящий способ создания новой совместимой таблицы для таблицы-потомка состоит в использовании опции `LIKE` в команде `CREATE TABLE`. Такая команда создаёт таблицу с теми же колонками и с теми же типами (смотрите однако замечание про предостережения ниже). В качестве альтернативы, совместимую таблицу можно создать если сперва создать новую таблицу-потом с помощью `CREATE TABLE`, а затем удалить связь наследования через `ALTER TABLE`.

Родительская таблица не может быть удалена пока существует хотя бы одна таблица-потомок. Если вы хотите удалить таблицу и всех её потомков, то наиболее простой способ состоит в удалении родительской таблицы с опцией `CASCADE`. Если колонки в таблицах-потомках наследуются из родительских таблиц, то их нельзя удалить или изменить.

Наследование в PostgreSQL

В PostgreSQL наследуются таблицы. При этом наследуются только столбцы с типами данных, а из ограничений только проверка значений. Ключ предка лучше иметь автоинкрементным (типа `serial`) иначе уникальность первичного ключа в рамках связки предок-потомок не обеспечить иначе чем триггерами. При этом для сохранения работы автоинкремента **нельзя принудительно ставить значения первичного ключа**.

Общий синтаксис наследования выглядит следующим образом:

```
createtablechild
(...
)inherits (parent)
```

Пример иллюстрации того, что наследуется:

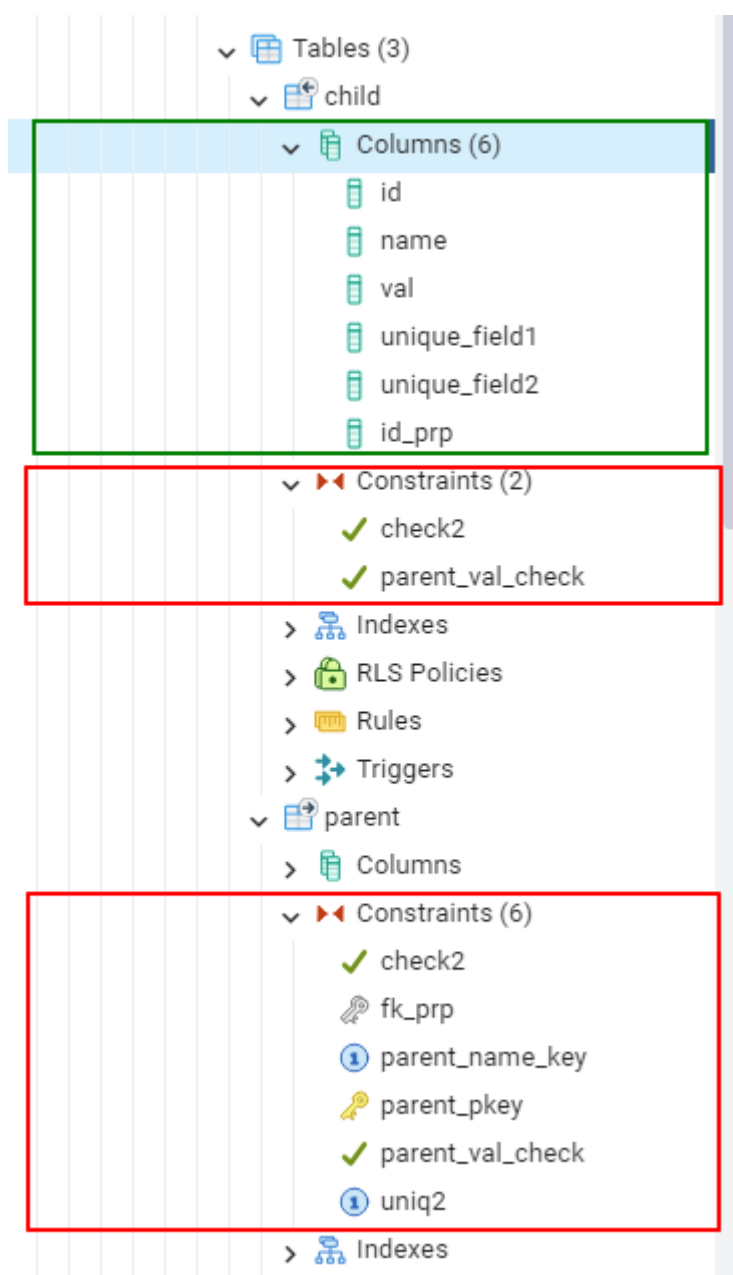
```
create table parent_realational_parent
(
  id_prp serial not null primary key,
  some_field varchar(20) null
);
create table parent
(
  id serial not null primary key,
  name varchar(20) null unique,
  val integer check(val>0),
  unique_field1 char(4),
  unique_field2 char(6),
  id_prp integer,
```

```

constraint fk_prp foreign key (id_prp) references
parent_realational_parent on delete cascade
on update cascade,
constraint uniq2 unique(unique_field1,unique_field2),
constraint      check2      check(unique_field1||unique_field2
!='00000000000')
);
create table child
()inherits (parent);

```

На рисунке ниже представлен скриншот PGAdmin с составляющими таблицы.(Зеленые унаследованные столбцы, красное- разница в ограничениях)



Чтобы в потомке правильно реализовать все ограничения необходимо, чтобы код потомка содержа все ненаследуемые ограничения. Т.е. выглядел следующим образом

```
create table child
(primary key(id),
constraint uniq unique(name),
constraint fk_prp1 foreign key (id_prp) references
parent_realational_parent on delete cascade
on update cascade,
constraint uniq2 unique(unique_field1,unique_field2),
)
inherits (parent);
```


Для примера запроса к предку и потомку вставим данные:

```
INSERT INTO public.parent_realational_parent(
some_field)
VALUES ('ref_val1'),('ref_val2'),('ref_val3');
INSERT INTO public.parent(
name, val, unique_field1, unique_field2, id_prp)
VALUES ('parent 1', 10, '1111', '111111', 1),
('parent 2', 20, '2222', '220022', 1);
INSERT INTO public.child(
name, val, unique_field1, unique_field2, id_prp)
VALUES ('child 1', 10, '1100', '10011', 2),
('child 2', 20, '2211', '223322', 3);
```

Чтобы получить информацию из потомка достаточно сделать запрос к самой таблице-потомку

```
select * from child
```

1



select * from child

Data Output

Messages

Notifications

	id integer	name character varying (20)	val integer	unique_field1 character	unique_field2 character	id_prp integer
1	3	child 1	10	1100	10011	2
2	4	child 2	20	2211	223322	3

При запросе к таблице-предку мы получим данные и из предка и из потомка

```
select * from parent
```

1 `select * from parent`

Data Output

Messages

Notifications

<

Если необходимо получить данные только предка, то запрос должен выглядеть:

```
select * from only parent
```

1

select * from only parent

Data Output

Messages

Notifications

≡+

	id [PK] integer	name character varying (20)	val integer	unique_field1 character	unique_field2 character	id_prp integer
1	1	parent 1	10	1111	111111	1
2	2	parent 2	20	2222	220022	1

Обратите внимание, что в таком запросе `only` должен стоять перед предком.

1	select * from only parent	
2	join public.parent_realational_parent using (id_prp)	I

Data Output Messages Notifications								
	id_prp integer	id integer	name character varying (20)	val integer	unique_field1 character	unique_field2 character	some_field character varying (20)	
1	1	2	parent 2	20	2222	220022	ref_val1	
2	1	1	parent 1	10	1111	111111	ref_val1	

1	select * from public.parent_realational_parent	
2	join only parent using (id_prp)	I

Data Output Messages Notifications								
	id_prp integer	some_field character varying (20)	id integer	name character varying (20)	val integer	unique_field1 character	unique_field2 character	
1	1	ref_val1	2	parent 2	20	2222	220022	
2	1	ref_val1	1	parent 1	10	1111	111111	

Основная проблема при реализации наследования в PostgreSQL связана с наличием внешнего ключа ссылающегося на предка. По логике наследования он должен видеть потомков, но в запросах он их видит, а во внешнем ключе нет.

Создания таблицы

```
create table tab_ref_to_parent
(id_tab_ref_to_parent serial primary key,
Value varchar(20),
Id_parent integer ,
Foreign key (Id_parent) references parent(id) on delete restrict on
update restrict
)
```

Вызовет ошибку при попытке вставить данные с ключом из потомка

То есть таблица должна выглядеть

```
create table tab_ref_to_parent
(id_tab_ref_to_parent serial primary key,
Value varchar(20),
Id_parent integer
```

)

И должны существовать триггеры действующие вместо внешнего ключа:

На вставку и обновление `tab_ref_to_parent`, чтобы нельзя было вставить ключи, не принадлежащие `parent` или `child`

На удаление и обновление предка- обрабатывая обновление ключа или удаление данных из родительской таблицы.

Пользовательскиетипы

Составные типы

Составной тип представляет структуру табличной строки или записи; по сути это просто список имён полей и соответствующих типов данных. PostgreSQL позволяет использовать составные типы во многом так же, как и простые типы. Например, в определении таблицы можно объявить столбец составного типа.

Ниже приведены два простых примера определения составных типов:

```
CREATE TYPE complex AS (  
  r double precision,  
  i double precision  
);  
CREATE TYPE inventory_item AS (  
  name text,  
  supplier_id integer,  
  price numeric  
);
```

Определив такие типы, мы можем использовать их в таблицах:

```
CREATE TABLE on_hand (  
  item inventory_item,  
  count integer  
);
```

Объявление перечислений

Тип перечислений создаются с помощью команды `CREATE TYPE`, например так:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Созданные типы `enum` можно использовать в определениях таблиц и функций, как и любые другие:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

```
CREATE TABLE person (  
  name text,  
  current_mood mood  
);  
INSERT INTO person VALUES ('Moe', 'happy');  
SELECT * FROM person WHERE current_mood = 'happy';
```

Все типы перечислений считаются уникальными и поэтому значения разных типов нельзя сравнивать.

2. Выполнение лабораторной работы

Спроектировать физическую модель базы данных, находящуюся в третьей нормальной форме и включающей наследование и хотя бы один пользовательский тип в соответствии с заданным вариантом. Написать соответствующий скрипт создания базы данных

Варианты заданий приведены в Приложении 3.

3. Содержание отчета

Содержание отчета:

- Текст задания.
- физическую модель БД.
- Скрипт создания базы данных

4. Варианты заданий

Варианты заданий приведены в Приложении 3.

Лабораторная работа № 10 Объектно-реляционные базы данных.

Манипуляция данными и пользовательские операторы

1. Теоретическая часть

Пользовательские операторы

CREATE OPERATOR — создать оператор

Синтаксис

```
CREATE OPERATOR имя (  
PROCEDURE = имя_функции  
[, LEFTARG = тип_слева ] [, RIGHTARG = тип_справа ]  
[, COMMUTATOR = коммут_оператор ] [, NEGATOR = обратный_оператор ]  
[, RESTRICT = процедура_ограничения ] [, JOIN = процедура_соединения ]  
[, HASHES ] [, MERGES ]  
)
```

Описание

CREATE OPERATOR определяет новый оператор, *имя*. Владелец оператора становится пользователь, его создавший. Если указано имя схемы, оператор создаётся в ней, в противном случае — в текущей схеме.

Имя оператора образует последовательность из нескольких символов (не более чем NAMEDATALEN-1, по умолчанию 63) из следующего списка:

+ - * / < > = ~ ! @ # % ^ & | ` ?

Однако выбор имени ограничен ещё следующими условиями:

- Сочетания символов -- и /* не могут присутствовать в имени оператора, так как они будут обозначать начало комментария.
- Многосимвольное имя оператора не может заканчиваться знаком + или -, если только оно не содержит также один из этих символов:

~ ! @ # % ^ & | ` ?

Например, @- — допустимое имя оператора, а *- — нет. Благодаря этому ограничению, PostgreSQL может разбирать корректные SQL-запросы без пробелов между компонентами.

- Использование => в качестве имени оператора считается устаревшим и может быть вовсе запрещено в будущих выпусках.

Оператор != отображается в <> при вводе, так что эти два имени всегда равнозначны.

Необходимо определить либо LEFTARG, либо RIGHTARG, а для бинарных операторов оба аргумента. Для правых унарных операторов должен быть определён только LEFTARG, а для левых унарных — только RIGHTARG.

Процедура *имя_функции* должна быть уже определена с помощью CREATE FUNCTION и иметь соответствующее число аргументов (один или два) указанных типов.

Другие предложения определяют дополнительные характеристики оптимизации.

Чтобы создать оператор, необходимо иметь право USAGE для типов аргументов и результата, а также право EXECUTE для нижележащей функции. Если указывается коммутативный или обратный оператор, нужно быть его владельцем.

Параметры

имя

Имя определяемого оператора. Допустимые в нём символы перечислены ниже. Указанное имя может быть дополнено схемой, например так: CREATE OPERATOR myschema.+ (...). Если схема не указана, оператор создаётся в текущей схеме. При этом два оператора в одной схеме могут иметь одно имя, если они работают с разными типами данных. Такое определение операторов называется *перегрузкой*.

имя_функции

Функция, реализующая этот оператор.

тип_слева

Тип данных левого операнда оператора, если он есть. Этот параметр опускается для левых унарных операторов.

тип_справа

Тип данных правого операнда оператора, если он есть. Этот параметр опускается для правых унарных операторов.

коммут_оператор

Оператор, коммутирующий для данного.

обратный_оператор

Оператор, обратный для данного.

процедура_ограничения

Функция оценки избирательности ограничения для данного оператора.

процедура_соединения

Функция оценки избирательности соединения для этого оператора.

HASHES

Показывает, что этот оператор поддерживает соединение по хешу.

MERGES

Показывает, что этот оператор поддерживает соединение слиянием.

Чтобы задать имя оператора с указанием схемы в *коммут_оператор* или другом дополнительном аргументе, применяется синтаксис `OPERATOR ()`, например:

```
COMMUTATOR = OPERATOR (myschema.===) ,
```

Пользовательские операторы пример

Любой оператор представляет собой «синтаксический сахар» для вызова нижележащей функции, выполняющей реальную работу; поэтому прежде чем вы сможете создать оператор, необходимо создать нижележащую функцию. Однако оператор — *не исключительно* синтаксический сахар, так как он несёт и дополнительную информацию, помогающую планировщику запросов оптимизировать запросы с этим оператором. Рассмотрению этой дополнительной информации будет посвящён следующий раздел.

Postgres поддерживает левые унарные, правые унарные и бинарные операторы. Операторы могут быть перегружены; то есть одно имя оператора могут иметь различные операторы с разным

количеством и типами операндов. Когда выполняется запрос, система определяет, какой оператор вызвать, по количеству и типам предоставленных операндов.

В следующем примере создаётся оператор сложения двух комплексных чисел. Предполагается, что мы уже создали определение типа `complex`. Сначала нам нужна функция, собственно выполняющая операцию, а затем мы сможем определить оператор:

```
CREATE FUNCTION complex_add(complex, complex)
RETURNS complex
AS 'имя_файла', 'complex_add'
LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (
  leftarg = complex,
  rightarg = complex,
  procedure = complex_add,
  commutator = +
);
```

Теперь мы можем выполнить такой запрос:

```
SELECT (a + b) AS c FROM test_complex;
```

```
c
-----
(5.2,6.05)
(133.42,144.95)
```

Мы продемонстрировали создание бинарного оператора. Чтобы создать унарный оператор, просто опустите `leftarg` (для левого унарного) или `rightarg` (для правого унарного). Обязательными в `CREATE OPERATOR` являются только предложение `procedure` и объявления аргументов. Предложение `commutator`, добавленное в данном примере, представляет необязательную подсказку для оптимизатора запросов. Подробнее о `commutator` и других подсказках для оптимизатора рассказывается в следующем разделе.

Функции для перечислений

Для типов перечислений предусмотрено несколько функций, которые позволяют сделать код чище, не «зашивая» в нём конкретные значения перечисления. Эти функции перечислены в таблице ниже.

В этих примерах подразумевается, что перечисление создано так:

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green',
  'blue', 'purple');
```

Функция	Описание	Пример	Результат примера
<code>enum_first(anyenum)</code>	Возвращает первое значение заданного перечисления	<code>enum_first(null::rainbow)</code>	red
<code>enum_last(anyenum)</code>	Возвращает последнее значение заданного перечисления	<code>enum_last(null::rainbow)</code>	purple
<code>enum_range(anyenum)</code>	Возвращает все значения заданного перечисления в упорядоченном массиве	<code>enum_range(null::rainbow)</code>	{red, orange, yellow, green, blue, purple}
<code>enum_range(anyenum, anyenum)</code>	Возвращает набор значений, лежащих между двумя заданными, в виде упорядоченного массива. Эти значения должны принадлежать одному перечислению. Если первый параметр равен NULL, функция возвращает первое значение перечисления, а если NULL второй — последнее.	<code>enum_range('orange'::rainbow, 'green'::rainbow)</code>	{orange, yellow, green}
		<code>enum_range(NULL, 'green'::rainbow)</code>	{red, orange, yellow, green}
		<code>enum_range('orange'::rainbow, NULL)</code>	{orange, yellow, green, blue, purple}

Пользовательские агрегатные функции

Агрегатные функции в PostgreSQL определяются в терминах *значений состояния* и *функций перехода состояния*. То есть агрегатная функция работает со значением состояния, которое меняется при обработке каждой последующей строки. Чтобы определить агрегатную функцию, нужно выбрать тип данных для значения состояния, начальное значение состояния и функцию перехода состояния. Функция перехода состояния принимает предыдущее значение состояния и входное агрегируемое значение для текущей строки и возвращает новое значение состояния. Также можно указать *функцию завершения*, на случай, если ожидаемый результат агрегатной функции отличается от данных, которые сохраняются в изменяющемся значении состояния. Функция завершения принимает конечное значение состояния и возвращает то, что она хочет вернуть в виде результата агрегирования. В принципе, функции перехода и завершения представляют собой просто обычные функции, которые также могут применяться вне контекста агрегирования. (На практике для большей производительности часто создаются специализированные функции перехода, которые работают, только когда вызываются при агрегировании.)

Таким образом, помимо типов данных аргументов и результата, с которыми имеет дело пользователь агрегатной функции, есть также тип данных внутреннего состояния, который может отличаться от этих типов.

Если мы определяем агрегат, не использующий функцию завершения, наш агрегат будет вычислять бегущее значение функции по столбцам каждой строки. Примером такой агрегатной функции является `sum`. Вычисление `sum` начинается с нуля, а затем к накапливаемой сумме всегда прибавляется значение из текущей строки. Например, если мы хотим сделать агрегатную функцию

`sum` для комплексных чисел, нам потребуется только функция сложения для такого типа данных. Такая агрегатная функция может быть определена так:

```
CREATE AGGREGATE sum (complex)
(
  sfunc = complex_add,
  stype = complex,
  initcond = '(0,0)'
);
```

Использовать её можно будет так:

```
SELECT sum(a) FROM test_complex;
```

```
sum
-----
(34,53.9)
```

(Заметьте, что мы задействуем перегрузку функций: в системе есть несколько агрегатных функций с именем `sum`, но PostgreSQL может определить, какая именно из них применима к столбцу типа `complex`.)

Определённая выше функция `sum` вернёт ноль (начальное значение состояния), если в наборе данных не окажется значений, отличных от `NULL`. У нас может возникнуть желание вернуть `NULL` в этом случае — стандарт SQL требует, чтобы `sum` работала так. Мы можем добиться этого, просто опустив фразу `initcond`, так что начальным значением состояния будет `NULL`. Обычно это будет означать, что в `sfunc` придётся проверять входное значение состояния на `NULL`. Но для `sum` и некоторых других простых агрегатных функций, как `max` и `min`, достаточно вставить в переменную состояния первое входное значение не `NULL`, а затем начать применять функцию перехода со следующего значения не `NULL`. PostgreSQL сделает это автоматически, если начальное значение состояния равно `NULL` и функция перехода помечена как «strict» (то есть не должна вызываться для аргументов `NULL`).

Ещё одна особенность поведения по умолчанию «строгой» функции перехода — предыдущее значение состояния остаётся без изменений, когда встречается значение `NULL`. Другими словами, значения `NULL` игнорируются. Если вам нужно другое поведение для входных значений `NULL`, не объявляйте свою функцию перехода строгой (`strict`); вместо этого, проверьте в ней поступающие значения на `NULL` и обработайте их, как требуется.

Функция `avg` (вычисляющая среднее арифметическое) представляет собой более сложный пример агрегатной функции. Ей необходимы два компонента текущего состояния: сумма входных значений и их количество. Окончательный результат получается как частное этих величин. При реализации этой функции для значения состояния обычно используется массив. Например, встроенная реализация `avg(float8)` выглядит так:

```
CREATE AGGREGATE avg (float8)
(
  sfunc = float8_accum,
  stype = float8[],
  finalfunc = float8_avg,
  initcond = '{0,0,0}'
);
```

Примечание

Функция `float8_accum` принимает массив из трёх, а не двух элементов, так как в дополнение к количеству и сумме значений она подсчитывает ещё сумму их квадратов. Это сделано для того, чтобы её можно было применять для `avg` и для некоторых других агрегатных функций.

Вызовы агрегатных функций SQL допускают указания `DISTINCT` и `ORDER BY`, которые определяют, какие строки и в каком порядке будут поступать в функцию перехода агрегата. Это реализовано на заднем плане и непосредственно не затрагивает функции, поддерживающие работу агрегата.

CREATE AGGREGATE

CREATE AGGREGATE — создать агрегатную функцию

Синтаксис

```
CREATE AGGREGATE имя ( [ режим_аргумента ] [ имя_аргумента ] тип_данных_аргумента [ ,  
... ] ) (  
SFUNC = функция_состояния,  
STYPE = тип_данных_состояния  
[ , SSPACE = размер_данных_состояния ]  
[ , FINALFUNC = функция_завершения ]  
[ , FINALFUNC_EXTRA ]  
[ , COMBINEFUNC = комбинирующая_функция ]  
[ , SERIALFUNC = функция_сериализации ]  
[ , DESERIALFUNC = функция_десериализации ]  
[ , INITCOND = начальное_условие ]  
[ , MSFUNC = функция_состояния_движ ]  
[ , MINVFUNC = обратная_функция_движ ]  
[ , MSTYPE = тип_данных_состояния_движ ]  
[ , MSSPACE = размер_данных_состояния_движ ]  
[ , MFINALFUNC = функция_завершения_движ ]  
[ , MFINALFUNC_EXTRA ]  
[ , MINITCOND = начальное_условие_движ ]  
[ , SORTOP = оператор_сортировки ]  
[ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]  
)
```

```
CREATE AGGREGATE имя ( [ [ режим_аргумента ] [ имя_аргумента ] тип_данных_аргумента [ ,  
... ] ]  
ORDER BY [ режим_аргумента ] [ имя_аргумента ] тип_данных_аргумента [ , ... ] ) (  
SFUNC = функция_состояния,  
STYPE = тип_данных_состояния  
[ , SSPACE = размер_данных_состояния ]  
[ , FINALFUNC = функция_завершения ]  
[ , FINALFUNC_EXTRA ]  
[ , INITCOND = начальное_условие ]  
[ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]  
[ , HYPOTHETICAL ]  
)
```

или старый синтаксис

```
CREATE AGGREGATE имя (  
BASETYPE = базовый_тип,  
SFUNC = функция_состояния,
```

```

STYPE = тип_данных_состояния
[ , SSPACE = размер_данных_состояния ]
[ , FINALFUNC = функция_завершения ]
[ , FINALFUNC_EXTRA ]
[ , COMBINEFUNC = комбинирующая_функция ]
[ , SERIALFUNC = функция_сериализации ]
[ , DESERIALFUNC = функция_десериализации ]
[ , INITCOND = начальное_условие ]
[ , MSFUNC = функция_состояния_движ ]
[ , MINVFUNC = обратная_функция_движ ]
[ , MSTYPE = тип_данных_состояния_движ ]
[ , MSSPACE = размер_данных_состояния_движ ]
[ , MFINALFUNC = функция_завершения_движ ]
[ , MFINALFUNC_EXTRA ]
[ , MINITCOND = начальное_условие_движ ]
[ , SORTOP = оператор_сортировки ]
)

```

Описание

CREATE AGGREGATE создаёт новую агрегатную функцию. Некоторое количество базовых и часто используемых агрегатных функций включено в дистрибутив. Но если нужно адаптировать их к новым типам или создать недостающие агрегатные функции, это можно сделать с помощью команды CREATE AGGREGATE.

Если указывается имя схемы (например, CREATE AGGREGATE myschema.myagg . . .), агрегатная функция создаётся в указанной схеме. В противном случае она создаётся в текущей схеме.

Агрегатная функция идентифицируется по имени и типам входных данных. Две агрегатных функции в одной схеме могут иметь одно имя, только если они работают с разными типами данных. Имя и тип(ы) входных данных агрегата не могут совпадать с именем и типами данных любой другой обычной функции в той же схеме. Это же правило действует при перегрузке имён обычных функций (см. [CREATE FUNCTION](#)).

Простую агрегатную функцию образуют одна или две обычные функции: функция перехода состояния `функция_состояния` и необязательная функция окончательного вычисления `функция_завершения`. Они используются следующим образом:

```

функция_состояния( внутреннее-состояние, следующие-значения-данных ) ---> следующее-
внутреннее-состояние
функция_завершения( внутреннее-состояние ) ---> агрегатное_значение

```

PostgreSQL создаёт временную переменную типа `тип_данных_состояния` для хранения текущего внутреннего состояния агрегата. Затем для каждой поступающей строки вычисляются значения аргументов агрегата и вызывается функция перехода состояния с текущим значением состояния и полученными аргументами; эта функция вычисляет следующее внутреннее состояние. Когда таким образом будут обработаны все строки, вызывается завершающая функция, которая должна вычислить возвращаемое значение агрегата. Если функция завершения отсутствует, просто возвращается конечное значение состояния.

Агрегатная функция может определить начальное условие, то есть начальное значение для внутренней переменной состояния. Это значение задаётся и сохраняется в базе данных в виде строки типа `text`, но оно должно быть допустимым внешним представлением константы типа данных переменной состояния. По умолчанию начальным значением состояния считается NULL.

Если функция перехода состояния объявлена как «strict» (строгая), её нельзя вызывать с входными значениями NULL. В этом случае агрегатная функция выполняется следующим образом. Строки со значениями NULL игнорируются (функция перехода не вызывается и предыдущее значение состояния не меняется) и если начальное состояние равно NULL, то в первой же строке, в которой все входные значения не NULL, первый аргумент заменяет значение состояния, а функция перехода вызывается для каждой последующей строки, в которой все входные значения не NULL. Это поведение удобно для реализации таких агрегатных функций, как `max`. Заметьте, что такое поведение возможно, только если *тип_данных_состояния* совпадает с первым *типом_данных_аргумента*. Если же эти типы различаются, необходимо задать начальное условие не NULL или использовать нестрогую функцию перехода состояния.

Если функция перехода состояния не является строгой, она вызывается безусловно для каждой поступающей строки и должна сама обрабатывать вводимые значения и переменную состояния, равные NULL. Это позволяет разработчику агрегатной функции полностью управлять тем, как она воспринимает значения NULL.

Если функция завершения объявлена как «strict» (строгая), она не будет вызвана при конечном значении состояния, равном NULL; вместо этого автоматически возвращается результат NULL. (Разумеется, это вполне нормальное поведение для строгих функций.) Когда функция завершения вызывается, она в любом случае может вернуть значение NULL. Например, функция завершения для `avg` возвращает NULL, если определяет, что было обработано ноль строк.

Иногда бывает полезно объявить функцию завершения как принимающую не только состояние, но и дополнительные параметры, соответствующие входным данным агрегата. В основном это имеет смысл для полиморфных функций завершения, которым может быть недостаточно знать тип данных только переменной состояния, чтобы вывести тип результата. Эти дополнительные параметры всегда передаются как NULL (так что функция завершения не должна быть строгой, когда применяется `FINALFUNC_EXTRA`), но в остальном это обычные параметры. Функция завершения может выяснить фактические типы аргументов в текущем вызове, воспользовавшись системным вызовом `get_fn_expr_argtype`.

Агрегатная функция может дополнительно поддерживать *режим движущегося агрегата*. Для этого режима требуются параметры `MSFUNC`, `MINVFUNC` и `MSTYPE`, а также могут задаваться `MSSPACE`, `MFINALFUNC`, `MFINALFUNC_EXTRA` и `MINITCOND`. За исключением `MINVFUNC`, эти параметры работают как соответствующие параметры простого агрегата без начальной буквы `m`; они определяют отдельную реализацию агрегата, включающую функцию обратного перехода.

Если в список параметров добавлено указание `ORDER BY`, создаётся особый тип агрегата, называемый *сортирующим агрегатом*; с указанием `HYPOTHETICAL` создаётся *гипотезирующий агрегат*. Эти агрегаты работают с группами отсортированных значений и зависят от порядка сортировки, поэтому определение порядка сортировки входных данных является неотъемлемой частью их вызова. Кроме того, они могут иметь *непосредственные* аргументы, которые вычисляются единожды для всей процедуры агрегирования, а не для каждой поступающей строки. Гипотезирующие агрегаты представляют собой подкласс сортирующих агрегатов, в которых непосредственные аргументы должны совпадать, по количеству и типам данных, с агрегируемыми аргументами. Это позволяет добавить значения этих непосредственных аргументов в набор агрегируемых строк в качестве дополнительной «гипотетической» строки.

Агрегатная функция может дополнительно поддерживать *частичное агрегирование*. Для этого требуется задать параметр `COMBINEFUNC`. Если в качестве *типа_данных_состояния* выбран `internal`,

обычно уместно также задать `SERIALFUNC` и `DESERIALFUNC`, чтобы было возможно параллельное агрегирование. Заметьте, что для параллельного агрегирования агрегатная функция также должна быть помечена как `PARALLEL SAFE` (безопасная для распараллеливания).

2. Выполнение лабораторной работы

1. Выполнить вставку тестовых данных в таблицы, созданные в ходе выполнения лабораторной работы 9.
2. Сделать запросы выборки с условием к таблицам предку и потомку, только потомку и только предку.
3. Придумать и создать пользовательский оператор для своей предметной области
4. Придумать и создать пользовательскую агрегатную функцию для своей предметной области

3. Содержание отчета

- Текст задания.
- физическую модель БД.
- наборы данных, содержащихся в таблицах БД
- текст запросов на SQL;
- код операторов
- пример выполнения операторов
- наборы данных, возвращаемые запросами.

4. Варианты заданий

Варианты заданий приведены в Приложении 3.

Приложение 1 Распределение баллов

Семестр 6

№	Наименование лабораторной	Количество баллов	Продельный № недели сдачи
1.	Разработка документной базы данных	10	4
2.	Манипулирование данными в документной базе данных	15	8
3.	Объектно-реляционные базы данных. Проектирование и создание	10	12
4.	Объектно-реляционные базы данных. Манипуляция данными и пользовательские операторы	15	16
	Итого	50	

Приложение 3 Варианты заданий лабораторных 9-12

1. Программа для рисования векторной графики: автор изображения (пользователь), примитивы (они же элементы графики), координаты левого верхнего угла отображаемого объекта, и её размеры
 - а. Элемент графики, название/текст которых содержит слово «выбор», но не начинается с него
 - б. Элементы изображения (графики), связанные друг с другом
 - в. Ширина изображения в пикселях (от максимальной суммы координаты по горизонтали с шириной отнять минимальную левую координату)
 - г. Пользователь- автор, изображений с минимальным количеством элементов
 - д. Вершина, нет смежных элементов-треугольников
2. Садоводческое товарищество: объекты общего пользования (колонки, помойка, магазин)

участки, в том числе общего пользователя, владельцы с учетом совместной собственности, линии/номер участка, площадь стоимость постройки, тип построек, взносы в фонд садоводства

 - а. номера участков владельцев с отчеством, заканчивающимся на «ич»
 - б. участки в том числе общего пользования, на которых зарегистрировано более 1 типа постройки
 - в. Владелец (владельцы) участка минимальной площади
 - г. Владельцы максимального количества участков
 - д. Участки, на которых есть колонки, но нет магазинов

3. Управляющая компания: квартиры, коммерческие помещения, владелец, количество комнат, площадь, жилая площадь, этаж, тарифы для юридических и физических лиц
- а. юр. лица владельцы, названия которых начинаются на 'и'
 - б. дом, где у одного владельца есть квартиры на 1 и 2 этажах
 - в. тариф для физ лиц с максимальной величиной
 - г. этаж, на котором больше всего коммерческих помещений
 - д. владелец или юр лицо, у которого нет неоплаченных квитанций в этом году, но есть двухкомнатные квартиры
4. парк: статуи, фонтаны, деревья ,породы, дата высадки, дата обрезки, расположение, аллеи
- а. аллеи, на которых встречаются разные виды кленов (клен в названии)
 - б. аллеи, на которых есть и статуи и фонтаны
 - в. дерево, которое было посажено позже всех
 - г. порода, деревьев которой больше всего
 - д. аллея, на которой нет фонтанов
5. расписание экзаменов и зачетов: , даты зачетов(диапазон- неделя),даты экзаменов, дисциплина, преподаватель, группа, аудитория
- а. дисциплины, имеющим в названии слова «базы данных»
 - б. аудитории, где в один день проходит несколько экзаменов
 - в. группы самого первого экзамена
 - г. аудитории, в которых проходит больше всего зачетов
 - д. преподаватель, не принимающий экзамены у группы 4031
6. детская поликлиника а: разграничен прием здоровых и больных детей датами (днями недели),прием, пациент, процедура/прием, врач, стоимость
- а. пациенты, приходившие на любые процедуры, связанные с электрофорезом
 - б. кабинет, где проходит приемы нескольких врачей
 - в. процедуры с наибольшей стоимостью
 - г. пациент, ходивший к наименьшему количеству врачей
 - д. пациент, не ходивший в дни здорового ребенка
7. личный кабинет студента: практики, вкр, дисциплина, работа, тип работы, студент, преподаватель, дата сдачи , баллы
- а. дисциплины, начинающиеся со слова «автоматизирован»
 - б. дисциплина, по которой есть несколько типов работ
 - в. студент, сдавший ВКР раньше всех
 - г. студент, прикрепивший в этом месяце наибольшее число работ

- д. преподаватель, которому не прикрепляли отчетов по практике
8. оборудование театра: роль, спектакль, реквизит (для роли /как часть костюма), название костюма, деталь костюма, размер, автор модели, дата разработки (учесть, чтобы в спектаклях идущих одновременно не использовались одинаковые объекты)
- а. реквизит, имеющие в названии слово «жезл»
 - б. спектакль в костюме к которому, есть и куртка и штаны
 - в. автор, разработавший самый старый из костюмов
 - г. спектаклю, к которому разработано наибольшее число костюмов
 - д. автор, не разрабатывавший реквизит к «Золушке»
9. охраняемые парковки: адрес парковки, машина, тип машины (грузовая, легковая, с прицепом и т.д.), , и места под разные типы машин, под ремонт,владелец, место, рег. номер машины, дата и время заезда, дата и время выезда
- а. все парковки, расположенные на линиях (не улицах или проспектах)(улица в адресе содержит «линия»)
 - б. владелец машины, у которого есть машины разных типов
 - в. владелец машин, заезжавший раньше всех
 - г. владелец машины, останавливавшийся на ремонт на минимальном числе парковок
 - д. владелец, не парковавшийся на Вознесенском проспекте
10. База данных по теме «сбор в поездку» с учетом поездок за границу и обязательного пакета документов должна иметь структуру, позволяющую реализовать следующие запросы:
- а. Найти какие поездки за текущий год имели в названии слово «поход», но не заканчивались на него
 - б. Найти в какую поездку брались предметы нескольких категорий
 - в. Найти пользователя, у которого в текущем месяце больше 2 поездок
 - г. Какие категории вещей не берутся в поездку типа «конференция»
 - д. Найти вид документов, с наибольшим количеством вхождений в обязательные для поездок за границу
11. База данных сериалов и фильмов.
- а. Сериал, в названии которого есть слово доктор, но оно не первое
 - б. Персонаж, которого в одном фильме играют разные актеры
 - в. Сезон и сериал этого сезона, с максимальным количеством эпизодов
 - г. Актер, играющий в самом старом фильме
 - д. Актер, который играл в комедийных сериалах, но не в исторических фильмах
12. вакансии: волонтерские позиции, название вакансии, организация работодатель, адрес работодателя, диапазон зарплаты, требования к образованию, Обязанности, график работы, требования обязательные, желательные, дата выставления вакансии

- а. вакансии, имеющие в названии SQL, но не заканчивающиеся на него
 - б. работодатели в Санкт-Петербурге, выставившие несколько вакансий
 - в. вакансии с наибольшей зарплатой
 - г. волонтерские позиции с максимальным количеством требований
 - д. вакансии, в которых нет требования к опыту работы
13. калькулятор бюджета физического лица: пользователь калькулятора (разные уровни доступа), категория дохода (продажа, зарплата), категория расхода (еда, счета за КУ, здоровье ...), статьи дохода и расхода, дата расхода/дохода
- а. статьи категорий, которые относятся к спорту (содержат слово спорт)
 - б. месяц, в котором были разные статьи дохода
 - в. категория, по которой наибольшие расходы в текущем году
 - г. категория, по которой не было расходов в феврале
 - д. пользователь, добавивший наименьшее количество статей
14. Книга контактов: организации с их контактами и временем работы, люди, метки (друзья, коллеги, семья), дни рождения, телефон, электронная почта, примечания к человеку или контакту, праздники, которые празднуют люди
- а. организации, у которых номер телефона начинается на 8-921,
 - б. люди, которые относятся 2 меткам одновременно
 - в. самые старые люди среди контактов, празднующих 8 марта
 - г. месяц, когда есть праздники у коллег, но нет у семьи
 - д. метки, к которым относится минимальное количество людей
15. вузы и колледжи для абитуриента: учебные заведения, разделенные по уровням образования, город, вуз, факультеты, направления, направленности, ЕГЭ которые нужно сдать, дата начала приемной кампании. СПО- среднее профессиональное образование (техникум , колледж) (Направление -09.03.04 «Программная инженерия», Направленность — его конкретизация «Разработка программно-информационных систем», именно направленность закреплена за кафедрой и соответственно факультетом)
- а. направленности СПО , в которых есть слово автоматизированный
 - б. направление, на которое надо сдавать историю и обществознание
 - в. факультет вуза, принимающий на наименьшее количество направлений
 - г. колледж, с самым поздним началом приемной кампании
 - д. направление, на которое не надо сдавать ЕГЭ по математике, но надо по истории
16. школьные экскурсии: тип (развлекательная/образовательная), дисциплины к которым имеет отношение образовательная экскурсия, стоимость с человека, список участников, ответственный за проведение учитель, категории участников экскурсии (участвует, организует, ассистирует при проведении (старшие классы у младших или родители))
- а. экскурсии в усадьбы (слово усадьба в любом месте названия)
 - б. экскурсии, относящиеся к 2 дисциплинам

- в. учащиеся, которые не ездили в музей истории религии
- г. экскурсия, собравшая наибольшее число ассистентов
- д. учителя, отвечающие за самые дорогие экскурсии

17. питомник собак и кошек: животное, владелец, порода, родители, медали с выставок, дата рождения

- а. породы, названия которых содержат слово «колли», но не начинается с него
- б. кошки, у родителей которых разные владельцы
- в. владелец, у которого есть йоркширские терьеры, но нет кошек
- г. порода, собак которой меньше всего
- д. владельцы самых старых кошек

18. служба доставки с учетом разницы физических и юридических лиц: адрес доставки, контактное лицо, стоимость посылки, диапазон желаемого времени доставки. время доставки фактическое, вес посылки, отметка о доставке, фирма отправитель (у юр. лица есть контактное лицо (человек))

- а. все посылки, отправляемые на улицу, в названии которой есть «ая», но это не единственные буквы в нем
- б. контакт(человек) юридического лица, получавшее посылки по разным адресам
- в. фирма, отправившая меньше всего посылок для физлиц.
- г. посылка с самым большим весом
- д. контактное лицо, никогда не получавшее посылок в марте

19. туристический путеводитель с учетом не только «адресных» достопримечательностей, но и природных: местность, город, достопримечательность, адрес, тип достопримечательности (памятник, архитектурный комплекс, природный комплекс), дата создания и выставок

- а. достопримечательности, в которых есть слово «мать», но с него название не начинается
- б. улица, на которой есть 2 разных вида достопримечательности
- в. город, в котором не проводится выставок
- г. местность, где больше всего природных достопримечательностей
- д. города с самыми старыми достопримечательностями

20. метрополитен и трамвай: линия, станция, время закрытия, время открытия, адрес выходов, время проезда между станциями, дата открытия станции

- а. станции трамвая, в названии которых есть слово «площадь», но оно на него не начинается
- б. пересадочные станции с линии 1 на линию 2
- в. маршрут трамвая с самым большим временем проезда
- г. станция метро с самым поздним открытием
- д. линия на которой нет перегона больше 3 минут

21. Медпункт вуза: студенты, сотрудники вуза, данные флюорографии (результат, дата), данные прививок, жалобы сотрудников (прием в медпункте)
- а. все сотрудники, женского пола (отчество, оканчивается на «на»)
 - б. сотрудники организации, делавшие за последний год прививки от 2 разных заболеваний
 - в. сотрудник, последним сделавший флюорографию
 - г. группа студента с самым большим количеством жалоб
 - д. сотрудник, не пошедший проф.осмотр в этом году
22. Инвентаризация: ответственное лицо, оборудование, категории имущества (компьютерная техника, канцелярские товары (могут не иметь помещения, а приписаны быть к отделу, не списываются), мебель, освещение, спец. оборудование и т.д.), помещения, дата поставки, дата списания
- а. любые помещения, где стоит компьютерная мебель (категория мебель и слово «комп»)
 - б. материально ответственное лицо, у которого подотчетны и канцелярия и оборудование
 - в. имущество, списанное последним
 - г. мат. ответственное лицо, ответственное за наибольшее количество канцелярских товаров
 - д. помещение, в котором нет компьютерной техники
23. Календарь корпоративных мероприятий: время дата, событие, статус события для участника (обязательное, рекомендуемое, нейтральное), должности и подразделения участников, комнаты проведения, отдельный учет сотрудника как организатора.
- а. Все мероприятия, текущего года, в названии которых есть слово конференция, но с него с названия не начинаются
 - б. Мероприятия, в которых участвуют сотрудники 2 и более подразделений
 - в. организаторы самых рано заканчивающихся мероприятий
 - г. Событие, с самым маленьким количеством пришедших на него участников.(участники на мероприятие могут быть приглашены и при этом могут прийти/не прийти).
 - д. Сотрудник, не посещавший мероприятий в июле этого года
24. Распределение обязанностей: обязанность, название, частота выполнения, категория обязанности (домашняя, оплата счетов, посещение мероприятий, общественная), разные статусы людей (выполнение и назначение обязанностей)
- а. Люди, выполняющие обязанности, имеющие в названии слово «уборка», но не начинающиеся на него
 - б. Обязанность, которую выполняет более одного человека
 - в. Ежегодная обязанность, которая выполняется последней по дате (дню)
 - г. Категория, в которой обязанностей больше среднего
 - д. Человек, который не выполняет обязанности из категории «ремонт», но назначал их
25. Багтрекинг: проекты, баги, фичи (новые функции), тестировщики, разработчики (разные статусы людей (исправление и добавление багов), тестировщики не исправляют)

- а. Баги, в названии которых содержится слово «размер», но оно не первое
- б. Функции(фичи), относящаяся к 2 различным проектам
- в. **Тривиальный** баг, который добавили последним в этом году
- г. Тестировщик, с самым маленьким количеством добавленных багов
- д. Разработчик, у которого нет незавершенных задач(реализация фичи или исправление бага) в этом году

26. Домашняя аптечка. Группа лекарств, срок(дата) годности, перевязочные средства и мед техника, расходные средства для мед техники (например, тест-полоски для глюкометра).

- а. Найти лекарство, которое содержит текст «септ», но не в конце слова
- б. Найти лекарство, у которого есть аналоги по действующему веществу в форме спрея
- в. Найти расходные средства для мед техники самым близким, истекшим сроком годности.
- г. Найти группы лекарств, с количеством лекарств больше среднего.
- д. Найти категорию перевязочных средств, в которой нет ни одного средства на букву б

27. Создайте базу спортзала: абонементы, свободное посещение тренажеров/залов, виды спорта, тренеры, дата покупки абонемента, тип абонемента, инвентарь к виду спорта. Свободное посещение часто подразумевает тренера, прикрепленного к клиенту

- а. виды спорта, на которые есть абонементы со словами «выходного дня» в конце названия типа;
- б. вид спорта, по которому занятия ведут различные тренеры;
- в. абонемент (предлагаемый) на румбу с максимальным сроком действия в абонементах;
- г. тренеры, которые ведут занятия по максимальному количеству видов спорта;
- д. тренеры, не ведущие самбо, но ведущие спорт, к которому нужны гантели;

28. Создайте базу данных фанфикшна (фанфикшн – это фанатские фантазии выраженные в дополнениях, переработках и продолжениях культовых произведений.) и фанарта для хранения следующих сведений: оригинальное произведение (фандом), автор фанфика/фанарта (произведения по мотивам), наименование фанфика/фанарта, дата его добавление и последней редакции, наименование и текст глав фанфика. Фан-арт (англ. fan art) — разновидность творчества поклонников популярных произведений искусства, производное рисованное произведение, основанное на каком-либо оригинальном произведении (как правило, литературном или кинематографическом), использующее его идеи сюжета и (или) персонажей

- а. перечень фанфиков и фанартов, с названием на начинающимся на «Приключения», на это не единственное слово в нем ;
- б. фанфики или фанарты, написанные более чем по одному произведению (кроссоверы), вроде «Чужой против хищника»);
- в. самый новый фандом (фандомы);
- г. фандом, по которому создано наибольшее количество фанфиков и фанартов вместе;
- д. авторов, пишущих только произведения по «Гарри Поттеру»;

29. Создайте базу данных идиом и крылатых выражений для хранения следующих сведений: отдельные слова идиомы, идиома, толкование, пример использования, дата добавления идиомы (если есть), пользователь добавивший идиому в словарь. Необходимо дополнительно хранить авторские идиомы, которые надо связать с произведением, в котором они появились. («Счастливые часов не наблюдают»- Горе от ума, «Может быть, тебе дать ещё ключ от квартиры, где деньги лежат?»-12 стульев)
- а. перечень идиом, содержащих слово «деньги», но не начинающихся с него;
 - б. пары идиом из одного произведения;
 - в. пользователь, добавивший меньше всего идиом;
 - г. идиомы, последние по алфавиту;
 - д. пользователи, добавлявшие идиомы со словом «труд», но не добавлявшие идиом в 2024;