Нерелационни бази данни

Ръководство

за лабораторни упражнения

Росен Иванов

2021

Ръководството е предназначено за студентите от катедра “Компютърни Системи и Технологии”, специалност “Софтуерно и Компютърно Инженерство”, които изучават дисциплината “Нерелационни Бази Данни” (НБД). Дисциплината НБД има за цел да запознае студентите със съвременните тенденции в областта на създаване на услуги, които използват нерелационни бази данни. След завършване на курса на обучение студентите трябва да могат да проектират документен тип бази данни MondoDB и IBM Cloudant (CouchDB), както и да създават приложения на Java, Node.js и Express.js за достъп до тези бази данни. Студентите трябва да имат начални знания за обектно ориентирано програмиране с Java и JavaScript.

Лабораторните упражнения са разработени достатъчно подробно, за да могат студентите с недостатъчни знания в областта на програмирането успешно да реализират поставените задачи. След завършване на курса те ще могат: 1) Да проектират MongoDB и CouchDB бази данни; 2) Да създават приложения и услуги използващи MongoDB чрез Java и Node.js; 3) Да създават serverless приложения чрез IBM Cloudant; 4) Да генерират заявки за търсене, агрегиране и филтриране на съдържание; 5) Да анализират производителността на своите бази данни; 6) Да работят с облачно базирани услуги и да създават своите бази в облака на Amazon и IBM и 7) Да създават приложения с достъп до облачно базирани NoSQL бази данни.

Ръководството може да се използва и от студентите от други специалности, които имат начални знания в областта на обектно ориентираното програмиране с Java и JavaScript. Всички примери от ръководството са достъпни за безплатно сваляне.

Нерелационни бази данни

Ръководство за лабораторни упражнения

ISBN 978-954-683-642-7

2021

СЪДЪРЖАНИЕ

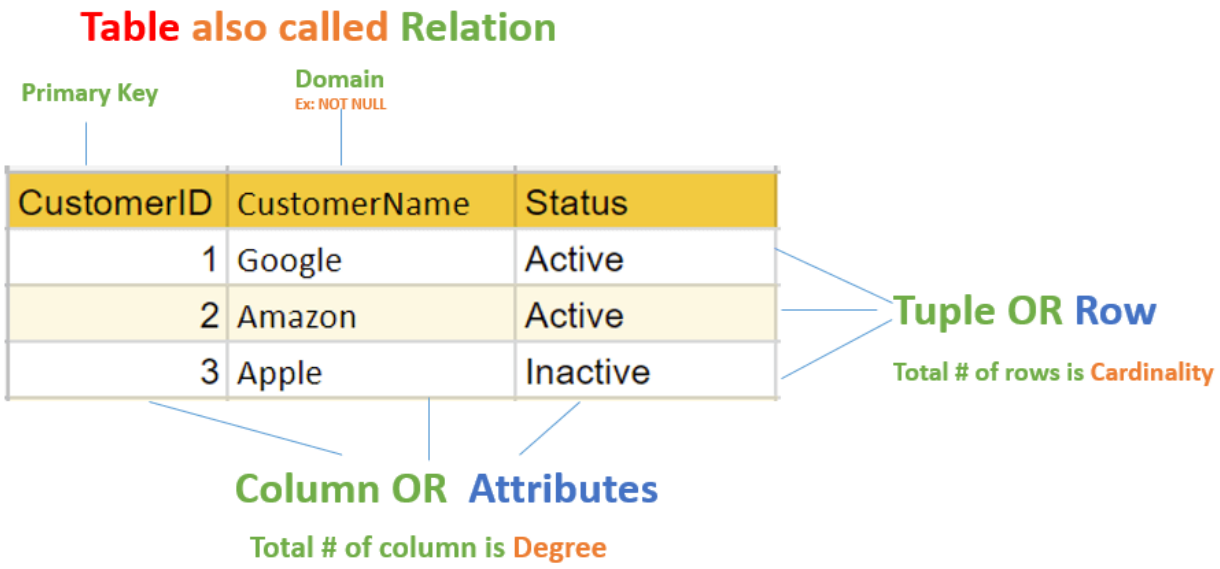
Упражнение № 1

Релационни бази данни. SQL заявки.

I. Въведение

Релационните бази данни съществуват от 70-те години на миналия век. Терминът "релационна база данни" е въведен от E. F. Codd през 1970 г. в своя научно-изследователския труд "Релационен модел на данните за големи споделени банки от данни." Според Microsoft релационната база данни е вид база данни, която съхранява и осигурява достъп до данни, които са свързани помежду си. Релационните бази данни използват **релационен модел** за описание на данните. Този модел дефинира начин за представяне на данни в таблици и връзките между тези таблици. При релационния модел логическите структури (таблици) са отделени от физическите структури за съхранение на данните. Това разделяне означава, че управлението на физическото съхранение на данните не се отразява на достъпа до тези данни като логическа структура. Например, преименуването на файл на база данни не води до преименуване на таблиците, съхранявани в него. Основният начин за изпращане на заявки към релационните бази данни е чрез езика за структурирани заявки **SQL**. Той е основан на релационната алгебра и предоставя вътрешно съгласуван математически език, който улеснява подобряването на производителността на всички заявки към бази данни. Предназначен е за работа със структурирани данни. Той е декларативен език, защото описва какъв да бъде желания резултат.

На Фиг. 1.1 са показани основните термини, използвани при релационен модел за описание на данните.



Фиг. 1.1 Основни термини в релационните бази данни

Таблица (Table): В релационния модел връзките се записват във формат на таблица. Тя се съхранява заедно с нейните същности. Една таблица има две свойства - редове и колони. Редовете представляват *записи*, а колоните - *атрибути*.

Атрибут (Attribute): Всяка колона в една таблица. Атрибутите са свойствата, които определят дадена връзка, например “customerName” и “Status”.

Област на атрибута (Attribute domain): Всеки атрибут има някаква предварително определена стойност и обхват, които са известни като домейн на атрибута.

Кортеж (Tuple): Един ред от таблица, който съдържа един запис.

Степен (Degree): Общият брой на атрибутите, които се съдържат в релацията.

Кардиналност (Cardinality): Общият брой на редовете в таблицата.

Колона: Колоната представлява набор от стойности за определен атрибут.

Схема на релацията (Relation Schema): Схемата на релацията представя името на релацията с нейните атрибути.

Ключ на релацията (Relation key): Всеки ред има един, два или няколко атрибута, които се наричат ключ на релацията.

Инстанция на връзката (Relation instance): Инстанцията на релацията е крайно множество от кортежи в системите за управление на бази данни (СУБД). Релационните инстанции никога нямат дублиращи се кортежи.

Релационният модел организира данните в една или повече **таблици** (релации), всяка съставена от колони и редове. Редовете се наричат още **записи** или **кортежи**, а колоните – **полета** или **атрибути**. **Записът** е смислен и последователен начин за обединяване на информацията за нещо. Полето е отделен информационен елемент – тип елемент, който присъства във всеки запис. Например, в таблица “Студенти” всеки ред (запис) ще съдържа информацията за един студент. **Полетата** съдържат някакъв вид информация за студента, например неговото име. Релационният модел е добър за поддържане на последователност на данните в приложенията и копията на базата данни (наречени инстанции). Например, когато клиентът тегли пари чрез банкомат и след това преглежда баланса по сметката на мобилния си телефон, той очаква тегленето да бъде отразено незабавно в актуализиран баланс по сметката. Релационните бази данни са отлични в този вид **съгласуваност на данните**, като гарантират, че многобройните екземпляри на базата данни имат едни и същи данни през цялото време. Повечето нерелационни бази данни (NoSQL) осигуряват само "евентуална съгласуваност" на данните. Евентуалната съгласуваност е приемлива за много услуги, но не и за критичните бизнес операции, като транзакции при онлайн пазаруване и банкиране.

Четири ключови свойства определят транзакциите в релационните бази данни: атомичност, съгласуваност, изолираност и трайност - **ACID** бази данни:

- **Atomicy** (атомичност) – всяко действие (транзакция) с базата данни или се изпълнява изцяло или не се изпълнява (връща се грешка). Ако една част от транзакция се провали, се анулира цялата транзакция.
- **Consistency** (съгласуваност) – всяка транзакция променя базата данни от едно състояние с консистентна информация в друго консистентно състояние.
- **Isolation** (изолираност) – няма възможност за промяна на данни, които участват в все още незавършила операция. Това се гарантира с различни видове блокировки. Трябва да се има предвид, че тези блокировки има опасност да доведат до мъртва хватка и затова някои бази данни не гарантират 100% изолация.
- **Durability** (трайност) – промените в данните стават постоянни, само след като транзакцията бъде завършена.

В релационната база данни всеки ред в таблицата е запис с уникален идентификатор, наречен ключ. **Първичният ключ** (primary key) е колона, която се използва за еднозначно идентифициране на всеки ред от една таблица, например факултетен номер на студент. Редовете в една таблица могат да бъдат свързани с редове в други таблици чрез доба-

вяне на колона за уникалния ключ на свързания ред. Такива колони са известни като външни ключове. **Външният ключ** (foreign key) е набор от атрибути в дадена таблица, които се отнасят към първичния ключ на друга таблица. При обработката на данните в базата данни се предполага възможност за избор или промяна на един и само на един ред в дадена таблица. Затова повечето физически реализации имат уникален първичен ключ за всеки ред в таблицата. В някои случаи се налага две или повече полета да формират първичен ключ. Когато се състои от повече от една колона, първичният ключ се нарича **съставен ключ** (composite key).

II. Проектиране на релационни бази данни

При проектиране на базата данни трябва да се следи за:

- Намаляване на излишните данни в базата чрез разделяне на изходната информация на необходимия брой таблици.
- Правилен подбор на първичните и външни ключове.
- Отговаря ли базата данни на изискванията на клиента от гледна точка на надеждност, сигурност, бързодействие, мащабируемост и изготвяне на справки и отчети?

Проектирането на базата данни преминава през следните по-важни стъпки:

- Определяне на предназначението на базата данни.
- Намиране и организиране на необходимата информация.
- Разделяне на информацията в таблици.
- Превръщане на информационните елементи в колони.
- Задаване на първични ключове.
- Дефиниране на релациите между таблиците.
- Прилагане на правила за нормализиране на базата данни.
- Населяване на базата с тестови данни. Провеждане на необходимите тестове.

Правилата за **нормализация** се прилагат последователно, така че на всяка стъпка да се уверите, че вашият проект отговаря на едно от така наречените "правила за нормализация". Има пет широко разпространени правила за нормализация – от първо до пето. В повечето практически случаи е достатъчно да достигнете до трета нормална форма. В Табл. 1.1. е показано кога една същност е в една от първите три нормални форми и какво трябва да се направи, за да се премине до трета нормална форма. Нормализацията има както предимства, така и недостатъци.

Основни предимства на нормализацията:

- Намаляване на излишъка от данни и необходимото пространство.
- Подобряване на последователността на данните.
- Налагане на целостта на данните.
- Намаляване на разходите за актуализация.
- Осигуряване на максимална гъвкавост при отговаряне на ad hoc заявки.
- Позволява използването на паралелизъм, може да намали общия брой редове в блок.

Основни недостатъци на нормализацията:

- Много сложни заявки ще бъдат по-бавни, тъй като трябва да се извършат обединения за извличане на подходящи данни от няколко нормализирани таблици.
- Потребителите на базата данни трябва да вникнат в основния модел на данните на приложението, за да извършат правилно заявки при които се обединяват атрибути от няколко таблици.
- Формулирането на заявки на няколко нива е нетривиална задача.

Табл. 1.1 Нормални форми

0NF	
Имат ли някои атрибути няколко стойности за една инстанция на същност?	**Да:** Премахнете повтарящите се атрибути и повтарящите се групи. Създайте същност, която описва тези атрибутите. Обикновено ще трябва да добавите връзка, за да свържете старите и новите същности.
	Не: Таблицата е в 1NF
1NF	
Състои ли се първичния ключ от повече от един атрибут? Ако е така, зависи ли някоя от стойностите на атрибут само от част от този ключ?	**Да:** Премахнете всички частични функционални зависимости. Преместете атрибутите към същност, в която стойностите им зависят от целия идентификатор. Обикновено трябва да създадете нова същност и да добавите връзка, за да свържете старата и новата същност.
	Не: Таблицата е в 2NF
2NF	
Зависи ли някоя от стойностите на атрибут от друг атрибут, който не е част от първичния ключ на същността?	**Да:** Премахнете преходната функционална зависимост или производния атрибут. Преместете атрибутите в същност, в която стойностите им зависят изцяло от идентификатора. Обикновено ще трябва да създадете нова същност и да добавите връзка, за да свържете старата и новата същност.
	Не: Таблицата е в 3NF

III. Език за заявки SQL

SQL е стандартният език за работа с релационни бази данни. SQL може да се използва за вмъкване, търсене, актуализиране и изтриване на записи в базата данни. SQL може да извършва и други операции, включително оптимизиране и поддръжка на бази данни. Синтаксисът на SQL, използван в различните бази данни, е почти сходен, въпреки че някои СУБД използват някои специфични команди и дори собствени синтаксиси на SQL. SQL е стандартен език на ANSI (Американски национален институт по стандартизация).

За какво се използва SQL?

- SQL помага на потребителите да имат достъп до данните в СУБД.
- Помага да се опишат данните.
- Позволява ви да дефинирате данните в базата данни и да манипулирате конкретни данни.
- С помощта на SQL можете да създавате и премахвате бази данни и таблици.
- SQL ви предлага да създавате изглед и да съхранявате резултата.

Следва списък на някои от най-често използваните команди на SQL:

- CREATE - определя схемата на структурата на базата данни.
- INSERT - вмъква данни в ред на таблица.
- UPDATE - актуализира данните в базата данни.
- DELETE - премахва един или повече редове от таблица.
- DROP - премахва таблици и бази данни.
- SELECT – извличане на данни.

Командата **INSERT INTO** се използва за съхраняване на данни в таблиците. Тази команда създава нов ред в таблицата.

Командата **DELETE** се използва за изтриване на редове от таблиците на базата данни, които вече не са необходими. Тя изтрива целия ред от таблицата и връща броя на изтритите редове. Командата DELETE е полезна за изтриване на временни или остарели данни от вашата база данни. Тя може да изтрие повече от един ред от таблица с една заявка. Това е предимство при премахване на голям брой редове от таблица. След като веднъж е изтрит ред, той не може да бъде възстановен. Поради това се препоръчва да се правят резервни копия на базата данни, преди да се изтриват каквито и да било данни от базата данни. Това може да ви позволи да възстановите базата данни и да прегледате данните по-късно, ако това се наложи.

Командата **UPDATE** се използва за промяна на редове в таблица. Тя може да се използва за актуализиране на едно поле или на няколко полета едновременно. Тя може да се използва и за актуализиране на таблица със стойности от друга таблица.

Команда **SELECT** се използва за извличане на данни от базата данни. Целта на SELECT е да върне от таблиците на базата данни един или повече редове, които отговарят на зададени критерии. Синтаксисът на команда SELECT е следния:

```
SELECT [DISTINCT|ALL ] { * | [fieldExpression [AS newName]} FROM tableName
[alias] [WHERE condition][GROUP BY fieldName(s)]  [HAVING condition] ORDER B
Y fieldName(s)
```

където:

- SELECT е ключова дума на SQL, която съобщава на базата данни, че искате да извлечете данни.
- [DISTINCT | ALL] са незадължителни ключови думи, които могат да се използват за прецизиране на резултатите, върнати от SQL заявката SELECT. Ако не е посочено нищо, по подразбиране се приема ALL.
- {*| [fieldExpression [AS newName]} трябва да се посочи поне една част, "*" избира всички полета от посоченото име на таблица, fieldExpression извършва някои изчисления върху посочените полета, като например събиране на числа или събиране на две полета с низове в едно.
- FROM tableName указва с коя таблица искате да работите. Ако таблиците са повече, те трябва да бъдат разделени със запетаи или обединени с ключовата дума JOIN.
- Условието WHERE не е задължително. То може да се използва за определяне на критерии в набора от резултати, върнати от заявката.
- Функцията GROUP BY се използва за обединяване на записи, които имат еднакви стойности на полетата.

- Условието HAVING се използва за задаване на критерии при работа с ключовата дума GROUP BY.
- ORDER BY се използва за определяне на реда на сортиране на набора от резултати.

Клаузата **WHERE** е ключова дума, която се използва за определяне на точните критерии за данните или редовете, които ще бъдат засегнати от зададения SQL оператор. Клаузата WHERE може да се използва със SQL оператори като INSERT, UPDATE, SELECT и DELETE с цел филтриране на записи и извършване на различни операции върху данните.

Клауза **ORDER BY** се използва в комбинация със заявката SELECT за подреждане на данните по подходящ начин. Клаузата ORDER BY се използва за сортиране на наборите от резултати от заявката във възходящ или низходящ ред.

Клаузата **GROUP BY** се използва за групиране на редове с еднакви стойности. Клаузата GROUP BY се използва с команда SELECT. По желание тя се използва в комбинация с функции за агрегиране, за да се изготвят обобщени отчети от базата данни. Именно това прави тя, като обобщава данните от базата данни. Заявките, които съдържат клауза GROUP BY, се наричат групирани заявки и връщат само по един ред за всеки групиран елемент.

Функции за агрегиране на съдържание

Агрегиращите функции се отнасят до извършване на изчисления върху множество редове на една колона от таблица и връщане на една стойност. Стандартът ISO дефинира пет агрегиращи функции, а именно: SUM, AVG, MAX, MIN, COUNT и DISTINCT.

COUNT: Функцията COUNT връща общия брой стойности в посоченото поле. Тя работи както с числови, така и с нечислови типове данни. Всички функции за агрегиране по подразбиране изключват нулевите стойности, преди да работят с данните.

DISTINCT: Ключовата дума DISTINCT ни позволява да изпускаме дубликати от нашите резултати. Това се постига чрез групиране на сходни стойности.

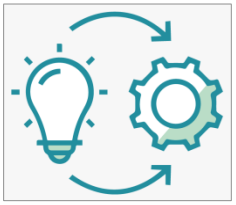
MIN: Функцията MIN връща най-малката стойност от посоченото поле на таблицата.

MAX: Функцията MAX връща най-голямата стойност от посоченото поле на таблицата.

SUM: Функцията SUM връща сумата от всички стойности в посочената колона. SUM работи само с числови полета. Нулевите стойности се изключват от върнатия резултат.

AVG: Функцията AVG връща средната стойност на стойностите от определена колона. Подобно на функцията SUM, тя работи само с числови типове данни.

IV. Задачи за изпълнение



Задача 1: *Да се проектира база данни, която позволява извличане на информация за състезание за програмиране GAPCOM. В състезанието могат да участват както студенти, така и ученици. Състезанието се провежда веднъж годишно. Класирането на участниците е на базата на получени точки, които са в интервала [0, 100]. В зависимост от тези точки, участниците получават или не медал (златен, сребърен или бронзов). Точките за медал са фиксирани и не подлежат на промяна в годините.*

Нека да започнем с това каква информация искаме да съдържа нашата база данни и какви отчети бихме желали да получаваме. Базата данни трябва да съдържа информация за всеки участник в състезанието GAPCOM – неговите имена, дали е студент или ученик, къде учи, в кой клас / курс и разбира се какво е класирането му (получени точки и медал) за всяко състезание в което е участвал. Тази информация е представена в Табл. 1.2.

Табл. 1.2 Изходни данни

pId	name	comId	affiliation	points	medal
10	Иван Георгиев	2017; 2018	„ТУ – Габрово, специалност КСТ, курс 2”; „ТУ – Габрово, специалност КСТ, курс 3”	76; 57	„сребърен”; „няма”
12	Петко Тодоров	2018	„ТУ - София”	66	„бронзов”

Вижда се, че само атрибут “pID” (participant id) и поле “name” не съдържат множество стойности. Следователно таблицата е в 0NF. Всеки ред от таблицата трябва да даде информация кой участник колко точки е получил при всяко състезание в което е участвал. Следователно, съставният ключ { pId, commId } е минимален набор от атрибути, които гарантират уникално идентифициране на реда. Това означава, че { pId, commId } е кандидат ключ за таблицата. Всеки участник може да участва в множество състезания. Теоретично, всяка година той може да участва от името на различно учебно заведение. За да преминем до 1NF, трябва да генерираме нови редове, за да няма атрибути с множество стойности (виж Табл. 1.3).

Табл. 1.3 Преминаване до 1NF

pId	name	comId	affiliation	points	medal
10	Иван Георгиев	2017	„ТУ – Габрово, специалност КСТ, курс 2”	76	„сребърен”
10	Иван Георгиев	2018	„ТУ – Габрово, специалност КСТ, курс 3”	57	„няма”
12	Петко Тодоров	2018	„ТУ - София”	66	„бронзов”

Функционалните зависимости са следните:

pId → name

{pId, comId} → affiliation

{pId, comId} → points

{pId, comId} → medal

Нека да анализираме дали някой непървичен атрибут зависи само част от първичния ключ. Атрибут “name” зависи само от “pId”. Останалите полета зависят от първичния ключ. Следователно, ще отделим участниците в таблица “Участници”, а резултатите от състезанията в различните години в друга таблица – “Състезания”. По този начин, всеки запис в таблицата с резултатите няма да изисква въвеждане на символно поле за име, а само идентификатора на участника. В таблица “Участници”, за по-голяма функционалност, сме добавили нов атрибут “lastName”. При конкретния случай всеки участник може да участва в множество състезания. Следователно имаме релация "един към много". Това налага да вземем първичния ключ от страната "един" на релацията и го добавим като допълнителна колона в таблицата от страната "много" на релацията.

Табл. 1.4 Преминаване до 2NF

Таблица “Участници”

pId	firstName	lastName
10	Иван	Георгиев
12	Петко	Тодоров

Таблица “Състезания”

comId	pId	affiliation	points	medal
2017	10	„ТУ – Габрово, специалност КСТ, курс 2”	76	„сребърен”
2018	10	„ТУ – Габрово, специалност КСТ, курс 3”	57	„няма”
2018	12	„ТУ - София”	66	„бронзов”

Сега остава да проверим, дали в тези таблици, които са в 2NF, има атрибути, които зависят от друг атрибут, който не е част от ключа. При таблица “Състезания” атрибут “medal” зависи от атрибут “points”. Тъй като знаем, че връзката „points-medal” не се променя в годините, можем да създадем нова таблица “Класиране” и така да премине към 3NF.

Табл. 1.5 Преминаване до 3NF

Таблица „Участници”

pId	firstName	lastName
10	Иван	Георгиев
12	Петко	Тодоров

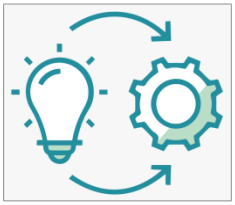
Таблица „Състезания”

comId	pId	affiliation	points
2017	10	„ТУ – Габрово, специалност КСТ, курс 2”	76
2018	10	„ТУ – Габрово, специалност КСТ, курс 3”	57
2018	12	„ТУ - София”	66

Таблица „Класиране”

points	medal
76	„сребърен”
66	„бронзов”
57	„няма”

Атрибут “affiliation” съдържа съставна информация и би било добре да се разбие на няколко атрибута: “institution”, “specialty” и “course”. По този начин ще се повиши функциалността на възможните заявки към базата данни.



Задача 2: *Създайте база данни GAPCOM, която да съдържа таблиците, описани в Задача 1. За целта използвайте Web платформата https://extendsclass.com/mysql-online.html чрез която може да със СУБД, без да се налага да инсталирате софтуер. Тествайте работоспособността на базата данни чрез SELECT команди.*

Стартирайте Web услугата. Използвайте файл **Task-2-create-database.txt** който съдържа всички необходими SQL команди чрез които да създадете трите таблици и да ги населите с данни. Създайте таблиците чрез команди CREATE:

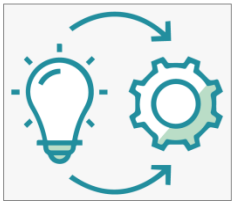
```
create table participants(id integer, firstname varchar(50),
    family varchar(50), lastname varchar(50));
create table competitions(id integer, pid integer,
    affiliation varchar(100), lang varchar(15), points integer);
create table ranking(points integer, medal varchar(10));
```

След това последователно населете всяка таблица с данни като използвате INSERT:

```
insert into participants (id, firstname, family, lastname)
    values (1, 'Йордан', 'Валентинов', 'Петков');
insert into competitions (id, pid, affiliation, lang, points)
    values (2017, 1, 'Математическа гимназия - Габрово', 'C++', 86);
insert into ranking(points, medal) values(86, 'gold');
```

Проверете дали базата е създадена успешно, като изведете последователно цялото съдържание на всяка една от таблиците:

```
select * from participants
select * from competitions
select * from ranking
```



Задача 3: *Използвайте команда SELECT, за да генерирате заявки към база данни GAPCOM. За бързо автоматично форматиране на заявките използвайте ресурс https://extendsclass.com/sql-formatter.html.*

Целта на задачата е да се научите да спазвате синтаксиса на команда SELECT, както и да извличате структурирано и сортирано съдържание чрез използване на клаузите на командата и функциите за агрегиране на съдържание.

***Заявка 1:** Получаване на имената и точките на всички участници, участвали в GAPCOM-2017.*

За да реализираме заявката трябва да обединим информацията от таблици participants и competitions. За целта ще използваме JOIN. Връзката между двете таблици е чрез атрибут “id” от participants и “pid” от competitions. Филтрирането на резултатите за точно определено състезание ще реализираме чрез клауза WHERE. Следва примерен синтаксис на SELECT заявката:

```
SELECT participants.firstname, participants.lastname,
    competitions.id, competitions.points
FROM participants
INNER JOIN competitions
ON participants.id = competitions.pid
WHERE competitions.id = 2017
```

Резултатът (15 записа), който се получава е следния:

firstname	lastname	id	points
Йордан	Петков	2017	86
Мартин	Копчев	2017	86
Георги	Генков	2017	82
Добрин	Башев	2017	80
Йордан	Пенев	2017	75
Адриян	Ибовски	2017	74

Заявка 2: *Получаване на имената и точките на всички участници, сортирани по име и фамилия (от А до Я).*

За желаното сортиране по име и фамилия ще използваме клауза ORDER BY. По подразбиране ORDER BY подрежда данните във възходящ ред. Можем да използваме ключовата дума DESC за сортиране на данните в низходящ ред и ключовата дума ASC за сортиране във възходящ ред.

```
SELECT participants.firstname, participants.lastname,
    competitions.id, competitions.points
FROM participants
INNER JOIN competitions
ON participants.id = competitions.pid
ORDER BY participants.firstname ASC, participants.lastname ASC
```

Резултатът (29 записа), който се получава, е следния:

firstname	lastname	id	points
Адриян	Ибовски	2017	74
Адриян	Ибовски	2018	82
Валентин	Владимиров	2017	58
Веселин	Димов	2017	72
Веселин	Димов	2018	74
Виктор	Венков	2017	57

Заявка 3: *Получаване на общият брой точки за всеки един състезател за всички състезания в които е участвал.*

В този случай се налага да обединим всички резултати на всеки един от състезателите. Това обединение може да се реализира чрез клауза GROUP BY. Атрибут “points” от таблица “competitions” трябва да се сумира за всяко едно обединение. За целта ще използваме функция SUM:

```
SELECT participants.firstname, participants.lastname,
    SUM(competitions.points) as "total points"
FROM participants inner join competitions
ON participants.id = competitions.pid
GROUP BY participants.firstname, participants.lastname
```

Резултатът (21 записа), който се получава, е следния:

firstname	lastname	total points
Йордан	Петков	178
Мартин	Копчев	178
Георги	Генков	82
Добрин	Башев	176
Йордан	Пенев	75
Адриян	Ибовски	156
Веселин	Димов	146

Заявка 4: *Получаване на общият брой точки за всеки един състезател за всички състезания в които е участвал и сортиране в низходящ ред на записите по общия брой точки.*

Трябва да използвате клауза ORDER BY като за параметър за сортиране подадете sum(competitions.points). Резултатът (21 записа), който се получава, е следния:

firstname	lastname	total points
Мартин	Копчев	178
Йордан	Петков	178
Добрин	Башев	176
Адриян	Ибовски	156
Пламен	Динев	152
Веселин	Димов	146
Никола	Максимов	145

Заявка 5: *Получаване на "Топ 10" на участниците във всички състезания, сортирани по брой на точките.*

Използвайте Заявка 4 като ограничите брой на върнатите записи до 10. За целта използвайте клауза LIMIT. Резултатът, който трябва да получите, е следния:

firstname	lastname	total points
Йордан	Петков	178
Мартин	Копчев	178
Добрин	Башев	176
Адриян	Ибовски	156
Пламен	Динев	152
Веселин	Димов	146
Никола	Максимов	145
Цвета	Тодорова	138
Георги	Генков	82
Йордан	Пенев	75

Заявка 6: *Получаване на имената на всички участници, които са избрали език за програмиране C#.*

Трябва да обедините таблици participants и competitions. Условието спрямо атрибут “lang” реализирайте с клауза WHERE. Трябва да получите 5 записа:

firstname	lastname	id	points
Тихомир	Галов	2017	67
Валентин	Владимиров	2017	58
Стефан	Георгиев	2018	75
Ивайло	Иванов	2018	75
Емил	Йорданов	2018	60

Заявка 7: *Получаване колко пъти е бил избран език за програмиране C#.*

Трябва да използвате функция COUNT:

```
SELECT COUNT(*) as 'Колко пъти е бил избран език C#?'
FROM participants inner join competitions
ON participants.id = competitions.pid
WHERE competitions.lang = 'C#'
```

Резултатът (1 запис), който се получава, е следния:

Колко пъти е бил избран език C#?
5

Заявка 8: *Получаване на имената на всички участници, които са използвали език за програмиране C++, но без повторение на имена, ако участникът е участвал в повече от едно състезание.*

Когато се налага изпускане на дублиращи се резултати се използва ключова дума DISTINCT:

```
SELECT DISTINCT participants.id,
    participants.firstname, participants.lastname
FROM participants inner join competitions
ON participants.id = competitions.pid
WHERE competitions.lang = 'C++'
```

Резултатът (14 записа), който се получава, е следния:

id	firstname	lastname
1	Йордан	Петков
2	Мартин	Копчев
3	Георги	Генков
4	Добрин	Башев
5	Йордан	Пенев
6	Адриян	Ибовски

Заявка 9: *Получаване само на броя на участници, които са избирали език за програмиране C++.*

Използвайте функция COUNT в команда SELECT. За параметър на функцията задайте `DISTINCT participants.id`.

Резултатът (1 запис), който се получава, е следния:

Брой участници избрали C++
14

Заявка 10: *Да се получат имената на участниците, които имат получен медал и какъв точно е медала (златен, сребърен или бронзов). Резултат да се сортира в зависимост от цвета на медала.*

В тази задача се налага да се обединят и трите таблици, за да се получи желания резултат. Трябва да изключим всички участници, които не са получили медал. За целта ще използваме клауза WHERE която проверява дали стойността на атрибут “medal” от таблица ranking не е “none”. Ако сортираме чрез ORDER BY по брой получени точки, то ще получим лесно желаната подредба по медали:

```
SELECT participants.firstname, participants.lastname,
      ranking.medal as 'Medal'
FROM participants INNER JOIN competitions
ON participants.id = competitions.pid
INNER JOIN ranking
ON competitions.points = ranking.points
WHERE ranking.medal != 'none'
ORDER BY competitions.points DESC
```

Резултатът (28 записа), който се получава, е следния:

firstname	lastname	Medal
Добрин	Башев	gold
Йордан	Петков	gold
Мартин	Копчев	gold
Йордан	Петков	gold
Мартин	Копчев	gold
Цвета	Тодорова	silver
Георги	Генков	silver

V. Задачи за самостоятелна работа

Задача 1: *Като използвате база данни GAPCOM получете колко участника са избрали език за програмиране C++ за всяко едно състезание.*

Групирайте резултата по атрибут “id” от таблица “competitions”. Резултатът, който трябва да получите, е следния:

id	Брой участници избрали C++
2018	10
2017	12

Задача 2: *Получете всяко едно учебно заведение, участвало със свои представители в състезанието GAPCOM и колко общо сребърни медала има.*

Проверката за тип медал ще реализираме чрез клауза WHERE. Сортирането трябва да бъде по атрибут “affiliation” от таблица “competitions”. Използвайте функция COUNT за намиране на броя на медалите. Трябва да получите два записа:

Учебно заведение	Брой сребърни медали
Математическа гимназия - Габрово	12
ТУ - Габрово	5

```
SELECT competitions.affiliation AS 'Учебно заведение',
    COUNT(*) AS 'Брой сребърни медали'
FROM competitions INNER JOIN ranking
ON competitions.points = ranking.points
WHERE ranking.medal = 'silver'
GROUP BY competitions.affiliation
```

Задача 3: *Трябва да получите статистика за броя на медалите, които всеки един участник в състезанието GAPCOM е получавал. Сортирайте резултата в зависимост от броя и цвета на медалите.*

Използвайте функция SUM да сумирате медалите от даден цвят. За да сравните цвета на медал, използвайте оператор LIKE. Той се използва най-често е в клауза WHERE за търсене на определен шаблон в дадена колона, но може да го използваме и в SUM. Има два заместващи символа, които често се използват в комбинация с оператора LIKE: знакът за процент (%) представлява нула, един или няколко символа, а знакът за подчертаване (_) представлява само един символ. Сумирането на златните медали и запазването им като “totalGold” може да се реализира по следния начин:

```
SUM(ranking.medal LIKE '%gold%') AS totalGold
```

По аналогичен начин получете “totalSilver” и “totalBronze”. Сортирайте резултата по стойността на “totalGold”, “totalSilver” и “totalBronze”. Трябва да получите 21 записа в следния формат:

firstname	lastname	totalGold	totalSilver	totalBronze
Йордан	Петков	2	0	0
Мартин	Копчев	2	0	0
Добрин	Башев	1	1	0
Адриян	Ибовски	0	2	0
Веселин	Димов	0	2	0
Пламен	Динев	0	2	0
Цвета	Тодорова	0	1	1

```
SELECT participants.firstname, participants.lastname,
    SUM(ranking.medal LIKE '%gold%') AS totalGold,
    SUM(ranking.medal LIKE '%silver%') AS totalSilver,
    SUM(ranking.medal LIKE '%bronze%') AS totalBronze
FROM participants INNER JOIN competitions
ON participants.id = competitions.pid
INNER JOIN ranking
ON competitions.points = ranking.points
GROUP BY participants.firstname, participants.lastname
ORDER BY totalGold DESC, totalSilver DESC, totalBronze DESC
```

Упражнение № 2

Използване на облачни платформи за работа с бази данни

I. Въведение

Физическото разположение на една база данни е важно за достъпността на услугата, чиято бизнес логика използва базата. В зависимост от разположението, базите данни биват:

- **Локални**. Може да инсталирате инстанция на сървъра за управление на базата данни на локален компютър, но това има смисъл само за развойна дейност и тестване. Практически смисъл имат локалните бази данни за мобилни устройства. Тяхната цел най-често е предоставяне на възможност за функциониране на една услуга, достъпна чрез мобилни устройства, дори в интервалите от време когато липсва мрежова свързаност. В този случай локалната база се синхронизира автоматично или при необходимост с база данни, хоствана на сървър в Интернет пространството. Подобни мобилни приложения се наричат offline-first приложения.
- **Отдалечени, хоствани на сървър**. В този случай сървърът за управление на базата данни е инсталиран на един (или няколко компютъра), който(които) има(т) необходимите апаратни и програмни ресурси, за да може да функционира конкретната база данни. Трябва да се има предвид, че в този случай администрирането на базата данни изцяло се поема от администратор. Той отговаря както за сигурността на услугата, която използва базата данни, така и за нейната мащабируемост.
- **Отдалечени, хоствани в облак.** Когато искате да се съсредоточите върху бизнес логиката на една услуга е добре самата услуга и базата данни да бъдат хоствани в облачна платформа. В този случай сигурността на данните и комуникационните канали, както и необходимото обновяването на софтуера се реализира независимо от вас. Най-често софтуерът за управление на базата данни се предлага като услуга (SaaS). Трябва да се отчете възможността за много по-големите възможности за мащабируемост на услугата в този случай. Освен вертикална мащабируемост (увеличаване на изчислителните ресурси като брой процесори и количество памет) в този случай е възможна и хоризонтална мащабируемост (разпределяне на базата данни на няколко физически сървъра). Следователно, хостването на една база данни е облачна инфраструктура има смисъл ако трябва да се работи с много големи обеми данни и вие искате да се абстрахирате от администриране на услугата с цел фокусиране върху логиката на услугата.

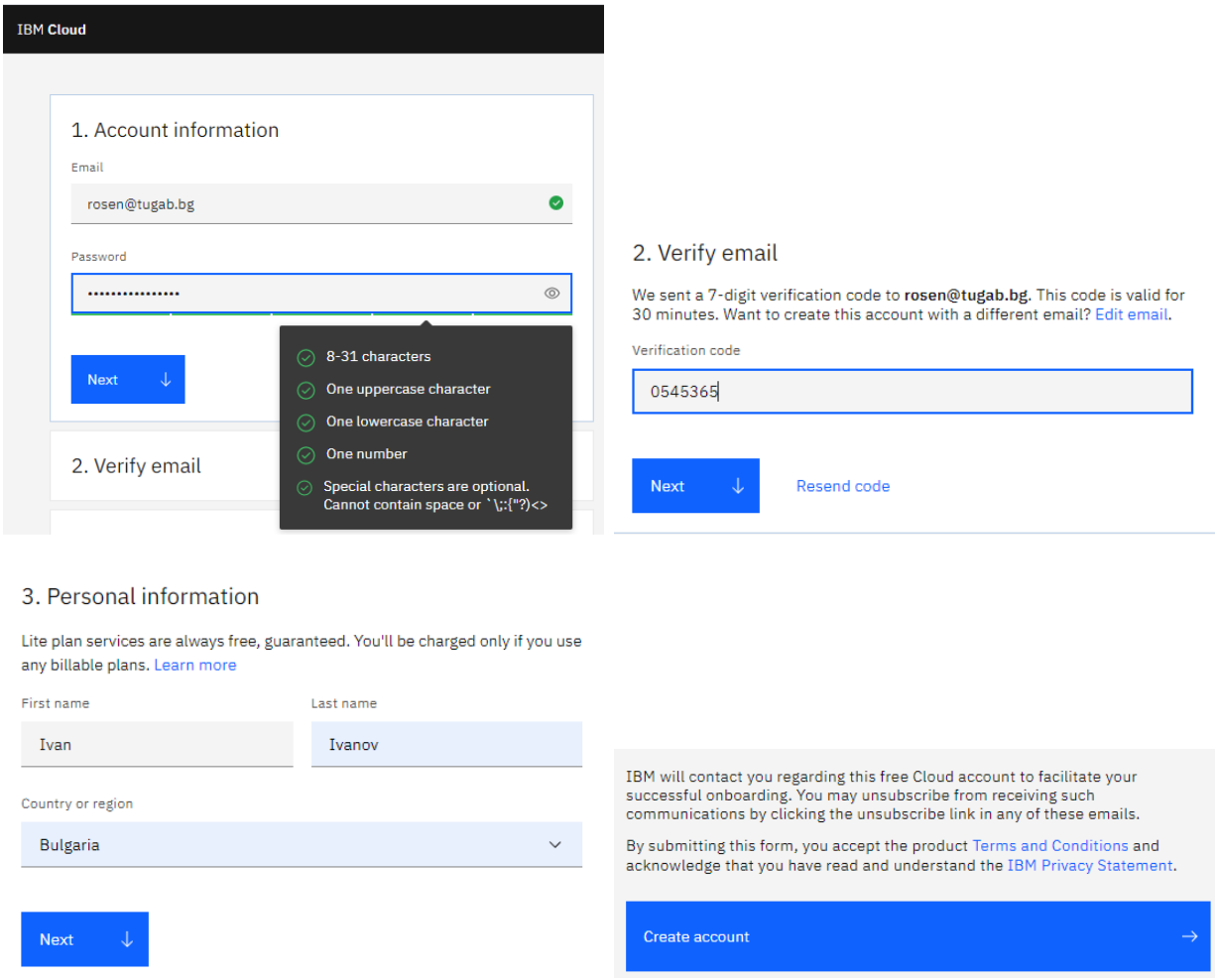
Ще използваме облака на фирма IBM (IBM Cloud) с цел използване на облачно дисково пространство за нашите бази данни, както и за получаване на достъп до различни услуги. На този етап ще използваме услугата “IBM SQL Query” чрез която ще бъде възможно да изпълняваме SQL заявки към нашите бази данни.

II. Регистриране в IBM Cloud

За да може да използвате услугите, достъпни чрез IBM Cloud, трябва да се регистрирате. За целта ви трябва само валиден e-mail адрес. Регистрацията се реализира чрез следния адрес:

https://cloud.ibm.com/registration

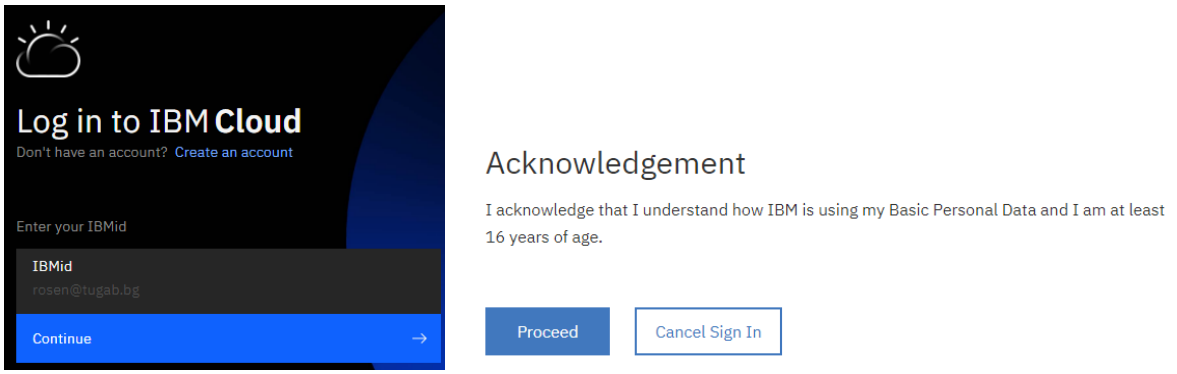
Попълнете формата за регистрация (стъпка 1), като въведете своя e-mail адрес и парола за достъп. Услугата ще изиска от вас въвеждане на код за верификация, който ще ви бъде изпратен като електронна поща (стъпка 2). Следва въвеждане на персонална информация (стъпка 3).



Фиг. 2.1 Регистрация в IBM Cloud

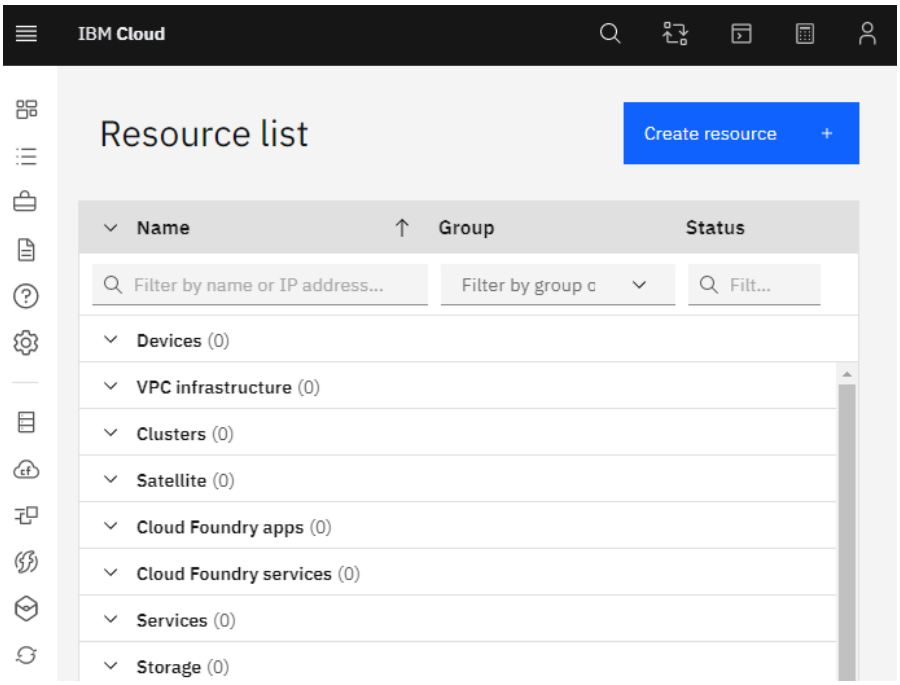
III. Създаване на ресурси в IBM Cloud

След като сте се регистрирали успешно ще бъдете препратени към форма за проверка на автентичността (login). Потвърдете, че сте съгласни IBM да обработва вашите персонални данни.



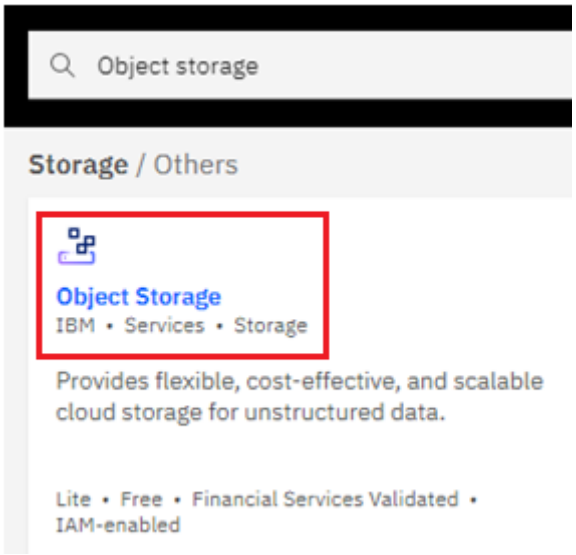
Фиг. 2.2 Достъп до IBM Cloud

След успешно ви удостоверяване ще бъдете пренасочени към Dashboard на IBM Cloud. От падащото меню изберете да видите списък с услугите, които сте инсталирали до този момент (Menu -> Resource list). Разбира се на този етап все още нямате инсталирани услуги, както се вижда на Фиг. 2.3.



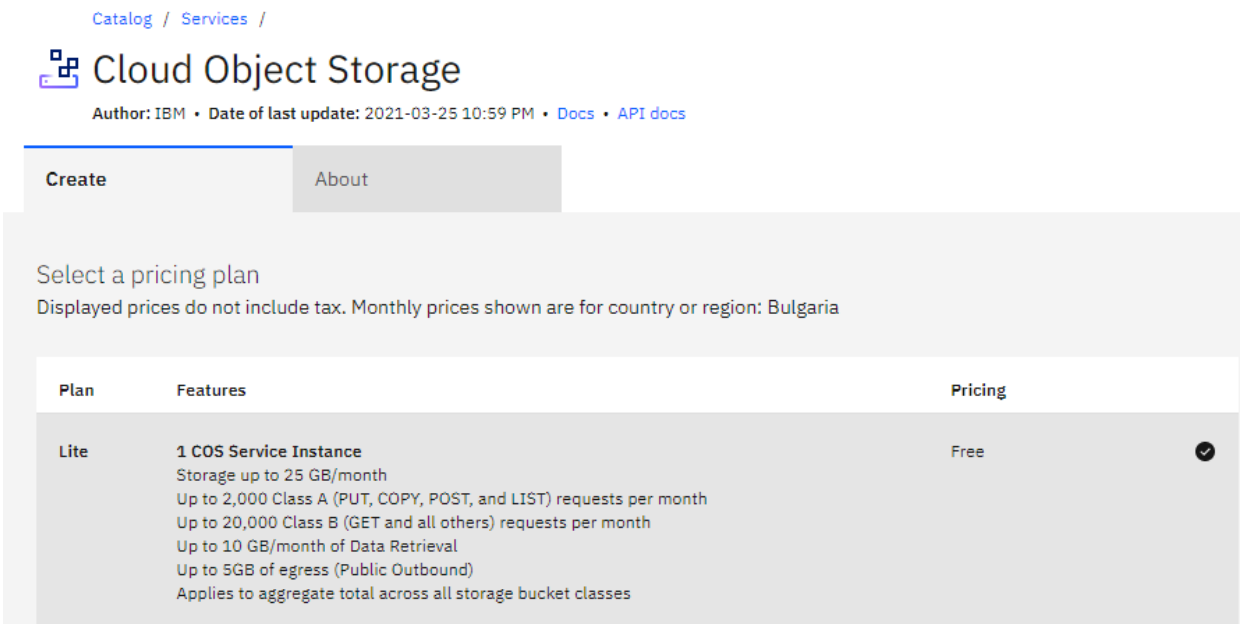
Фиг. 2.3 Списък на всички инсталирани от Вас ресурси

За да получим достъп до желан ресурс натиснете бутон “Create resource”. Първо, ще получим достъп до дисково пространство, където ще хостваме нашите бази данни. За целта в поле Search въведете “Object storage” (виж Фиг. 2.4). От получените ресурси след търсенето изберете “Object storage”.



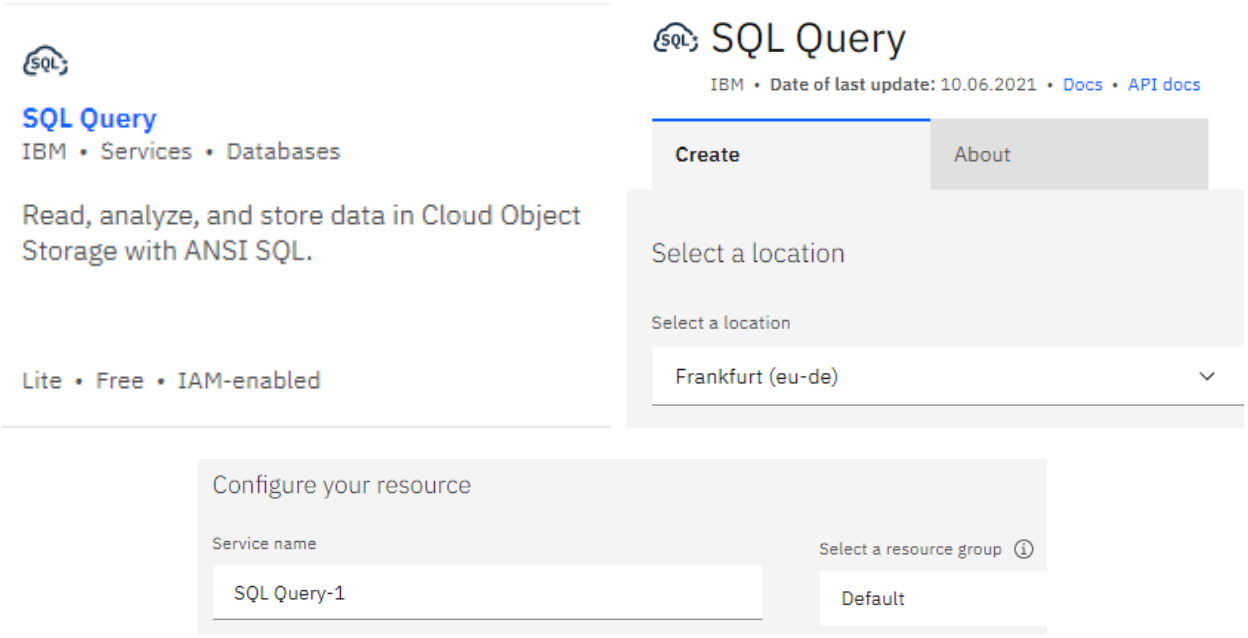
Фиг. 2.4 Получаване на достъп до Object storage

Създайте услуга за достъп до “Object storage”, като изберете безплатен абонаментен план Lite (виж Фиг. 2.5). При този план може да използвате до 25GB дисково пространство на месец и да правите до 20,000 заявки на месец, свързани с манипулиране на вашите данни. Задайте име на услугата (Cloud Object Storage-1) и натиснете бутон Create.



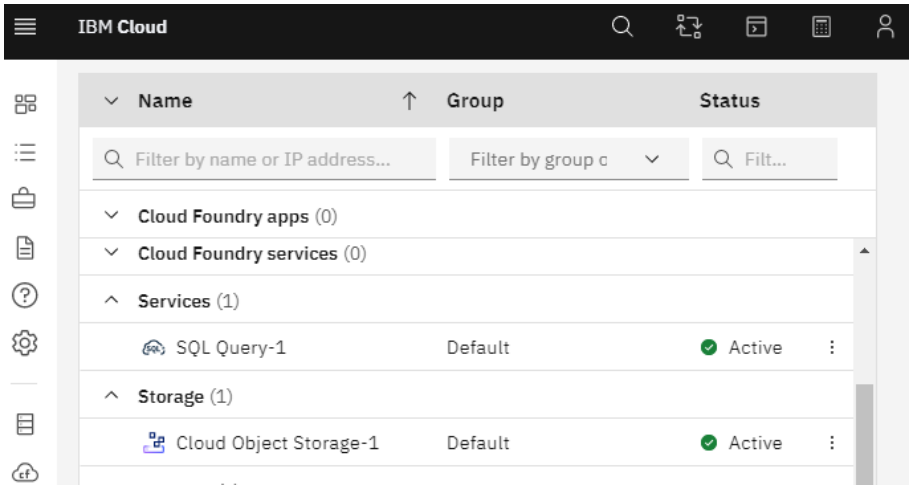
Фиг. 2.5 Създаване на услуга за достъп до Object storage

За да тествате вашите бази данни чрез стандартни ANSI SQL заявки е необходимо да създадете услуга от ресурса “SQL Query”. За целта отново изберете “Create resource” и потърсете “SQL Query”. Изберете ресурса с това име с цел да преминете към инсталиране на услуга (виж Фиг. 2.6).



Фиг. 2.6 Създаване на услуга за достъп до SQL Query

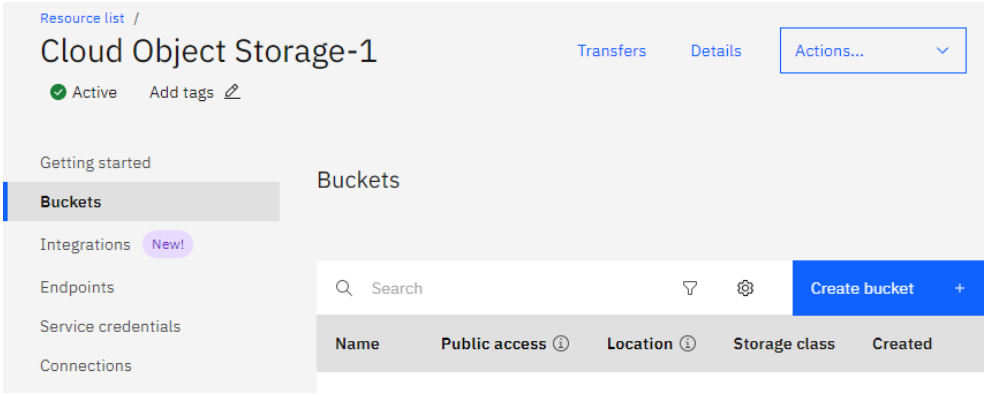
Проверете какво сте инсталирали до този момент чрез избор на “Resource list”. Трябва да имате активни две услуги, както е показано на Фиг. 2.7.



Фиг. 2.7 Инсталирани услуги

IV. Прехвърляне на базите данни в Object Storage

За да хоствате файлове в “Object storage” трябва да създадете ваше хранилище за данни (bucket). За целта от менюто на “Cloud Object Storage” изберете “Buckets” и натиснете бутон “Create bucket”:



Фиг. 2.8 Създаване на хранилище за вашите данни

Задайте име на хранилището, например “my-data-1”. Изберете къде физически да бъде разположено вашето хранилище. За целта изберете “Regional”, а за Location – “eu-de”. Забранете “Object versioning”, за да пестите дисково пространство. Изключете на този етап всичко от “Service integrations”.

Вече сте готови да прехвърлите вашите файлове в хранилището. За целта изберете създаденото хранилище и натиснете бутон “Upload”. Ще работим с три бази данни (виж Табл. 2.1):

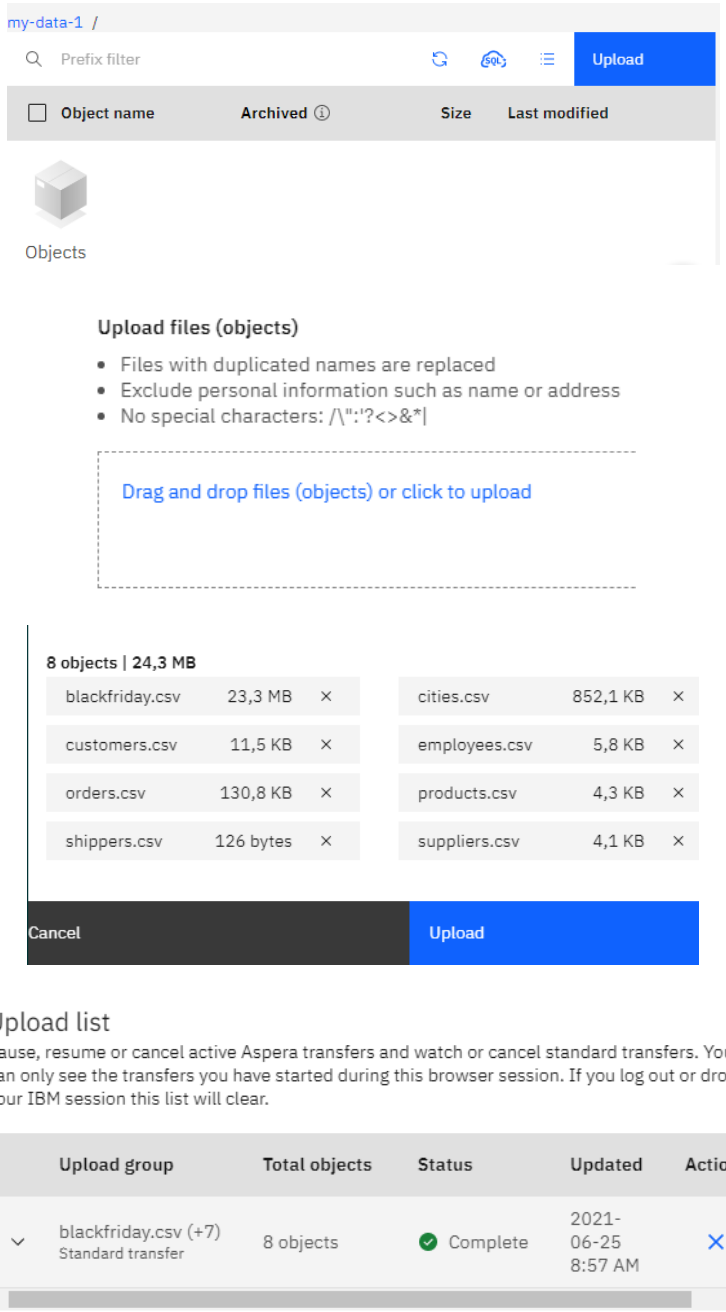
- База данни 1 – съдържа информация за покупките на клиенти по време на “черен” петък (7 таблици).
- База данни 2 – съдържа информация за класирането на участниците в състезание GAPCOM (3 таблици).
- База данни 3 – съдържа информация за градове (1 таблица).

Табл. 2.1 CSV файлове

База данни	Файлове	Брой записи	Съдържание атрибути
1.	blackfriday.csv	537,577	Информация за Id на клиенти и id на закупени от тях продукти по време на “черен” петък. User_ID, Product_ID, Gender,Age, Occupation, City_Category, Stay_In_Current_City_Years, Marital_Status, Product_Category_1, Product_Category_2, Product_Category_3, Purchase
	customers.csv	91	Информация за клиенти. customerID, companyName,contactName, contactTitle, address, city, region, postalCode, country, phone, fax
	employees.csv	9	Информация за работодатели. employeeID, lastName, firstName, title, titleOfCourtesy, birthDate, hireDate, address, city, region, postalCode, country, homePhone, extension, photo, notes, reportsTo, photoPath
	orders.csv	830	Информация за покупките на клиентите. orderID, customerID, employeeID, orderDate, requiredDate, shippedDate, shipVia,freight, shipName, shipAddress, shipCity, shipRegion, shipPostalCode, shipCountry
	products.csv	77	Информация за продуктите. productID, productName, supplierID, categoryID, quantityPerUnit, unitPrice, unitsInStock, unitsOnOrder, reorderLevel,discontinued
	shippers.csv	3	Информация за фирмите доставчици на продуктите. shipperID, companyName, phone
	suppliers.csv	29	Информация за фирмите продавачи. supplierID, companyName, contactName, contactTitle, address, city,region, postalCode, country, phone, fax, homePage
2.	participants.csv	21	Участници в състезанието GAPCOM id, firstname, family, lastname
	competitions.csv	29	Резултати за всяко състезание id, pid, affiliation, lang, points
	ranking.csv	101	Връзка точки – медали points, medal
3.	cities.csv	23,018	Информация за градовете в световен мащаб. city, country, subcountry, geonameid

Всички файлове може да получете от архива **Databases-csv-files.zip**

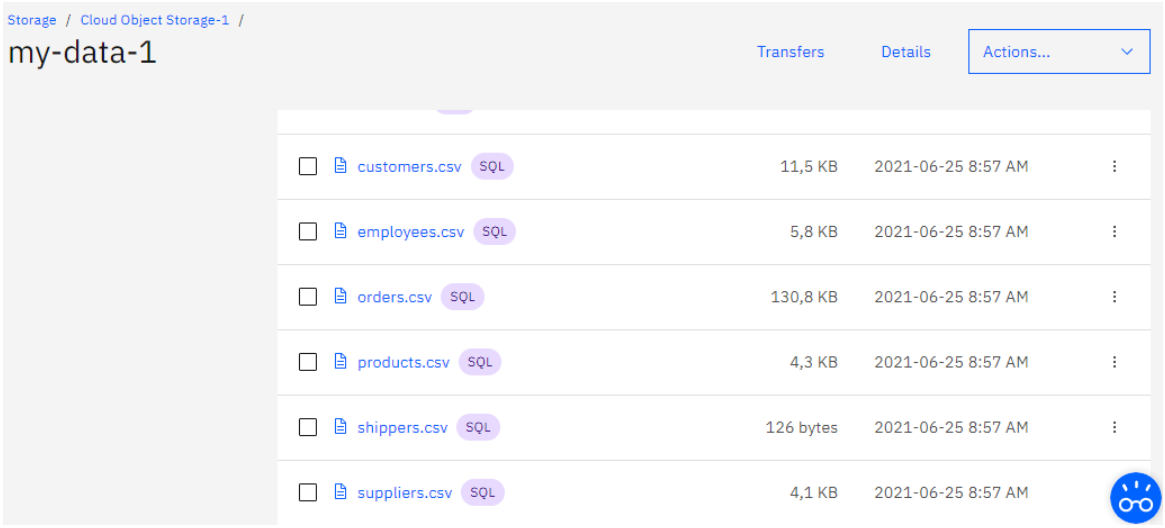
Качването на файловете в хранилището се реализира чрез натискане на бутон “Upload” (виж. Фиг. 2.9).



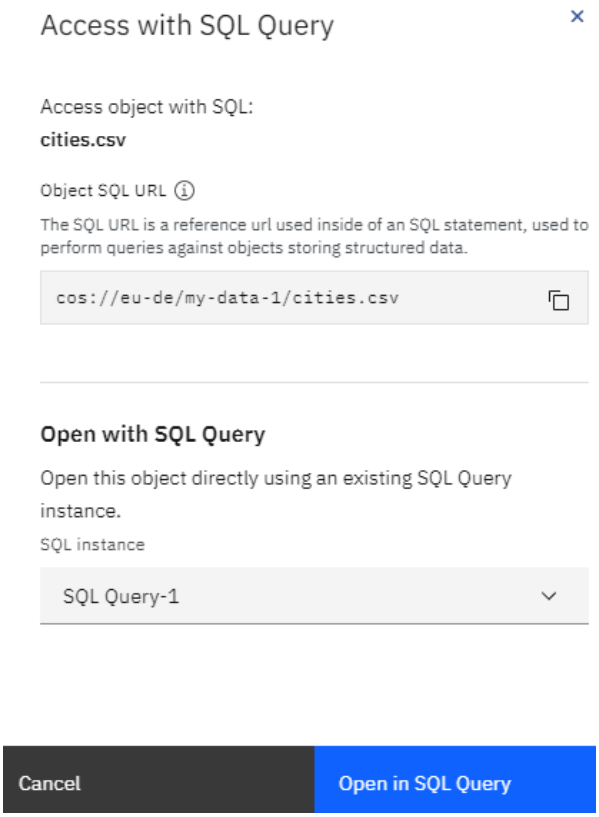
Фиг. 2.9 Прехвърляне на файлове в хранилище “my-data-1”

V. Генериране на SQL заявки

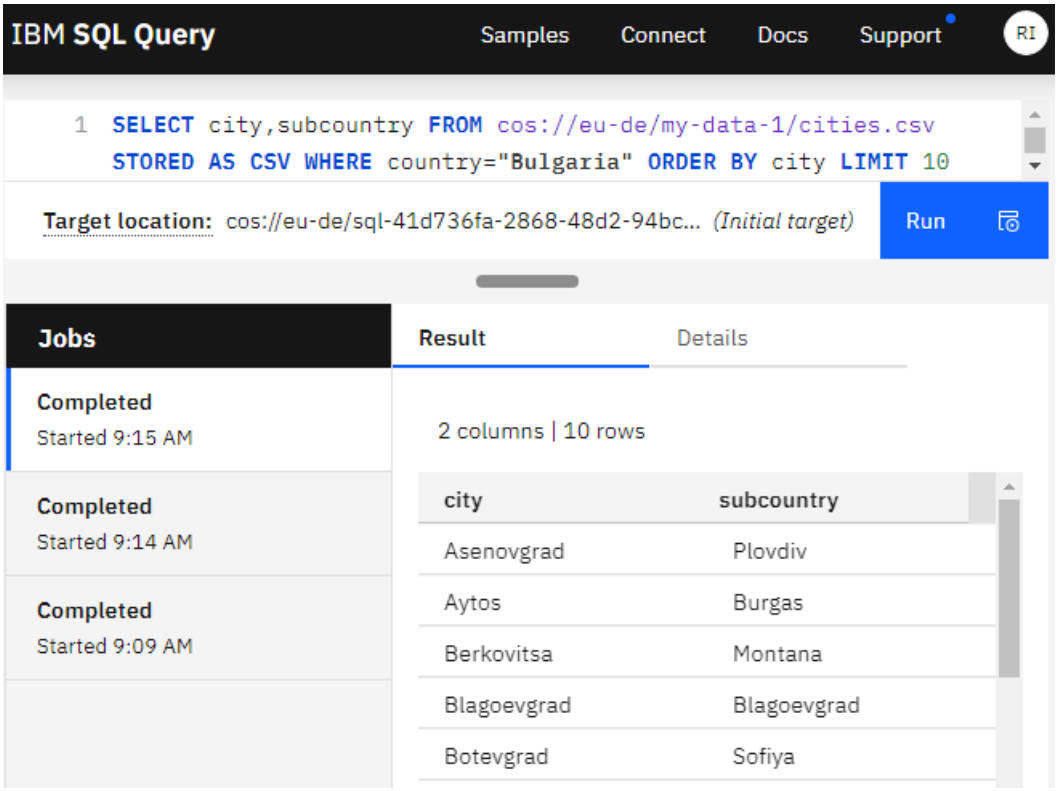
За да тествате базите данни трябва да използвате услуга “SQL Query”. За целта от менюто на Storage изберете Objects (виж. Фиг. 2.10). От менюто за всеки обект (три вертикални точки) изберете “Access with SQL”. Системата ще намери инсталираната от вас услуга “SQL Query-1” (виж Фиг. 2.11). Натиснете бутон “Open in SQL Query”. Отваря се интерфейса на SQL Query (виж Фиг. 2.12). Вече можете да генерирате SQL команди към избрана база данни и да виждате върнатия резултат. Заявката от Фиг. 2.12 връща първите 10 града в България и областите от които са те. Информацията се филтрира по името на града.



Фиг. 2.10 Списък на обектите в хранилище “my-data-1”

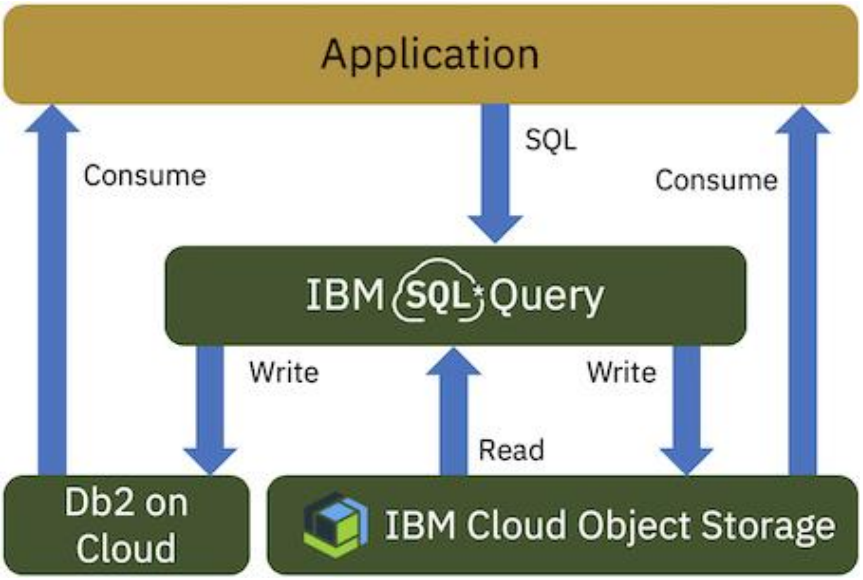


Фиг. 2.11 Активиране на услугата SQL Query



Фиг. 2.12 генериране на SQL заявка

IBM SQL Query е облачна услуга, която позволява изпълнение на SQL заявки с цел анализ, трансформиране и филтриране на данни. Може да използвате само команда SELECT. Команди CREATE, DELETE, INSERT и UPDATE са забранени. Входните данни могат да бъдат прочетени от CSV, JSON, ORC, Parquet или AVRO обекти, разположени в една или множество IBM Object Storage инстанции. Резултатът от всяка заявка се записва в CSV, JSON, ORC, Parquet или AVRO обект в указана от вас “Cloud Object Storage” инстанция (виж Фиг. 2.13).



Фиг. 2.13 Взаимодействие между “IBM Cloud Object Storage” и “IBM SQL Query”

Трябва да имате най-малко "writer" достъп до поне едно хранилище за съхранение в IBM Object Storage, така че там да могат да бъдат записани входните данни, както и резултатите (обекти съдържащи изходните данни). Идентификаторът за местоположението на входните данни и резултатите е следния:

cos://endpoint/bucket/path

където:

cos - cloud object storage;

endpoint - крайна точка на вашата инстанция за достъп до “Cloud Object Storage”, например "eu-de";

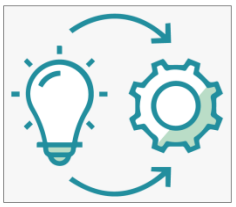
bucked - име на хранилище в което е базата данни, например "my-data-1";

path - име на файла с таблицата до която желаем достъп, например "cities.csv"

Например, следващата заявка връща броя на записите в таблицата от файл cities.csv:

```
SELECT COUNT(*) FROM cos://eu-de/my-data-1/cities.csv STORED AS CSV
```

VI. Задачи за изпълнение



Задача 1: *Получете списък на първите 10 града в България и областите от които са те.*

Ще използваме таблица cities.csv. Необходимата информация е записана в атрибути “city” и “subcountry”. С цел филтриране на имената на държавите ще използваме клауза WHERE:

```
SELECT city, subcountry FROM cos://eu-de/my-data-1/cities.csv STORED AS CSV
WHERE country = 'Bulgaria'
LIMIT 5
```

Резултатът, който се получава след изпълнение на заявката, е следния:

city	subcountry
Yambol	Yambol
Vratsa	Vratsa
Vidin	Vidin
Velingrad	Pazardzhik
Veliko Tŭrnovo	Veliko Tŭrnovo

Задача 2: *Получете списък на първите 10 града в България и областите от които са те. Сортирайте резултата по имената на градовете.*

Резултатът, който трябва да получите, е следния:

city	subcountry
Asenovgrad	Plovdiv
Aytos	Burgas
Berkovitsa	Montana
Blagoevgrad	Blagoevgrad
Botevgrad	Sofiya

Задача 3: *Получете списък на първите 10 града и областите от които са те за всички страни, имената на които започват с низа "Ви". Сортирайте резултата по имената на градовете.*

Използвайте клауза LIKE и заместващ символ %. Резултатът, който трябва да получите, е следния:

city	subcountry	country
Asenovgrad	Plovdiv	Bulgaria
Aytos	Burgas	Bulgaria
Banfora	Cascades	Burkina Faso
Berkovitsa	Montana	Bulgaria
Blagoevgrad	Blagoevgrad	Bulgaria

Задача 4: *Получете броя на градовете във всяка една държава от таблица "cities".*

Използвайте функция COUNT и клаузи GROUP BY и ORDER BY. Резултатът, който трябва да получите, е следния:

country	numberOfCities
D.C.	1
Afghanistan	48
Aland Islands	1
Albania	20
Algeria	247
American Samoa	1

Задача 5: *Извлечете всички записи с номера от 5 до 10 от таблица "customers.csv". Резултатът да съдържа номер на ред, име на купувача ("customerName") и град от който е купувача ("city").*

Предварително трябва да получим записите, които съответстват на редове от 5 до 10. За целта ще използваме клауза WITH и команда SELECT. След това ще генерираме команда SELECT, за да получим желаните атрибути:

```
WITH filteredByRowRecords AS
    (SELECT row_number() over(order by customerid) rowNumber, *
     FROM cos://eu-de/my-data-1/customers.csv STORED AS CSV)
SELECT rowNumber, contactName, city FROM filteredByRowRecords
WHERE rowNumber between 5 and 10
```

Резултатът, който трябва да получите, е следния:

rowNumber	contactName	city
5	Christina Berglund	Luleå
6	Hanna Moos	Mannheim
7	Frédérique Citeaux	Strasbourg
8	Martín Sommer	Madrid
9	Laurence Lebihan	Marseille
10	Elizabeth Lincoln	Tsawassen

Задача 6: *Получете информация за всички заявки на клиенти, които са генерирани на избрана от вас дата. Да се връща информация за:*

- *Идентификатор на заявката – атрибут "ordered" от таблица "orders".*
- *Име на клиента – атрибут "contactName" от таблица "customers".*
- *Име на спедитора – атрибут "companyName" от таблица "shippers".*

Задачата налага да обединим атрибути от три таблици. Ще използваме клауза WHERE, за да обединим таблиците по ключове и зададем условието за желана дата:

```
SELECT o.OrderID, c.ContactName AS CustomerName,
    e.FirstName AS EmployeerFirstName, e.LastName AS EmployeerLastName
FROM cos://eu-de/my-data-1/orders.csv STORED AS CSV o,
     cos://eu-de/my-data-1/employees.csv STORED AS CSV e,
     cos://eu-de/my-data-1/customers.csv STORED AS CSV c
WHERE e.EmployeeID = o.EmployeeID AND
      c.CustomerID = o.CustomerID AND
      o.OrderDate > '1998-03-23' AND
      o.OrderDate < '1998-03-24'
ORDER BY c.ContactName
```

Резултатът, който трябва да получите, е следния:

OrderID	CustomerName	EmployeerFirstName	EmployeerLastName
10967	Karin Josephs	Andrew	Fuller
10969	Pedro Afonso	Nancy	Davolio
10968	Roland Mendel	Nancy	Davolio

Задача 7: *За състезание GAPCOM получете за всяко учебно заведение какви медали е получило и какъв е техния брой.*

Използвайте обединяване на таблици “competitions” и “ranking”. Групирайте резултата по атрибут “affiliation” и атрибут “medal”. Чрез клауза WHERE изключете записите от таблица competitions, които не са свързани с получен медал.

```
SELECT c.affiliation, r.medal, COUNT(r.medal) as NumOfMedals
FROM cos://eu-de/my-data-1/competitions.csv STORED AS CSV  c
INNER JOIN cos://eu-de/my-data-1/ranking.csv STORED AS CSV r
ON c.points = r.points
WHERE r.medal != 'none'
GROUP BY c.affiliation, r.medal
```

Резултатът, който трябва да получите, е следния:

affiliation	medal	NumOfMedals
Математическа гимнази...	gold	5
Математическа гимнази...	silver	12
Математическа гимнази...	bronze	5
ТУ - Габрово	silver	5
ТУ - Габрово	bronze	1

VII. Задачи за самостоятелна работа

Задача Д1: *Получете броя на служителите и клиентите от всеки град, както и имената на градовете.*

Трябва да обедините таблици “ employees” и “ customers”. Сортирайте резултата по броя на служителите. Използвайте външното свързване (FULL OUTER JOIN) на таблиците, за да се гарантира, че градовете, които имат служители и градове, които имат клиенти, но нямат служители, са включени в списъка.

```
SELECT COUNT(DISTINCT e.EmployeeID) AS numOfEmployees,
       COUNT(DISTINCT c.CustomerID) AS numOfCompanies,
       e.City AS EmployeeCity, c.City AS CustomerCity
FROM cos://eu-de/my-data-1/employees.csv STORED AS CSV e
FULL OUTER JOIN cos://eu-de/my-data-1/customers.csv STORED AS CSV c
ON e.City = c.City
GROUP BY e.City, c.City
ORDER BY numOfEmployees DESC
```

Резултатът, който трябва да получите, е следния:

numOfEmployees	numOfCompanies	EmployeeCity	CustomerCity
4	6	London	London
2	1	Seattle	Seattle
1	1	Kirkland	Kirkland

Задача Д2: *Получете идентификатора на всички поръчки, идентификатора на клиентите които не са от Франция.*

Направете запитване за всички клиенти от Франция, а чрез предикат NOT IN към клауза WHERE създайте списък на всички клиенти извън Франция.

```
SELECT orderid, customerid, shipname, shipcountry
FROM cos://eu-de/my-data-1/orders.csv STORED AS CSV
WHERE customerid NOT IN (
    SELECT customerid
    FROM cos://eu-de/my-data-1/customers.csv STORED AS CSV
    WHERE country = 'France'
)
```

Трябва да получите 753 записа в следния формат:

orderid	customerid	shipname	shipcountry
10249	TOMSP	Toms Spezialitäten	Germany
10250	HANAR	Hanari Carnes	Brazil
10252	SUPRD	Suprêmes délices	Belgium
10253	HANAR	Hanari Carnes	Brazil
10254	CHOPS	Chop-suey Chinese	Switzerland

Задача Д3: *Получете списък на всички служители, които се намират в същия град като служител със зададено име*.

За да решим задачата ще използваме SELF JOIN. В случая трябва да получим две инстанции на таблица employees. Чрез клауза WHERE ще получим всички записи за които имената на града съвпадат, а името на служителя е например Nancy Davolio. За да изключим от самата Nancy Davolio от резултата ще търсим само клиентите с различни идентификатори.

```
SELECT e2.firstname AS colleagueFirstName,
       e2.lastname AS colleagueLastName,
       e1.city
FROM cos://eu-de/my-data-1/employees.csv STORED AS CSV e1,
     cos://eu-de/my-data-1/employees.csv STORED AS CSV e2
WHERE e2.city = e1.city
      AND e1.employeeid <> e2.employeeid
      AND e1.firstname = 'Nancy'
      AND e1.lastname = 'Davolio'
ORDER BY e1.city, e1.firstname
```

Трябва да получите, че само един служител с име Laura Callahan е от град Сиатъл, както Nancy Davolio:

colleagueFirstName	colleagueLastName	city
Laura	Callahan	Seattle

Упражнение № 3

NoSQL бази данни. MongoDB – инсталиране

I. Въведение

Терминът NoSQL е съкращение от „Not only SQL" и се използва за всички нерелационни бази данни. Този термин е въведен от Карло Строци през 1998 г. NoSQL базите данни са алтернатива на релационните бази данни. Те не ползват релационен модел на данните и не е задължително да поддържат SQL интерфейс. Основното им предназначение е да се справят с новите предизвикателства като: обработка в реално време на големи масиви от данни (Big Data), мащабируеми (основно хоризонтално) системи, системи с висока степен на отказоустойчивост и толерантни към грешки системи. Данните могат да се съхраняват не само в таблици. В случаите при които няма предварително зададени схеми за описание на данните (фиксиран брой колони) е за предпочитане да се използва нерелационна база данни. Типични примери за това са данните генерирани от GPS локацията на клиентите на дадена услуга; борсови данни; данни, генерирани от сензорни безжични мрежи от IoT инфраструктура; данни от социални мрежи и др. По принцип при всички изброени случаи се генерират много големи масиви от данни (милиони записи). Релационните бази данни трудно се справят с подобни обеми от данни тъй като трудно се мащабират хоризонтално – разпределени бази данни. Тук на помощ идват NoSQL базите данни. С тях можем лесно да избегнем сложните операции за обединяване на таблици, характерни за SQL. Много по-лесно се реализира и хоризонталното мащабиране.

Основните **предимства** на NoSQL базите данни са следните:

- Опростен дизайн.
- Не изискват прилагането на строга схема, която да важи за цялата база данни.
- По-лесна реализацията на хоризонтално мащабиране.
- Възможност за работа в реално време с много големи масиви от данни (Big Data), благодарение на ефективния начин за извличане и филтриране на данни чрез MapReduce.
- По-лесна реализация на силно разпределени бази данни.
- По-лесна реализация на синхронизация на данните между отделните сървъри за управление на базите данни при използване на силно разпределена архитектура.

Както релационните, така и NoSQL базите данни имат и **недостатъци**, например:

- Няма правила за стандартизация.
- Ограничени видове заявки (не поддържат стандартните SQL заявки).
- Не предлагат традиционни възможности на релационните бази данни, например съгласуваност.
- Когато обемът на данните се увеличава, е по-трудно да се поддържат уникални стойности като ключове.
- Не работят толкова добре със силно свързани данни.
- По-трудни за усвояване от начинаещи разработчици.
- Повечето са с отворен код, което ги прави от гледна точка на сигурност неатрактивни за бизнеса.

Основните разновидности на NoSQL базите данни са следните:

- **Документен** тип, например MongoDB, CouchDB и CouchBase.
- **Колонен** тип, например BigTable, Cassandra, HBase и Hypertable.
- **Key-value** тип, например Redis, Riak, Memcached и Scalaris.
- **Graph**-базирани, например Neo4j, HyperGraphDB, IngoGrid и Flock DB.

Документният тип бази данни свързват всеки ключ със структура от данни, която се нарича документ. Данните се съхраняват в документи, които могат да се обединяват в колекции. Документите могат да съдържат двойки ключ-масив или двойки ключ-стойност, както и вложени документи. Най-често документите се описват чрез JavaScript Object Notation (JSON) или Binary JSON (BSON) обекти. Примери за документни NoSQL са MongoDB, Apache CouchDB, Couchbase, IBM Domino, Cosmos DB и RavenDB..

При ключ-стойност базите данни всеки отделен елемент се съхранява като двойка ключ и стойности. Хранилищата за ключове и стойности са най-простата база данни сред всички NoSQL бази данни. Примери за Key-value NoSQL са Redis, Memcached, Apache Ignite, Riak и Scalaris.

Колонният тип бази данни са оптимизирани за заявки върху големи масиви от данни и вместо редове съхраняват заедно колони от данни. Примери за колонен тип NoSQL бази данни са HBase, Cassandra, Hypertable и Scylla.

Граф-базираните бази данни съхраняват информация чрез описание на връзките чрез графи. Пример за подобни връзки са връзки между потребителите на социални мрежи и транспортни връзки. Примери за граф-базираните бази данни са Neo4j, AllegroGraph, HyperGraphDB, IngoGrid и Flock DB.

1.1 Класация на базите данни

В Табл. 3.1 е показана класацията за всички бази данни за периода юни 2020 – юни 2021. Информацията е от сайта https://db-engines.com/en/ranking. Вижда се, че в Топ 10 на този етап попадат само две NoSQL бази данни – MongoDB и Redis. Причината за малкият брой NoSQL бази данни в класацията Топ 10 е основно огромният брой вече инсталирани релационни бази данни, които трябва да се поддържат. Дори да е необходимо преминаване към NoSQL база данни процесът на миграция е много бавен и скъп.

Табл. 3.1 Класация Топ 10 на всички бази данни за периода юни 2020 – юни 2021

Rank			DBMS	Database Model	Score		
Jun 2021	May 2021	Jun 2020			Jun 2021	May 2021	Jun 2020
1.	1.	1.	Oracle	Relational, Multi-model	1270.94	+1.00	-72.65
2.	2.	2.	MySQL	Relational, Multi-model	1227.86	-8.52	-50.03
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	991.07	-1.59	-76.24
4.	4.	4.	PostgreSQL	Relational, Multi-model	568.51	+9.26	+45.53
5.	5.	5.	MongoDB	Document, Multi-model	488.22	+7.20	+51.14
6.	6.	6.	IBM Db2	Relational, Multi-model	167.03	+0.37	+5.23
7.	7.	↑ 8.	Redis	Key-value, Multi-model	165.25	+3.08	+19.61
8.	8.	↓ 7.	Elasticsearch	Search engine, Multi-model	154.71	-0.65	+5.02
9.	9.	9.	SQLite	Relational	130.54	+3.84	+5.72
10.	10.	↑ 11.	MS Access	Relational	114.94	-0.46	-2.24

В Табл. 3.2 е показана същата класация, но само за NoSQL базите данни. Освен MongoDB и Redis в Топ 10 за NoSQL базите данни са Cassandra, Amazon DynamoDB и Neo4j.

Табл. 3.2 Класация за NoSQL базите данни за периода юни 2020 – юни 2021

Rank			DBMS	Database Model	Score		
Jun 2021	May 2021	Jun 2020			Jun 2021	May 2021	Jun 2020
5.	5.	5.	MongoDB	Document, Multi-model	488.22	+7.20	+51.14
7.	7.	↑ 8.	Redis	Key-value, Multi-model	165.25	+3.08	+19.61
11.	11.	↓ 10.	Cassandra	Wide column	114.11	+3.18	-4.90
16.	16.	16.	Amazon DynamoDB	Multi-model	73.76	+3.69	+8.90
18.	↑ 19.	↑ 22.	Neo4j	Graph	55.75	+3.52	+7.48
23.	23.	↓ 21.	HBase	Wide column	43.52	+0.27	-5.22
28.	↓ 26.	↓ 25.	Couchbase	Document, Multi-model	29.07	-1.16	-0.07
30.	30.	↓ 27.	Memcached	Key-value	25.18	+0.68	+0.37
39.	39.	↓ 35.	CouchDB	Document, Multi-model	16.12	+0.15	+0.05
81.	↑ 83.	↓ 77.	Oracle NoSQL	Multi-model	4.31	+0.61	+0.09
90.	↓ 87.	↓ 81.	RavenDB	Document, Multi-model	3.50	+0.25	-0.32
91.	↓ 89.	↓ 82.	IBM Cloudant	Document	3.36	+0.25	-0.32

За да получите повече информация за определена база данни щракнете с мишката върху името на базата.

1.2 Теорема CAP (Consistency, Availability, Partition Tolerance)

За нерелационните бази данни е в сила теоремата на Ерик Брюър, известна като Consistency, Availability, Partition tolerance (CAP) теорема. Тя гласи, че за една разпределена база данни не може да се гарантира едновременно всички данни да са налични, да са разделени на части между различните сървъри и структурата между тях да се съхрани.

Consistency (съгласуваност): Всеки клиент, независимо чрез кой нод се обслужва, получава едни и същи данни както останалите клиенти, независимо от протичащите конкурентните обновления (всички реплики на документите имат една и съща версия).

Availability (наличност): Дори да има нефункциониращи нодове, клиентите имат възможност да четат и записват информация.

Partition tolerance (възможност за разделяне на части): Базата данни може да бъде разпределена и да се обслужва от множество сървъри. Системата остава работоспособна при наличие на проблеми при някой от сървърите.

Всяка база данни в един момент от време гарантира два от тези три параметри от CAP теоремата:

- **AP** бази данни: CouchDB, Cassandra, SimpleDB, Riak, Dynamo.
- **CP** бази данни: Couchbase, MongoDB, Big Table, Hypertable, Hbase, Redis.
- **CA** бази данни: Всички релационни бази данни, например MySQL.

Основният проблем на NoSQL базите данни е, че те не гарантират съгласуваност във всеки момент от време. Това ограничава областите на приложение на NoSQL базите данни. Повечето NoSQL бази данни поддържа някаква форма на слаба консистентност.

Съгласуваността бива няколко вида:

- Силна съгласуваност. След приключване на обновяването, всеки един клиент ще вижда обновената стойност на данните.
- Слаба съгласуваност. Не се гарантира, че следващите заявки, след заявката за обновяване. ще върнат обновената стойност, освен ако не са изпълнени предварително зададени условия. Периодът до изпълняване на тези условията е период на съгласуваност.
- Евентуална съгласуваност. Това е форма на слаба съгласуваност, при която системата гарантира, че ако няма направени нови обновявания на данните, евентуално всички нодове ще върнат вярната стойност.

II. База данни MongoDB

Както вече видяхме MongoDB е най-популярната NoSQL база данни. Основните предимства за това са няколко: симулира някои свойства на SQL като заявки и индекси; много добра фирмена поддръжка; възможност за хостване на базата в облачна инфраструктура (AWS, Google Cloud и Microsoft Azure); поддържа много широк набор от езици за програмиране като използва драйвери за тях, например Java, JavaScript (node.js), C#, Python, Rubby, Swift и др.; възможност за генериране на динамични заявки и дефиниране на индекси. Тази база данни използва гъвкава структура, която може лесно да се променя и разширява.

Основните предимства на MongoDB са следните:

- Лесна настройка, много добра професионална поддръжка от MongoDB Inc.
- Осигурява висока производителност.
- Възможност за хоризонтална мащабируемост.
- Поддържа репликации Master-Slave.
- Данните се съхраняват под формата на BSON документи.
- Възможност за индексиране на всяко поле в документ.
- Има автоматична конфигурация за балансиране на натоварването.
- Поддържа търсене чрез използване на регулярни изрази.

Основните недостатъци на MongoDB са следните:

- Не поддържа обединения, но могат да се симулират с работа с няколко колекции.
- Вмъкването на документи е ограничено.
- Увеличава ненужното използване на паметта.
- Размерът на всяка база данни при 32-битови системи е ограничен до 2.5 GB.
- Един документ не може да е с размер по-голям от 16 MB.
- Поддържа евентуална консистентност.

2.1 Необходими инсталации за работа с MongoDB

Възможностите за работа с MongoDB са няколко:

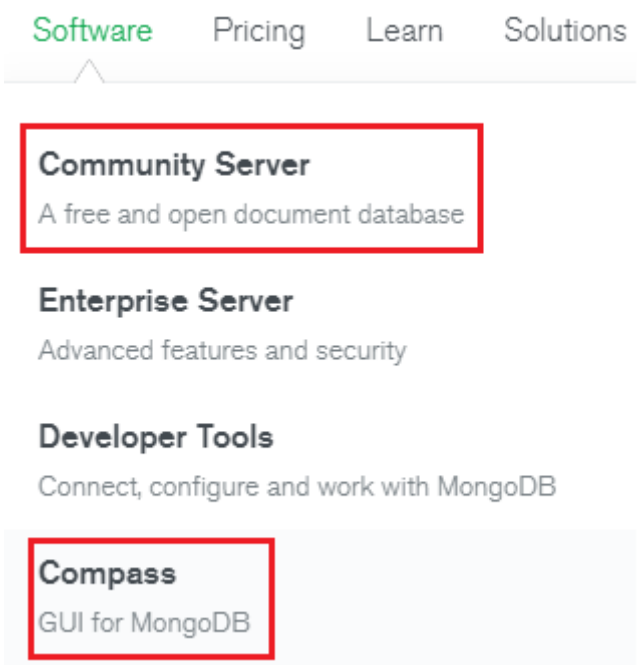
- Локално инсталиране на сървъра за управление на базата данни и останалите помощни програми (CLI или GUI интерфейс). Този подход е подходящ за развойна дейност, тестване и обучение.

- Инсталиране на сървъра за управление на базата данни на един или няколко хоста в Интернет пространството.
- Използване на MongoDB под формата на услуга (SaaS) в една от три облачни инфраструктури: Amazon AWS, Google Cloud или Microsoft Azure. Имате възможност да заявките един или няколко клъстера във всеки от които може да имате една или няколко бази данни. Наличната изчислителна мощност, RAM памет и дисково пространство зависи от типа на клъстера.

Имате възможност за създаване на напълно безплатен клъстър (shared tier cluster) или клъстер, който е напълно функционален, но е безплатен за определен период от време (free tier cluster). Трябва да имате предвид, че напълно безплатните клъстери са с не само ограничени ресурси (споделена RAM, 512MB дисково пространство), но и ограничена функционалност (например не могат да изпълняват MapReduce заявки). При безплатните за определен интервал от време клъстери получавате пълна функционалност и следните ресурси: 8GB RAM; 40GB дисково пространство и 3000 Input/Output Operations Per Second (IOPS).

2.1.1 Локална инсталация

За да използвате MondoDB локално трябва да свалите и инсталирате MongoDB Community Server и графичния интерфейс за него MongoDB Compass. Необходимите инсталации (виж Фиг. 3.1) може да получите от официалния сайт на фирма MongoDB https://www.mongodb.com/try/download.

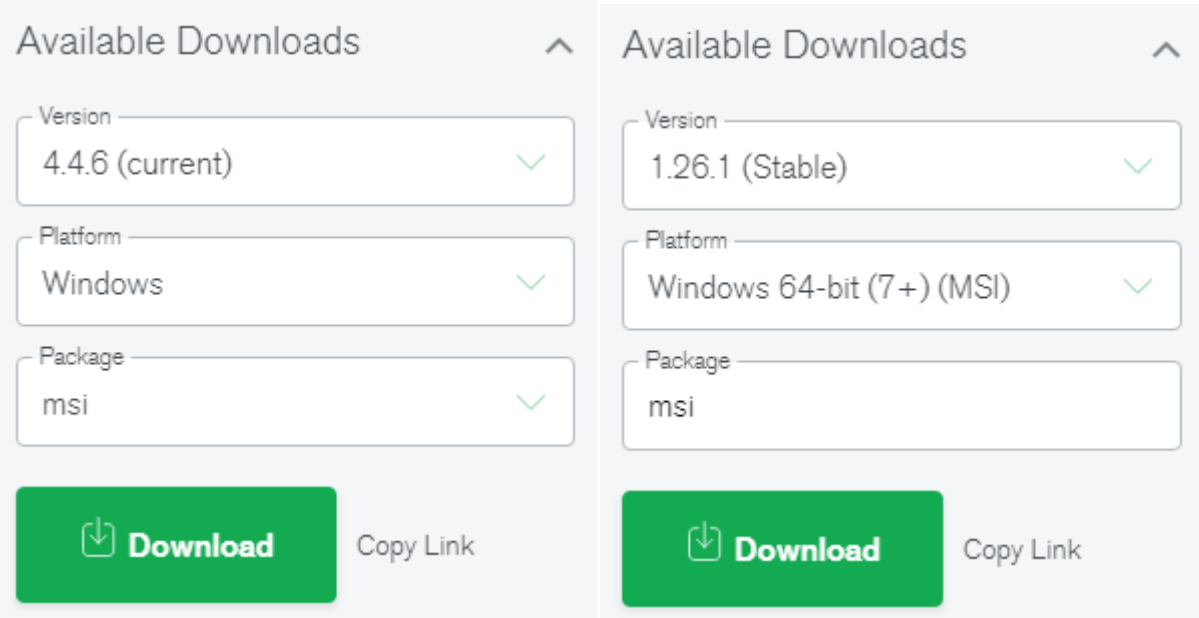


Фиг.3.1 Избор на необходимите инсталации

Изберете най-новата стабилна версия на желаната инсталация за избрана платформа и пакет. На Фиг. 3.2 е показан необходимия избор при ОС Windows.

Ако работите с ОС Windows 7 трябва да намерите следния инсталационен пакет за MongoDB Community Server:

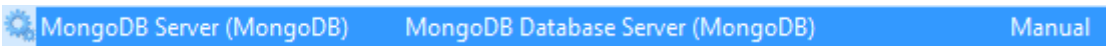
mongodb-win32-x86_64-2008plus-ssl-4.0.22-signed.msi



MongoDB Community Server MongoDB Compass

Фиг.3.2 Избор версия и тип инсталация за Windows

Първо, инсталирайте сървъра като локална услуга (local service), а след това и MongoDB Compass GUI. Ако не искате всеки път след включване на компютъра сървърът да се активира задайте ръчна настройка. За целта натиснете Start бутона на Windows и въведете “services.” Изберете „…local services.” Намерете услугата “MongoDB Server” и с десния бутон на мишката изберете Properties. За “Startup type” задайте “Manual” (виж Фиг. 3.3).



Фиг.3.3 Избор версия и тип инсталация за Windows

Когато искате да стартирате сървъра, изберете услугата и с десния бутон на мишката променете статуса й (Start / Stop).

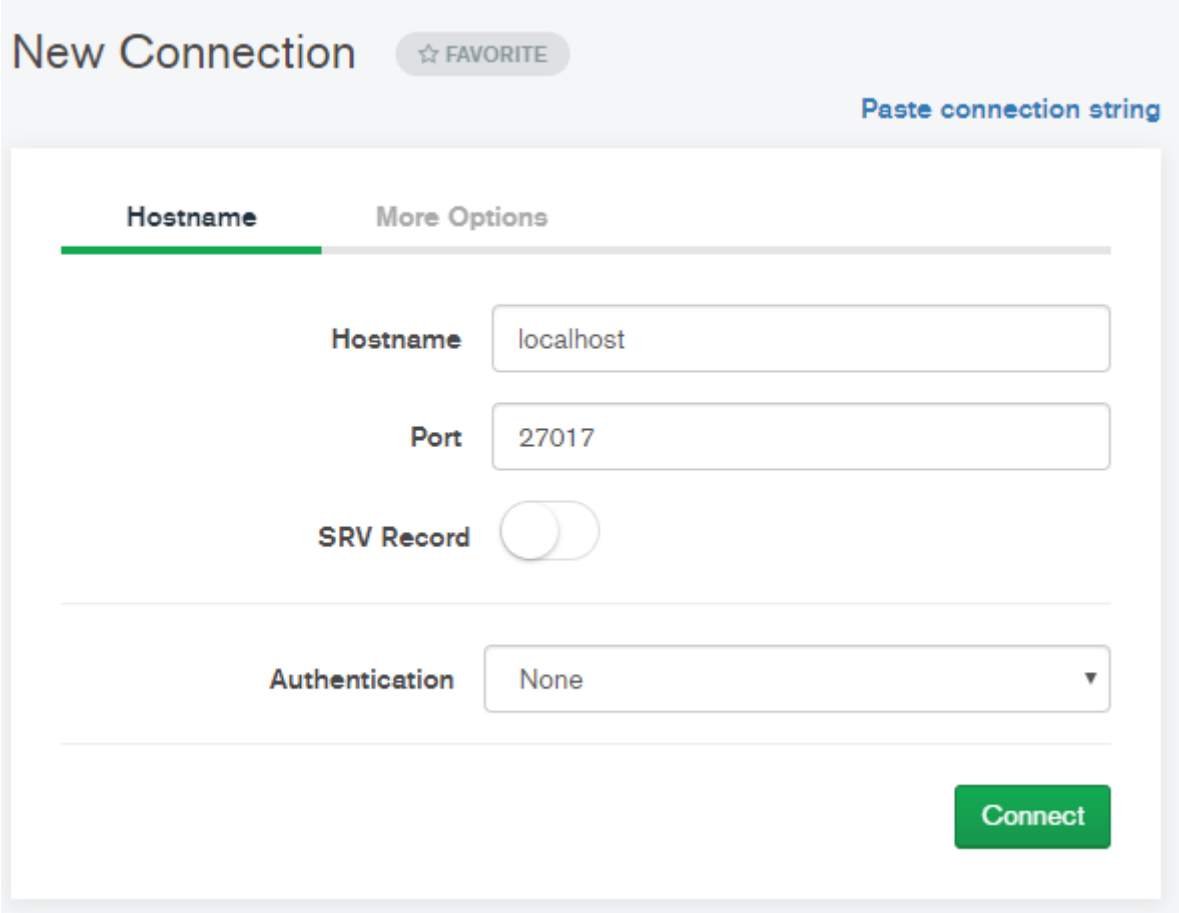
Намерете на твърдия диск къде е инсталацията на MongoDB Server. Ако не сте задали друг път за инсталиране, би трябвало изпълнимите файлове да са в папка

C:\Program Files\MongoDB\Server\X.Y\bin

където X.Y е версията на сървъра. В тази папка трябва да е приложението Mongo.exe. Чрез него може да реализирате комуникация със сървъра през командния ред (CLI). Направете препратка (shortcut) към десктопа, за да стартирате приложението по-бързо.

Тествайте работоспособността на сървъра. Стартирайте приложението mongo.exe и от конзолата въведете команда “help“. Ще получите списък на всички команди, които CLI поддържа. Изпълнете команда “show dbs”, за да получите списък на всички налични до този момент бази данни.

Много по-лесно ще работите със сървъра ако използвате графичния интерфейс на приложението MongoDB Compass (GUI). Чрез него ще получите достъп до сървъра, независимо дали е локален или в облачна инфраструктура. Стартирайте приложението, изберете “New Connection” и натиснете бутон “Connect” (виж Фиг. 3.4).



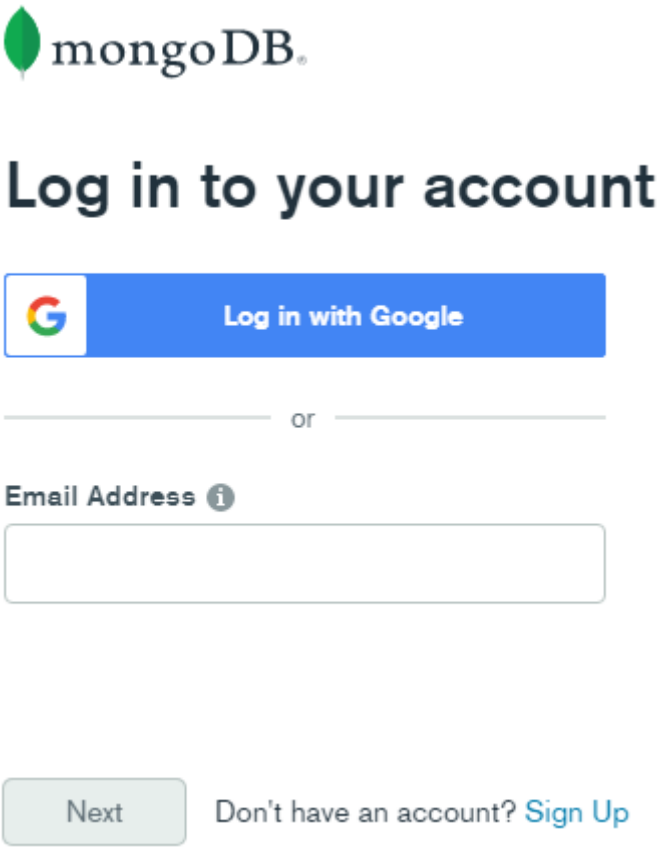
Фиг.3.4 Свързване с MongoDB Server чрез MongoDB Compass GUI

След успешен логин ще получите достъп до всички налични бази данни. Чрез това приложение може да създавате нови бази данни, да изтривате съществуващи бази данни, да създавате и изтривате колекции и документи. Чрез “Performance” можете да получите графична статистика за изпълняваните операции, за броя на четенията и записите, натовареността на връзките към сървъра и използваната памет.

2.1.2 Инсталация в облака на Amazon

MongoDB Atlas е най-иновативната услуга за бази данни в облак на пазара, с много добро разпределение на данните и мобилност. Може да използвате една от три облачни платформи, за да създадете клъстер: Amazon AWS, Google Cloud и Microsoft Azure. За да получите достъп до MongoDB Atlas е необходимо да имате валидна регистрация в mongodb.com. Регистрацията и авторизацията на достъпа се реализира от следната страница:

https://account.mongodb.com/account/login?signedOut=true



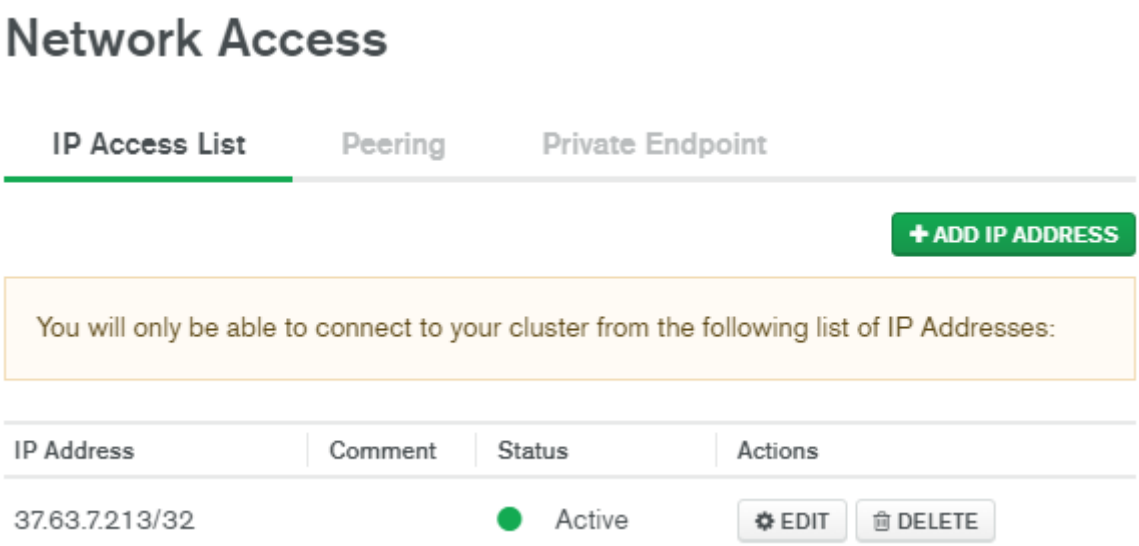
Фиг.3.5 Форма за регистрация и авторизация

Ако нямате валиден акаунт натиснете “Sign Up” и следвайте инструкциите за регистрация. След въвеждане на валидна двойка “име-парола” ще бъдете препратени до страницата за управление на услугата. Следвайте следната последователност на конфигуриране на услугата:

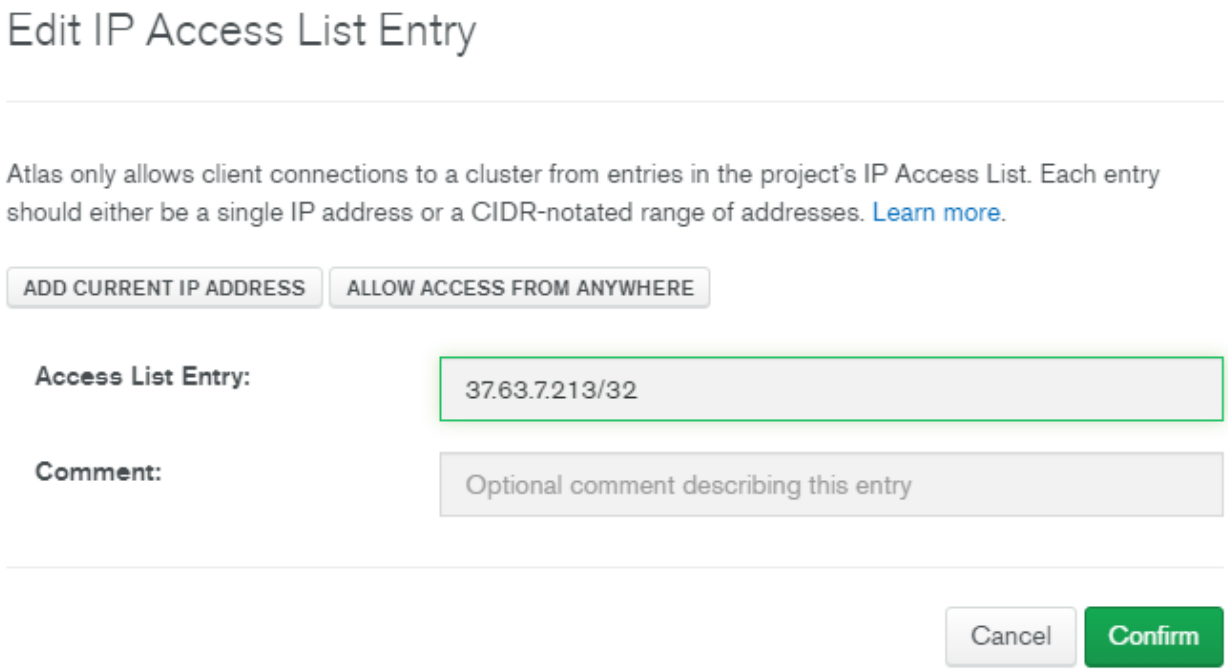
1. Създайте поне един **клъстер**. Като начало създайте “shared tier cluster”, който е безплатен. Както вече знаем, клъстерите могат да бъдат създадени в AWS, Google Cloud или Azure. Използвайте AWS.

2. Създайте **потребители** на услугата. Трябва да имате поне един администратор, който да има неограничен достъп до всички бази данни. Това се реализира от менюто в ляво. Изберете “Database Access” от секция SECURITY. За метод на автентикация изберете “Password”. Задайте име и парола с цел базова автентикация на достъпа. Задайте права на достъпа за съответния потребител (Database User Privileges). За администратора изберете “Atlas Admin”. За потребител, който трябва да може да чете и записва в базите данни изберете “Read and write to any database”. Ако е необходим достъп само за четене изберете “Only read any database”.

3. Задайте кой да има отдалечен **достъп** до базите данни. Изберете “Network Access” от секция SECURITY. Възможностите за избор са следните: списък от IP адреси; връзка между частен виртуален облак с вашият MongoDB Atlas виртуален частен облак (Peering Connection) и връзка чрез създаване на частна крайна точка на достъп (само за AWS и Azure). За безплатните регистрации е възможен достъп само чрез “IP Address List” (виж Фиг. 3.6).



Фиг.3.6 Мрежов достъп до клъстера



Фиг.3.7 Промяна на IP адреса за достъп

Чрез бутон "+ADD IP ADDRESS" имате възможност да зададете IP адреса от който да имате достъп до услугата. Ако вашият IP адрес се получава динамично трябва всеки път след включване на компютъра си да реализирате задаване на вашия нов IP адрес, тъй като той се променя динамично. За целта натиснете бутон <EDIT> (виж Фиг. 3.6) и от формата "Edit IP Access List Entry" (виж Фиг. 3.7) натиснете бутон <ADD CURRENT IP ADDRESS>. По този начин ще бъде получен вашия моментен IP адрес. Остава да натиснете бутон "Confirm". Ако искате базите данни да са достъпни от кой да е IP адрес натиснете бутон <ALLOW ACCESS FROM ANYWHERE> (0.0.0.0/0).

4. Създайте колкото са необходими **колекции**. За целта изберете CLUSTERS от секция "DATA STOREGE" и натиснете бутон <CONNECTIONS>. Всяка колекция съдържа една или няколко бази данни.

5. Създайте необходимите **бази данни**. От избрана колекция натиснете бутон <+ Create Database>.

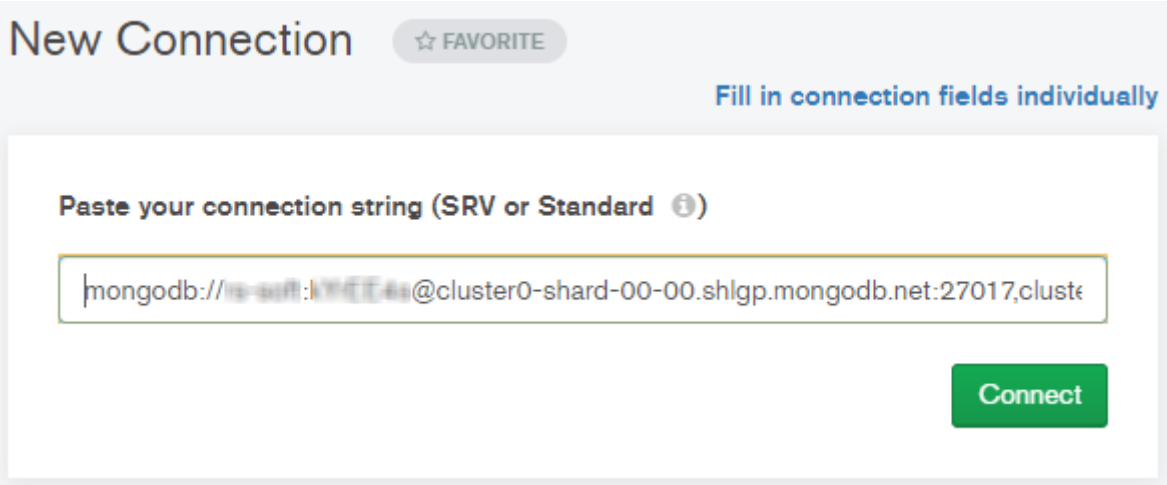
6. Създайте или импортирайте необходимите **документи** за всяка база данни. След като сте избрали база данни натиснете бутон <INSERT DOCUMENT>. Може да въведете документ по няколко начина: чрез редактора на услугата (включително и Copy/Paste) и програмно.

Свързването с базите данни от MongoDB Atlas може да се реализира по три начина. За да изберете начин на свързване и да получите низ за свързване (URL) с услугата натиснете бутон <CONNECT> след като сте избрали CLUSTERS. Наличните начини на свързване са:

- Чрез приложението Mongo shell (изисква сваляне и инсталиране).
- Чрез програмно приложение, написано на език за който има наличен MongoDB драйвер (C, C++, C#.NET, Go, Java, Node.js, Perl, PHP, Python, Ruby и Scala). Изберете език за програмиране и версия на драйвера с който ще работите. Остава да копирате низа за достъп, например:

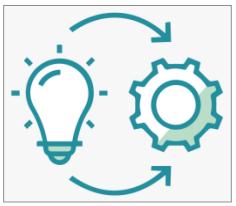
mongodb+srv://<user>:<password>@cluster0.shlgp.mongodb.net/<database>?retryWrites=true&w=majority

  където: <user> трябва да замените с валидно име на потребител; <password> - с паролата на този потребител, а <database> - с името на базата данни с която искате връзка.
- Чрез приложението MongoDB Compass. Трябва да изберете версията на Compass, която сте инсталирали и отново да копирате низа за достъп.



Фиг.3.8 Свързване с клъстер от MongoDB Atlas чрез интерфейса на MongoDB Compass

III. Задачи за изпълнение



Задача 1: *Инсталирайте MongoDB Community Server на вашия компютър като локална услуга. Задайте дали услугата да бъде стартирана автоматично след включване на захранването или да се активира от вас когато е необходимо.*

Сървърът използва ресурсите на вашия компютър, дори когато не се изпращат заявки към него. Затова е препоръчително да забраните автоматичното му стартиране след включване на захранването на компютъра.

Задача 2: *Стартирайте конзолата за MongoDB Server. Това е приложението mongo.exe, което предоставя конзолен интерфейс към сървъра (CLI). Проверете какви команди може да изпълнявате чрез команда help (виж. Фиг. 3.9).*

```
> help
        db.help()                    help on db methods
        db.mycoll.help()             help on collection methods
        sh.help()                    sharding helpers
        rs.help()                    replica set helpers
        help admin                   administrative help
        help connect                 connecting to a db help
        help keys                    key shortcuts
        help misc                    misc things to know
        help mr                      mapreduce

        show dbs                     show database names
        show collections             show collections in current database
        show users                   show users in current database
        show profile                 show most recent system.profile entries with time >= 1ms
        show logs                    show the accessible logger names
        show log [name]              prints out the last segment of log in memory, 'global' is default
        use <db_name>                set current database
        db.foo.find()                list objects in collection foo
        db.foo.find( { a : 1 } )     list objects in foo where a == 1
        it                           result of the last line evaluated; use to further iterate
        DBQuery.shellBatchSize = x   set default number of items to display on shell
        exit                         quit the mongo shell
>
```

Фиг.3.9 Списък на командите, които Mongo CLI предоставя

За да получите списък на всички вече създадени бази данни изпълнете команда

show dbs

Ако искате да получите достъп до дадена база данни от получения списък, използвайте команда

use database_name

Списък на колекциите на избраната от вас база данни може да получите чрез команда

show collections

за да работите с дадена колекция използвайте команда

use collection_name

Задача 3: *Като използвате MongoDB CLI създайте нова база данни с име "students". В тази база данни въведете 3 документа, които описват студенти.*

Вече знаем, че MondoDB организира документите в колекции, а колекциите формират база данни. Документите се въвеждат в JSON формат. Всеки документ (JSON обект) започва с отваряща фигурна скоба **{** и завършва със затваряща фигурна скоба **}**. В тялото на документа **{ }** има двойки "име на поле" и "стойност на поле" в следния синтаксис:

```
db.students.insert(
    {
        "id": "1234567",
        "name":
        [
            {
                "firstName": "Иван",
                "lastName": "Георгиев"
            }
        ],
        "department": "КСТ",
        "specialty": "СКИ",
        "course": 3
    }
)
```

При конкретният случай имената на полетата са следните: “id”, “name”, “department”, “specialty” и “course”. След символа за двоеточие **:** следват техните стойности. Те могат да бъдат числа, низове, масиви или други JSON обекти. Например, поле name има стойност масив, който има други две полета, които формират JSON обект.

За да създадем нова база данни students ще използваме команда use:

use students

Тъй като тази база данни не съществува, сървърът създава нова база данни. Следващата стъпка е да създадем колекция в която да бъдат разположени нашите документи. Най-лесният начин за създаване на колекция е да се вмъкне запис (документ). Ако колекцията не съществува, тя ще бъде създадена автоматично. В този случай името на колекцията ще съвпадне с името на базата данни. За вмъкване на документ ще използваме метод **insert**:

```
> use students
switched to db students
> db.students.insert(
... {
... "id":"1234567",
... "name":
... [
... {
... "firstName":"Иван",
... "lastName":"Георгиев"
... }
... ],
... "department":"КСТ",
... "specialty":"СКИ",
... "course":3
... }
... );
WriteResult({ "nInserted" : 1 })
> _
```

Вече имате 1 документ в базата данни. По аналогичен начин въведете и останалите документи.

Задача 4: *Като използвате създадената база данни “students” разпечатайте съдържанието на всички документи.*

Ще използваме функция **find**, за да реализираме задачата:

db.students.**find**()

Ще получите отговор, който връща неформатирано съдържанието на всички документи в колекция “students”. За да видите документите в JSON формат, изпълнете следната команда:

db.students.**find**().forEach(printjson)

```
> db.students.find().forEach(printjson)
{
        "_id" : ObjectId("60dd574d995feb53ad6c47ce"),
        "id" : "1234567",
        "name" : [
                {
                        "firstName" : "Иван",
                        "lastName" : "Георгиев"
                }
        ],
        "department" : "КСТ",
        "specialty" : "СКИ",
        "course" : 3
}
```

Вижда се, че сървърът е генерирал идентификационен код за вашите документи – поле “**_id**”. Всички полета, които започват с долна черта са системни и не трябва да бъда изтривани. Стойността на полето “_id” е обект от клас **ObjectId**. Стойността на тези полета е 12 байта, съставени от няколко вериги от по 2-4 байта. Всяка верига представлява и обозначава специфичен аспект от идентичността на документа: секундите от рождената дата на ОС Unix (4 байта); идентификатор на машината (3 байта); идентификатор на процеса (2 байта) и брояч, започващ със случайна стойност (3 байта). По този начин полето “_id” е уникално и се използва като основен ключ за елементите в колекция. Може да замените стойността на полето “_id”, но не може да го изтриете!

Задача 5: *Разпечатайте на конзолата в JSON формат студентите, които са от втори курс.*

Отново ще използваме метод **find**, но ще зададем филтър за резултата, който желаем. В случая трябва да филтрираме стойността на поле course. Задайте филтъра като JSON обект като атрибут на метод find. Това се реализира по следния начин:

```
> db.students.find({"course":2}).forEach(printjson)
{
        "_id" : ObjectId("60dd5933995feb53ad6c47cf"),
        "id" : "2345678",
        "name" : [
                {
                        "firstName" : "Георги",
                        "lastName" : "Петков"
                }
        ],
        "department" : "КСТ",
        "specialty" : "СКИ",
        "course" : 2
}
>
```

IV.Задачи за самостоятелна работа

Задача Д1: Реализирайте всички задачи от Секция III като използвате приложението MongoDB Compass. Създайте нова база данни, например SKI3, и въведете информация за студентите от вашата група. Разпечатайте на конзолата всички студенти чиято фамилия започва с низа “Иванов”.

Едно възможно решение на задачата е да използвате **регулярни изрази**, за да укажете какво е условието за филтриране на данните. Синтаксисът е следния:

db.collectionname.find({fieldName:{$regex:”filter string”}}).forEach(printjson)

където за стойност на полето, което трябва да филтрирате, се задава JSON обект. Името на полето е $regex, а стойността му низ, който съдържа шаблона за търсене. Припомнете си какъв трябва да е този шаблон ако търсим низ с който започва друг низ.

```
> db.students.find({"name.lastName":{$regex:"^Ив"}}).forEach(printjson);
{
        "_id" : ObjectId("60dd5955995feb53ad6c47d0"),
        "id" : "3456789",
        "name" : [
                {
                        "firstName" : "Петя",
                        "lastName" : "Иванова"
                }
        ],
        "department" : "КСТ",
        "specialty" : "СКИ",
        "course" : 3
}
>
```

Упражнение № 4

Проектиране на MongoDB бази данни

I. Въведение

Релационните бази данни имат фиксирана **схема** на описание на данните. Полетата от таблиците имат фиксиран тип и дори да не се налага да приемат в даден момент стойност, те трябва да имат стойност, дори тя да е null. Това води до неефективно използване на паметта. За нерелационните бази данни често се казва, че са бази без схема. На практика всяка база данни има схема, но при NoSQL базите данни схемата е *динамична*. Това означава, че не е наложително схемата твърдо да се прилага за всички документи в дадена колекция. Напълно допустимо е да липсват определени полета в част от документите. Схемата се дефинира на ниво приложение. Тя трябва да може да се развива с промяната на приложението. Такава схема се нарича и *полиморфна*.

Дизайнът на схемата на базата данни е важен процес тъй като от него до голяма степен зависи производителността на цялата услуга, както и мащабируемостта на базата данни. Преди да проектирате базата данни, трябва да отговорите на редица въпроси, например:

1. Кои са обектите с които приложението ви работи и които трябва да са част от база данни?
2. Колко и от какъв тип са връзките между тези обекти (1:1, 1:N или M:N)?
3. Колко е броят на потребителите на вашето приложение и колко често ще се извлича информация от базата данни?
4. Колко често се налага вмъкване на нови обекти в базата данни?
5. Колко често се налага изтриване на обекти от базата данни?
6. Колко често се налага модифициране на съдържанието на обекти от базата данни?
7. Какви заявки до базата данни ще реализира приложението ви?
8. Как ще се реализира достъпа до обектите от базата данни – чрез единичен ключ, чрез съставен ключ, чрез стойността на свойства или чрез прилагане на някакъв вид филтриране и сортиране на данните?

1.1 Видове връзки между обектите

Връзка 1:1

Връзката "едно към едно" е вид кардиналност, която описва връзката между две същности, при която един запис от същност A е свързан с един запис в същност B. Този тип връзка може да се моделира по два начина: чрез вграждане на връзката като поддокумент, или свързване с документ в отделна колекция. Всичко зависи от това колко често се прави достъп до данните, а също и от жизнения цикъл на набора от данни - ако обект A се изтрие, трябва ли обект B да продължи да съществува?

Връзка 1:N

Връзката “едно към много” се отнася до връзката между две същности A и B, при която едната страна може да има една или повече връзки с другата, докато обратната връзка е

едностранна (един студент изучава множество дисциплини). Подобно на връзката 1:1, тя също може да бъде моделирана чрез използване на вграждане или свързване.

Връзка N:M

Тази връзка е от тип „много към много" се отнася до връзка между две същности A и B, като и двете страни могат да имат една или повече връзки към другата. Ако имате опасение, че моделирането на тази връзка с един документ ще доведе до силно нарастване на размера му, то в този случай се предпочита A и B да се моделират като самостоятелни колекции. Това е компромисно решение, тъй като ще трябва да извършим втора заявка, за да получим данните за същност B, така че скоростта на четене може да бъде намалена. Не трябва да се забравя за възможността за обединение между A и B на ниво приложение. При правилното индексиране (за оптимизиране на паметта) обединенията от страна на сървъра са малко по-скъпи от тези, които се реализират от самата база данни.

От изложеното по-горе стана ясно, че при дизайн на схемата на базата данни се използват два основни подхода: **вграждане** и **свързване**.

1.2 Денормализиран (вграден) модел на данните

При документния тип NoSQL бази данни можете да вграждате свързани данни в една структура на документ. Тези схеми формират *денормализирани* модел на данните. Денормализацията на базата е процес при който се търсят обекти с връзки от тип 1:1. В този случай тези обекти трябва да се моделират чрез един документ. Това обаче трябва да се направи само ако обектите не са големи по размер и не се обновяват често. Предимството на денормализацията е бързодействието на обслужване на заявките – необходимо е да се достъпи само един документ, който съдържа цялата необходима информация. Вградените модели на данни позволяват на приложенията да съхраняват свързани части от информация в един и същи запис в базата данни. В резултат на това на приложенията може да се наложи да правят по-малко заявки и актуализации за извършване на общи операции.

Нека да създадем схема, която описва студенти при условие, че необходимата информация е следната: идентификационен номер на студента, име и фамилия, информация за връзка с него (e-mail адрес, номер на телефон, facebook профил). Ако информацията за връзка със студента се използва много често от вашето приложение, то тя трябва да се вгради в документа (виж Фиг. 4.1). Вграденият документ е поле "contact" и неговата стойност.

Вграждането осигурява по-добра производителност при операции четене, както и възможност за заявяване и извличане на свързани данни с една операция. Вградените модели на данни дават възможност да се актуализират свързаните данни с една атомична операция за запис. За да получите достъп до данни във вградени документи, използвайте точкова нотация за достигнете до вградените документи.

Фиг.4.1 Пример за схема, използваща вграждане

Вграден модел на данните е подходящо да се използва при:

- Наличие на връзки от тип “един към един”.
- Наличие на връзки от тип “един към много”. Необходимо е подчинените (“много”) документи да са в силна връзка с родителския документ (“един”).
- Гарантиране на интегритет при операции за четене и запис на данни от типа “едно към едно” и “едно към много”, които се изтриват заедно по подразбиране.
- Когато данните много по-често се четат отколкото обновяват.

Този модел не е подходящо да се използва при:

- Наличие на връзки от тип “много към много”.
- Тенденция за бързо увеличаване на размера на вградените документи. Трябва да се има предвид, че има физическо ограничение за размера на всеки документ и тенденция за намаляване на производителността при работа с много големи по размер файлове.
- Ако се налага много често модифициране на вградените данни.
- При независимост на заявките към документите от различните колекции

1.3 Нормализиран модел на данните

Нормализираните модели на данни описват взаимоотношенията с помощта на препратки между документите. Нормализацията е процес, който има за задача да намали излишъка и взаимозависимостта на информацията в базата. За целта трябва да се прецени кои обекти да се опишат в отделен документ. Едно от предимствата на нормализацията е, че базата ще стане по-малка по размер (няма да има повторение на обекти в рамките на една колекция. Един от недостатъците на нормализацията е, че се увеличават броя на заявките – получаването на крайният резултат може да изисква заявки към множество взаимосвързани документи. Когато връзката между два обекта е “1:N”, то най-логично е двата обекта да се моделират в два различни документа. Например, потребителят на една книжарница (1) и книгите (N), които той заема трябва да са отделни документи: всеки потребител може да заема множество книги и няколко потребители могат да заемат едни и същи книги. Книгите, които всеки потребител заема, ще се идентифицират с “bookId” на книгите, а за всяка книга ще има отделен документ, в който се описва идентификатора й, автора, наименованието, броя страници, жанра на книгата и др.

Ако използваме нормализиран модел за примера със описанието на студенти ще създадем две колекции. В едната с име “students” ще се съдържат документи, които описват студентите с идентификатор и име, а в другата с име “contact” ще бъдат всички документи, които описват информация за връзка със студентите. Връзката на документите от двете колекции ще се реализира чрез двойката “id-studentId” (виж Фиг. 4.2).

```
{
    "id": "1234567",
    "name":
     [
        {
            "firstname": "Иван",
            "lastname": "Георгиев"
        }
    ]
}
```

Документ от колекция “student”

```
{
    "studentId": "1234567",
    "mobile": "YYYYXXXXXX",
    "email": "you@domain.com",
    "facebook": "……"
}
```

Документ от колекция “contact”

Фиг.4.2 Пример за схема, използваща връзки между документите

Свързан модел на данните е подходящо да се използва при:

- Вграждането би довело до дублиране на данни, но не би осигурило достатъчно предимства в производителността при четене.
- Трябва да се моделират по-сложни взаимоотношения от тип "много към много".
- Трябва да се моделират големи йерархични масиви от данни.

Нека да разгледаме друг пример. Клиентите на дадена услуга могат да имат два адреса - домашен и на работното си място. Тук въпросът е дали тези адреси да са част от документа, който описва клиент или да бъдат в отделни документи. Тъй като домашният адрес е сравнително уникален за всеки клиент, нормално е той да е част от документа, който описва клиента. Множество клиенти обаче може да имат един и същ работен адрес. Ако вашата програма трябва да проверява често кои от вашите клиенти работят на едно и също място, то трябва да интегрирате и работния адрес в документа, който описва клиент. В противен случай, работното място трябва да е отделен документ, за да се намали размера на документите, описващи клиентите.

Нормализация или денормализация?

Заявките към една база данни могат да бъдат Create, Read, Update и Delete (CRUD). Критични са операциите при които се реализира промяна на съдържанието на базата данни. Целта е всички клиенти на услугата е да получат последно модифицираните данни, дори когато базата е с разпределена архитектура. Следователно, операциите, свързани със запис, трябва да са атомарни. Това означава, че само един процес трябва да има права да обновява един документ или колекция в даден момент от време. Ако документа е денормализиран, атомарността (невъзможни са блокировки на данни) на операция запис се гарантира лесно. Това не е в сила за нормализираните документи. В този случай едно обновяване може да изисква запис в множество документи. Повечето NoSQL бази данни решават проблема с атоматността на операции запис чрез репликиране на базата данни (дублиране на данните на множество инстанции на базата, които формират клъстер).

Съществуват три основни **правила** на базата на които се определя дали да използвате нормализация или денормализация:

Правило № 1: Ако обект B трябва да бъде достъпен самостоятелно (извън контекста на родителския обект A), тогава използвайте връзка, в противен случай - вграждане.

Правило № 2: Масивите не трябва да растат неограничено:

- Ако броят на документите от страна B е по-малък от няколко стотин и са малки по размер, то не е проблем да ги вградите в обекта A.
- Ако броят на документите от страна B е по-голям от няколко хиляди използвайте препратки като отделите обектите в отделна колекция.

Правило № 3: Връзките на ниво приложение са обичайна практика и не трябва да се отхвърлят; в тези случаи изборът на полета за индексиране до голяма степен определя производителността на заявките.

MongoDB дава възможност да проектираме схемата така, че да отговаря на нуждите на приложенията на потребителите. Не трябва да се забравя, че проблемите с производителността често се дължат на неправилен подбор на схемата на базата данни.

Избор на правилния брой на документите

Основателен е въпросът дали да се работи с много на брой малки по размер документи, или с малко документи, които съдържат много на брой полета и техните стойности. Ако бързодействието е важният за вас параметър, то трябва да се работи с колкото се може по-малко на брой документи, тъй като дисковите операции са много бавни в сравнение с операциите в паметта. Ако един документ в даден момент не е необходим за функционирането на вашата система, то най-добре е той да бъде изтрит. Затова още при проектирането трябва да се дефинира времето на живот на всеки документ.

1.4 Шаблони за моделиране на данните

Шаблоните за моделиране на данните са пряко свързани с опита на дизайнерите на бази данни и добрите практики в тази област. Всеки шаблон позволява бързо и надеждно решаване на определен проблем или набор от проблеми. Познанията в тази област са ключови от гледна точка време на дизайн и качество на дизайна. За да изберете подходящия шаблон трябва да знаете не само какви шаблони вече съществуват, но и кой шаблон при кой сценарий е най-подходящ (use case scenario). В Табл. 4.1 са описани имената на 12 шаблони за дизайн, които се използват при MongoDB и тяхното практическо приложение.

1. Шаблон “Approximation”

Моделът “Approximation” е добро решение за приложенията, които работят с данни, чието изчисляване е скъпо, тъй като изисква много изчислителни ресурси, а точността не е от решаващо значение. Апроксимирането на входните данни позволява да извършваме по-малко записи в базата данни, което увеличава производителността на услугата, и все пак да поддържаме статистически валидни числа. Например, ако трябва да изчисляваме средната стойност на температурата на годишна база в даден град няма смисъл да записваме в базата данни информация за температурата през малък интервал от време, а например веднъж на ден. В този случай не е толкова важна и точността на резултата. От страна на услугата може да покажем, че средната годишна температура в избран град е 18°C, а не реалното число, например 18.03°C.

Табл. 4.1 Шаблони за дизайн на MongoDB бази данни

Примерно приложение / Шаблон за дизайн	Каталог	Управление съдържанието	Интернет на нещата	Мобилни приложения	Персонализация	Анализ в реално време	Общ изглед
1. Approximation	■		■	■		■	
2. Attribute	■	■					■
3. Bucket			■			■	
4. Computed	■		■	■	■	■	■
5. Document Versioning	■	■			■		■
6. Extended Reference	■			■		■	
7. Outlier			■	■	■		
8. Preallocated			■			■	
9. Polymorphic	■	■		■			■
10. Schema Versioning	■	■	■	■	■	■	■
11. Subset	■	■		■	■		
12. Tree and Graph	■	■					

Проблемът при този шаблон е да преценим колко често да записваме в базата данни. Дали да бъде всяка минута, всеки час, всеки ден, през седмица или всеки месец. Друг типичен пример за използване на този модел е при обработка на данни от сензори от безжична сензорна мрежа (WSN). Тук основният въпрос е колко често трябва да записваме данните от сензорите в базата. Правилният отговор зависи от това колко инертни са тези данни и какво е значимото им изменение при което трябва да се активира някакво действие. Ако сензорът например измерва температура и сме преценили, че значимото изменение е 5 градуса, то в базата трябва да записваме, само когато се достигне този предварително дефиниран праг.

Плюсове

- По-малко записи в базата данни.
- Поддържане на статистически валидни данни.

Недостатъци

- Не се представят точните стойности на данните.
- Изисква се изпълнение на код от страна на приложението (поддържане на брояч, генератор на случайни числа).

2. Шаблон “Attribute”

Шаблонът "Attribute" е полезен при документи с много сходни полета, но имат и подмножество от полета с общи характеристики по които информацията трябва да се сортира или да се правят заявки. Шаблонът е подходящ и когато трябва да сортираме по стойностите на полета, които са налични само в малка част от документите. Следователно, този шаблон трябва да се използва за схеми, които имат набори от полета с един и същ тип стойност, например списъци с дати. Той работи добре и при работа с характеристики на продукти. Някои продукти, като например дрехи и обувки имат размери, които се описват по различен на чин в различните държави. Нека да опишем артикули от електронен магазин за облекло. Всички артикули имат общи полета като код на артикула, наименование на артикула, фирма производител и др. Всеки артикул има размер, но стойността му се различава в зависимост от локацията на клиента. На Фиг. 4.3а е показан част от документ, който описва артикул “обувки”. Размерите на обувките в различните локации са описани като отделни полета. Търсенето на размера на обувките ще изисква търсене в много полета едновременно. Поради тази причина ще ни трябват няколко индекса за колекцията ни от артикули (виж Фиг. 4.3б). С помощта на шаблон “Attribute” можем да преместим информацията за размера на артикула в масив от двойки „ключ-стойност” и така да намалим нуждата от използването на множество индекси (виж Фиг. 4.3в). В този случай търсенето ще бъде много по-бързо, тъй като ще използваме само един съставен ключ (виж Фиг. 4.3г).

```
{
   category: "shoes",
   manufacturer: "Ermenegildo Zegna",
   model: "Derby",
   ...
   size_EU: 42,
   size_USA: 8,
   size_UK: 8.5,
   size_JP: 26.5,
   ...
}
```

а) Документ от колекция “артикул”

```
{ size_EU: 1 }
{ size_USA: 1 }
{ size_UK: 1 }
...
```

б) Индекси

```
{
   category: "shoes",
   manufacturer: "Ermenegildo Zegna",
   model: "Derby",
   ...
   size: [
      {
         location: "EU",
         value: 42
      },
      {
         location: "USA",
         value: 8
      },
      ...
   ]
}
```

в) Документ използващ шаблон “Attribute”

```
{
   size.location: 1,
   size.value: 1
}
```

г) Нов индекс

Фиг.4.3 Използване на шаблон “Attribute”

Можем още да намалим размера на документите и да използваме само един ключ като използваме стойността на поле “location” за ключ. При този вариант на схемата документите ще имат следното съдържание:

```
{
   category: "shoes",
   manufacturer: "Ermenegildo Zegna",
   model: "Derby",
   ...
   size:
      {
         "EU": 42,
         "USA": 8,
         ...
      }
}
```

Фиг.4.4 Подобрен вариант на документа описващ обувки

3. Шаблон “Bucket”

Когато данните постъпват като поток за определен период от време (данни от сензори например), може да съхраняваме всяко измерване в отделен документ. Ако имаме сензор, който измерва температурата и я записва в базата данни всяка минута, нашият поток от данни може да изглежда по следния начин:

```
{
   sensor_id: 1,
   timestamp: ISODate("2021-09-01T08:00:00.000Z"),
   temperature: 23.4
}
{
   sensor_id: 1,
   timestamp: ISODate("2021-09-01T08:01:00.000Z"),
   temperature: 22.8
}
```

а) Потоково генерирани документи – по един на всяка минута

```
{
    sensor_id: 1,
    measurements: [
       {
          timestamp: ISODate("2021-09-01T08:00:00.000Z"),
          temperature: 23.4
       },
       {
          timestamp: ISODate("2021-09-01T08:01:00.000Z"),
          temperature: 22.8
       }
    ],
    measuremets_count: 2,
    sum_temperature: 46.2
}
```

б) Един документ, който ще съдържа данните от сензора за определен период от време

Фиг.4.5 Пример за използване на шаблон “bucket”

Ако използваме шаблон “bucket”, данните от сензора ще се записват в масив “measurements” за зададен период от време, например 1 час. В нашия случай в масива има само два записа, но той ще расте, докато записите станат 60. При всеки запис в масива се инкрементира брояч на измерванията “measurement_count” и се актуализирана текущата сумата на всички температури в масива - “sum_temperature”. Ако в даден момент е необходима средната температура, можем лесно и бързо да я получим чрез разделяне на стойностите на двете полета, sum_temperature / measurement_count.

Плюсове

- Необходими са по-малко индекси.
- Заявките стават по-прости за писане и като цяло са по-бързи.

4. Шаблон “Computed”

Този шаблон се използва с цел намаляване на натоварването на процесора и увеличаване на производителността на приложението. Винаги, когато вашата система извършва едни и същи изчисления многократно и имате високо съотношение между броя на операции четене и запис, може да използвате шаблон “Computed”. За да намалите броя на изчисленията може да добавите времеви печат (timestamp) към документа, който да показва кога е бил последно актуализиран. След това, приложението може да определи кога трябва да се извърши изчислението. Друг вариант би могъл да бъде опашка от изчисления, които трябва да се извършат. Изборът на стратегията за актуализация е най-добре да се остави на разработчика на приложението.

Плюсове

- Намаляване на натоварването на процесора при чести изчисления.
- Заявките стават по-прости за писане и като цяло са по-бързи.

Минуси

- Прилагането или прекомерното използване на шаблона трябва да се избягва, освен ако не е необходимо.

5. Шаблон “Document Versioning”

Този шаблон се използва когато трябва да се запазят старите версии на някои документи, вместо да се използва втора система за управление. За да постигнем това, добавяме поле към всеки документ, което ни позволява да следим версията на документа. След това базата данни ще има две колекции: една, която съдържа най-новите (и най-търсените данни), и друга, която съдържа всички стари версии на документите. Типични примери за услуги при които трябва да се запазват старите версии на документи са тези, свързани с: финанси, здравна, застрахователната и правна системи. Трябва да се има предвид,че някои NoSQL бази данни като CouchDB например поддържат автоматично версиите на документите, тъй като това се използва при репликация на данните.

Плюсове

- Лесен за внедряване шаблон.
- Не оказва влияние върху производителността на заявките към колекцията с последната версия на документите.

Минуси

- Удвоява броя на записите.
- Заявките трябва да са насочени към правилната колекция.

6. Шаблон “Extended Reference”

Шаблонът “Extended Reference” се използва, за да намали броя на операции JOIN при често достъпвани данни от различни колекции. За да ускорим времето за изпълнение на заявките трябва да използваме влагане на поддокумент в базов документ. Вграждането на цялата информация е един документ ще намали броя на операции JOIN, но ще доведе до много дублирана информация. При този шаблон вместо да вграждаме цялата информация или да включваме препратка за свързване на информацията, вграждаме само полетата с най-висок приоритет и тези които най-често се достъпват. По този начин се подобрява производителността на услугата.

Плюсове

- Подобрява производителността, когато са необходими много операции JOIN.
- По-бързо четене и намаляване на общия брой на JOIN операциите.

Недостатъци

- Дублиране на данни.

7. Шаблон “Outlier”

Нека да е необходимо една социална мрежа да пази идентификаторите на всички клиенти, които са ви последователи. Ако имате няколко стотин последователи, техните идентификатори могат да се стойностите на поле “ followers” от тип масив. Ако броя на последователите ви надмине някакъв праг, например 1000, това решение не е приемливо. В този случай се казва, че имаме неограничено разрастващ се масив. Шаблон “outlier” решава проблема като първите 1000 последователи се вграждат в основния документ като масив. Добавя се ново поле, което показва дали имате още последователи, описани в друг документ, например поле “more_followers” което приема стойност true или false. При стойност на това поле true ще се знае, че при този документ има отклонение (outlier) от базовата схема за описание на клиентите. Остатъкът от вашите последователи се премества в документ от друга колекция, като за връзка между документите се използва идентификатора на потребителя.

Плюсове

- Запитванията са пригодени за базовата схема на документите, но отклоненията все пак са налични.

Минуси

- Изисква се писане на програмен код за реализация на шаблона.

8. Шаблон “Pre-allocated”

Когато познавате структурата на документа и приложението ви просто трябва да го запълни с данни, моделът за предварително разпределение е правилният избор. На този етап няма голям практически смисъл, тъй като при MongoDB 3.2 се използва двигателя WiredTiger с цел съхранение на данните.

Плюсове

- Опростяване на дизайна, когато структурата на документа е предварително известна.

Недостатъци

- Опростеност срещу производителност.

9. Шаблон Polymorphic

Когато всички документи в една колекция са с подобна, но не идентична структура, наричаме това полиморфен модел. Групирането на документи заедно въз основа на заявките, които искаме да изпълним, помага за подобряване на производителността. Нека да взема за пример система, която трябва да обработва данните на спортисти. Всеки обект-спортист трябва да получава общите за всички спортисти свойства (полета), но и да съдържа и специфичните за неговия спорт характеристики. Шаблонът е лесно приложим ако сте изучавали обектно-ориентирано програмиране.

Плюсове

- Лесен за изпълнение.
- Запитванията могат да се изпълняват в една колекция.

10. Шаблон “Schema Versioning”

Този шаблон се използва ако често се налагат промени в схемата на данните. Възможно е за една част от документите в една колекция да се приложи една схема, а за други – друга. Този модел позволява предишни и текущи версии на документи да съществуват в една колекция. За целта е необходимо да се вмъкне поле “schema_version” със стойност версията на схемата, която трябва да се използва. Стойността на полето е необходима, за да може програмното осигуряване да се адаптира към полетата на документите (а не да ги проверява дали съществуват или не).

Плюсове

- Контрол на миграцията на схемата.

Недостатъци

- Може да са необходими два индекса за едно и също поле по време на миграцията.

11. Шаблон “Subset”

Шаблонът “Subset” се използва с цел оптимизиране на използването на паметта при работа с големи документи, основната част от данните от които приложението не използва в даден момент. Типичен пример за това са отзивите за всеки продукт, който се продава от даден електронен магазин. Вместо да запишем всички отзиви в масив от JSON обекти при шаблон “Subset” в документа на продукта се пазят само последните N отзива или подбрани N отзива на клиенти. Останалите отзиви се записват в отделен документ. Връзката на документите ще се реализира чрез идентификационния номер на продукта.

Плюсове

- По-кратко време за достъп до диска за най-често използваните данни.

Недостатъци

- Изтеглянето на допълнителни данни изисква допълнителни заявки до базата.

12. Шаблон “Tree”

Когато данните са с йерархична структура и често се търсят, шаблонът "Tree" е много подходящ за прилагане. Например, всеки продукт, който се продава в даден електронен магазин принадлежи на конкретна категория, която от своя страна е част от друга категория и така докато стигнем до върха (root) в тази йерархия. За да може лесно да се проследи йерархията на категориите, те най-често се записват в масив. Ако сте избрали да

разгледате мъжко сако, то в документа, който го описва, трябва да има поле, съдържащо йерархията от категории на които сакото принадлежи, например:

```
{
     _id: <ObjectId>,
     prod_code: "2YQS528U8_252",
    name: "BENETTON BLAZER IN LINEN BLEND",
    price: [
       value: "269,90",
       currency: "BGN"
    ],
    basic_features: [
    ...
    ],
    composition: "Outside 51% Linen 49% Cotton Lining 100% Cotton",
    clothing_care: "Machine washable at a maximum temperature of 30°C.",
    parent_category: "BLAZERS AND JACKETS",
     categories: [
       "BLAZERS AND JACKETS",
       "JACKETS",
       "JACKETS AND COATS",
       "MEN"
    ]
}
```

Фиг.4.6 Пример за използване на шаблон “Tree”

Плюсове

- Повишена производителност чрез избягване на множество операции JOIN.

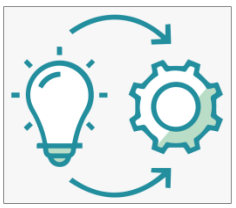
Недостатъци

- Актуализациите на йерархията на категориите трябва да се управлява от страна на приложението.

1.5 MongoDB Atlas - предложения за схеми

MongoDB Atlas предоставя възможност за автоматични персонализирани препоръки за вашите схеми. Тези предложения дават възможност да правите избори, които оптимизират модела на данни. Atlas може да сканира вашата схема за често срещани антимодели и да дава препоръки в реално време, които са в съответствие с най-добрите практики. Предложенията за схеми са групирани по общи антимодели, за които е известно, че влияят на производителността на MongoDB базите данни.

II. Задачи за изпълнение



Задача 1: *Трябва да проектирате база данни, която описва студенти чрез следните свойства: идентификатор на студента (факултетен номер), име и фамилия, както и оценките по дисциплините от последния семестър. Да се има предвид, че трябва да могат да се правят чести заявки, свързани с оценките на студентите.*

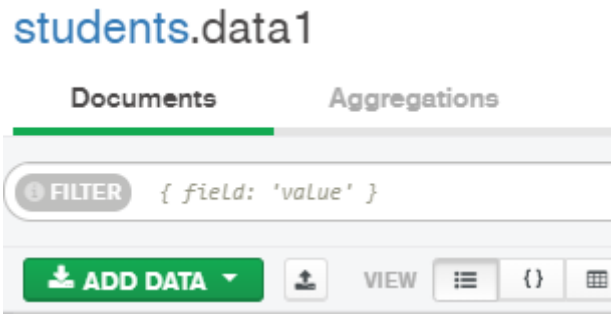
Тъй като по условие на задачата трябва да се правят чести заявки, свързани с оценките на студентите през един семестър, то тази информация трябва да се вгради в документа, описващ студент. На Фиг. 4.7 е показана примерната структура на документа.

```
{
   "id": "21705001",
   "name": {
       "firstName": "Георги",
       "lastName": "Котев"
   },
   "grade": [
       {
           "subject": "НБД", "value": 3
       },
       {
           "subject": "КМС", "value": 6
       },
       {
           "subject": "ММС", "value": 2
       },
       {
           "subject": "ДСП", "value": 5
       }
   ]
}
```

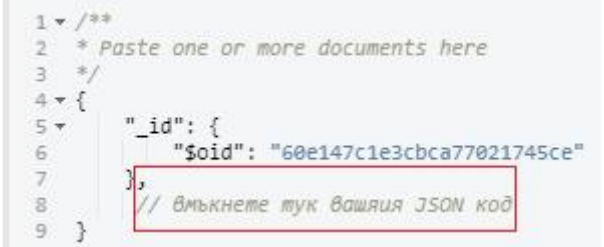
Фиг.4.7 Примерно съдържание на документ от колекцията описваща студентите

Ще работим с графичния интерфейс, който MongoDB Compass предоставя. Ще използваме вече създадената база данни “students”, която трябва да има една колекция със същото име. Чрез бутон <+> създайте нова колекция с име “data1”:

След като създадете колекцията трябва да я населите с данни. Това се реализира чрез бутон <ADD DATA>:



Ако имате съдържанието на документите в JSON или CSV файл, можете да ги импортирате чрез команда “Import File”. На този етап ще въведем три документа чрез редактора на Compass. За целта изберете вмъкване на документ – “Insert Document”. Редакторът ще генерира нов документ за вашата колекция в който има само поле “_id”.

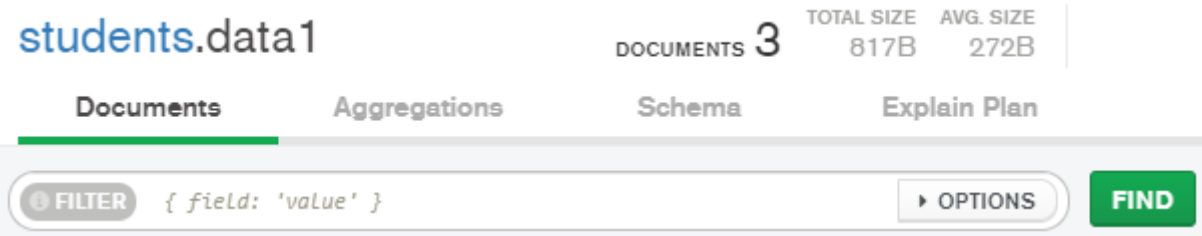


Веднага след него (не забравяйте символ запетая, който разделя полетата) въведете останалото съдържание на документа. След като въведете съдържанието на текущия документа, натиснете бутон <INSERT>. По аналогичен начин въведете още поне два документа в колекция “data1”.

Задача 2: *Трябва да генерирате заявка към база данни “students”, колекция “data1” която да връща имената на всички студенти, които имат:*

- *Отлична оценка по дисциплината НБД.*
- *Оценка в интервала [4, 6] по дисциплината НБД.*

Приложението Compass позволява генерирането на произволни по сложност заявки към документите от избрана колекция. За целта изберете колекция “students.data1”, а от хоризонталното меню - “Documents”:



Ще използваме заявка с филтриране на резултата, за да изпълним задачата. За целта правилото за филтриране трябва да се въведе в поле “FILTER”. Имате възможност да използвате операторите на MongoDB, свързани с филтриране. Те започват със знака за долар, например: $eq, $ne, $gt. $gte, $lt, $lte, $or, $nor, $and, $not, $exists, $in, $regex, $elemMatch. Използвайте документацията и опишете предназначението на всеки един от тези оператори. Както е показано на фигурата базовия формат на филтъра е следния

```
{"field": "value"}
```

Това означава, че полето с име “field” трябва да има стойност “value”. Нека да разгледаме първото условие на задачата - да намерим всички студенти, които имат оценка 6 по дисциплината НБД. В този случай трябва да филтрираме стойностите на две полета от масив “grade”: име на дисциплината – поле “subject” и стойността на оценката – поле “value”. Нека да пробваме със следния филтър:

```
{"grade.subject":"НБД", "grade.value":6}
```

След изпълнение на заявката ще получим списък на студентите, които са изучавали дисциплина “НБД” и имат отлична оценка, независимо по коя дисциплина. Следователно, тази заявка не изпълнява условието на задачата. Трябва да зададем, че търсим специфична двойка стойности за дисциплина и оценка, например:

```
{"grade":{"subject": "НБД","value": 6}}
```

Тази заявка ще върне всички документи, за които двойката “subject-value” има зададените чрез филтъра стойности. За да получим само имената на студентите, а не целия документ, ще филтрираме част от полетата от резултата (документа), който MongoDB връща. За целта натиснете бутон “OPTIONS” и въведете следния низ в поле “PROJECT”:

```
{"_id":0, "name":1}
```

Всяко поле, което има стойност 1 ще бъде включено в резултата. Тъй като системното поле “_id” по подразбиране се включва във всеки резултат, за да го премахнем за него задаваме стойност 0.

Нека да реализираме второто условие на задачата. За да получим студентите, които имат оценки между 3 и 5 по дисциплината НБД ще използваме следния филтър:

{"grade":{ "subject": "НБД" , "value": { $gte: 3, $lte: 5 }}}

Интервалът за оценката е зададен с оператори $gte (≥) и $lte (≤). Въпреки, че синтаксисът изглежда логичен, той няма да върне желания резултат. Причината за това е, че тук имаме заявка с няколко (две) условия, приложена за вложени полета. В този случай трябва да се използва оператор $elemMatch, за да зададете множество критерии за масив от вградени документи, така че поне един вграден документ да отговаря на всички зададени критерии. Следователно филтърът трябва да бъде:

{"grade":{$elemMatch: { "subject": "НБД", "value": { $gte: 3, $lte: 5 } }}}

Задача 3: *Трябва да генерирате заявка към база данни “students”, колекция “data1” която да връща имената на всички студенти, които имат оценка със стойност a или b по поне една от дисциплините НБД и КМС.*

Използвайте оператор $in чрез който да зададете от кои дисциплини се интересувате, както и кои оценка представляват интерес:

{"grade":{$elemMatch:{"subject":{$in:["НБД","КМС"]}, "value": {$in:[5, 6] }}}}

Задача 4: *Трябва да генерирате заявка към база данни “students”, колекция “data1” която да връща имената на всички студенти, които имат оценка по-висока 2 за дисциплината НБД и оценка 4 или 5 за дисциплината КМС.*

Трябва да използвате два пъти оператор $elemMatch и да обедините тези условия чрез оператор $and:

```
{
    "grade": {
      $all:
         [
            {
               $elemMatch:{"subject":"НБД", "value": {$gt:2}}
            },
            {
               $elemMatch:{"subject":"КМС", "value": {$in:[4,5]}}
            }
         ]
    }
}
```

Задача 5: *Оптимизирайте схемата на база данни “students”, колекция “data1” така, че да се намали размера на колекцията и да се повиши бързодействието на заявките.*

Ако разгледаме съдържанието на документ от колекция “students.data1” ще забележим, че описанието на оценките не е оптимално. Всяка дисциплина се описва от поле “subject”, а името на дисциплината е стойността на това поле. Можем да зададем името на дисциплините да е ключ, а не стойност. По този начин размерът на колекцията ще намалее, а заявките, свързани с име на дисциплина ще се опростят и ще връщат резултат по бър-

зо. На Фиг. 4.8 е показано новото съдържание на документа. Създайте нова колекция с име “data2” в базата данни “students” като спазвате новата схема.

```
{
    "id": "21705001",
    "name": {
       "firstName": "Георги",
       "lastName": "Котев"
    },
    "grade": {
       "НБД": { "value": 3 },
       "КМС": { "value": 6 },
       "ММС": { "value": 2 },
       "ДСП": { "value": 5 }
    }
}
```

Фиг.4.8 Ново съдържание на документ от колекцията описваща студентите

В Табл. 4.2 са дадени заявките от примерите от Упражнение 4 за всяка една от предложените схеми. Вижда се, че заявките към документите от колекция “data2” са много по кратки и разбираеми. Освен това те се изпълняват и по-бързо.

Табл. 4.2 Сравнение на заявките при използване на схемата от колекция “data1” и “data2”

Колекция “data1”	Колекция “data2”
{"grade":{"subject": "НБД","value": 6}}	{"grade.НБД.value":6}
{"grade":{$elemMatch:{"subject":"НБД","value":{$gte:3,$lte:5}}}}	{"grade.НБД.value":{$gte: 3, $lte: 5}}
{"grade":{$elemMatch:{"subject":{$in:["НБД","КМС"]}, "value": {$in:[5,6] }}}}	{$or:[{"grade.НБД.value": {$in:[5,6]}},{"grade.КМС.value": {$in:[5,6]}}]}

{
"grade": {
$all:
[
{
$elemMatch:{"subject":"НБД", "value": {$gt:2}}
},
{
$elemMatch:{"subject":"КМС", "value": {$in:[4,5]}}
}
]
}
}
	{$and:[{"grade.НБД.value":{$gt:2}},{"grade.КМС.value": {$in:[4,5]}}]}

Задача 6: *Напишете заявката, която променя оценката на студент за дадена дисциплина от база данни “students”, колекция “data1” и колекция “data2”.*

Промяната на стойностите на полета от документ се реализира чрез метод **update**. Синтаксисът на този метод е следния:

```
db.collection.update(query, update, options)
```

където “query” задава критериите за избор на обновяване. Налични са същите селектори за заявка като при метода find; “update” - промените, които трябва да се приложат.

Метод “update” променя съществуващ документ или документи в колекция. Може да се модифицират определени полета на съществуващ документ или документи или да се замени изцяло съществуващ документ в зависимост от стройността на параметър “update”. По подразбиране методът актуализира един документ. Чрез опция “multi”: true, може да актуализирате всички документи, които отговарят на критериите на заявката.

Приложението MongoDB Compass позволява изпълнение на конзолни команди чрез прозореца MONGOSH (Momgo Shell):

Промяната на стойност на едно или няколко полета или създаване на нови полета, без това да се отразява на стойностите на останалите полета се реализира чрез оператор $set от параметър “update”.

Нека да променим оценката на студент с id=”21705003” по дисциплината НБД. Методът update реализира промяна, само ако има разлика в старата и новата стойности на полето, което се променя. Следва пример как се реализира промяната за студентите от колекция “**data1**”:

```
> db.data1.updateOne({"id":"21705003","grade.subject":"НБД"}, {$set:{"grade.$.value":3}})
< { acknowledged: true,
    insertedId: null,
    matchedCount: 1,
    modifiedCount: 1,
    upsertedCount: 0 }
>
```

Стойността на параметър “query” е {“id”:”21705003”,”grade.subject.НБД”}, за да се избере желания студент и желаната дисциплина. Като стойност на параметър “update” е указано, че само се модифицира едно поле, а не целия документ – оператор $set. Стойността за $set е стойността на поле “value”. Тъй като “grade” е масив трябва да се използва оператора за позициониране **$**. Този оператор идентифицира елемент в масив, който да се актуализира, без да се посочва изрично позицията на елемента в масива. За да актуализирате всички елементи в масив, използвайте оператор $[].

Промяната на стойността на поле от колекция “**data2**” е значително по опростено и се изпълнява по-бързо:

```
>_MONGOSH BETA
> use students
< 'switched to db students'
> db.data2.updateOne({"id":"21705003"}, {$set:{"grade.НБД.value":3}})
< { acknowledged: true,
    insertedId: null,
    matchedCount: 1,
    modifiedCount: 1,
    upsertedCount: 0 }
>
```

Може да се направи извода, че схемата е от съществено значение за дизайна на една база данни. Схемата от колекция "data2" е по-добра от тази за колекция "data1" защото: 1) Намалява размера на базата данни; 2) Опростява и ускорява изпълнението на заявките към базата, включително и тези за обновяване на съдържанието на документите.

III. Задачи за самостоятелна работа

Задача Д1: *Проектирайте база данни, която описва студенти и техните оценки за всеки един семестър. Преценете дали трябва да използвате модел с вмъкване на резултатите от изпитите за всички семестри или да използвате свързване на документите, описващи студентите и документите, описващи оценките.*

Упражнение № 5

Достъп до MongoDB с Java – инсталации и търсене на съдържание

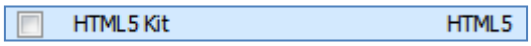
I. Въведение

Разработването на приложения, които комуникират с MongoDB сървъра, е много улеснено. Това е възможно благодарение на стандартизирания интерфейс който драйверите предоставят. Налични са драйвери за множество програмни езици, например Java, Node.js (JavaScript), C#, Python, Rubby, Swift и др. В това упражнение ще научим как да пишем приложения на Java чрез използване на Java MongoDB Driver. Трябва да се има предвид, че комуникацията с MongoDB Atlas изисква JDK версия не по-ниска от 117. Това е необходимо, тъй като предходните версии на JDK нямат налични в хранилището със сертификати тези с които Atlas работи. За развойна среда ще използваме IDE Netbeans версия 8.2, а приложенията ще реализираме чрез Java 8. Минималните знания, които са необходими, са следните:

- Работа с Java класове-колекции.
- Потокова (streaming) обработка на данни.
- Функционално програмиране (ламбда функции).
- Обработка на събития (try-catch-finally)

II. Необходими инсталации

Започнете с инсталация на развойната среда Netbeans 8.2. Необходимо е да изберете инсталатора, който предоставя разработка на приложения чрез Java и HTML5 / JavaScript. Ако сте инсталирали вариант при който няма поддръжка на HTML5, то това може да се реализира чрез инсталиране на модула като плъгин. За целта от менюто на Netbeabs изберете Tools и след това Plugins. Изберете “Available Plugins” и в поле Search въведете HTML5. Маркирайте “HTML5 Kit” и инсталирайте чрез натискане на бутон <Install>.



По този начин ще можете да разработвате и приложения за достъп до MongoDB и чрез Node.js. Плъгинът ще обнови версията на JDK до 239.

Възможни са два сценария на разработка на Java приложения за достъп до MongoDB сървъра, независимо дали той е локален или в облака:

- Свалете всички необходими библиотеки и ги вмъкнете във вашия проект – папка Libraries.
- Използвайте Maven с цел автоматично сваляне и интегриране на библиотеките към вашия проект.

Библиотеките (JAR файл), които ще използваме, са следните:

- MongoDB Java Driver.
- Библиотека за работа с JSON обекти.

2.1 MongoDB Java драйвер

Трябва да имате предвид, че има множество версии на MongoDB Java драйвера. Използвайте по възможност последната стабилна версия на драйвера. Използвайте хранилището MVNrepository.com, за да свалите драйвера:

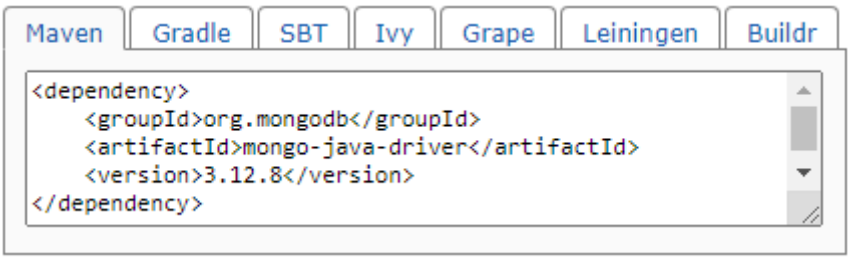
https://mvnrepository.com/artifact/org.mongodb/mongo-java-driver

На Фиг. 5.1 е показана формата, която се визуализира при избор на версия 3.12.8 на драйвера.

Фиг.5.1 Сваляна на MongoDB Java Driver

Щракнете върху препратката “jar”, за да свалите драйвера. Запишете драйвера в папка, например “MongoDBLibraries”.

Ако искате да инсталирате драйвера автоматично, трябва да създадете Maven проект и да интегрирате следния код в конфигурационния файл pom.xml:



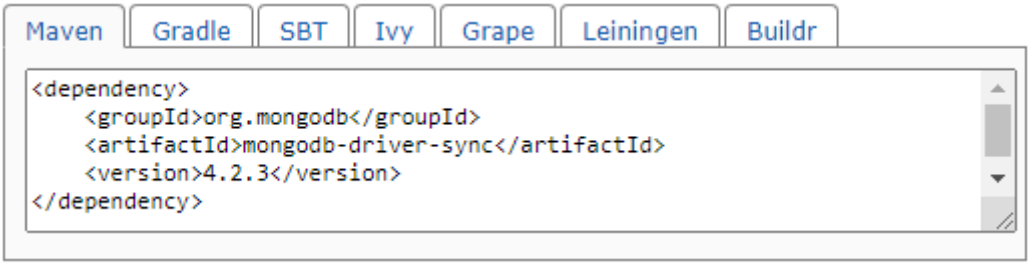
Фиг.5.2 Maven код за сваляне на MongoDB Java Driver 3.12.8

Препоръчително е да работите със синхронната версия на MongoDB Java драйвера - mongodb-driver-sync. Трябва да имате предвид, че има разлика в класовете на старите и новите драйвери. Това означава, че ако вече сте писали приложение за стара версия на драйвера, ще се наложи да направите корекции във вашия код при смяна на драйвера.

Синхронната версия на MongoDB Java Driver може да свалите от следното хранилище:

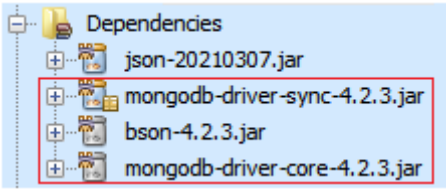
https://mvnrepository.com/artifact/org.mongodb/mongodb-driver-sync

Изберете последната стабилна версия, например 4.2.3. Свалете драйвера и го запишете в папка “MongoDBLibraries”. Копирайте и Maven кода, за автоматично сваляне на този драйвер:



Фиг.5.4 Maven код за сваляне на синхронния MongoDB Java Driver версия 4.2.3

Ако работите с Maven проект, автоматично ще бъдат свалени всички библиотеки (заградени в червено), необходими за функциониране на драйвера:



На практика свалянето се активира след като изберете с мишката името на проекта и чрез десния бутон на мишката изберете “Build with Dependencies”.

Ако не използвате Maven, трябва да свалите:

- mongodb-driver-core-4.2.3.jar
- bson-4.2.3.jar

Тези библиотеки съдържат ядрото на драйвера и класове за работа с BSON обекти. Класовете, които изграждат ядрото на MongoDB Java Driver може да свалите от следното хранилище:

https://mvnrepository.com/artifact/org.mongodb/mongodb-driver-core

Класовете, които позволяват работа с BSON обекти може да свалите от следното хранилище:

https://mvnrepository.com/artifact/org.mongodb/bson

Внимание: Версията и за трите библиотеки трябва да съвпадат!

Винаги използвайте Maven, за да създадете вашия проект, за да е сигурно свалянето на всички необходими библиотеки и с цел бърза смяна на версията на драйвера.

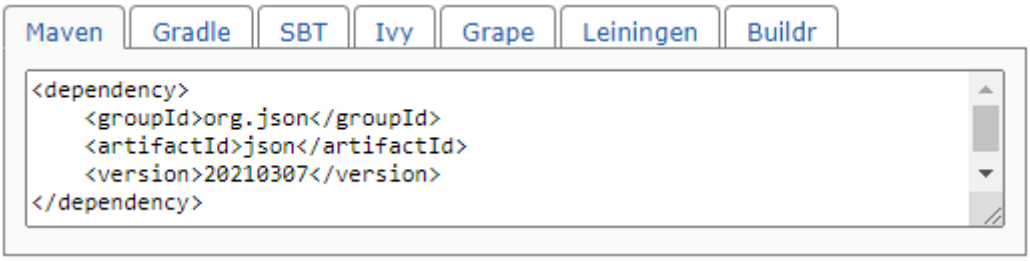
2.2 JSON coder/decoder

За да може да обработвате данните, които MongoDB сървъра връща като отговор на вашите заявки, ще ви трябва библиотека за работа с JSON обекти. Тъй като такава библиотека не е интегрирана в Java 8, ще използваме библиотека, която може да свалите от следния адрес:

https://mvnrepository.com/artifact/org.json/json

Изберете последната налична версия, например 20210307. Щракнете върху препратката “bundle”, за да свалите библиотеката. Запишете файла в папка “MongoDBLibraries”.

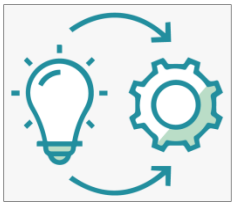
Ако искате да инсталирате библиотеката автоматично, трябва да интегрирате следния Maven код в конфигурационния файл pom.xml:



Фиг.5.3 Maven код за сваляне на библиотека JSON версия 20210307

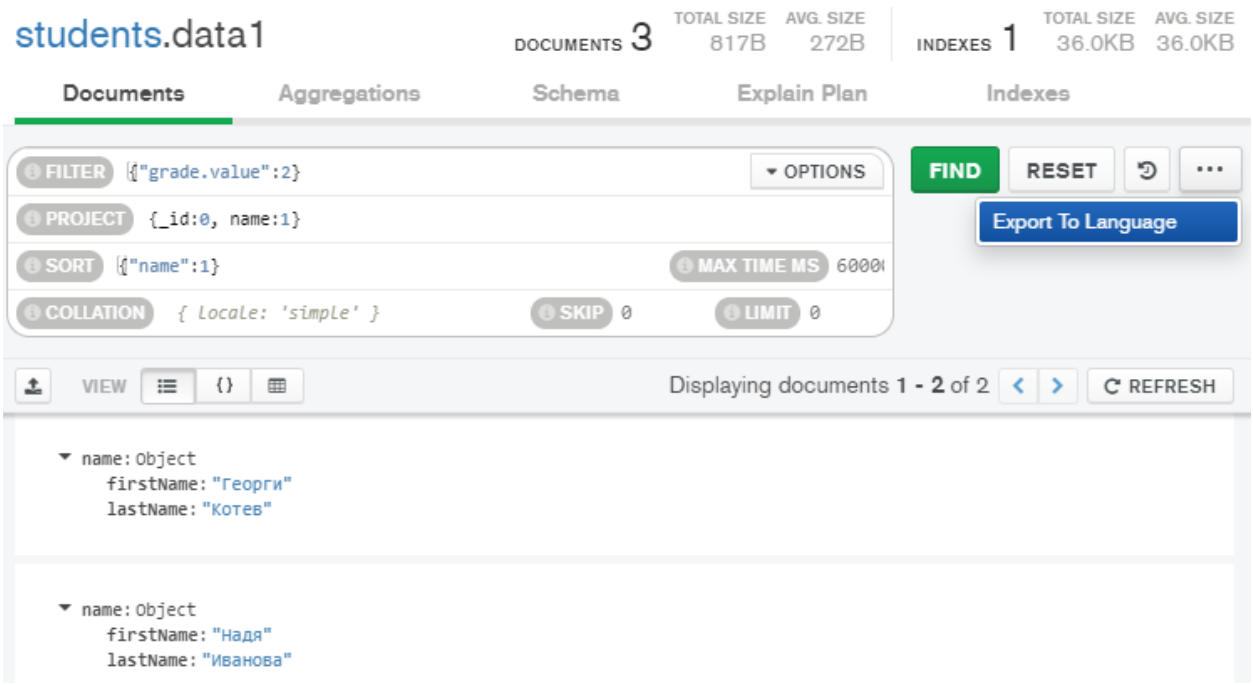
III. Задачи за изпълнение

Всички задачи от това упражнение са свързани със генериране на заявки към MongoDB бази данни и форматиране на отговора, който сървъра връща. Ще използваме приложението MongoDB Compass с цел тестване на заявките и конвертирането им до Java код. Тъй като Compass генерира код за по-старите версии на драйвера на този етап ще работим с версия 3.12.8 на драйвера.



Задача 1: *Напишете Java конзолно приложение, което връща имената на всички студенти, които имат поне една слаба оценка. Списъкът с имена да е във възходящ азбучен ред. Използвайте вече създадената база данни "students", колекция "data1".*

Ще използваме приложението MongoDB Compass, за да реализираме задачата. За целта от хоризонталното меню изберете "Documents", а след това "Options" (виж Фиг. 5. 4).



Фиг.5.4 Реализация на Задача 1 чрез използване на MongoDB Compass

В поле "FILTER" ще въведем кода, който търси всички слаби оценки:

{"grade.value": 2}

Тъй като се интересуваме само от имената на студентите, които имат двойки, от поле “PROJECT” задайте кои полета искате да формират отговора. Премахнете поле “_id” и вмъкнете поле “name”:

{ _id: 0, name:1 }

За да сортираме имената на студентите (от А към Я) ще използваме следния код от поле “SORT”:

{ name:1 }

Стойност 1 означава, че сортирането е възходящо. При стойност -1 сортирането е низходящо.

Когато се уверите, че въведения код работи коректно натиснете бутон <…> и изберете “Export To Language”. Чрез падащото меню “Export Query To:” изберете Java. Изберете всичко от чек боксовете:

- “Include Import Statements” - включва кода за импортиране на всички необходими класове и интерфейси.
- “Include Driver Syntax” - включва кода за връзка с базата данни чрез драйвера.
- “Use Builders” - включва кода за симулиране на filter, project и sort.

Създайте нов проект, Java конзолно приложение:

“File” -> “New Project …” -> Java -> “Java Application”

Нека да именуваме проекта MONGODB_EX_5_1. След като бъде създаден проекта, щракнете с десния бутон на мишката върху папка “Libraries”. Изберете “Add JAR/Folder…” и прехвърлете Java драйвера “mongo-java-driver-3.12.8.jar” от папка “MongoDBLibraries”. По аналогичен начин прехвърлете към проекта и библиотека “json-20210307.jar”.

Копирайте кода за импортиране на всички класове и интерфейси и го вмъкнете след package и преди базовия клас.

Вмъкнете в тялото на метод **main** програмния код за комуникация със сървъра, който приложението Compass генерира (виж Фиг. 5.5).

```
20  public class MONGODB_EX_5_1 {
21      public static void main(String[] args) {
22          Bson filter = eq("grade.value", 2L);
23          Bson project = and(eq("_id", 0L), eq("name", 1L));
24          Bson sort = eq("name", 1L);
25
26          MongoClient mongoClient = new MongoClient(
27                  new MongoClientURI(
28                          "mongodb://localhost:27017"
29                  )
30          );
31          MongoDatabase database = mongoClient.getDatabase("students");
32          MongoCollection<Document> collection = database.getCollection("data1");
33          FindIterable<Document> result = collection.find(filter)
34                  .projection(project)
35                  .sort(sort);
36      }
37  }
```

Фиг.5.5 Програмен код за комуникация с базата данни (метод main)

Създадени са три обекта от клас Bson чрез които се описва какъв е филтъра, проектора и правилата за сортиране (редове 22, 23 и 24). Използвани са статични методи **eq** и **and** от “com.mongodb.client.model.Filters”.

Започваме със създаване на обект “mongoClient”, който описва клиента. Чрез него ще реализира достъп до сървъра (редове 26-29). За целта са използвани конструкторите на класове MongoClientURI и MongoClient.

Следва свързване с желаната база данни чрез метод **getDatabase** (ред 31). Чрез обект database и метод **getCollection** получаваме обект за достъп до желаната колекция (ред 32). Изпращането на заявката към сървъра се реализира чрез метод **find** (ред 33). На този метод се подава един аргумент – Bson обект, който описва филтъра. Следва задаване кои точно полета да останат като краен резултат. Това се реализира на ред 34 - метод **projection**.. И накрая, задаваме правилата за сортиране чрез метод **sort** (ред 35). Резултатът, който се получава е колекция от документи (интерфейс FindIterable).

Остава ни да обработим документите в обекта “result”. За целта ще използваме библиотека JSON. Програмният код, който реализира необходимото парсване на резултата, е показан на Фиг. 5.6.

```
36          for (Document document : result) {
37              String json = document.toJson();
38              JSONObject obj = new JSONObject(json);
39              JSONObject nameObj = obj.getJSONObject("name");
40              String firstName = nameObj.getString("firstName");
41              String lastName = nameObj.getString("lastName");
42              System.out.printf("%s %s\n", firstName, lastName);
43          }
```

Фиг.5.6 Програмен код за парсване на резултата

За всеки документ в обекта “result” реализираме следната последователност от действия:

- Конвертираме документа до JSON обект (ред 37).
- Извличаме съдържанието на поле “name”. Тъй като полето е обект, използваме метод **getJSONObject** (ред 39).
- Чрез метод **getString** получаваме името и фамилията на студента (редове 40 и 41).
- Печатаме ма конзолата имената на студента (ред 42).

Стартирайте приложението. Ако нямата грешки, трябва да получите списък с имената на студентите, които имат поне една двойка:

```
Весела Иванова
Георги Котев
BUILD SUCCESSFUL (total time: 1 second)
```

Задача 2: *Напишете Java конзолно приложение, което връща имената на всички студенти и техния среден успех от всички изучавани дисциплини. Използвайте база данни “students”, колекция “data1”.*

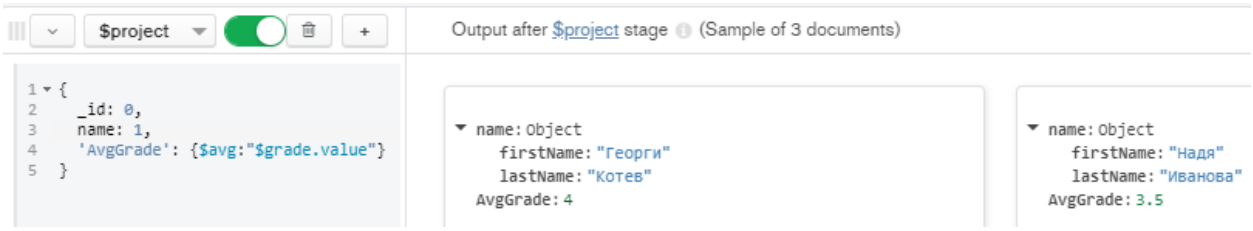
При MongoDB има възможност за агрегиране на съдържание чрез множество оператори. Обединяването на съдържание се реализира чрез метод **aggregate** от ниво приложение. За целта от MongoDB Compass трябва да изберете от хоризонталното меню “Aggregations”. При агрегиране може да свързвате последователно множество оператори

за агрегиране, всеки от които започва със знака за долар $. Всеки от тези оператори формира фаза от конвейера за агрегиране. Най-често използваните от тях са следните:

- $addFields – променя стойността на поле или вмъква нови полета в документа и задава тяхната стойност чрез изчислителен израз.
- $count – връща броя на документите за текущото ниво на агрегиране.
- $group – групира документи от една колекция по зададени признаци.
- $limit – ограничава броя на документите, които се предават към следващия оператор за агрегиране.
- $lookup – реализира обединение на документите от две колекции.
- $match – филтрира документите, които достигат до следващата фаза на агрегиране.
- $project – дава възможност за премахване или включване на съществуващи или нови полета, които да достигнат до следващата фаза на агрегиране.
- $skip – пропускане на определен брой документи преди преминаване към следваща фаза на агрегиране.
- $sort – позволява сортиране на документите по зададен ключ и подреждане на стойностите на полетата (възходящо или низходящо).
- $sortByCount – групира документите по зададено условие, а след това изчислява броя на документите във всека от така получените групи.
- $unwind – връща към следваща фаза на агрегиране по един документ за всеки елемент от зададен масив.

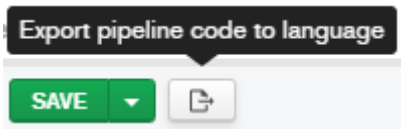
За целите на статистическата обработка на данните MongoDB поддържа оператори $max, $min и $avg, които могат да участват при изчисляване на стойността на полета. За да решим конкретната задача ще използваме оператор $avg, който изчислява средна стойност. Ще приложим оператора за стойностите на оценките, поле “grade.value”. Ще създадем ново поле в изходния документ с име “AvgGrade”, което ще съдържа желаната информация. Към крайния резултат трябва да включим поле “name” и новото поле “AvgGrade”. Тази функционалност е възможно да се получи само с оператор $project.

Първо, ще реализираме задачата чрез MongoDB Compass, а след това и като Java приложение. От Compass изберете работа с база “students” и колекция “data1”. От хоризонталното меню изберете “Aggregations”. В конкретния случай конвейера за агрегиране ще има само една фаза в която ще използваме оператор $project. За целта от падащото меню “Select…” изберете точно този оператор (виж Фиг. 5. 7).



Фиг.5.7 Агрегиране на съдържание чрез оператор $avg

Можете да получите програмния код, който реализира желаното агрегиране програмно. За целта натиснете бутон <Export pipeline code to language>:

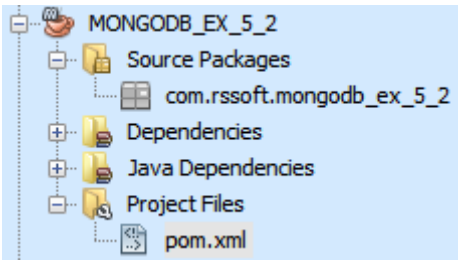


Изберете за програмен език Java (“Export Pipeline To:”). Разрешете включването на всички необходими класове, както и програмния код за комуникация с Java драйвера.

Създайте нов проект, Java конзолно приложение чрез използване на Maven:

“File” -> “New Project …” -> Maven -> “Java Application”

Нека да именуваме проекта MONGODB_EX_5_2:

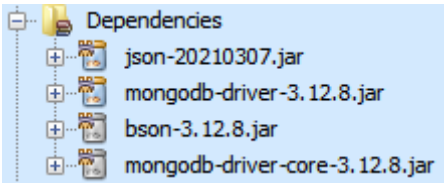


Отворете конфигурационния файл и включете XML кода чрез който се задава кои библиотеки трябва да бъдат свалени и интегриране към проекта:

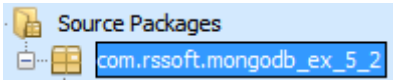
```
<dependencies>
   <dependency>
      <groupId>org.mongodb</groupId>
      <artifactId>mongodb-driver</artifactId>
      <version>3.12.8</version>
   </dependency>

   <dependency>
      <groupId>org.json</groupId>
      <artifactId>json</artifactId>
      <version>20210307</version>
   </dependency>
</dependencies>
```

Фиг.5.8 Код от конфигурационния файл pom.xml

Свалените библиотеки ще бъдат прехвърлени в проектната папка “Dependencies”:



Следва да създадете Java файл, който да съдържа кода, генериран от приложението Compass. За целта, с десният бутон на мишката щракнете върху пакета от папка “Source Packages”:



Изберете New -> “Java Class…”. Задайте име на Java файла, който ще съдържа програмния клас в поле “Class Name”. Името може да е произволно, например MONGODB_EX_5_2. Натиснете бутон “Finish”. Въведете програмния код в този файл. Прехвърлете кода за импортиране на необходимите класове и интерфейси веднага след

package. В тялото на класа създайте метод **main** (виж Фиг. 5.9) и въведете програмния код от Compass в тялото на метод main:

```
20      public static void main(String[] args) {
21
22          MongoClient mongoClient = new MongoClient(
23                  new MongoClientURI(
24                          "mongodb://localhost:27017/"
25                  )
26          );
27          MongoDatabase database = mongoClient.getDatabase("students");
28          MongoCollection<Document> collection = database.getCollection("data1");
29
30          AggregateIterable<Document> result = collection
31                  .aggregate(Arrays.asList(new Document("$project",
32                                                  new Document("_id", 0L)
33                  .append("name", 1L)
34                  .append("AvgGrade", new Document("$avg", "$grade.value")))));
35
36          for (Document document : result) {
37              String json = document.toJson();
38              JSONObject obj = new JSONObject(json);
39              JSONObject nameObj = obj.getJSONObject("name");
40              String firstName = nameObj.getString("firstName");
41              String lastName = nameObj.getString("lastName");
42              double avgGrade = obj.getDouble("AvgGrade");
43              System.out.printf("%s %s: %.2f\n", firstName, lastName, avgGrade);
44          }
45      }
46  }
```

Фиг.5.9 Програмен код – метод main

Метод **aggregate** (ред 31) се прилага към обекта, описващ избраната колекция. Атрибутите към метода описват кой оператор се използва и какъв е кода от тялото на този оператор. Тъй като може да има множество оператори при агрегиране, може да използвате метод **append** за обединяване на съдържание, предавано към aggregate. Кодът за парсване на резултата е от ред 36 до ред 44. След стартиране на приложението и избор на базов клас ще получите резултат подобен на следния:

```
Георги Котев: 4.00
Надя Иванова: 3.50
Весела Иванова: 3.50
---------------------
BUILD SUCCESS
---------------------
```

Задача 3: *Напишете Java конзолно приложение, което връща за всеки студент по кои дисциплини има избрана оценка, например Слаб(2). Използвайте база данни "students", колекция "data1".*

За да решим задачата ще трябва да използваме няколко фази на агрегиране:

- **Фаза 1**: Получаване на всички студенти, които имат поне една слаба оценка. За целта ще използваме оператор $match. Ще проверим дали в масив "grade" има поне един елемент за който стойността на поле "value" има стойност 2. Тази проверка налага да използваме оператор $elemMatch. Този оператор съпоставя докумен-

ти, които съдържат поле от масив с поне един елемент, който отговаря на всички зададени критерии на заявката.

"grade": { $elemMatch: { "value": 2 } }

След тази фаза ще получим толкова документи, колкото студенти имат поне една двойка.

- **Фаза 2**: Ще генерираме по един документ за всеки елемент от масив “grade”. За целта ще използваме оператор $unwind:

path: "$grade"

- **Фаза 3**: Ще филтрираме чрез оператор $match получените при Фаза 2 документи и ще оставим само тези от тях за които стойността на поле “value” е 2:

"grade.value": 2

- **Фаза 4**: Ще групираме документите, които се отнасят за един и същи студент. За целта ще използваме оператор $group. Групирането може да стане по някой уникален ключ за студента, например “id” или “name” или и двата заедно. Нека да групираме само по име. Ще направим така, че ако студент има повече от една слаба оценка те да формират масив. За целта ще използваме оператор $addToSet:

_id: "$name", grades: {$addToSet: "$grade"}

Фиг.5.10 Фази на изпълнение на конвейера за агрегиране – Задача 3

Копирайте програмния код, който приложението Compass генерира и създайте нов Maven Java проект, който да реализира Задача 3. Вмъкнете този код (виж Фиг. 5.11) в тялото на метод main. След това трябва да обработите върнатия от сървъра резултат - обект “result” (ред 28). Това може да стане по различни начини. На Фиг. 5.12 е показано едно възможно решение. След изпълнение на приложението трябва да получите резултат подобен на следния:

```
Весела Иванова: ММС КМС
Георги Котев: ДСП
BUILD SUCCESSFUL (total time: 1 second)
```

```
20          MongoClient mongoClient = new MongoClient(
21                  new MongoClientURI(
22                          "mongodb://localhost:27017"
23                  )
24          );
25          MongoDatabase database = mongoClient.getDatabase("students");
26          MongoCollection<Document> collection = database.getCollection("data1");
27
28          AggregateIterable<Document> result
29                  = collection.aggregate(Arrays.asList(
30                      new Document("$match",
31                          new Document("grade",
32                              new Document("$elemMatch",
33                                  new Document("value", GRADE)))),
34                      new Document("$unwind",
35                          new Document("path", "$grade")),
36                      new Document("$match",
37                          new Document("grade.value", GRADE)),
38                      new Document("$group",
39                          new Document("_id", "$name")
40                              .append("grades",
41                                  new Document("$addToSet", "$grade")))));
```

Фиг.5.11 Програмен код за комуникация със сървъра – Задача 3

```
43          for (Document document : result) {
44              String json = document.toJson();
45              JSONObject obj = new JSONObject(json);
46              JSONObject nameObj = obj.getJSONObject("_id");
47
48              String firstName = nameObj.getString("firstName");
49              String lastName = nameObj.getString("lastName");
50              System.out.printf("\n%s %s: ", firstName, lastName);
51
52              JSONArray gradeObj = obj.getJSONArray("grades");
53              Iterator<Object> iterator = gradeObj.iterator();
54              while (iterator.hasNext()) {
55                  JSONObject itemObj = (JSONObject)iterator.next();
56                  String subject = itemObj.getString("subject");
57                  System.out.printf("%s ", subject);
58              }
59          }
```

Фиг.5.12 Програмен код за парсване на резултата – Задача 3

Задача 4: *Напишете Java конзолно приложение, което връща общия брой на двойките, които студентите от колекция " data1" имат.*

Използвайте Задача 3 и оператор $count.

Задача 5: *Напишете Java конзолно приложение, което връща списък на оценките за всички дисциплини на избран по име и фамилия студент. Името на студента да се въвежда от клавиатурата. Използвайте база данни "students", колекция "data1".*

Упражнение № 6

Достъп до MongoDB с Java – CRUD заявки

I. Въведение

Заявките към една база данни могат да бъдат Create, Read, Update и Delete (CRUD). На този етап тествахме възможността за четене от вече създадени бази данни чрез методи **find** и **aggregate**. В това упражнение ще се научим как програмно с Java код може да създаваме бази данни, колекции и документи, както и да обновяваме стойностите на полета и да изтриваме документи.

1.1 Създаване на бази данни

Създаването на една база данни програмно се реализира в следната последователност:

- Задаване на името на базата данни с която ще работим. За целта ще използваме метод **getDatabase**. Той изисква за атрибут името на базата данни. Методът връща обект от клас MongoDatabase. Ако базата с указаното име не съществува, методът я създава.
- Създаване на колекция. Ще използваме метод **getCollection**, който приема като аргумент името на колекцията. Ако колекцията не съществува тя ще бъде създадена. Методът връща обект, колекция от документи: MongoCollection<Document>.
- Вмъкване на необходимите документи в избрана колекция. Вмъкването може да се реализира един по един или множество наведнъж. При първия случай се използва метод **insertOne**, а при втория – метод **insertMany**. Методите изискват като аргумент обект-документ или масив от обекти-документи. Документите може да бъде импортирано от файл или създадени динамично програмно.

1.2 Обновяване на полета

Промяната на стойностите на полета от документ се реализира чрез метод **update**. Синтаксисът на метода е следния:

```
db.collection.update(query, update, options)
```

където:

- “query” задава критериите за избор на обновяване. Налични са същите селектори за заявка като при метода find;
- “update” - промените, които трябва да се приложат към документите, избрани чрез find.
- “options” – параметри чрез които се конкретизира обновяването. Например, при параметър “multi” със стойност “true” обновяването се отнася за всички документи, отговарящи на филтъра зададен чрез find. По подразбиране се обновява съдържанието на един документ.

Чрез метод **update** може да се модифицират определени полета на съществуващ документ или документи или да се замени изцяло съществуващ документ в зависимост от стройността на параметър “update”. По подразбиране методът актуализира един документ. Чрез опция “multi”: true, може да актуализирате всички документи, които отговарят на критериите на заявката.

В Табл. 6.1 са описани основните оператори, които могат да бъдат използвани при обновяване на полета:

Табл. 6.1 Оператори при обновяване

Оператор	Описание
$currentDate	Задава стойността на полето като текуща дата, като дата или времеви печат.
$inc	Увеличава текущата стойността на полето с указана стойност, която може да е различна от 1 (както положителна, така и отрицателна).
$min	Актуализира полето само ако зададената стойност е по-малка от съществуващата стойност на полето.
$max	Актуализира полето само ако зададената стойност е по-голяма от съществуващата стойност на полето.
$mul	Умножава стойността на полето по посочената стойност.
$rename	Преименува поле.
$set	Задава стойността на поле в документ.
$setOnInsert	Задава стойността на поле, ако актуализацията води до вмъкване на документ. Няма ефект върху операции за актуализация, които променят съществуващи документи.
$unset	Премахва посоченото поле от документ.

Внимание: Ако аргумент “update” съдържа само двойка от вида “fieldname:value”, то съдържанието на документа се изтрива и в него се вмъква новото поле и неговата стойност. Ако искате само да обновите стойността на поле, използвайте оператор $set.

Когато се налага обновяване на полета от масив могат да бъдат използвани операторите, описани в Табл. 6.2:

Табл. 6.2 Оператори при обновяване на полета от масиви

Оператор	Описание
$	Действа като заместител за актуализиране на първия елемент, който отговаря на условието на заявката.
$[]	Действа като заместител за актуализиране на всички елементи в масив за документите, които отговарят на условието на заявката.
$[<идентификатор>]	Действа като заместител, за да актуализира всички елементи, които отговарят на условието зададено чрез опция “arrayFilters”, за документите, които отговарят на условието в заявката.
$addToSet	Добавя елементи към масив само ако те вече не съществуват в множеството.
$pop	Премахва първия или последния елемент от масив.
$pull	Премахва всички елементи на масива, които отговарят на зададена заявка.
$pullAll	Премахва всички съвпадащи стойности от масив.
$push	Добавя елемент в масив.

Към някои от описаните оператори могат да бъдат приложени модификатори които променят тяхното действие. Тези модификатори са описани в Табл. 6.3.

Табл. 6.3 Модификатори към оператори за работа с масиви

Модификатор	Описание
$each	Модифицира действието на оператори $push и $addToSet с цел добавяне на множество елементи при актуализиране на масива.
$position	Модифицира оператор $push - указва позицията в масива за добавяне на елементи.
$slice	Модифицира оператор $push - ограничава размера на актуализираните масиви.
$sort	Модифицира оператор $push - сортира документите, съхранявани в масив.

Като трети аргумент към метод update могат да бъдат зададени множество опции. Те са описани в Табл. 6.4.

Табл. 6.4 Опции към метод update

Опция	Описание
upsert (true/false)	Създава нов документ, ако няма документи, които да отговарят на заявката или актуализира един документ, който отговаря на заявката. Ако и двете стойности upsert и multi са true и няма документи, които да отговарят на заявката, операцията update вмъква само един документ. По подразбиране стойността на upsert е false - не се вмъква нов документ, когато не е намерено съвпадение.
multi (true/false)	Ако е зададена стойност true, се актуализират няколко документа, които отговарят на критериите на заявката. Ако е зададена стойност false, се актуализира само един документ. Стойността по подразбиране е false.
writeConcern (document)	От MongoDB 4.4 наборите от реплики и клъстерите с шардове поддържат задаване на глобална грижа за запис по подразбиране. Операциите, които не задават изрично загриженост при запис, наследяват глобалните настройки за значение за запис по подразбиране. Не задавайте изрично загриженост за запис за операции, които се изпълнява в транзакция.
collation (document)	Collation позволява на потребителите да задават специфични за езика правила за сравняване на низове.
arrayFilters (масив)	Масив от документи за филтриране, които определят кои елементи на масива да се модифицират при операция за актуализиране на поле от масив.
hint (document или string)	Документ или низ, който определя индекса, който да се използва за поддържане на предиката на заявката. Опцията може да приеме документ за спецификация на индекса или низ от името на индекса. Ако укажете индекс, който не съществува, операцията ще бъде грешна

1.3 Изтриване на документ(и)

Може да изтривате един или няколко документа от избрана колекция. За целта използвайте метод **deleteOne** или **deleteMany**. Операциите за изтриване не премахват зададените от вас индекси, дори ако се изтрият всички документи от дадена колекция. Когато се налага да изтриете само един документ, който отговаря на дадено условие (независимо,

че може да има и други документи, които отговарят на да отговарят на зададения филтър) може да използвате метод **deleteOne**. Синтаксисът на методите за изтриване на документи е следния:

db.collection.delete(query, options);

Чрез атрибут “query” се задава филтъра (критерии) на базата на който ще бъдат избрани кои документи да бъдат изтрити. Може да зададете празен документ { }, за да изтриете първия документ, върнат в колекцията. Вторият аргумент задава опции “writeConcern”, “collation” и “hint”. Те имат аналогично приложение, както едноименните опции при метод update. Следва пример, който изтрива първия срещнат документ, за който стойността на поле “status” не е “ok”:

collection.deleteOne(ne("status", "ok"))

Ако трябва да изтриете всички документи, които отговарят на дадено условие, използвайте метод **deleteMany**. Можете да зададете критерии или филтри, които определят документите, които да бъдат изтрити. Филтрите използват същия синтаксис като при четене (метод find). Следващият пример премахва всички документи от колекция за които поле “toDelete” има стойност "yes":

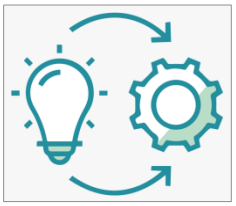
collection.deleteMany(eq("toDelete", "yes"))

За да изтриете всички документи от дадена колекция, подайте празен обект като филтър на метода deleteMany:

collection.deleteMany(new Document());

Методът deleteMany връща обект от тип DeleteResult, който съдържа информация за статуса на операцията.

II. Задачи за изпълнение



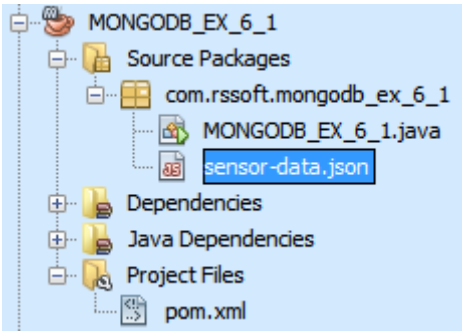
Задача 1: *Напишете Java конзолно приложение, което създава база данни “my-sensors”. Тя трябва да съдържа данните от сензори за температура и влажност на въздуха. Тези данни са в JSON файл. Името на колекцията трябва да бъде “data”.*

Имате данни от 5 сензора, записани като JSON масив във файл “**sensor-data.json**”. Всеки документ има 4 полета (виж Фиг. 6.1). Поле “timestamp” приема за стойност дата и час в ISODate формат. За целта неговата стойност е JSON обект с поле със специфичното име “$date”. Това гарантира, че сървърът ще създаде поле от тип ISODate.

```
{
    "sensorId": 13,
    "timestamp": {
       "$date": "2021-06-16T12:34:42Z"
    },
    "type": "humidity",
    "value": 49
}
```

Фиг.6.1 Документ, описващ данните от сензор

Създайте Java Maven проект с име MONGODB_EX_6_1. В конфигурационния файл pom.xml задайте интегриране към проекта на библиотеки Java Driver и JSON парсер. Създайте базов клас с име MONGODB_EX_6_1. Вмъкнете в тялото на класа метод main, за да създадете конзолно приложение. Структурата на проекта е показана на Фиг. 6.2.



Фиг.6.2 Структура на проект MONGODB_EX_6_1

За да вмъкнете JSON файла “sensors-data.json” към проекта копирайте файла (Ctrl+C) и го прехвърлете в папката с пакета чрез “Paste (Ctrl+V)”. За да може да достъпите файла трябва да знаете пътя до него. Той е в следния формат:

"src\\main\\java\\com\\rssoft\\mongodb_ex_6_1\\sensor-data.json”

```
20      final String DATABASE_NAME = "my-sensors";
21      final String COLLECTION_NAME = "data";
22      final String ARRAY_NAME = "data";
23      final String FILE_NAME = "sensor-data.json";
24      String fpath = "src\\main\\java\\com\\rssoft\\mongodb_ex_6_1\\"+FILE_NAME;
25      try {
26          // ---------------------------------------- Read the whole file
27          String content = new String(Files.readAllBytes(Paths.get(fpath)),
28                                      StandardCharsets.UTF_8);
29          // ---------------------------------- Convert content to JSON object
30          JSONObject obj = new JSONObject(content);
31          JSONArray data = obj.getJSONArray(ARRAY_NAME);
32          // ------------------------------------------------ Connect to server
33          MongoClient mongoClient = new MongoClient("localhost", 27017);
34          //------------------------------------------------- Create database
35          MongoDatabase db = mongoClient.getDatabase(DATABASE_NAME);
36          //------------------------------------------------- Create collection
37          MongoCollection<Document> collection =
38                                          db.getCollection(COLLECTION_NAME);
39          // ------------------------------ Write docs to selected collection
40          int n = data.length();
41          for (int i = 0; i < n; i++) {
42              String dataAsString =
43                      data.getJSONObject(i).toString();
44              Document doc = Document.parse(dataAsString);
45              collection.insertOne(doc);
46          }
47          System.out.println("OK, " + n + " documents are added.");
48          // ------------------------------------------------------------------
49      } catch (IOException | JSONException e) {
50          System.out.println(e);
51      }
```

Фиг.6.3 Програмен код от тялото на метод main – Задача 1

На Фиг. 6.3 е показан програмния код от тялото на метод **main**, който реализира задачата. Първо, декларираме като константи името на базата данни, колекцията, масива от JSON файла и името на файла (редове 20-23). След това формираме пътя до JSON файла – обект “fpath” (ред 24). Програмният код, който чете данните от файла, преобразува ги до JSON обект, създава базата данни и колекцията и вмъква документите в колекцията, е от ред 25 до ред 51. Съдържанието на JSON файла се прочита чрез метод **readAllBytes**. Не забравяйте да зададете правилната кодова таблица с която са кодирани символите – в случая UTF8. След това, съдържанието на файла от низ се конвертира до JSON обект (ред 30). От него се извлича масива с документите (ред 31). Следва свързване с MongoDB сървъра (ред 33), създаване на базата данни (ред 35) и колекцията (редове 37-38). В цикъл (редове 41-46) извличаме документите последователно от обект-масив “data” и чрез метод **insertOne** записваме документ в колекцията. Този метод изисква един атрибут – документа, който се записва. За целта всеки JSON обект от масив “data” се конвертира до низ (ред 43), а след това чрез метод **parse** (ред 44) до желания документ.

След стартиране на приложението, трябва да получите:

```
OK, 5 documents are added.
------------------------
BUILD SUCCESS
------------------------
```

Задача 2: *Напишете Java конзолно приложение, което променя оценката по зададена дисциплина на студент, зададен чрез неговия факултетен номер. Работете с база данни “students”, колекция “data1”.*

Да си припомним как реализирахме тази задача чрез конзолата на приложение Compass (виж Задача 6 от Упражнение № 4):

```
> db.data1.updateOne({"id":"21705003","grade.subject":"НБД"}, {$set:{"grade.$.value":3}})
```

Стойността на атрибут “query” е {“id”:”21705003”,”grade.subject.НБД”}, за да се избере желания студент и желаната дисциплина. Като стойност на атрибут “update” е указано, че само се модифицира едно поле, а не целия документ – оператор $set. Стойността за $set е стойността на поле “value”. Тъй като “grade” е масив, трябва да се използва оператора за позициониране $.

Създайте нов Java Maven проект с име MONGODB_EX_6_2. Предвидете идентификатора на студента и името на дисциплината и новата оценка да се задават като константи. На Фиг. 6.4 е показан примерен код от тялото на метод **main**, който реализира заданието. Използван е метод **updateOne** (ред 39) чрез който се обновява съдържанието само на един документ. Атрибутите, които предаваме към този метод, са декларирани като Bson обекти “query” (редове 35-36) и “update” (редове 37-38). Използвани са статични методи **and** и **eq**, за да се реализира необходимата логика. Метод **set** се използва, за да симулира оператор $set.

```java
final String DATABASE_NAME = "students";
final String COLLECTION_NAME = "data1";
final String STUDENT_ID = "21705001";
final String SUBJECT_NAME = "НБД";
final int GRADE = 6;

try {
    // ------------------------------------------------ Connect to server
    MongoClient mongoClient = new MongoClient("localhost", 27017);
    //------------------------------------------------- Create database
    MongoDatabase db = mongoClient.getDatabase(DATABASE_NAME);
    //------------------------------------------------- Create collection
    MongoCollection<Document> collection
            = db.getCollection(COLLECTION_NAME);
    // ------------------------------------------------- Update field
    Bson query =
            and(eq("id",STUDENT_ID), eq("grade.subject", SUBJECT_NAME));
    Bson update =
            set("grade.$.value", GRADE);
    collection.updateOne(query, update);

    System.out.println("Document update successfully...");
    // ------------------------------------------------------------------

} catch (JSONException e) {
    System.out.println(e);
}
```

Фиг.6.4 Програмен код от тялото на метод main – Задача 2

Задача 3: *Напишете Java конзолно приложение, което променя оценката по зададена дисциплина на студент, зададен чрез неговия факултетен номер. Работете с база данни "students", колекция "data2".*

Припомнете си как реализирахме тази задача чрез конзолата на приложение Compass (виж Задача 6 от Упражнение № 4):

```
> db.data2.updateOne({"id":"21705003"}, {$set:{"grade.НБД.value":3}})
```

Задача 4: *Напишете Java конзолно приложение, което променя оценката по зададена дисциплина на всички студенти от колекция "data1".*

Използвайте метод **updateMany**.

Задача 5: *Напишете Java конзолно приложение, което увеличава с 1 оценката по зададена дисциплина на всички студенти от колекция "data1".*

За да увеличите оценката по избрана дисциплина задайте за стойност на аргумент "update" оператор $inc. В Java можете да симулирате този оператор чрез статичния метод **inc**. Филтърът, зададен чрез аргумент "query" трябва да е съставен от два елемента – единият избира конкретна дисциплина, а другият трябва да предотврати увеличаване на оценките над стойност 6. За целта използвайте оператор за по-малко $lt. Този оператор в Java се реализира чрез статичния метод **lt**.

Задача 6: *Напишете Java конзолно приложение, изтрива от база данни "my-sensors" всички документи, който са генерирани след определена дата.*

Използвайте метод **deleteMany**. Сравняването на дати изисква описание на датата като ISOData обект. Класът Instant в Java 8+ за дата и време (java.time.Instant) позволява създаване на обект, който описва конкретен момент във времето. Метод **parse** от този клас се използва за конвертиране на низ, съдържащ дата и час до обект от тип Instant. За да формираме заявка към MongoDB сървъра е необходимо този Instant обект да преобразуваме до обект от тип Date. Това се реализира чрез статичния метод **from** от клас Date. Програмният код, който реализира условието на задачата е следния:

```
Instant instant = Instant.parse("2021-06-20T00:00:00.000Z");
Date timestamp = Date.from(instant);
Bson query = gt("timestamp", timestamp);
collection.deleteMany(query);
```

Фиг.6.5 Програмен код от тялото на метод main – Задача 6

След изпълнението на този програмен код от колекцията ще бъдат изтрити всички документи за които полето “timestamp” е със стойност дата, по късна от 20.06.2021 г.

III. Задачи за самостоятелна работа

Задача Д1: *Реализирайте всички задачи от Упражнение № 6 като работите с MongoDB Atlas.*

За целта трябва да смените низа чрез който указвате къде се намира сървъра за управление на базата данни, например:

"mongodb://username:password@cluster0-shard-00-00.shlgp.mongodb.net:27017,cluster0-shard-00-01.shlgp.mongodb.net:27017,cluster0-shard-00-02.shlgp.mongodb.net:27017/DATABASE?ssl=true&replicaSet=atlas-lhm95z-shard-0&authSource=admin&retryWrites=true&w=majority"

Упражнение № 7

Достъп до MongoDB с Java – Map-Reduce

I. Въведение

Map-reduce е парадигма за обработка на данни, която позволява да се обработват бързо големи масиви от данни и да се получават обобщени резултати. Map-reduce прилага фаза Map към всеки входен документ, по-точно към документите в колекцията, които отговарят на условието на заявката. Това условие се задава чрез фаза “query”. Функцията Map работи с двойки „ключ-стойност”. За ключовете, които имат няколко стойности, MongoDB прилага фазата за редуциране, която използва израз за обработка, който води до желаните обобщени данни, например получаване на сума или средна стойност. След това MongoDB съхранява резултатите в колекция. По желание изходът на функцията за редуциране може да премине през функция за финализиране, чрез която може да получим допълнително обобщаване на резултатите.

За извършване на операции Map-reduce MongoDB предоставя метод **mapReduce**. Синтаксисът на този метод е следния:

db.collection.mapReduce(map, reduce, options)

където:

- **map** е низ, който описва анонимна JavaScript функция от тялото на която се вика метод **emit** чрез който се задава двойката „ключ-стойност” с която ще се работи, например:

  "function() { emit(KEY, VALUE) }"

  След фаза Map, всички документи с еднаква стойност на ключ KEY се обединяват, като стойностите на поле “VALUE” от всички документи се записват в масив. Следователно тази фаза реализира операция групиране.

- **reduce** е низ, който описва анонимна JavaScript функция която като аргументи получава ключа и стойността. От тялото на функцията се реализира редуцирането на документите, например:

  "function(KEY, VALUE) { return Array.avg(VALUE); }"

  При конкретния пример функцията връща средната стойност на стойностите на всички полета с име “VALUE”. За целта е използвана статична функция avg.

- **options** е JSON обект чрез който можем да зададем първоначално филтриране на документите чрез поле “query”, както и името на новата колекция чрез поле “out”, например:

```
{
    query: {status: “ok”},
    out: “new_collection”
}
```

В този случай ще се обработват само документите, за които поле “status” има стойност “ok”. Изходната колекция се преименува до “new_collection”.

Нека за пример да вземем база данни “my-sensors”. В колекция “data” има 5 документа, които съдържат информация, получена от сензори за температура и влажност на въздуха в определен момент от време (поле “timestamp”). Типът на сензора се задава чрез поле “type”, а стойността на измерената величина чрез поле “value”. Трябва да намерим средната стойност за температурата и влажността на въздуха, получени от всички сензори преди дата 20.06.2021 до текущия момент. Необходимо е да групираме документите по техния тип. Следователно KEY=”type”. Търсим средна стойност на показанията на сензорите. Следователно VALUE=”value”. Зададено е ограничение над входните за Map-Reduce операции документи. Това налага да използваме поле “query” чрез което ще зададем желания филтър. Филтрира се стойността на поле “timestamp” което е от тип ISO-Date. На Фиг. 7.1 е показан резултата, който се получава на всяка фаза от реализация на операции Map-Reduce. На фаза “query” се изключва документа с “timestamp” поле от дата 26.06.2021 г., тъй като не отговаря на зададения филтър. На фаза “map” остават само два документа. Единият описва показанията всички сензори за температура, а другият – сензорите за влажност на въздуха. Стойностите на сензорите са обединени в масив. При последната фаза “reduce” се реализира операция получаване на средна стойност (avg) за всички стойности от масивите.

Фиг.7.1 Реализация на операция Map-Reduce за база данни my-sensors

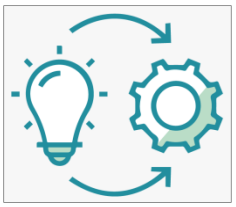
Основното предимство на mapReduce е възможността да персонализираме както map функцията, така и reduce функцията. Това се реализира чрез вмъкване на необходимия

JavaScript код в тялото на функциите. Можете да използвате пълните възможности на езика JavaScript и така да създавате сложни както map, така и reduce функции.

Основният недостатък на метод mapReduce е, че ако използвате map и reduce функциите в базовия им вариант няма да получите по-високо бързодействие при сравнение с използване на операторите за агрегиране на съдържание. Конвейерът за агрегиране на MongoDB осигурява по-добра производителност от Map-Reduce.

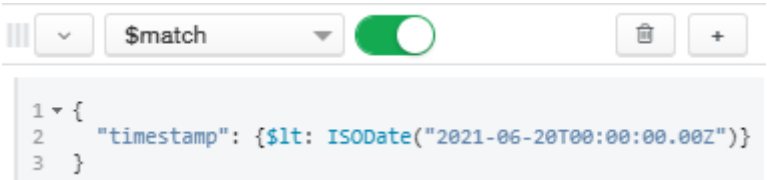
Всички операции Map-Reduce могат да бъдат пренаписани, като се използват операторите на конвейера за агрегиране, например $group и $merge. За операциите Map-Reduce, които изискват персонализирана функционалност, MongoDB от версия 4.4 предоставя операторите за агрегиране $accumulator и $function. Може да използвате тези оператори, за да дефинирате потребителски изрази за агрегиране на съдържание. Оператор **$function** дефинира потребителска функция за агрегиране или израз чрез JavaScript. Можете да използвате оператор $function, за да дефинирате потребителски функции за агрегиране, което не се поддържа от езика за заявки на MongoDB. Акумулаторите са оператори, които поддържат своето състояние (например общи суми, максимални стойности, минимални стойности и свързани с тях данни), докато документите преминават през конвейера. Използвайте оператора **$accumulator**, за да изпълнявате свои собствени функции на JavaScript, за да реализирате поведение, което не се поддържа от езика за заявки на MongoDB. Трябва да се има предвид, че изпълнението на JavaScript в израз за агрегиране може да намали производителността. Използвайте оператора $function само ако предоставените конвейерни оператори не могат да изпълнят нуждите на вашето приложение.

II. Задачи за изпълнение



Задача 1: *Използвайте приложението MongoDB Compass, за да получите чрез агрегиране средната стойност за показанията на всички сензори, групирани по типа им. Работете с база данни "mysensors", колекция "data". Задайте филтриране по дата на получаване на показанията на сензорите: да се обработват данните получени преди дата 20.06.2021 г.*

Задачата можем да решим като използваме две фази на агрегиране. При първата фаза ще филтрираме документите по стойността на поле "timestamp". За целта ще използваме оператор $match:



След тази фаза ще отпадне един от документите. Следващата фаза трябва да групира документите по типа на сензорите. Ще използваме оператор $group, като за стойност за поле "_id" ще зададем "$value". Ще създадем и ново поле с име "avg". Стойността на това поле трябва да е средната стойност на показанията на сензорите. За целта ще използваме оператор $avg:

Задача 2: *Реализирайте условието на Задача 1 като Java конзолно приложение. Закръглете резултата (средна стойност на показанията на сензорите) до втория знак. Включете в резултата и мерните единици: ℃ за температурата и % за относителната влажност на въздуха.*

Използвайте възможността за генериране на Java код чрез приложението Compass. Разгледайте програмния код, който е генериран автоматично. Няма да използваме този код и ще напишем приложението по-кратко и по-разбираемо.

Създайте Java Maven конзолно приложение с име на проекта MONGODB_EX_7_2. Декларирайте в тялото на метод main всички необходими константи:

```
final String DATABASE_NAME = "my-sensors";
final String COLLECTION_NAME = "data";
final String KEY = "type";
final String VALUE = "value";
```

Формирайте стойностите на аргументите на метод mapReduce – map функцията и reduce функцията:

```
String mapFunction =
        String.format("function() {emit(this.%s, this.%s)}",
                KEY, VALUE);
String reduceFunction =
        String.format("function(%s, %s) {return Array.avg(%s);}",
                KEY, VALUE, VALUE);
```

Създайте Bson обект, който описва правилото за филтриране:

```
String date = "2021-06-20T00:00:00.000Z";
Instant instant = Instant.parse(date);
Date timestamp = Date.from(instant);
Bson query = lt("timestamp", timestamp);
```

Следва свързване към базата данни и изпълнение на метод mapReduce. На Фиг. 7.2 е показан програмния код, който реализира комуникацията със сървъра и форматира резултата, съгласно заданието. След като получим обект за достъп до желаната колекция (редове 46-47) може да извикаме метод mapReduce (ред 50). Ще използваме варианта на метода., който изисква задаване на два аргумента: map функцията като низ и reduce функцията като низ. Ще използваме метод **filter** (ред 51) от интерфейс MapReduceIterable, за да зададем необходимото първоначално филтриране на документите по дата. Като аргумент на този метод подаваме обект query, който описва правилото за филтриране.

След изпълнение на метод mapReduce се получава обект “result”, който е от тип MapReduceIterable (ред 49). Създаваме обект-итератор, за да обработим последователно всички документи в обект “result” (ред 53). За целта използваме метод **hasNext**. За да разпечатаме коректно мерната единица проверяваме дали данните са от сензор за тем-

пература или сензор за относителна влажност (ред 59). За да визуализираме °C използваме уникод “\u00B0C”.

След стартиране на приложението трябва да получите следния резултат:

```
Average humidity: 59.67%
Average temperature: 26.00°C
```

```
40          try {
41              // ------------------------------------------------ Connect to server
42              MongoClient mongoClient = new MongoClient("localhost", 27017);
43              //------------------------------------------------- Create database
44              MongoDatabase db = mongoClient.getDatabase(DATABASE_NAME);
45              //------------------------------------------------- Create collection
46              MongoCollection<Document> collection
47                      = db.getCollection(COLLECTION_NAME);
48              // ------------------------------------------------------ Map-Reduce
49              MapReduceIterable<Document> result = collection
50                      .mapReduce(mapFunction, reduceFunction)
51                      .filter(query);
52              // ------------------------------------------------------ Show result
53              MongoCursor<Document> iterator = result.iterator();
54              while (iterator.hasNext()) {
55                  Document document = iterator.next();
56                  String json = document.toJson();
57                  JSONObject obj = new JSONObject(json);
58                  String sensorType = obj.getString("_id");
59                  String unit = sensorType.equals("temperature") ? "\u00B0C" : "%";
60                  double val = obj.getDouble("value");
61                  System.out.printf("Average %s: %.2f%s\n", sensorType, val, unit);
62              }
63              // ---------------------------------------------------------------------
64
65          } catch (JSONException e) {
66              System.out.println(e);
67          }
```

Фиг.7.2 Програмен код - Задача 2

Задача 3: *Напишете Java конзолно приложение което чрез метод mapReduce намира минималната и максимална стойност за показанията на сензорите за температура и относителна влажност на въздуха. Работете с база данни “my-sensors”, колекция “data”.*

Получаването на минималната или максимална стойност на елементите от масив може да се реализира на JavaScript по следния начин:

Math.min.apply(Math,array);

Math.max.apply(Math,array);

Задача 4: *Напишете Java конзолно приложение, което чрез метод mapReduce намира средната стойност за показанията на сензорите за температура само ако температурата е по-висока от 20°C. Работете с база данни “my-sensors”, колекция “data”.*

Формирайте query обекта чрез обединяване чрез метод and на две условия: 1) типът на сензора е “temperature” и 2) Стойността на поле “value” е по-висока от зададената прагова стойност.

Задача 5: *Напишете Java конзолно приложение, което чрез метод mapReduce намира средната стойност за показанията на сензорите за температура само ако температурата е по-висока от 20°C. Реализирайте задачата като дефинирате собствени map и reduce JavaScript функции. Работете с база данни "my-sensors", колекция "data".*

Задачата има за цел да демонстрира възможностите, които метод mapReduce предоставя при използване на собствени map и reduce функции. Ще реализираме необходимото филтриране в тялото на map функцията, а редуцирането (получаване на средна стойност) – в тялото на reduce функцията. На Фиг. 7.3 е показан програмния код, който дефинира необходимите функции. От тялото на map функцията (редове 22-26) емитираме типа на сензора като ключ и показанието на сензора като стойност, само ако типа на сензора е SENSOR_TYPE, а праговата стойност е по-голяма от VALUE_TH. Изчисляването на средна стойност се реализира чрез програмния код от тялото на функция reduce (редове 31-36). За целта използваме цикъл, който сумира стойностите на сензорите от всички входни документи. Крайният резултат получаваме като тази сума разделим на броя на показанията.

```
20          String mapFunction =
21                  "function(){ "
22                      + "if (this.type === \""+ SENSOR_TYPE +"\") {"
23                          + "if (this.value > "+ VALUE_TH +") {"
24                              + "emit(this.type, this.value)"
25                          + "}"
26                      + "}"
27                  + "}";
28
29          String reduceFunction =
30                  "function(type, value){"
31                      + "var n = value.length;"
32                      + "var sum = 0;"
33                      + "for (var i=0; i<n;i++) {"
34                          + "sum += value[i];"
35                      + "}"
36                      + "return sum/n;"
37                  + "}";
```

Фиг.7.3 Дефиниране на функции map и reduce, Задача 5

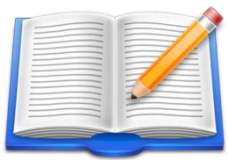
Резултатът, който трябва да получите, е следния:

```
Average temperature: 26.00°C
------------------------------
BUILD SUCCESS
------------------------------
```

Задача 6: *Напишете Java конзолно приложение, което чрез метод mapReduce намира средната стойност за показанията на сензорите за относителна влажност на въздуха за избран месец от годината. Работете с база данни "my-sensors", колекция "data".*

Формирайте "query" обекта чрез обединяване чрез метод **and** на две условия: 1) типът на сензора е "humidity" и 2) Стойността на поле "value" е по-висока или равна на първия ден от месеца и по-малка или равна на последния ден от месеца.

III. Задачи за самостоятелна работа



Задача Д1: *Реализирайте програмно генериране на база данни sensors и документите в нова колекция “data1”. Работете с MongoDB Atlas. Броя на записите да бъде 50,000. При формиране на запис за сензор да се задава на случаен принцип: типа на сензора, стойността на показанието в граници [a, b], идентификатора на сензора в граници [c, d] и времевата марка за която денят е в интервала [1, 31]. Реализирайте всички заявки, описани в това упражнение. Анализирайте времето за получаване на отговор.*

Упражнение № 8

Достъп до MongoDB с Node.js – необходими инсталации

I. Въведение

Node.js е междуплатформена среда за изпълнение на JavaScript с отворен код, която работи с двигателя V8 и изпълнява JavaScript код без участието на Web браузър. V8 е двигател за изпълнение на JavaScript, който първоначално е създаден за Google Chrome през 2008 г. Началният вариант на Node.js е написан от Райън Дал (Ryan Dahl) през 2009 г. Преди проектът Node.js се управляваше от фондация Node.js, а сега от OpenJS Foundation. През 2019 г. фондациите JS Foundation и Node.js Foundation се сляха, за да създадат OpenJS Foundation.

Node.js позволява създаването на Web сървъри и мрежови услуги с помощта на JavaScript и набор от модули, които управляват различни основни функционалности. Предвидени са модули за входно/изходни операции на файловата система, работа в мрежа (DNS, HTTP, TCP, UDP и TLS/SSL комуникации), криптографски функции, работа с потоци от данни и др. Модулите на Node.js използват API, предназначен да намали сложността на писането на сървърни приложения.

Node.js позволява на разработчиците да използват JavaScript за писане на код от страна на сървъра (backend) - изпълнение на скриптове от страна на сървъра за създаване на динамично съдържание на Web страници, преди страницата да бъде изпратена към браузъра на потребителя. Следователно, Node.js позволява разработването на Web приложения чрез един-единствен език за програмиране, а не чрез различни езици от страна на сървъра и от страна на клиента (full-stack програмиране).

Node.js се управлява от събития. Всички комуникации (входно-изходни операции) по подразбиране са асинхронни и следователно са неблокиращи. Този избор на дизайн има за цел да оптимизира пропускателната способност и мащабируемостта на Web приложения с много входно/изходни операции, както и за Web приложения, които работят в реално време. Node.js използва една програмна нишка за изпълнение на JavaScript модули, използвайки неблокиращи I/O извиквания. Това позволява да се поддържат десетки хиляди едновременни комуникационни канали (връзки). При използване на Node.js програмистът не трябва да има познания в областта на thread-safe програмирането. За да приспособи еднонишковия цикъл на събитията, Node.js използва библиотеката libuv - която на свой ред използва пул от нишки с фиксиран размер, който обработва някои от неблокиращите асинхронни I/O операции. Пулът от нишки се грижи за изпълнението на паралелни задачи в Node.js. Извикването на функцията на главната нишка изпраща задачи в обща опашка за задачи, които нишките в пула от нишки изтеглят и изпълняват. Неблокиращите по същността си системни функции, като например мрежовите, се превеждат към неблокиращи сокети от страна на ядрото, докато блокиращите по същността си системни функции, като например файловите входно-изходни операции, се изпълняват блокиращо чрез собствени нишки. Когато някоя нишка в пула от нишки завърши своята задача, тя информира за това главната нишка, която на свой ред се събужда и изпълнява регистрираното обратно извикване (callback функция).

Node.js поддържа WebAssembly, а от Node 14 има експериментална поддръжка на WASI (системния интерфейс на WebAssembly). Node.js предоставя начин за създаване на до-

бавки (addons) чрез N-API (базиран на програмния език C API), който може да се използва за създаване на зареждащи се Node.js модули от изходен код, написан на C или C++.

II. Необходими инсталации

Сървърите за управление на MongoDB бази данни позволяват използване на Node.js за достъп до базите чрез Node.js драйвер. Официалният драйвер MongoDB Node.js позволява на Node.js приложенията да се свързват с MongoDB сървъра по унифициран начин, характерен за всички програмни езици, които MongoDB поддържа. Драйверът разполага с асинхронен API, който позволява да се получи достъп до върнатите стойности (след изпълнение на заявки) чрез JavaScript обещания (promises) или чрез callback функции.

3.3 Инсталиране на Node.js

Може да свалите необходимия инсталатор за Node.js от следния адрес:

https://nodejs.org/en/download/

Изберете инсталатор за вашата операционна система. При ОС Windows е най-добре да свалите съответния msi инсталатор, например:

node-v14.17.3-x86.msi

С тази инсталация ще получите съответната версия на Node.js, както и версия на Node Package Manager (NMP). Приложението NPM е по подразбиране мениджър за инсталиране на Node.js програмни пакети. Състои се от клиент за командния ред, също наречен NPM, и онлайн база данни с публични и платени частни пакети, наречена регистър NPM. Достъпът до регистъра се осъществява чрез клиента, а наличните пакети могат да бъдат разглеждани и търсени чрез уебсайта на NPM Inc.

Направете проверка дали инсталацията е успешна. За целта от командния ред изпълнете следните команди:

```
node –v
npm –v
```

Като резултат трябва да получите версиите на инсталираното програмно осигуряване.

3.4 Инсталиране на MongoDB Node.js драйвер

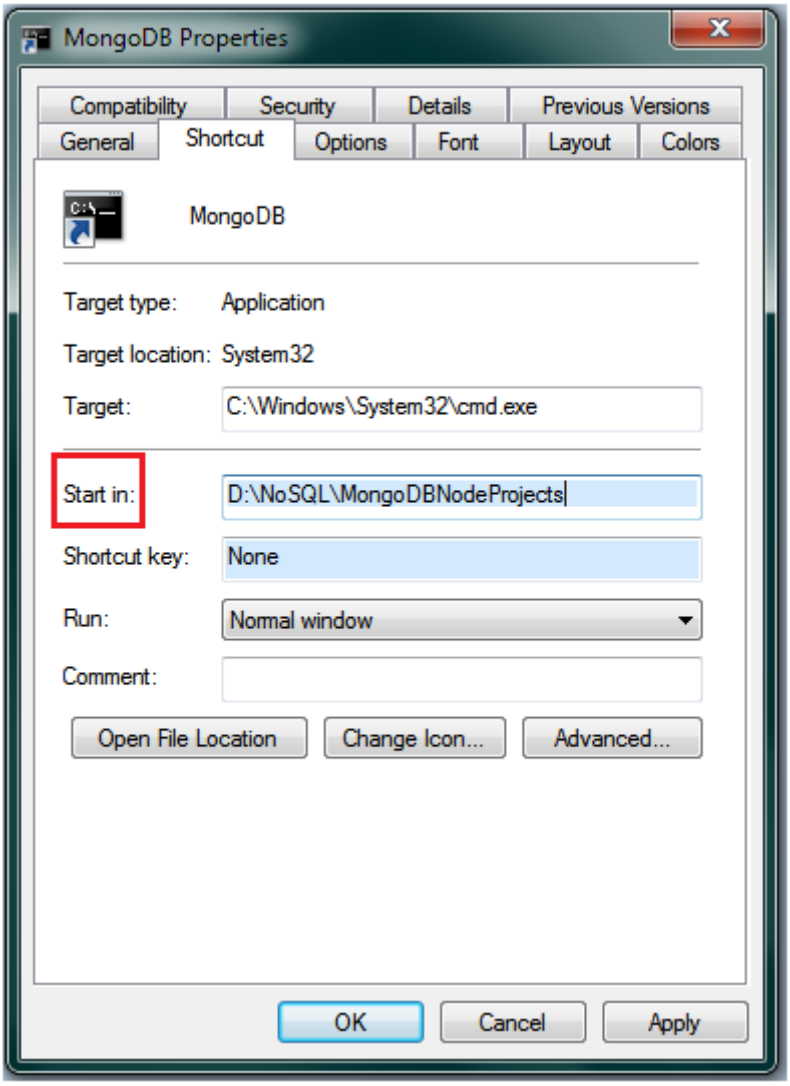
Създайте папка (например MongoDBNojeProjects) в която ще бъдат вашите MongoDB Node.js проекти. За да имате бърз достъп от командно ниво до тази папка, направете за командния интерпретатор на Windows (cmd.exe) препратка на Desktop (виж Фиг. 8.1). За целта изберете с десния бутон на мишката приложението cmd.exe и изберете “Desktop (create shortcut)”. Къде се намира cmd.exe се вижда от поле “Target” (Фиг. 8.1).

След избор на тази препратка, вашата папка ще бъде текущата папка. От командния ред стартирайте следната команда:

```
npm install mongodb --save
```

Тази команда ще изтегли MongoDB Node.js драйвераи ще добави запис за зависимост във вашия файл “package.json”. Проверете дали драйверът е инсталиран коректно:

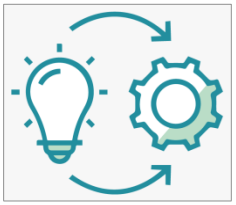
```
npm view mongodb version
```



Фиг. 8.1 Задаване на желана папка да бъде активна след стартиране на cmd.exe

След като имате наличен драйвер, може да го използвате като модул към вашите MongoDB Node.js проекти.

III. Задачи за изпълнение



Задача 1: *Създайте Node.js приложение, което разпечатва на конзолата стойността на показанията на определен тип сензори (за температура или влажност на въздуха) от база данни "my-sensors", колекция "data". Да се обработват само данните, получени преди дата 15.06.2021. Резултатът да се сортира в зависимост от стойността на идентификатора на сензорите.*

Създайте нов JavaScript файл в папката с Node.js проекти, например QUERY_8_1.js. Първо ще декларираме всички необходими константи (виж Фиг. 8.2). Зареждането на желан клас от програмен модул се реализира чрез метод **require**. При конкретния пример от модул "mongodb" създаваме обект от клас MongoClient. Този обект ще е необходим при свързване с базата данни.

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017';
const dbName = 'my-sensors';
const collectionName = "data";
const options = {
    serverSelectionTimeoutMS: 3000,
    connectTimeoutMS: 3000,
    socketTimeoutMS: 3000,
    useUnifiedTopology: true,
};
const sensorType = 'humidity';
```

Фиг. 8.2 Деклариране на константи

Останалите константи, които приложението ще използва, са следните:

- url – низ, съдържат спецификацията за връзка с локално инсталиран MongoDB сървър.
- collectionName – име на колекцията с която ще работим.
- options – JSON обект, който се използва от метод connect чрез който се реализира свързване със MongoDB сървъра. Използвани са полета чрез които се задава времена на изчакване при свързване с MongoDB.
- sensorType – кой тип сензор се интересуваме (“temperature” или “humidity”).

Ще създадем функция **startQuery**, която съдържа програмния код за свързване с базата данни, изпращане на необходимата заявка и парсване на резултата. Тъй като Noje.js работи по подразбиране асинхронно, след като се извика една функция не се чака да се получи резултат, а се вика следващата функция. Следователно ни трябва механизъм чрез който да знаем кога една функция е върнала резултат. Има два основни подхода за това:

- Задаване чрез ключова дума **await**, че трябва да се изчака изпълнението на текущата функция.
- Използване на JavaScript обект-обещание (**promise**) чрез който може да разберем кога една функция е върнала резултат чрез викане на наша callback функция.

И двата подхода могат да бъдат използвани, но за предпочитане е да се работи с обещания. Чрез тях може да синхронизирате работата на своите функции, да регистрирате callback функция при завършване на една или няколко функции. Освен това, използването на обещания позволява много по-добра обработка на възникналите изключения.

При конкретния пример ще използваме ключова дума await. За да използвате await от тялото на дадена функция трябва функцията да се декларира като **async**. Например, декларацията на функция startQuery (виж Фиг. 8.3) започва с ключова дума async. Това е необходимо, тъй като ще използваме ключова дума await за функцията за свързване с базата данни (connect) и функцията за изпращане на заявка (find).

Тъй като реализираме комуникация в мрежова среда, вероятността за получаване на изключения е висока. Обработката им става чрез клаузи **try-catch-finally**. Започване с получаване на обект “client” чрез който ще изградим комуникационен канал между клиента и сървъра. За целта се използва метод **connect**. При проблем се изпълнява програмния код от тялото на клауза catch – печата се съобщение за грешката и се прекратява програмата. Чрез методи **time** и **timeEnd** се изчислява за колко време се изгражда връзката.

Получаването на обект за работа с базата данни се реализира чрез метод **db**, а обект за работа с конкретна колекция – метод **collection**. В нов try блок се изпълнява програмния код за изпращане на заявката и обработка на резултата. Заявката се изпраща чрез метод **find**. Неговият синтаксис е идентичен със синтаксиса на метод find от конзолата на сървъра MongoDB. В конкретния пример метод find получава два JSON обекта. Първият описва филтъра (ограничения за полета “type” и “timestamp”, а втория задава чрез поле “projection” какви полета да останат в резултата. Зададено е възходящо сортиране на резултата (sensorId: 1). Чрез метод **toArray** резултатът се конвертира до списък от масиви. Всеки елемент от списъка е един JSON документ. Обхождането на документите в обекта за резултата “results” се реализира чрез метод forEach и използване на ламбда функция (оператор =>). От тялото на функцията се извлича и разпечатва идентификатора на сензора (sensorId) и стойността на температурата или влажността на въздуха.

```
async function startQuery() {

    let client;
    console.time('Connect time ');
    try {
        client = await MongoClient.connect(url, options);
    }
    catch(error) {
        console.log(`Timeout:\n${error.message}`);
        process.exit();
    }
    finally {
        console.timeEnd('Connect time ');
    }

    const collection = client.db(dbName).collection(collectionName);
    console.time('Find time: ');
    try {
        await collection
            .find(
                {
                    type: sensorType,
                    timestamp: { $lt: new Date('2021-06-15') }
                },
                {
                    projection: {_id: 0, value: 1, sensorId: 1}
                }
            ).sort(
                {
                    sensorId: 1
                }
            ).toArray(function(error, results) {
                if (error) throw error;
                let units = sensorType === 'temperature' ? '\u00B0C' : '%';
                results.forEach(result => {
                    let id = result.sensorId;
                    let value = result.value;
                    console.log(`Sensor ${id}: ${sensorType} ${value}${units}`);
                });
                client.close();
            });
    }
    catch(error) {
        console.log(error.message);
    }
    finally {
        console.timeEnd('Find time: ');
    }
}
```

Фиг. 8.3 Метод startQuery - Задача 1

Не забравяйте да стартирате метод startQuery:

```
startQuery();
```

След като въведете програмния код, запишете JavaScript файла в папката с вашите Node.js проекти. Стартирайте приложението от командния ред чрез следната номада:

```
node QUERY_8_1.js
```

Трябва да получите резултат, подобен на следния:

```
Connect time : 46.494ms
Find time: : 5.39ms
Sensor 15: humidity 57%
Sensor 20: humidity 73%
```

Задача 2: *Създайте Node.js приложение, което разпечатва на конзолата средната стойност на показанията на всички сензори, групирани по техния тип (температура или влажност на въздуха) от база данни "my-sensors", колекция "data". Да се обработват само данните, получени преди дата 20.06.2021.*

Създайте нов JavaScript файл в папката с Node.js проекти, например QUERY_8_2.js. В този случай, ще използваме функцията за агрегиране на съдържание, тъй като трябва да получим средна стойност. В Node.js тази функция е с име **aggregate**. Следователно трябва да замените метод find с метод aggregate. Чрез оператор $match ще зададем нужния филтър (поле "timestamp"). Чрез оператор $group ще обединим всички файлове с еднакъв тип на сензора и ще въведем в изходния документ ново поле с име "avg", чиято стойност ще получим чрез оператор $avg приложен към стойността на показанията на сензора – поле "value". На Фиг. 8.4 е показан програмния код който вика метод aggregate.

```
    try {
        await collection
            .aggregate(
                [
                    {
                        $match:
                        {
                            timestamp: { $lt: new Date('2021-06-20') }
                        }

                    },
                    {
                        $group:
                        {
                            _id : '$type',
                            avg : { $avg : '$value' }
                        }
                    }
                ]
            ).toArray(function(error, results) {
                if (error) throw error;
                results.forEach(result => {
                    let type = result._id;
                    let units = result._id === 'temperature' ? '\u00B0C' : '%';
                    console.log(`Average ${type}:
                                ${result.avg.toFixed(2)}${units}`);
                });
                client.close();
            });
    }
```

Фиг. 8.4 Програмен код на метод aggregate - Задача 2

Трябва да получите следния резултат:

```
Connect time : 57.631ms
Aggregate time: : 5.446ms
Average temperature: 26.00°C
Average humidity: 59.67%
```

Задача 3: *Тествайте програмния код от Задача 1 и Задача 2 но с бази данни, разположени в MongoDB Atlas. Направете анализ на времето за свързване със сървъра и времето, необходимо за изпълнение на заявките.*

Използвайте следният синтаксис за обект url:

```
"mongodb://username:password@cluster0-shard-00-00.shlgp.mongodb.net:27017,cluster0-shard-00-01.shlgp.mongodb.net:27017,cluster0-shard-00-02.shlgp.mongodb.net:27017/DATABASE?ssl=true&replicaSet=atlas-lhm95z-shard-0&authSource=admin&retryWrites=true&w=majority"
```

Не забравяйте да зададете коректни стойности за “username”, “password” и “DATABASE”!

IV.Задачи за самостоятелна работа

Задача 1: *Напишете Node.js приложение което разпечатва на конзолата средния успех на всички студенти от база данни “students”, колекция “data1”.*

Задача 2: *Напишете Node.js приложение което разпечатва на конзолата кои студенти по коя дисциплина имат двойки. Списъкът да е сортиран по име на студента. Работете с база данни “students”, колекция “data1”.*

Задача 3: *Напишете Node.js приложение което разпечатва на конзолата броя на слабите оценки за всички студенти от база данни “students”, колекция “data1”.*

Задача 4: *Напишете Node.js приложение което разпечатва на конзолата имената на студентите, които имат повече от една слаба оценка. Работете с база данни “students”, колекция “data1”.*

Упражнение № 9

Достъп до MongoDB с Node.js – CRUD операции

I. Въведение

Създаването на нова база данни, колекция и документи, обновяването и изтриване на документи чрез използване на Node.js е аналогично на това при използване на Java. Използват се същите методи, благодарение на MongoDB Node.js Driver. Трябва да отбележим, че JavaScript има вградени възможности за работа с JSON документи и не се налага да използвате външни модули. Операциите Create, Read, Update, Delete (CRUD) позволяват да работите с данните, съхранявани в базата данни. Те се делят на две групи:

- Операции за четене.
- Операции за запис.

II. Операции за четене

Към тази група принадлежат операции за намиране и връщане на съдържанието на документи. Съществуват няколко вида операции за четене, които осъществяват достъп до данните по различни начини. Вече знаем, че извличането на информация от базата данни въз основа на набор от критерии от съществуващия набор от данни, можете да използвате един от вариантите на метод **find**. Можете също така да уточните допълнително информацията, която искате да получите, например:

- Сортиране на резултатите ($sort).
- Пропускане на върнатите резултати ($skip).
- Ограничаване на броя на върнатите резултати ($limit).
- Избор кои полета да бъдат върнати ($project).

Можете да използвате операция за агрегиране, за да извлечете желаните данни. За целта използвайте метод **aggregate**. Този тип операция ви позволява да приложите конвейер от трансформации към данните.

III. Операции за запис

Към тази група принадлежат операции за вмъкване, промяна или изтриване на документи от бази данни.

3.1 Вмъкване на документ

Ако искате да добавите нов(и) документ(и) в колекция, можете да използвате метода **insertOne** или **insertMany**. Тези методи приемат като аргумент(и) един или множество документи. Драйверът автоматично генерира уникално поле „_id” за документите, освен ако не е посочено да се работи с друг идентификатор. Синтаксисът на метод insertOne е следния:

```
const result = await collection.insertOne(myDocument);
```

където myDocument е JSON обект, описващ документа, който искате да вмъкнете. Може да проверите броя на реално вмъкнатите документи чрез стройността на поле “insertedCount” от обект “result”:

```
console.log(result.insertedCount);
```

Ако искате да вмъкнете множество документи в базата данни чрез един метод, използвайте insertMany. Синтаксисът на този метод е следния:

```
const result = await collection.insertMany(myDocuments);
```

където myDocuments е масив от JSON обекти - документите, които искате да вмъкнете.

3.2 Изтриване на документ

Ако искате да премахнете съществуващ документ от колекция, можете да използвате метод **deleteOne**. Използвайте метод **deleteMany**, за да премахнете един или повече документи. Тези методи изискват като аргумент документ за заявка, който съответства на документите, които искате да изтриете.

```
const result1 = await collection.deleteOne(query);
const result2 = await collection.deleteMany(query);
```

където "query" е JSON обект, който описва кой обект(и) искаме да изтрием, например:

```
const query = {
    timestamp: {
        $lte: new Date(2021-06-20),
        $gte: new Date(2021-06-10)
    }
};
```

В този случай се задава да се изтрият всички документи от дата 10.06.2021 до 20.06.2021. Ако използваме метод deleteOne, ще бъде изтрит само първият срещнат в колекция документа за който условието е в сила. Реалният брой на изтритите документи можете да получите чрез стойността на поле "deletedCount", например:

```
console.log(deleteResult.deletedCount);
```

3.3 Обновяване на съдържанието на документ(и)

Обновяването (актуализация) на съдържанието на документ(и) се реализира чрез метод **update**. Променят се определени полета, а останалите полета остават непроменени. За да извършите актуализация на един или повече документи, създайте документ за актуализация, в който са посочени операторът за актуализация (типът актуализация, който трябва да се извърши) и полетата и стойностите, които описват промяната. Методите за актуализация са два: **upadeOne** и **updateMany**. Синтаксисът е следния:

```
const result = await collection.updateOne(filter, updateDocument);
```

където "filter" е JSON обект, който описва правилата за филтриране на документите, а втория аргумент "updateDocument" е JSON обект, който описва какво се актуализира. Форматът на "updateDocument" е следния:

```
{
   <update operator>: {
     <field> : {...}
     },
     <field> : {...}
   },
   <update operator>: {
      ...
   }
}
```

Операторите за актуализация, които може да използвате са следните:

- $set - заменя стойността на дадено поле с определена стойност.
- $inc - увеличава или намалява стойностите на определени полета.
- $rename - преименува полета.
- $unset - премахва полета.
- $mul - умножава стойността на полето по определено число.

Следва пример, който променя една от оценките на студент с идентификатор “21705001”:

```
const filter = {id: "21705001"}
const updateDocument = {
   $set: {
      "grade.НБД.value": 5
   }
};
const result = await collection.updateOne(filter, updateDocument);
console.log(`Брой обновени полета: ${result.modifiedCount}`);
```

За да обновите множество документи с една заявка използвайте метод updateMany. Синтаксисът е същият като този за updateOne. В зависимост от филтъра ще бъдат обновени 0, 1 или повече документи.

Обновяване на полета от масиви

Ако трябва да модифицирате масив, можете да използвате оператор за актуализиране на масива в извикването на метода за актуализиране, например:

- $ - позиционен оператор - първи елемент от масива.
- $[] - оператор за съвпадение на всички елементи на масива.
- $[<идентификатор>] - филтриран позиционен оператор.

За да извършите актуализация само на първия елемент от масива на всеки документ, който съвпада с документа на заявката ви, използвайте оператора за актуализация на позиционен масив **$**. Този оператор за актуализация се позовава на масива, съответстващ на филтъра на заявката, и не може да се използва за позоваване на масив, вложен в този масив. Ако искате да обновите поле или полета от всички документи в масива използвайте оператор **$[]**. За случаите, в които трябва да получите достъп до вложените масиви, използвайте позиционния оператор **$[<идентификатор>]**. Стойността на „идентификатор” се задава в обекта “arrayFilters”, който се предава като последен аргумент за метод update. Този обект представлява масив от филтри, които определят кои елементи от масива да бъдат включени в актуализацията, например:

```
const filter = { };
const updateDocument = {
    $set: {"arrayName.$[arrayItem].fieldName": value }
};
const updateOptions =
{
    arrayFilters: [{
      "arrayItem.fieldName1": value1,
      "arrayItem.fieldName2": value2
    }]
};
const result
= await collection.updateMany(filter, updateDocument, updateOptions);
```

Ако искате да обновите стойността на елемент от масив използвайте оператор $set. Ако трябва да вмъкнете нов елемент в масив - използвайте оператор $push.

3.4 Замяна на съдържанието на документ(и)

Методите за замяна премахват всички съществуващи полета в един или повече документи и ги заместват с определени полета и техните стойности. За целта се използват методи **replaceOne** и **replaceMany**. Синтаксисът е следния:

```
const result = await collection.replaceOne(filter, replacementDocument);
```

За да извършите операция за замяна, създайте документ за замяна, който се състои от полетата и стойностите, които искате да вмъкнете в операцията за замяна. Документите за замяна използват следния формат:

```
{
   <field>: {
      <value>
   },
   <field>: {
      ...
   }
}
```

Следва пример, който заменя всички полета от документ с поле “id” = “21705001” с поле “newField”=5. Не може да заменяте само служебното поле “_id”. Броят на модификациите може да получите чрез поле “modifiedCount” от обект result.

```
const filter = {id: "21705005"}
const replacementDocument = {
   newField: 5
};
const result = await collection.replaceOne(filter, replacementDocument);
console.log(`Брой заместени документи: ${result.modifiedCount}`);
```

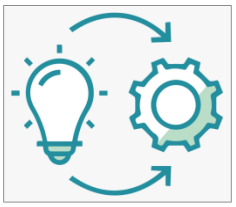
3.5 Вмъкване или актуализиране с една операция

В някои случаи може да се наложи да избирате между операция за вмъкване и актуализация в зависимост от това дали документът съществува. В тези случаи можете да оптимизирате логиката на приложението си, като използвате опцията “**upsert**”. Тя е налична в следните методи:

- updateOne
- updateMany
- replaceOne

Ако филтърът от заявката, предаден на тези методи, не намери съвпадения и стойността на “upsert” е true, сървърът за управление на MongoDB ще вмъкне нов документ. В противен случай зададените полета ще се обновят (update) или ще заместят старите полета (replace).

IV.Задачи за изпълнение



Задача 1: *Създайте Node.js приложение, което* обновява оценката по дисциплина НБД на всички студенти от база “students”, колекция “data1”.

Тъй като трябва да се обновят оценките за всички студенти ще използваме метод **updateMany**. Поради тази причина не е необходимо да се задава условие за филтриране на документите. При подобни случаи аргумент “filter” трябва да бъде празен JSON обект. Чрез атрибут “updateDocument” трябва да се зададе, че се обновява оценката (поле “value”) по дисциплината “НБД”. Следователно, трябва да се използва оператор $set и позиционен идентификатор с филтриране. От всички дисциплини в масив “grade” трябва да се работи само с тези за които поле “subject” е със стойност “НБД”. Това се постига чрез опция “arrayFilters”. Филтрирането е в зависимост от стойност на поле “subject”. Едно примерно решение на задачата е показано на Фиг. 9.1.

```
const filter = { };
    const updateDocument = {
      $set: {"grade.$[arrayItem].value": 4}
    };
const updateOptions = {
    arrayFilters: [{
        "arrayItem.subject": "НБД"
    }]
};
const result =
   await collection.updateMany(filter, updateDocument, updateOptions);
console.log(`Брой обновени документи: ${result.modifiedCount}`);
```

Фиг. 9.1 Примерно решение на Задача 1

Задача 2: *Създайте Node.js приложение, което създава нова база данни, колекция и документи към колекцията. Документите да се четат от JSON файл. Да се използват JavaScript обещания, за да се синхронизира работата на използваните методи.*

Ще използваме възможностите на Node.js за работа с файлове чрез модул “fs”. Константите, които приложението ще използва, са показани на Фиг. 9.2. Зареждането на модул “fs” се реализира чрез метод “require”. Останалите константи са свързани с имената на базата данни, колекцията и опциите, които се предават като аргумент на метод connect.

```
const fs = require('fs');
const MongoClient = require('mongodb').MongoClient;
const url = "mongodb://localhost:27017";

const jsonFile = 'filename.json';
const databaseName = 'databaseName';
const collectionName = "collName";

const options = {
    serverSelectionTimeoutMS: 3000,
    connectTimeoutMS: 3000,
    socketTimeoutMS: 3000,
    useUnifiedTopology: true
};
```

Фиг. 9.2 Използвани константи, Задача 2

```
const createDatabase = (file, database, collection) =>

    new Promise((resolve, reject) => {

        process.stdout.write("Read data from file: ");
        fs.readFile(file, (error, data) => {
            if (error) return reject(error);
            process.stdout.write('OK\n');
            var jsonData = JSON.parse(data);

            process.stdout.write("Connect to server: ");
            MongoClient.connect(url, options, (error, conn) => {
                if (error) return reject(error);
                process.stdout.write('OK\n');
                let db = conn.db(database);

                process.stdout.write("Create collection: ");
                db.createCollection(collection, (error, result) => {
                    if (error) return reject(error);
                    process.stdout.write('OK\n');

                    process.stdout.write("Insert docs: ");
                    result.insertMany(jsonData, (error, result) => {
                        if (error) return reject(error);
                        conn.close();
                        return resolve(result.insertedCount +
                                                " documents inserted.");
                    });

                });

            });

        });

    });
```

Фиг. 9.3 Метод createDatabase - Задача 2

На Фиг. 9.3 е показан програмния код на метод **createDatabase** чрез който се създава базата данни, колекция и се записват документите в колекцията. Методът изисква три аргумента:

- file – файлова спецификация на JSON файла с документите.
- database – име на базата данни.
- collection – име на колекцията.

Форматът на JSON файла трябва да съдържа масив с документи. Съдържанието на файла се прочита чрез метод **readFile** от модул “fs”. Методът приема два аргумента - файловата спецификация и ламбда функция, която се активира след като readFile върне отговор. При успешно прочитане на данните, посредством метод parse конвертираме съдържанието на файла от низ до JSON обект – jsonData.

Следва свързване с MongoDB сървъра чрез метод **connect**. На метода се предават три аргумента: URL, опции към connect и ламбда функция, която се активира след като метод connect върне резултат. При успешно създаване на комуникационен канал се създава базата данни чрез метод **db**. Следва създаване на желаната колекция. Това се реализира чрез метод **createCollection**. Остана само да вмъкнем документите (jsonData) в новата колекция. Това се реализира чрез метод **insertMany**.

На Фиг. 9.4 е показан програмния код, който вика метод **createDatabase**.

```
createDatabase(jsonFile, databaseName, collectionName)
  .then(result => console.log(result))
  .catch(error => {
      process.stdout.write(`(ERROR: ${error.message})`);
      process.exit()
  });
```

Фиг. 9.4 Викане на метод createDatabase - Задача 2

Вижда се, че е използван JavaScript обект-обещание, за да се синхронизира момента на получаване на информация от сървъра. Освен това, по този начин се съсредоточава обработката на изключенията само на едно място – тялото на клауза catch. Програмният език JavaScript използва интерфейс $Promise$, за да създаде обект-обещание. Чрез този обект е възможно да дефинираме асинхронно работещ код, както и да получим информация в момента, когато кодът завърши без или с грешка. Синтаксисът на създаване на обещание е показан на Фиг. 9.5. Ако асинхронния код от тялото на анонимната функция завърши успешно се вика метод **resolve** (ред 5). При грешка се вика метод **reject** (ред 8). И на двата метода се предава един аргумент – JSON обект с резултата или JSON обект, който описва грешката.

```
1  var promise = new Promise(function(resolve, reject) {
2      // асинхронен код, изпълнението на който
3      // завършва с уставяване на флаг статус
4      if (статус === "ок") {
5          resolve(обект);
6      }
7      else {
8          reject(обект);
9      }
10 });
```

Фиг. 9.5 Създаване на обект-обещание

Когато се изпълни метод **resolve** се вика метод **then**, приложен към обекта-обещание. Следователно, той изпълнява ролята на callback функция. При извиквана на метод **reject**, управлението се предава на **catch** клауза (виж Фиг. 9.6).

```
1  promise.then(function(обект) {
2      // успешно изпълнен код (resolve)
3  }).catch(function(обект) {
4      // неуспешно изпълнен код
5  })
```

Фиг. 9.6 Обработка на събития resolve и reject

Задача 3: *Създайте Node.js приложение, което създава нова база данни с име "sensors" и колекция с име "data". Всеки документ в тази колекция трябва да съдържа информация за сензор от безжична сензорна мрежа. Сензорите могат да бъдат два типа: за измерване на температура и за измерване на относителна влажност на въздуха. Всеки сензор има идентификатор, тип, показание и времева марка (кога е направено измерването).*

За да решим задачата ще ни трябват следните константи:

```
const databaseName = 'sensors';
const collectionName = "data";

const numberOfRecords = 10000;
const numberOfSensors = 20;
const sensorTypes = ['temperature', 'humidity'];
const numberOfSensorTypes = sensorTypes.length;
```

Фиг. 9.7 Използвани константи - Задача 3

Тъй като ще генерираме случайни стойности за идентификатор на сензор, за тип на сензор и времева марка ще използваме два метода, които генерират случайни числа в интервала [min, max]. Техния програмен код е показан на Фиг. 9.8. Първият метод връща случайно число, а вторият – случайно число като форматиран низ. Последното е необходимо, за да генерираме случайна дата и час.

```
function getRandomNumberInRange(min, max) {
    return Math.floor((Math.random() * (max - min + 1) + min));
}
function getRandomNumberInRangeAsString(min, max) {
    let random = getRandomNumberInRange(min, max);
    return String(random).padStart(2,'0');
}
```

Фиг. 9.8 Методи за генериране на случайни числа в зададен интервал, Задача 3

Програмният код на метод **createDatabase**, който създава базата данни и я заселва с документи, е показан на Фиг. 9.9. След създаване на базата данни и колекцията се преминава към формиране на масив от документи в паметта. За целта се използва масив с име “records”. Чрез for цикъл се получават всички необходими полета и формиране на JSON обект от тях – обект “records”. Вмъкването на всеки документ в масива се реализира чрез метод **push**. След това остава само да въведем документите в колекцията чрез метод **insertMany**. При успешно изпълнение на метод **createDatabase** се връща броя на вмъкнатите документи чрез метод **resolve**. При някак тип грешка, обектът, който описва грешката, се предава за обработка към клауза **catch** чрез викане на метод **reject**.

Викането на метод createDatabase се реализира с програмния код показан на Фиг. 9.10.

V. Задачи за самостоятелна работа

Задача Д1: *Напишете Node.js приложение което разпечатва на конзолата имената на всички участници, които не са от университет, както и името на учебното заведение. Работете с база данни “students”, колекция “data1”.*

```
const createDatabase = (database, collection) =>
    new Promise((resolve, reject) => {
        process.stdout.write("Connect to server: ");
        MongoClient.connect(url, options, (error, conn) => {
            if (error) return reject(error);
            process.stdout.write('OK\n');
            let db = conn.db(database);
            process.stdout.write("Create collection: ");
            db.createCollection(collection, (error, result) => {
                if (error) return reject(error);
                process.stdout.write('OK\n');
                process.stdout.write("Insert docs: ");
                let records = [];
                for (let i=0; i<numberOfRecords; i++) {
                    let sensorTypeId =
                    getRandomNumberInRange(0, numberOfSensorTypes-1);
                    let sensorTypeName = sensorTypes[sensorTypeId];
                    let temperature = getRandomNumberInRange(-10, 38);
                    let humidity = getRandomNumberInRange(20, 99);

                    let day = getRandomNumberInRangeAsString(1, 31);
                    let hour = getRandomNumberInRangeAsString(0, 24);
                    let min = getRandomNumberInRangeAsString(0, 59);
                    let sec = getRandomNumberInRangeAsString(0, 59);
                    let sensorId = getRandomNumberInRange(1, numberOfSensors);
                    let record = {
                        sensorId: sensorId,
                        timestamp:
                           new Date(`2021-06-${day}T${hour}:${min}:${sec}Z`),
                        type: sensorTypeName,
                        value: sensorTypeName ===
                                'temperature' ? temperature : humidity
                    }
                    records.push(record);
                }
                result.insertMany(records, (error, result) => {
                    if (error) return reject(error);
                    conn.close();
                    return resolve(result.insertedCount +
                                            " documents inserted.");
                });

            });

        });

    });
```

Фиг. 9.9 Програмен код на метод createDatabase - Задача 3

```
createDatabase(databaseName, collectionName)
  .then(result => console.log(result))
  .catch(error => {
      process.stdout.write(`(ERROR: ${error.message})`);
      process.exit();
  });
```

Фиг. 9.10 Викане на метод createDatabase, Задача 3

Упражнение № 10

Достъп до MongoDB с Node.js – търсене и агрегиране на съдържание

I. Въведение

Ще използваме методи **find** и **aggregate** с цел търсене и агрегиране на съдържание. Ще създадем нова база данни GAPCOM, която описва участниците в състезанието GAPCOM за 2017 и 2018 година, както и техните резултати. За целта ще започнем с програмно създаването на базата данни чрез приложението разработено в Задача 2 от Упражнение № 9. Базата данни трябва да има 4 колекции с имена “data-2017”, “data-2018” и “data-2017+2018” и “data-2017-1018”. При първите две колекции се описват участниците и техните резултати за конкретна година, третата колекция обединява документите от първите две колекции, а при последната колекция се описват резултатите за всички години чрез подходящ избор на схемата на базата данни. За да създадем необходимата база данни ще стартираме приложението четири пъти, като променяме стойностите на следните константи:

- jsonFile – име на файла, който съдържа необходимите документи като масив от JSON обекти.
- collectionName – име на колекция, съответства на името на файла без разширението.

```
const jsonFile = "data-2017.json";
const collectionName = "data-2017";
const databaseName = 'GAPCOM';
```

Стартирайте MongoDB Compass и проверете дали базата данни GAPCOM е създадена коректно (виж Фиг. 10.1).

data-2017	15	298.5 B	4.4 KB
data-2017+2018	21	403.7 B	8.3 KB
data-2017-2018	21	402.0 B	8.2 KB
data-2018	14	293.7 B	4.0 KB

Фиг. 10.1 Колекции в база данни GAPCOM

Съдържанието на документите от колекции data-2017 и data-2018 използва една и съща схема (виж Фиг. 10.2). Всеки участник се описва със следните полета:

- Поле “name” – имена на участника.
- Поле “affiliation” – принадлежност на участника (училище или университет, клас или година на обучение).
- Поле “language” – програмен език, който участникът е избрал.

- Поле “points” – брой точки от теста.
- Поле “medal” – получил или не е медал участника и ако да – какъв?
- Поле “status” – участвал ли е реално в състезанието или само се е записал?

```
{
   "name": {
      "firstname": "Йордан",
      "family": "Валентинов",
      "lastname": "Петков"
   },
   "affiliation": {
      "type": "school",
      "name": "Математическа гимназия - Габрово",
      "class": 11
   },
   "language": "C++",
   "points": 86,
   "medal": "gold",
   "status": "ok"
}
```

Фиг. 10.2 Съдържание на документ от колекция data-2017

В колекция “data-2017+2018” е използвана схема, която обединява данните за резултата от всички състезания (виж Фиг. 10.3).

```
{
   "name": {
      "firstname": "Йордан",
      "family": "Валентинов",
      "lastname":"Петков"
   },
   "results": [
      {
         "year": 2017,
         "affiliation": {
            "type": "school",
            "name": "Математическа гимназия - Габрово", "class": 11
         },
         "language": "C++",
         "points": 86,
         "medal": "gold",
         "status": "ok"
      },
      {
         "year": 2018,
         "affiliation": {
            "type": "school",
            "name": "Математическа гимназия - Габрово", "class": 12
         },
         "language": "C++",
         "points": 92,
         "medal": "gold",
         "status": "ok"
      }
   ]
}
```

Фиг. 10.3 Съдържание на документ от колекция data-2017+2018

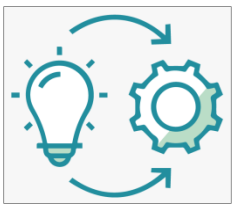
В колекция “data-2017-2018” е използвана друга схема за описание на участниците и техните резултати (виж Фиг. 10.4).

```
{
   "name": {
      "firstname": "Йордан",
      "family": "Валентинов",
      "lastname":"Петков"
   },
   "results": [
      {
         "2017":
         {
            "affiliation": {
               "type": "school",
               "name": "Математическа гимназия - Габрово",
               "class": 11
            },
            "language": "C++",
            "points": 86,
            "medal": "gold",
            "status": "ok"
         },
         "2018":
         {
            "affiliation": {
               "type": "school",
               "name": "Математическа гимназия - Габрово",
               "class": 12
            },
            "language": "C++",
            "points": 92,
            "medal": "gold",
            "status": "ok"
         }
      }
   ]
}
```

Фиг. 10.4 Съдържание на документ от колекция data-2017-2018

При тази схема се използва поле “results” което обединява в масив резултатите от всички участия на всеки един от участниците. Полетата в този масив имат за ключ годината на провеждане на състезанието. Стойността на този ключ са всички данни, които могат да приемат различни стойности в зависимост от конкретното състезание.

II. Задачи за изпълнение



Задача 1: *Създайте Node.js приложение, което разпечатва сортирано по име всички участници в GAPCOM 2018, които имат повече от 70 точки.*

Нека да използваме данните от колекция “data-2018”. На Фиг. 10.5 е показано едно примерно решение на задачата. Резултатът, който трябва да получите е показан на Фиг. 10.6.

```
MongoClient.connect(url,{useUnifiedTopology: true}, (err, conn) => {
  if (err) throw err;
  let db = conn.db(dbName);
  db.collection(collectionName)
    .find(
      {
        status: "ok",
        points: { $gt: 70 }
      },
      {
        projection: {_id: 0, name: 1, points: 1}
      }
    ).sort(
      {
        "name": 1
      }
    ).toArray( (error, docs) => {
      if (error) throw error;
      docs.forEach((doc) => {
        let name = doc.name;
        console.log(`${name.firstname} ${name.family}: ${doc.points}`);
      });
      conn.close();
    });
});
```

Фиг. 10.5 Примерно решение на Задача 1

```
Адриян Ивов: 82
Веселин : 74
Добрин Венциславов: 96
Емил Иво: 72
Ивайло Николаев: 75
Йордан Валентинов: 92
Мартин Даниел: 92
Никола Мирославов: 82
Пламен Венелинов: 82
Стефан Георгиев: 75
Цвета Огнянова: 84
Цветомир : 72
```

Фиг. 10.6 Резултат - Задача 1

Задача 2: *Създайте Node.js приложение, което разпечатва сортирано по име участниците в GAPCOM 2017 и GAPCOM 2018. Работете с колекции "data-2017" и "data-2018".*

Тъй като данните, от които се нуждаем, са в две колекции трябва да използваме оператор **$lookup**. Този оператор симулира JOIN и може да се използва при агрегиране на съдържание. По-точно, операторът извършва ляво външно присъединяване (left outer join) към колекция, която не е разпределена, и е в същата база данни. Целта е филтриране и агрегиране на съдържание за документите от базовата и присъединените колекции.. Към всеки входен документ, оператор $lookup добавя ново поле от масив, чиито елементи са съвпадащите документи от "присъединената" колекция. Синтаксисът на оператора е следния:

```
{
   $lookup:
     {
       from: <колекция за присъединяване>,
       localField: <поле от входните документи, базова колекция>,
       foreignField: <поле от документите на колекцията "from">,
       as: <поле от изходния масив - резултат>
     }
}
```

Нека базовата колекция е “data-2017”. Колекцията за присъединяване е “data-2018”. Локалното поле трябва да бъде “name”. Външното поле от колекция “data-2018” трябва също да е “name”. Нека резултата да се запише в масив “students”. На следващия етап от агрегиране ще използваме оператор **$unwind** чрез който ще създадем нова колекция, която ще има по един документ за всеки елемент от масива “students”.

```
db.collection('data-2017')
    .aggregate(
    [
        {
            $lookup: {
                from: 'data-2018',
                localField: 'name',
                foreignField: 'name',
                as: 'students'
            }
        },
        {
            $unwind: '$students'
        },
        {
            $project: { _id: 0, name: '$students.name' }
        },
        {
            $sort: { name: 1 }
        }
    ]).toArray(function(error, docs) {
        if (error) throw error;
        docs.forEach((doc) => {
            console.log(`${doc.name.firstname} ${doc.name.family}`);
        });
        conn.close();
    });
```

Фиг. 10.7 Примерно решение на Задача 2

```
Адриян Ивов
Добрин Венцислaвов
Йордан Валентинов
Мартин Даниел
Никола Мирославов
Пламен Венелинов
Цвета Огнянова
```

Фиг. 10.8 Резултат - Задача 2

Задача 3: *Създайте Node.js приложение, което разпечатва сортирано по общ брой точки участниците в GAPCOM 2017 и в GAPCOM 2018. Работете с колекции “data-2017” и “data-2018”.*

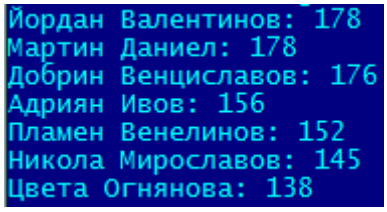
Ще използваме програмния код от Задача 2. Промените са само две:

- Ще създадем ново поле “total”, което ще съдържа сумата от точките на всеки състезател за 2017 и 2018 година. Това поле ще вмъкнем в тялото на оператор $project. Сумирането на точките може да се реализира чрез оператор $add:

  total: {$add: ['$students.points', '$points']}

- Сортирането трябва да се реализира по стойността на поле “total”, "total": -1.

На Фиг. 10.9 е показан резултата, който трябва да получите.



Фиг. 10.9 Резултат - Задача 3

Задача 4: *Създайте Node.js приложение, което разпечатва имената участниците в състезанието GAPCOM, годината на участие и получените точки, ако за поне едно от състезанията са получили повече от 85 точки. Сортирайте резултата по имената на участниците. Работете с колекции “data-2017+2018”.*

За да решим задачата ще използваме агрегиране. При първата фаза на агрегиране ще филтрираме документите съгласно условието на задачата чрез оператор $match. Ще използваме оператор $elemMatch, за да проверим за кои обекти в масив “results” условието „годината да е в интервала [2017, 2017] и броя на точките да е над 85” е изпълнено.

При втората фаза на агрегиране ще използваме оператор $project, за да зададем как да се формира резултата. Ще премахнем идентификатора на документите и ще оставим поле “name”. Ще формираме ново поле с име “result” което ще съдържа само желаната от нас информация: годината на провеждане на състезанието и точките, които участниците са получили. За целта ще използваме оператор $map.Синтаксисът на този оператор е следния:

```
{
   $map: {
      input: <входен масив>,
      as: <Променливата, която представлява всеки отделен елемент на
           входния масив. Ако не е зададено име, името на променливата по
           подразбиране е this.>
      in: <Израз, който се прилага към всеки елемент на входния масив>
   }
}
```

В конкретния случай стойността на поле “input” трябва да бъде поле “results”. Нека да зададем име “result” на променливата, която ще представлява всеки отделен елемент на входния масив. Стойността на поле “in” трябва е JSON обект с две полета: годината на провеждане на състезанието и оценката на участника за тази година.

Програмният код на метод aggregate е показан на Фиг. 10.10. Забележете как се адресират полетата от “result” – “$$this.year”, “$$this.points” Резултатът след изпълнение на приложението е показан на Фиг. 10.11.

```
db.collection(collectionName)
    .aggregate(
       [
          {
             $match: {
                results: {
                   $elemMatch: {
                      year: { $in: [2017, 2018] },
                      points: { $gt: 85 }
                   }
                }
             }
          },
          {
             $project: {
                _id: 0, name: 1,
                'results': {
                   '$map': {
                      'input': '$results',
                      'in': {
                        'year': '$$this.year',
                        'points': '$$this.points'
                      }
                   }
                }
             }
          }
        ]
        ).sort( { "name": 1 }
        ).toArray(function(error, docs) {
         if (error) throw error;
         docs.forEach((doc) => {
            let name = doc.name;
            let results = doc.results;
            console.log(`\n${name.firstname} ${name.family}:`);
            results.forEach(result => {
              console.log(`Година: ${ result.year},
                                              точки: ${ result.points}`)
            });
         });
         conn.close();
    });
```

Фиг. 10.10 Примерно решение на Задача 4

```
Добрин Венцеславов:
Година: 2017, точки: 80
Година: 2018, точки: 96

Йордан Валентинов:
Година: 2017, точки: 86
Година: 2018, точки: 92

Мартин Даниел:
Година: 2017, точки: 86
Година: 2018, точки: 92
```

Фиг. 10.11 Резултат - Задача 4

Задача 5: *Създайте Node.js приложение, което разпечатва "Топ 10" на участниците с най висок сумарен резултат от всички състезания. Сортирайте резултата по сумарния брой точки. Работете с колекция "data-2017+2018".*

Използвайте 4 фази на агрегиране, за да решите задачата. При фаза 1 реализирайте филтриране на документите чрез оператор $match. За да е участвал един състезател в конкретно състезание е необходимо поле "status" да има стойност "ok". Фаза 2 трябва да формира изходната колекция чрез оператор $project. Създайте ново поле "total", стойността на което да е сумата на точките, които всеки участник е получил при всички свои участия. Използвайте оператор $sum:

```
'total': { $sum: "$results.points" }
```

При Фаза 3 реализирайте ограничаване на броя на документите в резултата до 10. Използвайте оператор $limit. И накрая, във Фаза 4 сортирайте по стойността на поле "total" чрез оператор $sort.

Резултатът, който трябва да получите, е показан на Фиг. 10.12.

```
1. Йордан Валентинов: 178
2. Мартин Даниел: 178
3. Добрин Венциславов: 176
4. Адриян Ивов: 156
5. Пламен Венелинов: 152
6. Веселин Александров: 146
7. Георги Делянов: 82
8. Йордан Руменов: 75
9. Тихомир Тихомиров: 67
10. Димитър Константинов: 65
```

Фиг. 10.12 Резултат - Задача 5

Задача 6: *Създайте Node.js приложение, което разпечатва имената на участниците които са избрали конкретен език за програмиране, зададен като константа. Сортирайте резултата по имената на участниците. Работете с колекции "data-2017+2018".*

Декларирайте константа "language" чрез която да задавате търсения език за програмиране, например:

```
const language = 'C#';
```

Използвайте 3 фази на агрегиране, за да решите задачата. При Фаза 1 реализирайте филтриране на документите чрез оператор $match. Трябва да останат само документите при които "status" = "ok" и "language" = language. При Фаза 2 използвайте оператор $project, за да укажете, че в изходната колекция трябва да участва само поле "name". Сортирайте чрез оператор $sort по име във Фаза 3.

Резултатът, който трябва да получите, е показан на Фиг. 10.13.

```
Списък на участниците, избрали език за програмиране C#:
1. Валентин Иванов
2. Георги Делянов
3. Емил Йорданов
4. Ивайло Николаев
5. Йордан Руменов
6. Стефан Георгиев
7. Тихомир Тихомиров
```

Фиг. 10.13 Резултат - Задача 6

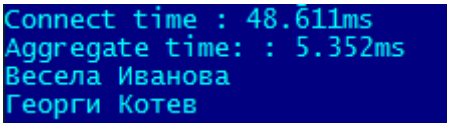
III. Задачи за самостоятелна работа

Задача Д1: *Напишете Node.js приложение което разпечатва имената на всички студенти, които нямат слаба оценка по дисциплината НБД и имат оценка по КМС Добър(4) или Отличен(6). Сортирайте резултата по имената на студентите. Работете с база данни "students", колекция "data2".*

```
const filter = {
  '$and': [
    {
      'grade.НБД.value': { '$gt': 2 }
    },
    {
      'grade.КМС.value': { '$in': [4, 6] }
    }
  ]
};
const projection = {
  '_id': 0, 'name': 1
};
const sort = {
  'name': 1
};
async function startQuery() {
    let client;
    console.time('Connect time ');
    try {
        client = await MongoClient.connect(url, options);
    }
    catch(error) {
        console.log(`Error:\n${error.message}`);
        process.exit();
    }
    finally {
        console.timeEnd('Connect time ');
    }
    const collection = client.db(dbName).collection(collectionName);
    console.time('Aggregate time: ');
    try {
        await collection
            .find(filter, { projection: projection, sort: sort}
            ).toArray(function(error, results) {
                if (error) throw error;
                results.forEach(result => {
                    let firstName = result.name.firstName;
                    let lastName = result.name.lastName;
                    console.log(`${firstName} ${lastName}`);
                });
                client.close();
             });
    }
    catch(error) {
        console.log(error.message);
    }
    finally {
        console.timeEnd('Aggregate time: ');
    }
}
```

Фиг. 10.14 Примерно решение на Задача Д1

За да решим задачата може да използваме само метод **find**. Първият аргумент ще реализира филтриране на документите в колекция “data2”. Тъй като условието се състои от две подусловия ще използваме оператор $and. Вторият аргумент към метод find ще съдържа информация за формирането на изходната колекция (projection) и сортирането (sort). Фиг. 10.14 е показано примерно решение на задачата (необходими константи и метод find). Резултатът, който трябва да получите е показан на Фиг. 10.15. Оценките на Весела са следните: НБД: 5 и КМС: 4, оценките на Георги са: НБД: 3 и КМС: 6 и оценките на Надя са: НБД: 4 и КМС: 5. Вижда се, че оценката на Надя по КМС не отговаря на филтъра.



Фиг. 10.15 Резултат - Задача Д1

Задача Д2: *Напишете Node.js приложение което разпечатва имената на всички участници в GAPCOM, които са избрали език за програмиране C++. Сортирайте резултата по имената на участниците. Работете с колекция “data-2017-2018” от база данни “GAPCOM”.*

Използвайте агрегиране на данните. На Фаза 1 от конвейера задайте желания филтър. При използваната схема за колекция “data-2017-2018” той ще бъде по-сложен отколкото същия филтър за колекция “data-2017+2018” (виж Задача 6). Правилото за филтриране е следното: Върни всички документи за които: 1) поле “status” за 2017 г. е “ok” И поле “language” за 2018 г. е “C++” ИЛИ 2) поле “status” за 2018 г. е “ok” И поле “language” за 2018 г. е “C++”:

```
{
   $match: {
      results: {
         $elemMatch: {
            $or: [
               {
                  $and: [{"2017.status": {$eq: "ok"}},
                         {"2017.language": {$eq: language}}]
               },
               {
                  $and: [{"2018.status": {$eq: "ok"}},
                         {"2018.language": {$eq: language}}]
               }
            ]
         }
      }
   }
}
```

Резултатът, който трябва да получите, е показан на Фиг. 10.16.

```
Списък на участниците, избрали език за програмиране C++:
1. Адриян Ивов
2. Веселин Александров
3. Димитър Константинов
4. Добрин Венциславов
5. Емил Иво
6. Йордан Валентинов
7. Мартин Даниел
8. Никола Мирославов
9. Пламен Венелинов
10. Цвета Огнянова
11. Ясен Пламенов
```

Фиг. 10.16 Резултат - Задача Д2

Задача Д3: *Използвайте приложението за създаване на нова база данни и колекция с документи от Задача 3, Упражнение № 9 и създайте в MongoDB Atlas база данни "sensors", която има една колекция "data" с 10,000 документа, съдържащи информация от сензори за температура и относителна влажност на въздуха. Напишете Node.js приложение което връща информация за показанията на конкретен сензор (зададен с неговия номер и тип) за определен интервал от време (начална дата, крайна дата). Сортирайте резултата в зависимост от датата на измерване на показанията.*

Задачата може да бъде решена с използване на метод **find**. Филтърът, който трябва да се използва, обединява три условия: 1) Стойност на поле "sensorId"; 2) Стойност на поле "type" и 3) Стойност на поле "timestamp". За да зададем интервал за "timestamp" ще използваме оператори $lte и $gte. Сортирането трябва да е стойността на поле "timestamp".

Ще използваме следните константи с цел достъп до MongoDB Atlas:

```
const urlAtlas = "mongodb://username:password@cluster0-shard-00-
00.shlgp.mongodb.net:27017,cluster0-shard-00-
01.shlgp.mongodb.net:27017,cluster0-shard-00-
02.shlgp.mongodb.net:27017/sensor?ssl=true&replicaSet=atlas-lhm95z-
shard-0&authSource=admin&retryWrites=true&w=majority";
const dbName = 'sensors';
const collectionName = "data";
const options = {
    serverSelectionTimeoutMS: 4000,
    connectTimeoutMS: 4000,
    socketTimeoutMS: 4000,
    useUnifiedTopology: true,
};
```

Не забравяйте да зададете коректни стойности за "username" и "password".

Константите, които са необходими, за формиране на заявката и отговора са следните:

```
const sensorId = 15;
const sensorType = 'temperature';
const startDate = new Date("2021-06-01");
const endDate = new Date("2021-06-3");
const dateOptions = {
    year: 'numeric',
    month: '2-digit',
    day: '2-digit'
};
```

Избираме от кой сензор се интересуваме ("sensorId") и какъв е неговия тип ("sensorType"). Задаваме начална ("startDate") и крайна дата ("endDate") за измерванията. Чрез обект

“dateOptions” се задава формата на датата с който искаме да работим: по две цифри за деня и месеца и четири за годината.

Константите, които задават стойността на атрибутите към метод **find** са следните:

```
const filter = {
    "sensorId": sensorId,
    "type" : sensorType,
    "timestamp": {
        "$gte": startDate,
        "$lte": endDate
    }
};
const sort = {
  'timestamp': 1
};
```

Програмният код на метод **makeQuery**, който генерира заявка и обработва отговора на сървъра, е показан на Фиг. 10.17. Започваме с опит за свързване с MongoDB Atlas. Ако това не е възможно на конзолата се разпечатва низа “Няма връзка със сървъра” и се конкретизира причината за това. Причините за това може да се три: 1) Нямате мрежова свързаност и възниква timeout; 2) Грешни име на потребител и/или парола и 3) Невалиден формат на низа “urlAtlas”. Грешките, които може да получите, са следните:

```
Няма връзка със сървъра:
connection <monitor> to 35.157.194.65:27017 closed
```

```
Няма връзка със сървъра:
bad auth : Authentication failed.
```

```
Няма връзка със сървъра:
Invalid connection string
```

При успешно свързване ще получите информация за времето за свързване със сървъра, времето за изпълнение на заявката и отговора на сървъра, например:

```
Connect time : 963.792ms
Find time: : 13.118ms
Записи за сензор 15 за периода [01.06.2021 г. - 03.06.2021 г.]:
  1. Дата 01.06.2021 г.: 38°C
  2. Дата 01.06.2021 г.: 7°C
  3. Дата 01.06.2021 г.: -2°C
  4. Дата 01.06.2021 г.: 6°C
  5. Дата 01.06.2021 г.: 2°C
  6. Дата 01.06.2021 г.: 27°C
  7. Дата 02.06.2021 г.: 35°C
  8. Дата 02.06.2021 г.: 25°C
  9. Дата 02.06.2021 г.: 1°C
 10. Дата 02.06.2021 г.: -7°C
```

```
async function makeQuery() {
    let client;
    console.time('Connect time ');
    try {
        client = await MongoClient.connect(urlAtlas, options);
    }
    catch(error) {
        console.log(`Няма връзка със сървъра:\n${error.message}`);
        process.exit();
    }
    finally {
        console.timeEnd('Connect time ');
    }
    const coll = client.db(dbName).collection(collectionName);
    console.time('Find time: ');
    try {
        await coll
            .find(filter,{ sort: sort})
            .toArray(function(error, records) {
                if (error) throw error;
                let index = 1;
                let ts1 = startDate.toLocaleString('bg-bg', dateOptions);
                let ts2 = endDate.toLocaleString('bg-bg', dateOptions);
                console.log(`Записи за сензор
                                ${sensorId} за периода [${ts1} - ${ts2}]:`);
                if (records.length != 0) {
                    records.forEach((record) => {
                        let unit =
                          record.type === 'temperature' ? '\u00B0C' : '%';
                        let ts =
                    record.timestamp.toLocaleString('bg-bg', dateOptions);
                        let id = String(index++).padStart(3,' ');
                        console.log(`${id}.
                                    Дата ${ts}: ${record.value}${unit}`);
                    });
                }
                else {
                    console.log(`Няма данни за сензор номер ${sensorId}.`);
                }
                client.close();
            });
    }
    catch(error) {
        console.log(error.message);
    }
    finally {
        console.timeEnd('Find time: ');
    }
}
```

Фиг. 10.17 Програмна реализация на Задача Д3

В програмния код е използван метод **toLocaleString** с цел конвертиране и форматиране по шаблон на дата до низ. За разпечатване на °C е използван уникод `\u00B0C'`. За подравняване на номерата на показанията е използван метод **padStart**, който в конкретния случай вмъква в номера на показанието интервали, за да изравни от дясно номерата.

Задача Д4: *Напишете Node.js приложение което връща информация за броя на записите от конкретен сензор (идентификатор и тип). Работете с MongoDB Atlas, база данни "sensors", колекция "data".*

Използвайте метод **aggregate** за агрегиране на съдържание. При Фаза 1 на конвейера реализирайте филтрирането на записите чрез оператор $match (полета "sensorId" и "type"). От Фаза 2 реализирайте изброяването на филтрираните документи чрез оператор $count.

Резултатът, който трябва да получите, трябва да е подобен на следния:

```
Connect time : 1.081s
Find time: : 12.596ms
Брой записи за сензор 5 (temperature): 262
```

Задача Д5: *Напишете Node.js приложение което връща статистика (минимална, максимална и средно аритметична стойност) за показанията на конкретен сензор (идентификатор и тип). Работете с MongoDB Atlas, база данни "sensors", колекция "data".*

Използвайте метод **aggregate** за агрегиране на съдържание. При Фаза 1 на конвейера реализирайте филтрирането на записите чрез оператор $match (полета "sensorId" и "type"). Чрез Фаза 2 реализирайте групиране на показанията на избрания сензор чрез оператор $group. За да изчислите минималната, максималната и средно-аритметичната стойност на показанията използвайте оператори $min, $max и $avg, например:

```
"$group":
{
   "_id" : null,
   "maxVal" : { "$max" : "$value" },
   "minVal" : { "$min" : "$value" },
   "avgVal" : { "$avg" : "$value" },
}
```

Резултатът, който трябва да получите, трябва да е подобен на следния:

```
Connect time: : 911.75ms
Find time: : 12.791ms
Статистика за сензор 5 (temperature):
Min: -10°C,Max: 38°C,Average: 15.14°C
```

```
Connect time: : 999.184ms
Find time: : 9.719ms
Статистика за сензор 5 (humidity):
Min: 20%,Max: 99%,Average: 59.41%
```

За да ограничите показанията за средната стойност до втория знак след десетичната точка използвайте метод **toFixed**.

Упражнение № 11

Достъп до MongoDB с Express.js

I. Въведение

Express.js или само Express е сървърна рамка за Web приложения базирани на Node js. Тя позволява бързо изграждане на приложения с една страница - Single page Apps (SPA), приложения с множество страници, както и хибридни Web приложения. Express поддържа повечето популярни JavaScript стекове за разработка, например MEAN (MongoDB, Express, AngularJS, Node.js), MERN (MongoDB, Express, React, Node.js) и MEVN (MongoDB, Express, Vue, Node.js). Вижда се, че разликата е в рамката, реализираща потребителския интерфейс (AngularJS, React или Vue).

Express предоставя механизми за:

- Обслужване на различни видове HTTP(S) заявки (GET, POST и PUT) чрез пренасочването им към обслужващи функции чрез техните URL пътища (маршрути).
- Възможност за интегриране на двигател за визуализиране на "изгледи", за да се генерират отговори чрез вмъкване на данни в шаблони.
- Настройки на сървъра, обслужващ заявките.
- Добавяне на допълнителен междинен софтуер (middleware) за обработка на заявки във всяка точка от конвейера за обработка на заявки.

Макар че самият Express е с минималистичен дизайн, разработчиците са създали съвместими пакети за междинен софтуер, за да се справят с почти всеки проблем при разработката на Web приложения. Съществуват библиотеки за работа с бисквитки, сесии, следене на потребителски влизания, извличане на параметри, предавани към ресурс и много други. Функциите на междинния софтуер са функции, които имат достъп до обекта на заявката “request”, обекта на отговора “response”, както и до следващата функция на междинния софтуер в цикъла “заявка-отговор” на приложението. Функциите на междинния софтуер могат да изпълняват следните задачи:

- Изпълняват всякакъв код.
- Да правят промени в обектите на заявката и отговора.
- Да приключат цикъла “заявка-отговор”.
- Извикване на следващата функция на междинния софтуер в стека.

Едно Express приложение може да използва следните типове междинен софтуер:

- Междинен софтуер на ниво приложение.
- Междинен софтуер на ниво маршрутизатор.
- Междинен софтуер за обработка на грешки.
- Вграден междинен софтуер.
- Междинно програмно осигуряване от трети страни.

Междинното програмно осигуряване на ниво приложение се свързва с инстанцията на обекта, описващ Express (**app**), като се използват функциите app.**use** и app.**method**, където “method” е HTTP методът, използван при HTTP заявката, например **get**, **post** или **put**. Следва пример, при който се обслужват “сляпа” get заявки:

```
var express = require('express');
var app = express();
...
app.get('/', (request, response, next) => {
  response.send('Отговор на заявката, най-често HTML код.')
});
```

Зареждането на модул Express се реализира чрез метод **require**. Инстанция на обекта за достъп до Express **app** се получава чрез метод **express**. При сляпата GET заявка не се задава конкретен ресурс, който да обслужи заявката – първият аргумент е "/".

Обработка на грешки също се реализира чрез междинен софтуер. Междинният софтуер за обработка на грешки винаги приема четири аргумента, например:

```
app.use((error, request, response, next) => {
  console.error(error.stack)
  response.status(500).send(Информация за грешката.')
})
```

Код 500 за статус на отговора отговаря на вътрешна сървърна грешка.

Express има следните вградени функции за междинен софтуер:

- "express.static" обслужва статични ресурси, като HTML файлове и изображения.
- "express.json" анализира входящите заявки с JSON съдържание. Налична е при Express 4.16.0+.
- "express.urlencoded" анализира входящите заявки със съдържание кодирано чрез URL кодер. Налична е при Express 4.16.0+.

Много често се налага използване на междинен софтуер на трети страни с цел получаване на липсваща, но желана функционалност. Първо трябва да инсталирате модула за необходимата функционалност след което трябва да го заредете в приложението си на ниво приложение или на ниво маршрутизатор. Следва пример за използване на парсер на HTTP заявки:

```
const bodyParser= require('body-parser');
...
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
```

Информация за най-често използвания междинен софтуер на трети страни можете да получите от следната страница:

https://expressjs.com/en/resources/middleware.html

II. Последователност за създаване на приложения с използване на Express

1. Създайте папка в която ще създадете своя проект, например MongoWithExpress. Стартирайте cmd.exe така, че текущата папка да е проектната папка. Създайте нов проект чрез команда

npm init

Задайте последователно: име на проекта; версия на проекта; описание на проекта; входна точка (например index.js); git хранилище (не е задължително); ключови думи; автор и тип на лицензия. В резултат на командата ще бъде създаден файл **package.json**, който съдържа информацията за проекта, въведена до момент, например:

```
{
  "name": " MongoWithExpress",
  "version": "1.0.0",
  "description": "Generate CRUD requests using Express",
  "main": "server.js",
  "dependencies": {
    "mongodb": "^3.6.9"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
  "author": "rs-soft-bg",
  "license": "ISC"
}
```

2. Инсталирайте Express. Express е програмна рамка за back-end Web приложения базирани на Node.js. По своята същност Express позволява стартиране на един или множество сървъри, обслужващи заявките на клиентите. Клиентите могат да бъдат както хора, така и smart модули. За да инсталирате Express изпълнете следната команда:

npm install express --save

Командата ще прехвърли в проекта ви всички файлове, които изграждат Express и ще обнови съдържанието на файл package.json, поле “dependencies”.

3. Създайте в проектната папка файл server.js. Той ще съдържа програмния код на сървъра чрез който ще обслужваме заявките на клиентите. Първо ще тестваме дали сървъра работи като въведем кода чрез който се стартира сървър, който подслушва зададен свободен порт и функция чрез която се обработват HTTP GET заявки:

```
const express = require('express');
const app = express();

app.listen(8080, () => {
  console.log('Start listening on port 8080...')
});
app.get('/', (request, response) => {
  response.send('Здравей!');
});
```

Първо, ще заредим Express като модул чрез метод **require**. Следва създаване на инстанция на програмната рамка с референция **app**. Стартирането на сървъра се реализира чрез метод **listen**. Методът изисква два аргумента: номера на порта, който се подслушва (8080) и ламбда функция, която се активира при нова заявка към сървъра. Всяка заявка, в зависимост от типа й (GET, POST, PUT), се пренасочва за обработка чрез съответния й метод. На този етап сървърът обслужва само “слепи” GET заявки (не е зададено име на ресурс) чрез метод **get**. Първият аргумент е URI (‘/’), а втория е ламбда функция, която обслужва GET заявките. На функцията се предават два обекта: request (достъп до входящия комуникационен канал - параметрите на заявката) и response (достъп до изходящия комуникационен канал). На този етап сървърът връща чрез метод **send** низа “Здравей!”.

4. Стартирайте сървъра чрез командата:

node server.js

След като на конзолата се появи низа “Start listening on port 8080...” стартирайте браузър и изпратете “сляпа” GET заявка към сървъра:

http://localhost:8080

Ако всичко е наред трябва да видите низа “Здравей!”.

5. Ако промените кода на сървъра трябва да го спрете чрез Ctrl+C и рестартирате чрез node. За да автоматизирате този процес може да използвате инструмента **Nodemon**. Той следи промените в кой да е файл от сървърната страна и при всяка промяна рестартира сървъра автоматично. Инсталирайте Nodemon чрез следната команда:

npm install nodemon --save-dev

Тук използваме флаг “--save-dev”, защото ще използваме Nodemon само в процеса на разработка на приложението. Този флаг ще вмъкне информация за версията на Nodemon в поле “devDependency” във файл package.json, например:

```
"devDependencies": {
    "nodemon": "^2.0.12"
}
```

6. Ще създадем HTML форма в index.html за изпращане на заявки към сървъра. Нека заявката да проверява кои студенти имат определена оценка по зададена от потребителя дисциплина. Програмния код, който реализира това, е следния:

```
<form action="/query" method="POST">
    <input type="text" placeholder="Въведи дисциплина" name="subject">
    <input type="text" placeholder="Въведи оценка" name="grade">
    <button type="submit">Изпрати заявката</button>
</form>
```

Ще използваме HTTP POST заявка до ресурс “query”, за да изпращаме запитванията към базата данни. В тази форма се изисква въвеждане на две стойности: име на дисциплината “subject” и оценка “grade”.

7. За да формираме заявка към MongoDB сървъра трябва да прочетем стойностите на полета “subject” и “grade”, които се изпращат чрез POST заявката. За да реализираме това ще използваме модул “**body-parser**”. Инсталацията му се реализира чрез следната команда:

npm install body-parser –save

За да използваме този модул, ще въведем следната информация във файл server.js:

```
const bodyParser= require('body-parser');
...
app.use(bodyParser.urlencoded({ extended: true }));
// Следва обработка на GET и POST заявки.
...
app.post('/query', (request, response) => {
    let subject = request.body.subject;
    let grade = request.body.grade;
    response.send(`${subject}: ${grade}`);
});
```

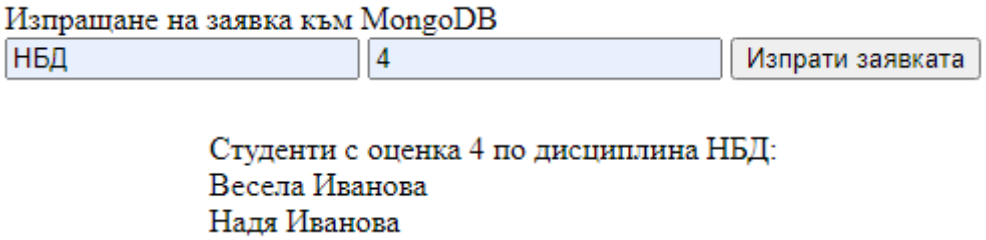
Чрез метод **urlencoded** се задава данните, предавани чрез HTML форми, да се декодират и добавят към свойството “body” в обекта на заявката (request). Чрез метод **post** се прехваща POST заявката към ресурс “query”. От тялото на ламбда функцията се извличат стойностите на двете полета от HTML формата (“subject” и “grade”). За да тестваме работата на метод post връщаме прочетените параметри на браузъра чрез метод send.

8. Вече можем да генерираме заявки към MongoDB сървъра. На практика се създават определен брой комуникационни канали към MongoDB (pool) и постъпващите заявки се разпределят да използват някои от тези канали. За да опростим задачата ще създадем само един комуникационен канал към сървъра. Това се реализира чрез метод **connect**. От тялото на функцията, която се активира при успешно свързване ще реализираме методите, които прехващат заявките от страна на браузъра (get и post). Чрез метод **get** ще обработваме слепите GET заявки, а чрез метод **post** – данните от HTML формата. На Фиг. 11.1 е показан примерен код, който реализира свързване с MongoDB сървъра и изпращане на заявка чрез метод **find**. В случая се търсят имената на студентите, които имат точно определена оценка по избрана дисциплина. Програмният код използва следните константи:

```
const url = 'mongodb://localhost:27017';
const dbName = 'students';
const collectionName = "data1";
const options = {
    serverSelectionTimeoutMS: 3000,
    connectTimeoutMS: 3000,
    socketTimeoutMS: 3000,
    useUnifiedTopology: true,
};
```

Ще работим с локално инсталиран сървър, база данни с име “students” и колекция “data1”. Чрез JSON обекта options се задава необходимите timeout времена. След изпълнение на метод connect се получава обект “client”. Първо, получаваме обект db за достъп до базата данни. След това получаваме обект “collection” за достъп до желаната колекция. Използваме метод **urlencoded** с цел достъп до параметрите, предавани от HTML формата. Следва програмния код на callback методи get и post. Метод **get** връща съдържанието на файла index.html. От тялото на метод **post** се получава стойностите на избраната дисциплина и оценка (“qSubject” и “qGrade”). Следва дефиниране на три константи, които се използват като аргументи на метод **find** (filter, projection и sort). За метод find е зададено резултатът да се преобразува до масив (метод **toArray**).

От тялото на **then** прочитаме резултата от сървъра (обект docs) и формираме резултата, който да върнем на браузъра. Това се реализира чрез метод **send**. Следва пример за конкретна заявка към базата данни и отговора, който приложението връща:



```
MongoClient.connect(url, options)
    .then(client => {
        const db = client.db(dbName);
        const collection = db.collection(collectionName);

        app.use(bodyParser.urlencoded({ extended: true }));

        app.get('/', (request, response) => {
          response.sendFile(__dirname + '/index.html')
        });
        app.post('/query', (request, response) => {
            let qSubject = request.body.subject;
            let qGrade = request.body.grade;
            const filter = {
              grade: {
                subject: `${qSubject}`,
                value: parseInt(`${qGrade}`)
              }
            };
            const projection = { _id: 0, name: 1 };
            const sort = { name: 1 };

            collection
                .find(filter, { projection: projection, sort: sort })
                .toArray()
                .then(docs => {
                    let result =
                    `Студенти с оценка ${qGrade} по дисциплина
                                                ${qSubject}:<br/>`;
                    if (docs.length == 0) {
                        response.send(result + 'Няма');
                    }
                    else {
                        docs.forEach((doc) => {
                            let name = doc.name;
                            result +=
                              `${name.firstName} ${name.lastName}<br/>`;
                        });
                        response.send(result);
                    }
            })
            .catch(error => {
                console.log(error.message);
            });

        });

        // listening on server port
        app.listen(SERVER_PORT, () => {
            console.log(`Start listening on port ${SERVER_PORT}...`)
        });
    })
    .catch(error => {
        console.log(error.message);
    });
```

Фиг. 11.1 Програмен код на метод connect

На Фиг. 11.2 е показан програмния код от файл server.js (без метод connect).

```
const express = require('express');
const bodyParser= require('body-parser');
const MongoClient = require('mongodb').MongoClient;
const app = express();

const url = 'mongodb://localhost:27017';
const SERVER_PORT = 8080;

const dbName = 'students';
const collectionName = "data1";
const options = {
    serverSelectionTimeoutMS: 3000,
    connectTimeoutMS: 3000,
    socketTimeoutMS: 3000,
    useUnifiedTopology: true,
};
// Следва метод connect: свързване към базата данни, генериране на заявка и формиране на отговор (виж Фиг. 11.1).
```

Фиг. 11.2 Програмен код – файл server.js

Програмният код на файл index.html е показан на Фиг. 11.3.

```
<!DOCTYPE html>
<html lang="en">
    <head>
      <meta charset="UTF-8">
      <title>Query MongoDB</title>
    </head>
    <body>
        <div id="container">
            <header>
                Изпращане на заявка към MongoDB
            </header>
            <section data-position="query">
            <form action="/query" method="POST">
                <input type="text"
                        placeholder="Дисциплина" name="subject">
                <input type="text" placeholder="Оценка" name="grade">
                <input type="submit" value="Изпрати заявката"></input>
            </form>
            </section>
        <div>

    </body>
</html>
```

Фиг. 11.3 Програмен код – файл index.html

В този си вид приложението има следните по-важни недостатъци:

- При връщане на отговор се отваря нова HTML страница. Това налага връщане към страницата index.html, за да генерираме нова заявка.
- Липсва стилово форматиране.

Добре би било да можем да интегрираме отговора на MongoDB сървъра в страницата с която се генерира заявката. За целта е необходимо да можем да добавяме динамично съдържание към HTML файл. Това се реализира чрез така наречените страници-шаблони. Шаблонът съдържа статичен HTML код, който не се променя и динамичен код,

който се променя динамично. Съществуват множество двигатели за шаблони, например: Embedded JS (EJS), Handlebars, Nunjucks и др.

Ще използваме шаблон **EJS**, тъй като синтаксисът е много подобен на Java Server Pages (JSP). Чрез него ще може да комбинирате HTML и JavaScript код, както и да интегрирате стойностите на променливи и обекти в HTML страниците. За да инсталирате модул EJS използвайте следната команда:

```
npm install ejs --save
```

След това трябва да зададем изгледа да се формира от EJS:

```
app.set('view engine', 'ejs');
```

Вече можете да използвате EJS за динамично вмъкване на съдържание в HTML документ. Всички документи, които използват EJS трябва да са с разширение ejs. Създайте папка views и запишете в нея индексния файл index.htm. След това променете разширението на файла до ejs. Ще използваме метод **render**, който е част от Express, с цел визуализиране на съдържанието на ejs файла. Синтаксисът е следния:

```
response.render('index.ejs', { results: data });
```

Метод render предава на EJS модул index.ejs JSON обект. Може да извлечете стойността на полетата от JSON обекта чрез използване на EJS израз, например:

```
<%= results.title %>
```

На мястото на <%= … %> в HTML кода се вмъква стойността на поле “title” от обект “results”. Ако трябва да интегрирате JavaScript код в HTML страница използвайте <% код %>.

За да форматираме стилово HTML страницата ще напишем необходимия CSS код. Всички допълнителни модули ще запишем в папка public. Създайте тази папка и вмъкнете в нея файла style.css. Трябва да зададем на Express да направи тази папка достъпна:

```
app.use(express.static('public'));
```

Задайте в хедъра на index.html и index.ejs, че ще се използва файл style.css с цел стилово форматиране:

```
<link rel="stylesheet" href="/styles.css" />
```

На Фиг. 11.4 е показан програмния код на метод **connect**. Новият код е на жълт фон. При успешно получаване на резултат се формира JSON обект results, който съдържа следните полета: status (при стойност “ok” заявката е изпълнена успешно); title (информация какво представлява заявката) и docs (резултат от сървъра). Ако заявката не може да бъде изпълнена от тялото на клауза catch отново се създава обект results, но в този случай той съдържа следните полета: status със стойност “fail” и info (съдържа текстово описание на грешката).

```
MongoClient.connect(url, options)
    .then(client => {
        const db = client.db(dbName);
        const collection = db.collection(collectionName);

        app.set('view engine', 'ejs');
        app.use(bodyParser.urlencoded({ extended: true }));
        app.use(bodyParser.json());
        app.use(express.static('public'));

        app.get('/', (request, response) => {
          response.sendFile(__dirname + '/index.html')
        });
        app.post('/query', (request, response) => {
            let qSubject = request.body.subject;
            let qGrade = request.body.grade;
            console.log(`${qSubject}: ${qGrade}`);
            const filter = {
              grade: {
                subject: `${qSubject}`,
                value: parseInt(`${qGrade}`)
              }
            };
            const projection = { _id: 0, name: 1 };
            const sort = { name: 1 };
            collection
                .find(filter, { projection: projection, sort: sort } )
                .toArray()
                .then(docs => {
                    let title =
                    `Студенти с оценка ${qGrade} по дисциплината
                                                    ${qSubject}:`;
                    let results = {title:title,docs:docs,status:"ok"};
                    response.render('index.ejs', { results: results });
            })
            .catch(error => {
                let results = {info:`Грешка:${error.message}`,
                                                    status: "fail" };
                response.render('index.ejs', { results: results });
            });

        });
        // Listening on server port
        app.listen(SERVER_PORT, () => {
            console.log(`Start listening on port ${SERVER_PORT}...`)
        });
    })
    .catch(error => {
        console.log(error.message);
    });
```

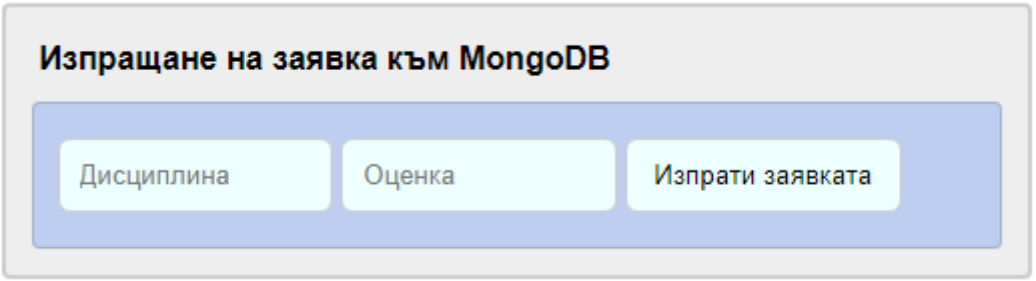
Фиг. 11.4 Програмен код на метод connect

На Фиг. 11.5 е показан програмния код от файл index.ejs. Съдържанието, което се визуализира, е в тялото на контейнер. При стойност на поле “status” е “ok” се обхождат всички документи от поле docs чрез метод **forEach**. При стойност “fail” на поле “status” се визуализира информация за грешката (параграф от клас “error”).

```
<!DOCTYPE html>
<html lang="en">
     <head>
        <meta charset="UTF-8">
       <title>Query MongoDB</title>
       <link rel="stylesheet" href="/styles.css" />
     </head>
     <body>
        <div id="container">
           <header>
              Изпращане на заявка към MongoDB
           </header>
           <section data-position="query">
              <form action="/query" method="POST">
                 <input type="text" placeholder="Дисциплина"
                                                      name="subject">
                 <input type="text" placeholder="Оценка" name="grade">
                 <input type="submit" value="Изпрати заявката"></input>
              </form>
           </section>
           <section data-position="results">
              <% if(results.status == "ok") { %>
                 <header><%= results.title %></header>
                 <ul class="results">
                    <% results.docs.forEach(doc => { %>
                       <li class="results">
                          <%= doc.name.firstName%> <%= doc.name.lastName%>
                       </li>
                    <%}); %>
                 </ul>
                 <% }
                 else { %>
                    <p class="error"><%= results.info %></p>
                 <%} %>
           </section>
        </div>
     </body>
</html>
```

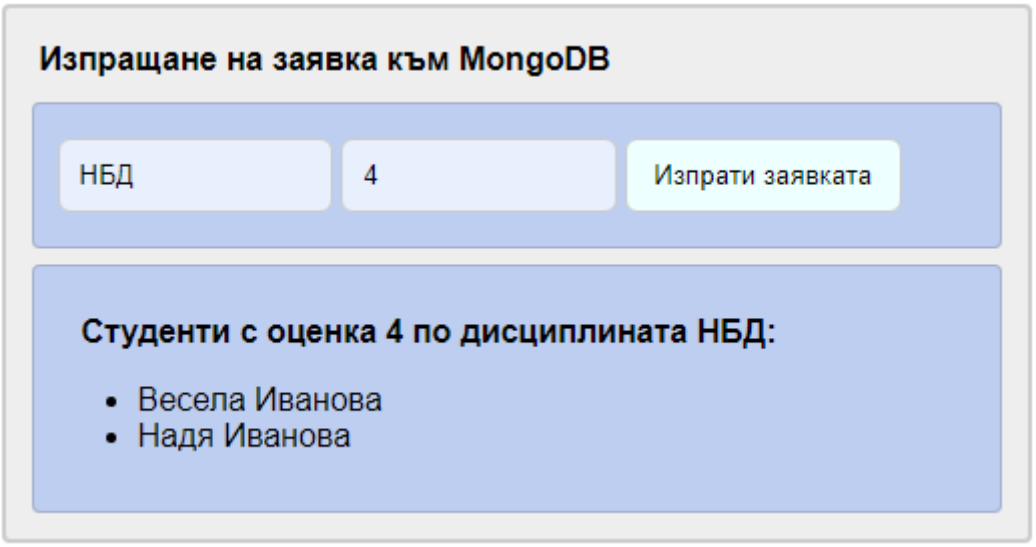
Фиг. 11.5 Програмен код– файл index.ejs

Стартирайте server.js и от браузъра изпълнете сляпа GET заявка (http://localhost:8080). Тя ще предизвика зареждане на съдържанието на файл index.html (виж Фиг. 11.6).



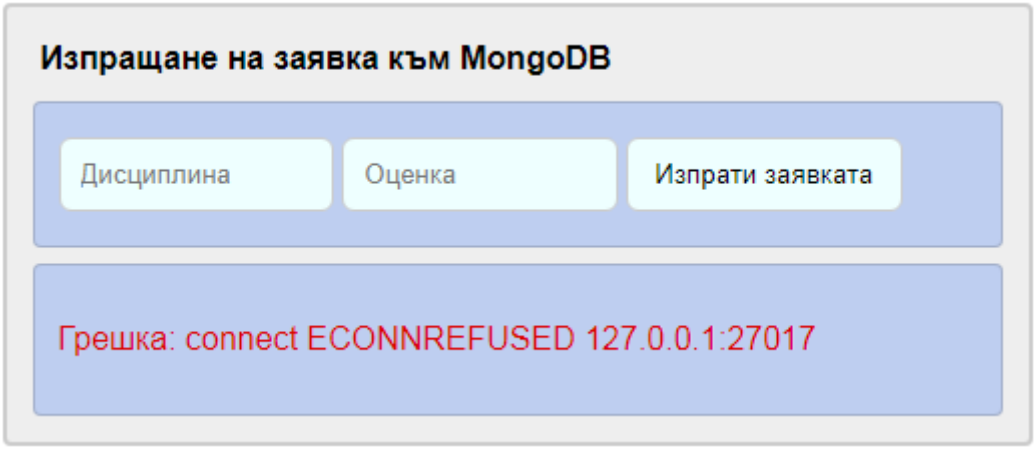
Фиг. 11.6 Изглед след сляпа GET заявка

Въведете двойката дисциплина и оценка и натиснете бутон “Изпрати заявката”. Трябва да бъдат намерени и разпечатани сортирано имената на студентите, които имат зададената оценка за зададената дисциплина (виж Фиг. 11.7).



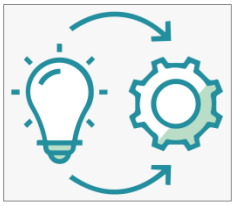
Фиг. 11.7 Изглед след успешно изпълнена POST заявка

Спрете временно MongoDB сървъра и генерирайте отново заявката. Трябва да получите резултат, подобен на този, показан на Фиг. 11.8.



Фиг. 11.8 Изглед след неуспешно изпълнена POST заявка

III. Задачи за изпълнение



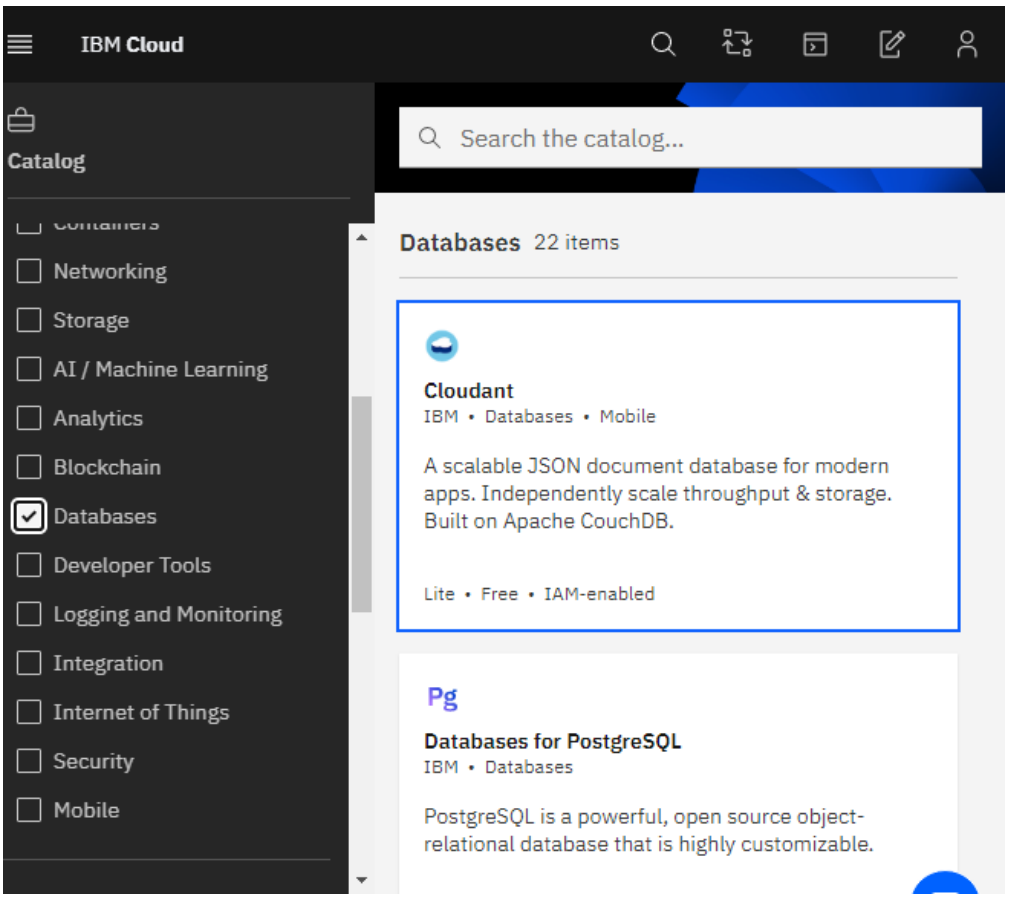
Задача 1: *Въведете във файл style.css необходимите CSS правила така, че изгледът на приложението да имитира показания на Фиг. 11.7 и Фиг. 11.8 интерфейс с потребителя.*

Упражнение № 12

IBM Cloudant – инсталации и работа с документи

I. Въведение

IBM Cloudant е NoSQL документен тип база данни. Достъпът до тази база данни се реализира чрез протокол HTTPS. Има опция да използвате IBM Cloudant безплатно. За целта трябва да се регистрирате като клиент на IBM Cloud. След като се регистрирате, запомнете своята парола. След успешна авторизация на достъпа (login) създайте нов ресурс. Натиснете бутон “Create resource”, изберете “Databases”, а след това Cloudant. Изберете безплатен абонаментен план и задайте име на ресурса.



Фиг. 12.1 Избор на ресурс Cloudant

Попълнете полетата от меню “Create”. Изберете “Frankfurt” или “London” за “Available regions”. За поле “Authentication method” изберете “IAM and legacy credentials”. Този избор ще позволи отдалечен достъпа до базите данни чрез базова авторизация. И накрая, изберете безплатен достъп до услугата – план “Lite”. При този план ограниченията са следните:

- 20 четения/сек.
- 10 записа/сек.
- 5 глобални заявки/сек.
- 1 GB място за съхранение

За да активирате услугата натиснете бутон “Create”. При успешно активиране ще бъдете препратени към страницата с вашите ресурси. Изчакайте докато статуса на услугата стане “”Active”.

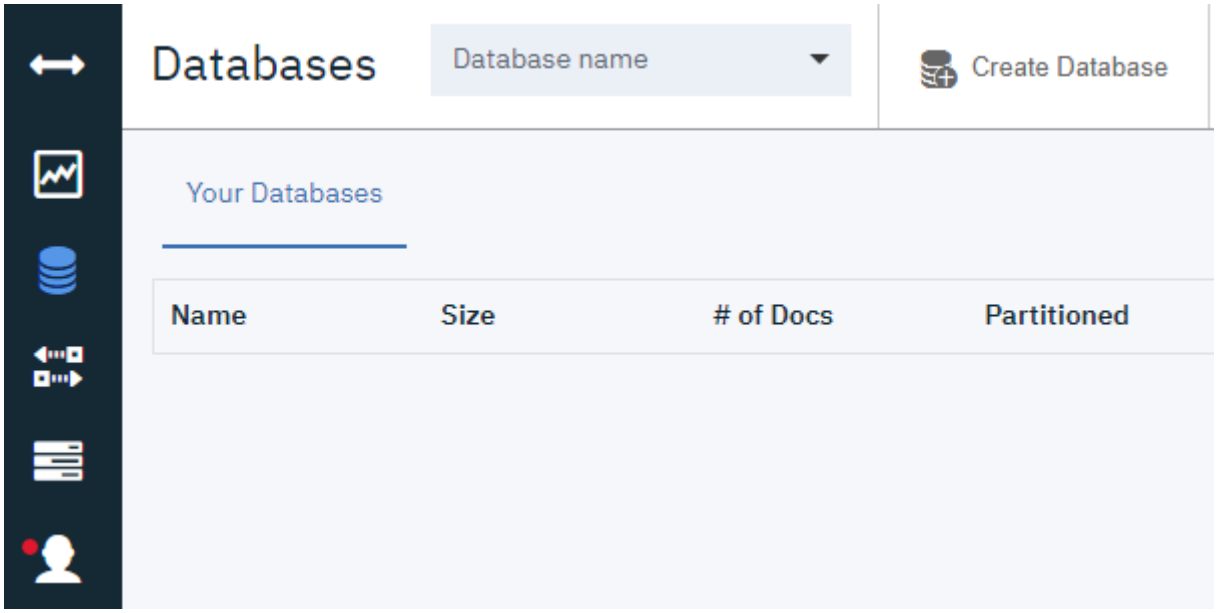
Изберете услугата, за да достигнете до менюто за настройки. Изберете “Manage”. Копирайте URL низа, съответстващ за “External Endpoint (preferred)”, например:

https://71176003-6559-48ff-95d9-9ff0384b35d7-bluemix.cloudantnosqldb.appdomain.cloud

Чрез този адрес ще достъпвате вашата база данни отдалечено чрез HTTPS клиент.

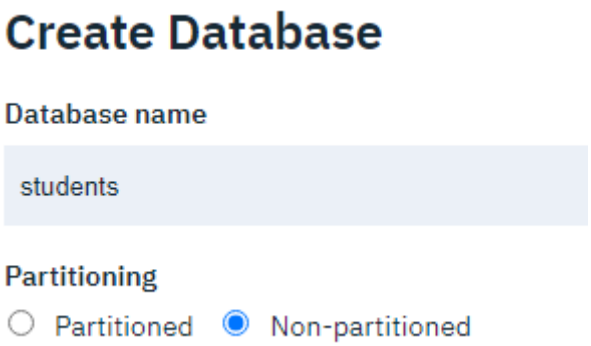
Изберете “Service credentials” и натиснете бутон <New credential +>. Задайте име на удостоверяването и “Manager” за роля. Натиснете бутон <Add>.

Натиснете бутон <Launch Dashboard>. Така ще достигнете до графичния интерфейс на IBM Cloudant чрез който може да управлявате своите бази данни (виж Фиг. 12.2).



Фиг. 12.2 IBM Cloudant dashboard

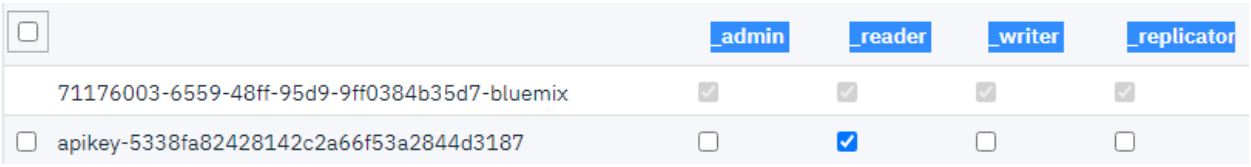
Вече сте готови да създадете своята първа база данни. За целта натиснете бутон <Create Database>. Изберете име на базата данни, например “students” и забранете разделянето на базата данни между множество нодове (сървъри) – “Non partitioned”.



Фиг. 12.3 Създаване на нова база данни

Натиснете бутон <Create>. Вече имате база данни, но в нея все още няма документи. Преди да създадем необходимите документи ще зададем необходимите разрешения за

достъп до услугата. За целта от менюто изберете “Permissions” и натиснете бутон <Generate API Key>. Ключовете са необходими, за да се предостави програмен достъп до базите данни на Cloudant. Можете да генерирате толкова ключа, колкото са ви необходими, например: един за администратора на базите данни, един или повече за четене и запис и един или няколко само за четене. Системата ще генерира двойка “ключ-стойност”. Трябва да го копирате и запишете, за да може да го използвате на по-късен етап. За всяка двойка “ключ-стойност” може да зададете различни права за достъп (виж Фиг. 12.4).



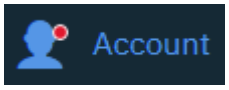
Фиг. 12.4 Задаване на права за достъп (permissions)

За да въведете документ в базата данни “students” натиснете бутон <Create document>. Документът ви автоматично получава идентификатор чрез системното поле “_id”. Може да зададете нова стойност на полето, но то трябва да е уникално спрямо останалите документи. Нека да въведем в документа информация за студентите от MongoDB базата данни “students”, колекция “data1”. Чрез JSON редактора на Cloudant въведете информацията за един студент (виж Фиг. 12.5). Натиснете бутон <Create Document> с което ще запишете документа в база данни “students”.

```
{
  "_id": "student-0001",
  "id": "21705001",
  "name": {
    "firstName": "Георги", "lastName": "Котев"
  },
  "grade": [
    {
      "subject": "НБД", "value": 4
    },
    {
      "subject": "КМС", "value": 2
    },
    {
      "subject": "ММС", "value": 4
    },
    {
      "subject": "ДСП", "value": 3
    }
  ]
}
```

Фиг. 12.5 Създаване на нов документ

След като запишете документа, той получава още едно системно поле “_rev”. То съдържа информация за версията на документа и не трябва да бъде манипулирано. Това поле се използва за системни нужди (репликация на базата данни). По аналогичен начин създайте още два записа в базата данни. Вече имате база данни с три документа в нея. Остава да разрешим отдалечения достъп до тази база данни. За целта от вертикалното меню изберете “Account”:

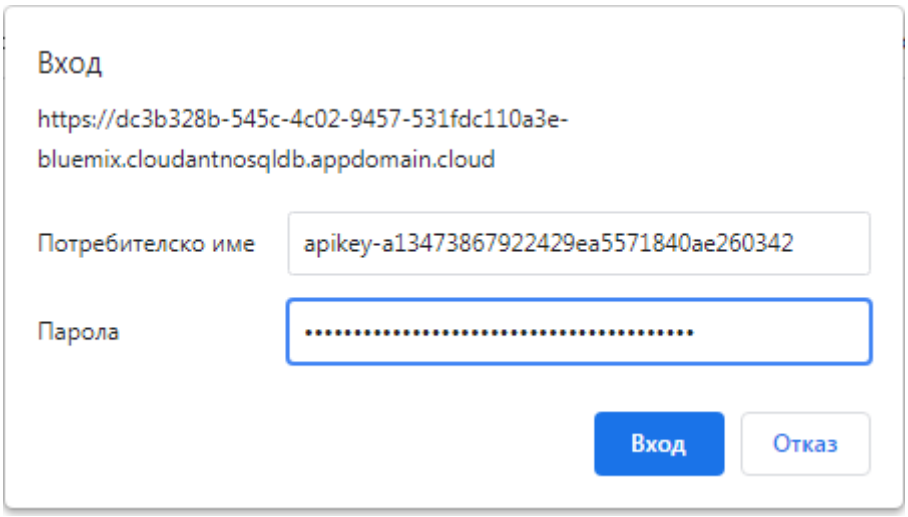


От хоризонталното меню изберете Cross-Origin Resource Sharing (CORS). Споделянето на ресурси от различни източници (CORS) ви позволява да се свързвате с отдалечени сървъри директно от браузъра. Натиснете бутон <Enable CORS>. От “Origin domains” изберете “All Domains (*)”. По този начин достъпът до вашите бази данни ще е разрешен от всеки IP адрес. Защитата на връзката се реализира чрез протокол HTTPS и избрания механизъм за автентикация на достъпа. По подразбиране се използва базова автентикация при която ключа и паролата се кодират чрез алгоритъм Base64.

Тествайте достъпа до услугата като от браузъра направете заявка до един от документите в база данни “students”, например:

https://dc3b328b-545c-4c02-9457-531fdc110a3e-bluemix.cloudantnosqldb.appdomain.cloud/students/student-0001

Тъй като достъпът изисква базова авторизация чрез име и парола, браузърът ще визуализира прозорец за тяхното въвеждане (виж Фиг. 12.6).

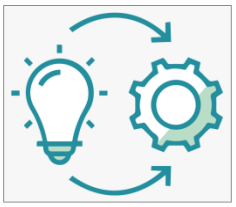


Фиг. 12.6 Форма за авторизация на достъпа

След натискане на бутон <Вход> се визуализира съдържанието на избрания документ:


```
{"_id":"student-0001","_rev":"1-
9c8f87b7104684e21c832ffae329fd69","id":"21705001","name":
{"firstName":"Георги","lastName":"Котев"},"grade":
[{"subject":"НБД","value":4},{"subject":"КМС","value":1},
{"subject":"ММС","value":1},{"subject":"ДСП","value":1}]}
```

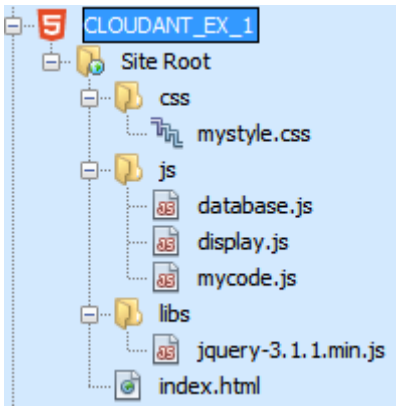
II. Задачи за изпълнение



Задача 1: *Направете своя регистрация в IBM Cloud. Следвайте стъпките за получаване на достъп до ресурса IBM Cloudant. Създайте база данни "students" и запишете поне три документа, които описват студенти и техните оценки.*

Задача 2: *Напишете Web приложение, което реализира връзка с IBM Cloudant, база данни "students" и визуализира информация за всеки студент за който има запис в базата данни.*

Създайте нов HTML5 NetBeans проект с име CLOUDANT_EX_1. Структурата на приложението е показана на Фиг. 12.7.



Фиг. 12.7 Структура на проекта – Задача 2

Индексният файл (**index.html**) съдържа HTML5 кода чрез който се изгражда изгледа на приложението. Информацията, което се визуализира е следната:

Горен колонтитул (header)
Информация В даден момент от време се визуализира съдържанието на една от четири възможни секции.
Долен колонтитул (footer)

За да реализираме SPA приложение (с една HTML5 страница) ще използваме няколко HTML секции, като в даден момент ще визуализираме само една от тях (другите ще бъдат скрити). На Фиг. 12. 8 е показано съдържанието на файл index.html. За да използваме кирилица сме задали да бъде активна кодова таблица UTF-8. Заредени са и всички JavaScript и CSS модули, необходими за работа на приложението. Информацията, която се визуализира е в тялото на етикет <main>. Тя е разпределена в 4 секции със следното предназначение:

- Секция "loading-box" – показва низа "Свързване…" при всяко начало на мрежов обмен.
- Секция "query-box" – показва бутон чрез който се реализира изпращане на заявка към IBM Cloudant.
- Секция "info-box" – показва форматираното съдържание на получените документи.
- Секция "error-box" – показва информация за грешката, ако такава възникне.

```
<!DOCTYPE html>
<html>
    <head>
        <title>IBM Cloudant</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width,
                                       initial-scale=1.0">
        <link rel="stylesheet" type="text/css" href="css/mystyle.css"/>
        <script src="libs/jquery-3.1.1.min.js"></script>
        <script src="js/database.js"></script>
        <script src="js/display.js"></script>
        <script src="js/mycode.js"></script>
    </head>
    <body>
        <header>IBM Cloudant Query</header>
        <main>
            <section data-position="loading-box">
                Свързване ...
            </section>
            <section data-position="query-box">
                <button id="submit">Изпрати заявка</button>
            </section>
            <section data-position="info-box">
            </section>
            <section data-position="error-box">
                <div id="error-info"></div>
            </section>
        </main>
        <footer>&copy; 2021 rs-soft-bg</footer>
    </body>
</html>
```

Фиг. 12.8 Програмен код на index.html

Останалият програмен код е структуриран в няколко папки. В папка “**libs**” са библиотеките, необходими за реализация на проекта. На този етап ще използваме само библиотека jQuery. В папка “js” са записани три JavaScript модула:

- **mycode.js** – съдържа програмния код чрез който се изчаква зареждане на библиотека jQuery, прехваща се събитието, което се генерира при натискане на бутона за изпращане на заявка, прехващат и събитията “Начало на AJAX заявка” и “Край на AJAX заявка”. Тези събития се използват от интерфейса с потребителя - уведомяват клиента за всяко начало и край на AJAX заявки. Те се използват с цел адресиране на ресурси от IBM Cloudant.
- **display.js** – съдържа програмния код чрез който се реализира SPA приложение. Предвидени са функции за показване и скриване на съдържание от различни секции от HTML5 кода.
- **database.js** – съдържа програмния код за изпращане на заявка къмIBM Cloudant, получаване на отговора и динамично формиране на отговор за браузъра.

В папка “**css**” е CSS кода, който реализира желаното стилово форматиране.

Програмният код от модул “**mycode.js**” е показан на Фиг. 12.9. След зареждане на библиотека jQuery - $(function() { ... }); - се прехваща три събития: “click” от бутона за изпращане на заявка и “ajaxStart” и “ajaxStop” от библиотека jQuery. При натискане на бутона управлението се предава на функция **sendQuery** от модул “database.js”. Веднага след зареждане на jQuery се визиализира секция “query-box” чрез метод **view** (модул display.js).

```
$(function () {
    $("#submit").click(function() {
        sendQuery();
    });
    $(document).on({
        ajaxStart: function () {
            view('loading-box');
        },
        ajaxStop: function () {
            hide('loading-box');
        }
    });
    view('query-box');
});
```

Фиг. 12.9 Програмен код на mycode.js

```
function view(viewName) {
    $('main > section').hide();
    $(`[data-position=${viewName}]`).show();
}
function hide(viewName) {
    $(`[data-position=${viewName}`).hide();
}
function showInfo(data) {
    $('[data-position=info-box]').append(data);
    view('info-box');
}
function showError(errorMsg, time) {
    var errorBox = $('#error-info');
    errorBox.text(errorMsg);
    view('error-box');
    setTimeout(function () {
        $('[data-position=error-box]').fadeOut();
        view('query-box');
    }, time*1000);
}
```

Фиг. 12.10 Програмен код на display.js

На Фиг. 12.10 е показан програмния код от файл “**display.js**”. Функция **view** скрива всички секции, които са в тялото на етикет <main> и показва само секцията, чието име се предава като аргумент. Функция **hide** скрива съдържанието на указана по име секция. Информацията, която е получена от базата данни, се визуализира чрез функция **showInfo**. Тази функция се вика толкова пъти, колкото документа има в базата данни. Затова тя добавя всеки път ново HTML съдържание към секцията чрез функция **append**. Обработката на грешки се реализира от тялото на функция **showError**. Тя получава два аргумента: низ, който описва грешката “errorMsg” и времето в секунди за което този низ да се вижда на екрана “time”.

Програмният код, който реализира комуникация с базата данни IBM Cloudant е в модул **database.js**. Тази услуга в облака на IBM ни позволява да пишем Web приложения без реален код от страна на сървъра (serverless app). Комуникацията между клиента и базата данни е чрез протокол HTTPS. Целият трафик е шифриран чрез SSL/TLS. Тъй като ще използваме базова авторизация на достъпа до Cloudant е необходимо да кодираме ключа и паролата за достъп до база данни “students” чрез алгоритъм Base64. В JavaScript това се реализира чрез вградената функция **btoa**. Ще използваме библиотека jQuery за да изпратим заявката като AJAX заявки. За целта ще използваме функция **ajax**. За да можем

да зададем timeout интервал трябва да използваме асинхронния вариант на тази функция. Това означава, че функцията веднага ще върне от управление след изпълнението ѝ. За да разберем кога реално е получен отговор ще използваме обект-обещание (promise). Функцията, която реализира комуникация с IBM Cloudant, е с име **getResult**. Тя се вика от функция **sendQuery**. Синхронизацията между тези две функции се реализира чрез обекта-обещание. Когато се получи валиден резултат от услугата се вика анонимна функция (**then**). Грешките се обработват също чрез анонимна функция (**catch**). Ако резултатът е валиден, но се съдържа само един документ, неговото съдържание се парсва и добавя в HTML страницата чрез функция **addResultToPage**. Ако резултатът съдържа множество JSON обекти, те се обработват последователно (вътрешен цикъл чрез функция **forEach**) отново със същия метод.

Когато заявяваме съдържание на документ от Cloudant база данни трябва да използваме следния URL низ:

```
const url = PROTOCOL + SERVER_NAME + ':' + PORT + '/' + DATABASE + '/' +
            DOCUMENT + REC_FIELDS;
```

където:

- PROTOCOL е низа ‘https://’.
- SERVER_NAME е името на сървъра (получавате го при инсталиране на услугата).
- PORT – номера на порта, който Claudant подслушва – 443 (HTTPS).
- DATABASE – име на базата данни, до която желаете достъп – “students”.
- DOCUMENT – име на документа до който желаете достъп. В този случай резултатът ще бъде съдържанието на указания документ:

```
▶ grade: (4) [{…}, {…}, {…}, {…}]
  id: "21705001"
▶ name: {firstName: "Георги", lastName: "Котев"}
  _id: "student-0001"
  _rev: "1-9c8f87b7104684e21c832ffae329fd69"
```

Ако трябва да получите имената или съдържанието на всички документи, стойността на това поле трябва да бъде **“_all_docs**”.

- RES_PARS – допълнителни параметри, които се предават към заявката. Ако полето е празен низ, при DOCUMENT=“_all_docs” ще получите масив от JSON обекти, които чрез поле “id” описват имената на всеки един документ, например:

```
▼ rows: Array(3)
  ▶ 0: {id: "student-0001", key: "student-0001", value: {…}}
  ▶ 1: {id: "student-0002", key: "student-0002", value: {…}}
  ▶ 2: {id: "students-0003", key: "students-0003", value: {…}}
```

Ако искате с една заявка да получите и съдържанието на документите, то задайте за стойност на RES_PARS низа “ ?include_docs=true”:

```
▼ rows: Array(3)
  ▶ 0: {id: "student-0001", key: "student-0001", value: {…}, doc: {…}}
  ▶ 1: {id: "student-0002", key: "student-0002", value: {…}, doc: {…}}
  ▶ 2: {id: "students-0003", key: "students-0003", value: {…}, doc: {…}}
```

Ако към документите има прикачени файлове и искате да получите и тяхното съдържание трябва да изпратите втори параметър с име “attachments” със стойност true:

“&attachments=true”.

Припомнете си за какво служеха специални символи “?” и “&” при изпращане на HTTP заявки.

На Фиг. 12. 11 са показани константите, които модул database.js използва. Не забравяйте да смените стойностите на константите SERVER_NAME, KRY и PASWWORD с тези за вашата база данни. Времето за timeout се задава в ms и следователно е 4 секунди.

```
const PORT = 443;
const TIMEOUT = 4000;
const PROTOCOL = 'https://';
const SERVER_NAME =
 'dc3b328b-545c-4c02-9457-531fdc110a3e-
bluemix.cloudantnosqldb.appdomain.cloud';
const DATABASE = 'students';
const DOCUMENT = '_all_docs';              // 'student-0001'
const RES_PARS = '?include_docs=true';     // ''
const KEY = 'apikey-a13473867922429ea5571840ae260342';
const PASSWORD = '53a874f75d9825ff9aa9c10d3c6ace4449da2045';
```

Фиг. 12.11 Използвани константи – database.js

Програмният код от функция **getResult** е показан на Фиг. 12.12.

```
function getResult(hash) {
    var url = PROTOCOL + SERVER_NAME + ':' + PORT + '/'
            + DATABASE + '/' + DOCUMENT + RES_PARS;
    var promise = new Promise(function (resolve, reject) {
        $.ajax({
            type: 'GET',
            url: url,
            async: true,
            cache: false,
            contentType: 'application/json',
            timeout: TIMEOUT,
            beforeSend: function (req) {
                req.setRequestHeader('Accept', 'application/json');
                req.setRequestHeader('Authorization', 'Basic ' + hash);
            },
            success: function (data) {
                resolve(data);
            },
            error: function (data) {
                reject(data);
            }
        });
    });
    return promise;
}
```

Фиг. 12.12 Функция getResult

Функция **ajax** от библиотека jQuery изисква един атрибут – JSON обект който съдържа множество полета чрез които се указва каква точно заявка искаме да изпратим. Използван е HTTP метод GET (получаване на съдържание). Заявките не трябва да се кешират (cache: false). Задължително трябва да е разрешено асинхронното изпълнение на заявката (async: true), за да можем да зададем timeout интервал. Задайте за тип на отговора MIME типа на JSON документ (application/json). За да зададем използването на базова авторизация ще използваме функция **beforeSend** и чрез функция **setRequestHeader** ще

вмъкнем в заглавния блок на заявката поле “Authorization” със стойност “Basic xxx”, където “xxx” е Base64 кодиран низ, който съдържа “ключа + символ **:** + паролата”.

При успешно изпълнение на заявката се вика функция **success**. От тялото ѝ се вика функция **resolve**, която предава управлението на **then**. При грешка се вика функция **error**. От тялото ѝ се изпълнява функция **reject** и управлението се предава на **catch**.

Програмният код от функция **sendQuery** е показан на Фиг. 12.13.

```
function sendQuery() {
   var hash = btoa(KEY + ':' + PASSWORD);
   getResult(hash)
      .then(function (result) {
         var n = Object.keys(result).length;
         if (n === 1) {
            addResultToPage(result);
         }
         else {
            var rows = result.rows;
            rows.forEach(function(row) {
               addResultToPage(row.doc);
            });
         }
      }).catch(
         function (error) {
            var status = error.status;
            if (status === 0) {
               showError('Невъзможна е връзката с базата данни!', 3);
            }
            else if (status === 401) {
               showError('Грешка при авторизация!', 3);
            }
         }
   );
}
```

Фиг. 12.13 Функция sendQuery

Функция **sendQuery** вика функция **getResult** и синхронизира завършването ѝ (успешно или неуспешно) чрез обект-обещание. За да проверим резултата дали съдържа един или множество JSON обекта използваме функция **keys** и свойство “length”. Ако стойността на променлива *n* е 1, това означава, че сме заявили съдържанието на точно определен документ. В противен случай сме заявили съдържанието на всички документи. Обработката на резултата и в двата случая се реализира чрез функция addResultToPage.

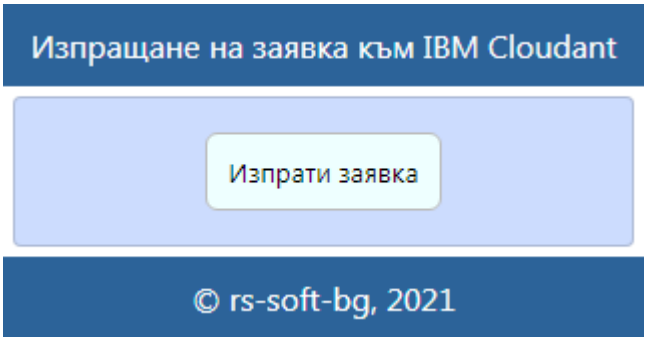
Програмният код от функция **addResultToPage** е показан на Фиг. 12.14. Функцията извлича името на студента и факултетния му номер. След това вмъква тази информация в div контейнер с идентификатор “info-name”. След това информацията за оценките се извлича и вмъква в контейнер с идентификатор “info-grades”. Чрез функция **showInfo** тази информация се добавя към секция “info-box”.

```
function addResultToPage(result) {
    var id = result.id;
    var firstName = result.name.firstName;
    var lastName = result.name.lastName;
    var name = `${firstName} ${lastName} - ${id}`;
    var divName = '<div id="info-name">' + name + '</div>';
    var grades = result.grade;
    var gradesHtmlStart = '<ul class="grades">';
    var gradesHtmlEnd = '</ul>';
    var gradesHtml = gradesHtmlStart;
    grades.forEach(function (grade) {
        var subject = grade.subject;
        var value = grade.value;
        var pair = `${subject}: ${value}`;
        gradesHtml += `<li class="grades">${pair}</li>`;
    });
    gradesHtml += gradesHtmlEnd;
    var divGrades = '<div id="info-grades">' + gradesHtml + '</div>';
    showInfo(divName + divGrades);
}
```

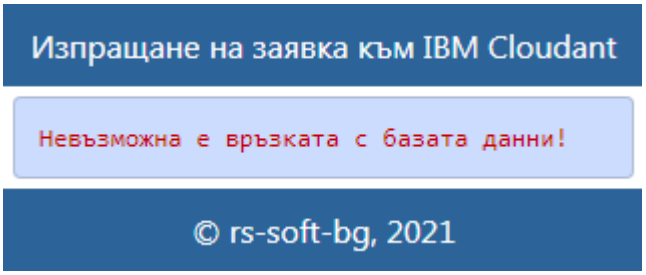
Фиг. 12.14 Функция addResultToPage

На Фиг. 12.15 са показани снимки от потребителския интерфейс на приложението.

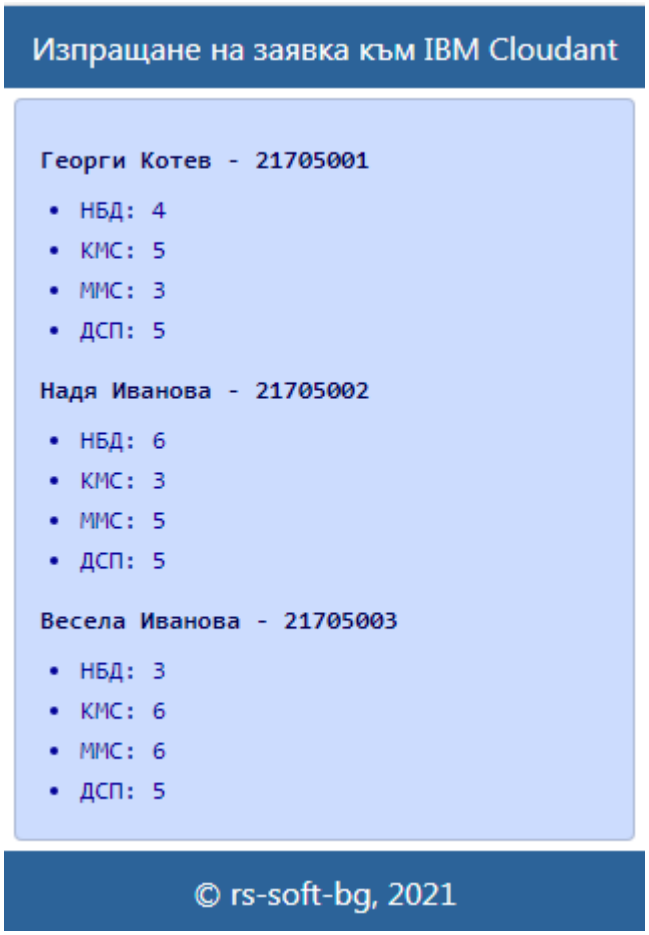
Задача 2: *Разширете функциите на приложението от Задача 1 така, че да се дава възможност за търсене на студентите, които имат конкретна оценка по избрана дисциплина.*



a)



b)



c)

Фиг. 12.15 Потребителски интерфейс: a) Начална страница; b) Липсва мрежова свързаност; c) Резултат при успешно изпълнена заявка

Упражнение № 13

IBM Cloudant – търсене на съдържание

I. Въведение

Търсенето на съдържание в IBM Cloudant (CouchDB) бази данни може да се реализира по два начина:

- Използване на дизайн документи (design) и ресурси, формиращи изгледа (view).
- Използване на Mango заявки.

Базовият подход за обработка на заявки към IBM Cloudant е създаването на View ресурси. Всеки дизайн документ съдържа един или колкото е необходимо view ресурси чрез които се формира отговора посредством функция **emit**. Всеки view ресурс може да реализира MapReduce функционалност. За целта **map** функцията е стойността на поле “map”, а **reduce** функцията е стойността на поле “reduce”.

Предпочитаният подход е да използвате Mango заявки. Mango е акроним, който идва от “MongoDB inspired query language interface for Apache CouchDB”. Целта е предоставяне на възможност за генериране на заявки с цел извличане на информация от базата данни, без да е необходимо да се пишат view ресурси и без да се използва MapReduce. Тези заявки се изпълняват от 2 до 10 пъти по-бързо отколкото обслужваните чрез Erlang заявки. Заявките Mango се обработват от Mango Query Server, който е част от всеки CouchDB сървър версия 2.0.0+. Този сървър позволява обработка на декларативен тип заявки, без да е необходимо да се пишат ресурси на JavaScript. Препоръчва се да се използват само Mango заявки при разработване на приложенията. Причините за това са няколко:

- Mango заявките се обработват по-бързо от Erlang заявките.
- Erlang Query Server не е достатъчно сигурен. За разлика от JavaScript Query Server той не функционира в sandbox режим. Следователно Erlang програмния код има пълен достъп до апаратните и програмни ресурси на хоста, в зависимост от правата на достъп на текущият потребител.

II. Използване на view ресурси

За да може да се обработват заявките от страна на клиента, трябва да създадете необходимите дизайн документи и view ресурси, които генерират отговор, който се визуализира от браузъра. Базата данни CouchDB използва MapReduce механизъм, за да формира отговор. За целта всеки **view ресурс** може да има **map** и **reduce** функции. Те се стартират за всеки документ от базата данни и логиката им не зависи от нищо друго извън документа. Тази независимост позволява паралелната обработка на view ресурсите. Тъй като тези функции формират отговор във формат “key:value” те се индексират по ключ чрез B+ дърво. Това позволява бързо търсене на информация по стойността на ключ или по минимална и максимална стойност на ключа. На Фиг. 13.1 е показан синтаксиса на дизайн документ, който съдържа два view ресурса.

```
{
   "_id": "_design/name",
   "_rev": "1-11b803fe24ddfbc6fd8d47dd7150225c",
   "language": "javascript",
   "views": {
      "viewFuncName1": {
         "map": "function(doc) {
           emit(key, value);
         }",
         "reduce": "function(key, value, rereduce) {
           return sum(value)/value.length;
         }"
      },
      "viewFuncName2": {
         "map": "function(doc) {
           emit(key, value);
         }",
         "reduce": "_stats"
      }
   }
}
```

Фиг. 13.1 Дизайн документ – структура и синтаксис

Както всеки друг документ, всеки дизайн документ има системни полета “_id” и “_rev”. Може да смените стойността на системния идентификатор, както е показано на Фиг. 13.1. За име на документа задайте низ, който ще ви подсеща за какво точно служи документа. На този етап поле “language” име стойност “javascript”. Това подсказва, че документа съдържа view ресурси в които има JavaScript код. Всички view ресурси са в тялото на поле “views”. Избирайте имената на view ресурсите така, че да са максимално информативни. Всеки ресурс трябва да има **map** функция и незадължителна **reduce** функция. Тези функции се декларират като анонимни JavaScript функции. Всяка map функция трябва да завършва с **emit** функция, която указва какво представлява ключа (key) и каква е стойността (value), която се връща (JSON обект). Функциите map получават един аргумент – обект, който е един от обработваните документи (doc). От тялото на map функцията може да проверите дали този документ трябва да се обработва или не (например, това може да е дизайн документ). Следва логика чрез която се реализира желаното филтриране. Ключът показва кое свойство ще се използва като ключ. Това позволява да филтрираме информацията в документите, в зависимост от стойността на един или няколко ключа. Ако трябва да се използват няколко ключа се използват няколко **emit** функции, или се използва съставен ключ (масив от ключове).

За да използвате view ресурсите е необходимо да създадете URL низ в който е описано кой view ресурс да се използва и в кой дизайн документ е този ресурс, например:

```
https://hostName/databaseName/_design/designDocName/_view/viewResName?key=keyValue
```

Името на хоста “hostName” зависи дали работите локално с CouchDB или с IBM Cloudant. Към всеки view ресурс може да предавате конкретна стойност за ключа, “key=keyValue”. Стойността на ключа може да бъде число, низ или масив от елементи, ако ключа е съставен.

III. Използване на Mango заявки

Основният начин за изпращане на Mango заявки е чрез ресурс _**find**. Заявката до _find се изпраща като се използва HTTP POST метод. Всяка заявка може да използва собствено индексиране или това, което е активно за "_all_docs". В HTTP заглавния блок задължително трябва да се зададе "Content-type: application/json", тъй като заявката е JSON обект. Този обект съдържа следните по-важни свойства:

- **selector** – Задължително поле, което задава критерия по които се търси информация в базата данни като JSON обект;
- **sort** – Незадължително поле - JSON обект, който описва декларативно какво сортиране трябва да се приложи към заявката;
- **fields** – Незадължително поле - JSON масив, чрез който се задава кои полета на документа трябва да формират отговора. При пропускане на това свойство се връща съдържанието на целия обект.
- **limit** – Незадължително поле - максимален брой на върнатите обекти. По подразбиране 25.
- **skip** – Незадължително поле - пропускане на първите *n* резултата, където *n* е зададената стойност след skip (skip: n).
- **use_index** - Незадължително поле - инструктира сървъра, че заявката ще използва точно определен индекс. Индексът се описва като низ, съдържащ името на дизайн документа с индекса или масив съдържащ името на дизайн документа и името на индекса: ["design_document", "index_name"].
- **execution_stats** – Незадължително поле - при стойност true в отговора се включва и статистическа информация за обслужената заявка, например: общ брой анализирани документи; брой документи, върнати като резултат, време за изпълнение на заявката в ms.

Ще опишем синтаксиса на така изброените свойства. За примерна база данни ще използваме "students", която съдържа информация за студенти.

Свойство selector

Стойността на свойство selector (JSON) задава каква точно информация търси клиента, например:

```
{
   "specialty": "КСТ"
}
```

В този случай ще се върне информация за студентите, които са от специалност КСТ. Всеки селектор може да съдържа толкова полета, колкото са необходими, например:

```
{
   "specialty": "КСТ",
   "course": 1
}
```

В този случай ще се върнат всички студенти от специалност КСТ, които в момента са в първи курс. Някои от полетата от документите може да съдържат други JSON обекти или да са JSON масиви. В този случай трябва да се укаже кое вложено поле ще се използва с цел генериране на заявката. За да намерим всички студенти с фамилия Иванов трябва да използваме следния селектор:

```
{
   "name.family": "Иванов"
}
```

За да създадете по-сложни като логика заявки можете да използвате **Mango оператори** или комбинация от Mango оператори. Всеки оператор започва със знака за долар $. Операторите се делят на:

- Оператори за комбиниране: $and, $or, $not, $nor, $all, $elemMatch, $allMatch, $keyMapMatch.
- Условни оператори: $eq, $ne, $gt, $gte, $lt, $lte, $exist, $type, $in, $min, $mod, $size, $regex

Операторите за комбиниране се използват за комбиниране на селектори. В допълнение към общите логически оператори има три оператора ($all, $elemMatch и $allMatch), които ви помагат да работите с JSON масиви, и един, който работи с JSON карти ($keyMapMatch).

Условните оператори са специфични за дадено поле и се използват за оценка на стойността, съхранена в това поле.

Можете да създавате по-сложни изрази за селектор чрез комбиниране на оператори. За най-добра производителност е най-добре да обединявате оператори за комбиниране или условни оператори, например оператора за търсене чрез регулярни изрази $regex с условни оператори, като например $eq, $gt, $gte, $lt и $lte (но не и $ne). Не използвайте оператор $regex с големи масиви с данни, тъй като производителността ще е ниска.

Ако трябва да намерим всички студенти от специалност КСТ, които са във втори курс, може да използваме следния селектор:

```
"$and": [
   {
      "specialty": { "$eq": "КСТ" }
   },
   {
      "course": { "$eq": 2 }
   }
]
```

Ако трябва да получите имената на студентите, които са от няколко курса, например 2-ри и 4-ти, може да използвате оператор $in. Стойността трябва да е масив, всеки елемент от който са номерата на курсовете от които се интересувате:

```
"course": { "$in": [2, 4] }
```

Когато трябва да извлечем информация за параметър, който приема минимална и максимална стойност можем да използваме условни оператори, например:

```
"course": {"$gte": 1},  "course": {"$lte": 3}
```

Този селектор връща всички студенти от 1-ви до 3-ти курс.

Ако трябва да намерим всички студенти, които имат фамилия, която започва с буква “И” можем да използваме регулярен израз, например:

```
"name.family": { "$regex": "^И" }
```

Нека да намерим имената на всички студенти, които имат двойки по дисциплината НБД. За целта можем да използваме следния селектор:

```
"grade": {
  $elemMatch: {
    "$and": [
      {
        "subject": { "$eq": "КСТ" }
      },
      {
        "value": { "$eq": 2 }
      }
    ]
  }
}
```

Свойство sort

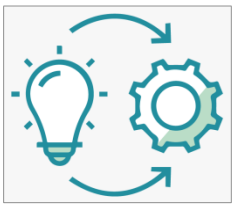
Стойността на поле soft съдържа масив от JSON обекти или имена на полета чрез които се задава какво сортиране на данните трябва да се приложи. Може да се използват множество полета при сортирането. Данните се сортират с приоритет, от първото към последното поле. Подредбата на резултата може да бъде във низходяща или възходяща посока. Посоката се задава като стойност на полето:

- asc – възходяща посока;
- desc – низходяща посока.

Нека да сортираме трите имена на всички студенти с фамилия Иванов. Следва съдържанието на Mango заявката:

```
{
  "selector": {
    "name.family": "Иванов"
  },
  "sort": [ {"name": "desc"} ],
  "fields": ["name"]
}
```

IV.Задачи за изпълнение



Задача 1: *Създайте база данни "students2", която позволява извличане на информация за студентите от дадена специалност: факултетен номер, трите имена, курс на обучение, оценки по всяка дисциплина за всеки семестър на обучение. Напишете необходимите view ресурси, които позволяват получаване на следната информация:*

1. *Име на студента и курс на обучение по зададен факултетен номер.*
2. *Факултетен номер на студента по зададено име.*
3. *Имената на студентите от зададена специалност.*
4. *Оценките на студент по зададено име за даден семестър.*
5. *Статистика на оценките за зададена специалност.*
6. *Статистика за оценките за зададен семестър.*
7. *Статистика на оценките за зададена дисциплина.*

За всеки студент ще създадем документ, който ще описва неговите имена, специалност, курс и оценките по дисциплините по семестри. Идентификационният код на документите може да съвпада с факултетния номер на студента, тъй като той е уникален за всеки студент в дадено учебно заведение. Тъй като е възможно един студент да промени факултетния си номер по време на следването (издаване на нова студентска книжка) или да има две студентски книжки (изучава две специалности едновременно) е по-добре факултетният номер да е свойство, а не уникален ключ.

Създайте нова база данни “students2” в IBM Cloud. Получете двойката “ключ-стойност” за достъп до базата данни с цел четене на информация от нея. Създайте поне три документа, всеки от които описва студент, съгласно заданието. На Фиг. 13.2 е показано примерно съдържание на един документ от тази база данни.

```
{
  "_id": "student-0001",
  "_rev": "1-96fa0a50e00ef79eaf3efacb39012b38",
  "type": "student",
  "name": {
    "firstname": "Стоян",
    "family": "Иванов",
    "lastname": "Стоянов"
  },
  "id": "123457",
  "specialty": "АИУТ",
  "course": 1,
  "semester": [
    {
      "number": 1,
      "subjects": [
        {
          "name": "Висша математика I",
          "grade": 2
        },
        {
          "name": "Химия",
          "grade": 3
        },
        // Други дисциплини от Семестър 1
      ]
    },
    {
     // Информация за следващ семестър
    }
  ]
}
```

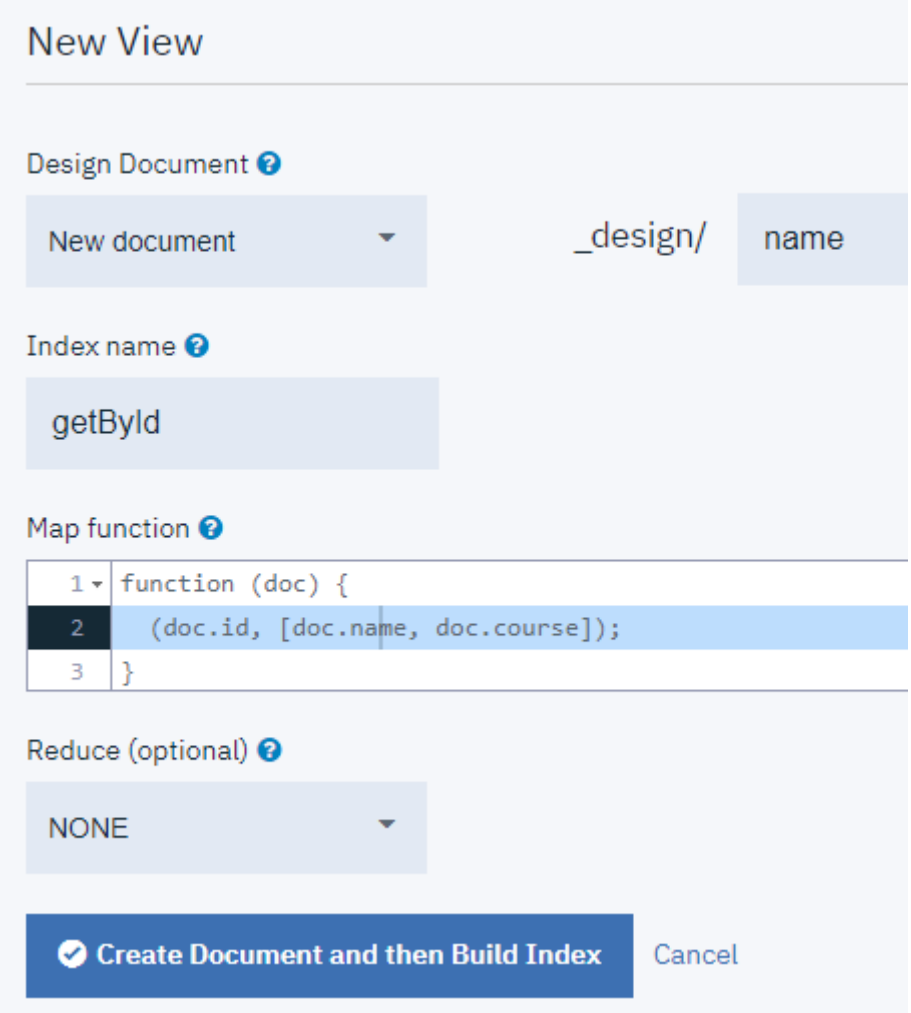
Фиг. 13.2 Примерно съдържание на документите от база данни “students2”

Основните полета и тяхното предназначение са следните:

- Поле “**type**” – чрез това поле view ресурсите могат бързо да филтрират документите, които не съдържат описание на студенти.
- Поле “**name**” – съдържа трите имена на студента.
- Поле “**semester**” – масив от JSON обекти - съдържа информация за номера на семестъра (поле “number”) и информация за изучаваните дисциплини (поле “subjects”). Това поле е масив, елементите от който съдържат информация за име на дисциплина (поле “name”) и оценката (поле “grade”) по тази дисциплина.

1. Получаване на името на студента и курса на обучение по зададен факултетен номер

Ще създаден дизайн документ с име name и view ресурс с име getById. За целта изберете от менюто на IBM Cloudant Dashboard “Design Document” и след това “New View” (виж Фиг. 13.3). Задайте “New document” за “Design Dicument” и име на документа “name”. Името на view документа е “getById”. Редактирайте съдържанието на **map** функцията. Задайте за ключ “doc.id”, а за стойност масив, който съдържа името на студента “doc.name” и курса на обучение “doc.course”. Създайте документа чрез натискане на бутон “Create document and then build index”.



Фиг. 13.3 Създаване на нов view ресурс

Съдържанието на новия документ “_design/name” е следното:

```
{
  "_id": "_design/name",
  "_rev": "1-274dc0094371ce7984babb146e5a242f",
  "views": {
    "getById": {
      "map": "function (doc) {\n (doc.id, [doc.name, doc.course]);\n}"
    }
  },
  "language": "javascript"
}
```

Остава да изпратим заявката, като използваме създадения view ресурс, например:

https://dc3b328b-545c-4c02-9457-531fdc110a3e-bluemix.cloudant.com/students2/_design/name/_view/getById

При тази заявка ще получим информация за всички студенти във формат ключ-стойност:

```
{"total_rows":3,"offset":0,"rows":[
{"id":"student-0002","key":"123456","value":[{"firstname":"Иван","family":"Иванов","lastname":"Иванов"},2]},
{"id":"student-0001","key":"123457","value":[{"firstname":"Стоян","family":"Иванов","lastname":"Стоянов"},1]},
{"id":"student-0003","key":"123458","value":[{"firstname":"Надя","family":"Иванова","lastname":"Петкова"},2]}
]}
```

Ако желаете тази информация за конкретен студент трябва да предадете към view ресурса конкретна стойност за ключа, например:

https://dc3b328b-545c-4c02-9457-531fdc110a3e-bluemix.cloudant.com/students2/_design/name/_view/getById?key=”123457”

В този случай сървърът ще върне информация само за студента със зададения идентификатор (факултетен номер):

```
{"total_rows":3,"offset":1,"rows":[
{"id":"student-0001","key":"123457","value":[{"firstname":"Стоян","family":"Иванов","lastname":"Стоянов"},1]}
]}
```

Ако искате да получите данните за няколко студента може да използвате параметър **keys**, например:

https://dc3b328b-545c-4c02-9457-531fdc110a3e-bluemix.cloudant.com/students2/_design/name/_view/getById?keys=[”123457”, “123456”]

Ако искате да получите данните на студенти с факултетни номера попадащи в интервал трябва да използвате параметри **startkey** и **endkey**, например:

https://dc3b328b-545c-4c02-9457-531fdc110a3e-bluemix.cloudant.com/students2/_design/name/_view/getById?startkey”123456”&endkey=”123458”

2. Получаване на факултетен номер на студента по зададено име

Създайте нов view ресурс при който ключът е името на студента, а стойността - факултетния му номер:

```
{
   "_id": "_design/id",
   "_rev": "1-11b803fe24ddfbc6fd8d47dd7150225c",
   "language": "javascript",
   "views": {
      "getByName": {
         "map": "function(doc) {emit(doc.name,doc.id);}"
      }
   }
}
```

3. Получаване на името на студентите от зададена специалност

Създайте нов view ресурс при който ключът е името на студента, а стойността - специалността която учи:

```
{
  "_id": "_design/name",
  "_rev": "1-d861ad1dc20a8df29cbfed9f1a3c062b",
  "language": "javascript",
  "views": {
    "getBySpecialty": {
      "map": "function(doc) { emit(doc.specialty, doc.name); }"
    }
  }
}
```

4. Получаване на оценките на студент за даден семестър по зададено име

В този случай трябва да използваме съставен ключ, който се състои от името на студента и номера на семестъра. Стойността, която връща функция **emit** трябва да е JSON обект, който съдържа името на дисциплината и оценката по тази дисциплина. За целта създайте view ресурс **getByNameAndSemester**. От тялото на map функцията трябва да обходим програмно два масива - “semester” и “subjects”:

```
{
   "_id": "_design/grade",
   "_rev": "1-d861ad1dc20a8df29cbfed9f1a3c062b",
   "language": "javascript",
   "views": {
      "getByNameAndSemester": {
        "map": "function(doc) {
           doc.semester.forEach(function(semester){
              semester.subjects.forEach(function(subject) {
                 emit([doc.name,semester.number], subject);
              });
           });
        }"
      }
   }
}
```

Следва примерна заявка, която връща оценките на студента Иван Иванов за първи семестър:

https://dc3b328b-545c-4c02-9457-531fdc110a3e-bluemix.cloudant.com/students2/_design/grade/_view/getByNameAndSemester?key=[{"firstname":"Иван","family":"Иванов","lastname":"Иванов"}, 1]

Резултатът, който се получава, е следния:

```
{"total_rows":33,"offset":0,"rows":[
{"id":"student-
0002","key":[{"firstname":"Иван","family":"Иванов","lastname":"Иванов"},
1],"value":{"name":"Химия","grade":4}},
{"id":"student-
0002","key":[{"firstname":"Иван","family":"Иванов","lastname":"Иванов"},
1],"value":{"name":"Физика I","grade":4}},
...
]}
```

5. Получаване на статистика на оценките за зададена специалност

При тази задача за ключ трябва да използваме специалността на студента, а за стойност оценките по отделните дисциплини. Напишете нов view ресурс с име **getBySpecialty**, който да е част от вече създадения дизайн документ "_design/grade". Освен необходимата **map** функция, трябва да напишете и **reduce** функция чрез която да се върне статистика за оценките:

```
{
   "_id": "_design/grade",
   "_rev": "1-d861ad1dc20a8df29cbfed9f1a3c062b",
   "language": "javascript",
   "views": {
      "getBySpecialty": {
         "map": "function(doc) {
            doc.semester.forEach(function(semester){
               semester.subjects.forEach(function(subject) {
                  emit(doc.specialty, subject.grade);
               });
            });
         }",
         "reduce": "_stats"
      }
   }
}
```

В IBM Cloudant има вградени reduce функции като **_sum**, **_count** и **_stat**. Използвайте тях винаги, когато е възможно, за да намалите времето необходимо за формиране на отговор. При конкретния пример, стойността на reduce функцията е "_stats". Това означава, че ще се върне статистическа информация (сума на оценките, брой оценки, минимална оценка, максимална оценка и сумата от квадрата на оценките). Ако искаме да получим статистика за оценките на всички студенти от катедра КСТ трябва да генерираме следната заявка:

https://dc3b328b-545c-4c02-9457-531fdc110a3e-bluemix.cloudant.com/students2/_design/grade/_view/getBySpecialty?key="КСТ"

Резултатът, който се получава, е следния:

```
{"rows":[
{"key":null,"value":{"sum":100,"count":22,"min":2,"max":6,"sumsqr":476}}
]}
```

Ако не зададем стойност за параметър **key** ще получим статистика за оценките за студентите от всички специалности:

```
{"rows":[
{"key":null,"value":{"sum":144,"count":33,"min":2,"max":6,"sumsqr":662}}
]}
```

Ако искате да получите статистика на оценките за всяка специалност по отделно чрез една заявка трябва да използвате параметър **group** със стойност true:

https://dc3b328b-545c-4c02-9457-531fdc110a3e-bluemix.cloudant.com/students2/_design/grade/_view/getBySpecialty?group=true

Резултатът, който се получава, е следния:

```
{"rows":[
{"key":"АИУТ","value":{"sum":44,"count":11,"min":2,"max":5,"sumsqr":186}
},
{"key":"КСТ","value":{"sum":100,"count":22,"min":2,"max":6,"sumsqr":476}
}
```

6. Получаване на статистика за оценките за зададен семестър

В този случай трябва да добавим нов view ресурс с име **getBySemester** към дизайн документа с име “grade”:

```
{
   "_id": "_design/grade",
   "_rev": "1-d861ad1dc20a8df29cbfed9f1a3c062b",
   "language": "javascript",
   "views": {
      "getBySemester": {
        "map": "function(doc) {
          doc.semester.forEach(function(semester){
            semester.subjects.forEach(function(subject) {
              emit(semester.number, subject.grade);
            });
          });
        }",
        "reduce": "_stats"
      }
   }
}
```

Ако трябва да получим статистика за оценките за всеки семестър поотделно ще използваме следната заявка:

https://dc3b328b-545c-4c02-9457-531fdc110a3e-bluemix.cloudant.com/students2/_design/grade/_view/getBySemester?group=true

Резултатът, който се получава, е следния:

```
{"rows":[
{"key":1,"value":{"sum":73,"count":18,"min":2,"max":6,"sumsqr":317}},
{"key":2,"value":{"sum":71,"count":15,"min":3,"max":6,"sumsqr":345}}
]}
```

Ако интерес представлява конкретен семестър (например 1-ви), задайте номера му като стойност на параметър key:

https://dc3b328b-545c-4c02-9457-531fdc110a3e-bluemix.cloudant.com/students2/_design/grade/_view/getBySemester?key=1

Ако трябва да получим статистика на оценките за точно определени семестри, трябва да се използва параметър keys. Стойността на параметъра е масив, който в случая трябва да съдържа номерата на семестрите за които желаем статистика на оценките. Трябва да зададете и параметър group със стойност true:

https://dc3b328b-545c-4c02-9457-531fdc110a3e-bluemix.cloudant.com/students2/_design/grade/_view/getBySemester?keys=[1,3,2]&group=true

7. Получаване на статистика на оценките за зададена дисциплина

В този случай трябва да добавим нова view ресурс с име **getBySubject** към дизайн документа с име “grade”. Ключът в този пример трябва да бъде името на дисциплина, а стойността – оценката на студента:

```
{
   "_id": "_design/grade",
   "_rev": "1-d861ad1dc20a8df29cbfed9f1a3c062b",
   "language": "javascript",
   "views": {
      "getBySubject": {
        "map": "function(doc) {
          doc.semester.forEach(function(semester){
            semester.subjects.forEach(function(subject) {
              emit(subject.name, subject.grade);
            });
          });
        }",
        "reduce": "_stats"
      }
   }
}
```

Ако трябва да получите статистика за оценките за конкретна дисциплина, например “Висша математика I”, използвайте следния синтаксис на заявката:

https://dc3b328b-545c-4c02-9457-531fdc110a3e-bluemix.cloudant.com/students2/_design/grade/_view/getBySubject?key="Висша математика I"

Резултатът, който се получава, е следния:

```
{"rows":[
{"key":null,"value":{"sum":11,"count":3,"min":2,"max":6,"sumsqr":49}}
]}
```

Използвайте параметър group със стойност true, за да получите статистика за всички дисциплини:

https://dc3b328b-545c-4c02-9457-531fdc110a3e-bluemix.cloudant.com/students2/_design/grade/_view/getBySubject?group=true

Резултатът, който се получава, е следния:

```
{"rows":[
{"key":"Висша математика
I","value":{"sum":11,"count":3,"min":2,"max":6,"sumsqr":49}},
{"key":"Висша математика
II","value":{"sum":12,"count":3,"min":3,"max":5,"sumsqr":50}},
{"key":"Електроматериалознание","value":{"sum":16,"count":3,"min":5,"max":6,
"sumsqr":86}},
{"key":"Инженерна
графика","value":{"sum":15,"count":3,"min":5,"max":5,"sumsqr":75}},
{"key":"Програмиране и използване на
компютри","value":{"sum":9,"count":3,"min":2,"max":4,"sumsqr":29}},
{"key":"Теоретична електротехника
I","value":{"sum":13,"count":3,"min":4,"max":5,"sumsqr":57}},
{"key":"Физика I","value":{"sum":13,"count":3,"min":4,"max":5,"sumsqr":57}},
...
]}
```

Задача 2: *Разпечатайте имената на всички студенти, които имат Добър(4) по дисциплината „Физика I”. Използвайте Mango заявка.*

Трябва да проверим стойностите на полета “name” и “grade”. Тъй като те са в масив “subjects”, а той от своя страна в масив “semester” трябва да използваме два пъти оператор “$elemMatch”.

Можете да симулирате Mango заявки от ниво програмен интерфейс на IBM Cloudant. Изберете желаната база данни (“students2”), а от менюто - “Query”. Въведете желания Mango код в поле “Cloudant Query” (виж Фиг. 13. 4).

```
{
  "selector": {
    "semester": {
      "$elemMatch": {
        "subjects": {
          "$elemMatch": {
            "$and": [
              {
                "grade": {
                  "$eq": 4
                }
              },
              {
                "name": "Физика I"
              }
            ]
          }
        }
      }
    },
    "fields": [
      "name"
    ]
}
```

Фиг. 13.4 Симулиране на Mango заявки чрез интерфейса на IBM Cloudant

Натиснете бутон <Run query>, за да видите резултата. Студентите, които имат оценка Добър(4) по дисциплината “Физика I” са двама (виж Фиг. 13.5).

```
{
 "name": {
  "firstname": "Стоян",
  "family": "Иванов",
  "lastname": "Стоянов"
 }
}
```

```
{
 "name": {
  "firstname": "Иван",
  "family": "Иванов",
  "lastname": "Иванов"
 }
}
```

Фиг. 13.5 Резултат, получен след изпълнение на кода от Фиг. 13.4

Задача 3: *Разпечатайте имената на всички студенти, които имат поне една оценка Отличен(6). Използвайте Mango заявка. За колко време се изпълнява заявката?*

Тъй като оценките се намират в масив “subjects”, а той от своя страна в масив “semester” ще използваме два пъти оператор “$elemMatch” (виж Фиг. 13. 6). За да получим автоматично статистика за изпълнението на заявката ще използваме поле “execution_stats” със стойност true.

```
{
   "selector": {
      "semester": {
         $elemMatch: {
            "subjects": {
               $elemMatch: {
                  "grade": {"$eq":6}
               }
            }
         }
      }
   },
   "fields": ["name"],
   "execution_stats": true
}
```

Фиг. 13.6 Mango заявка – Задача 3

Резултатът, който се получава, е следния:

```
{ "firstname": "Иван", "family": "Иванов", "lastname": "Иванов" }
{ "firstname": "Надя", "family": "Иванова", "lastname": "Петкова" }
```

Времето за изпълнение на заявката е 2.028 ms.

Задача 4: *Разпечатайте имената на всички студенти, които са от 1-ви или 2-ри курс и всичките им оценки за 2-ри семестър са по-високи от Среден(3). Използвайте Mango заявка. За колко време се изпълнява заявката?*

Тъй като условието за филтриране на студентите е всички оценки да са по-високи от Среден(3) то трябва да използваме оператор “$allMatch” приложен за поле “grade”. За да зададем за кои курсове търсим оценки, ще използваме оператор “$in” (виж Фиг. 13.7).

```
{
   "selector": {
      "course": {"$in": [1,2]},
      "semester": {
         "$elemMatch": {
            "number": 2,
            "subjects": {
               "$allMatch": {
                  "grade": {
                     "$gt": 3
                  }
               }
            }
         }
      }
   },
   "fields": [
      "course","name"
   ],
   "execution_stats": true
}
```

Фиг. 13.7 Mango заявка – Задача 4

Резултатът, който се получава, е следния:

	course	name
1		{ "firstname": "Стоян", "famil...
2		{ "firstname": "Надя", "family...

Времето за изпълнение на заявката е 2.062 ms.

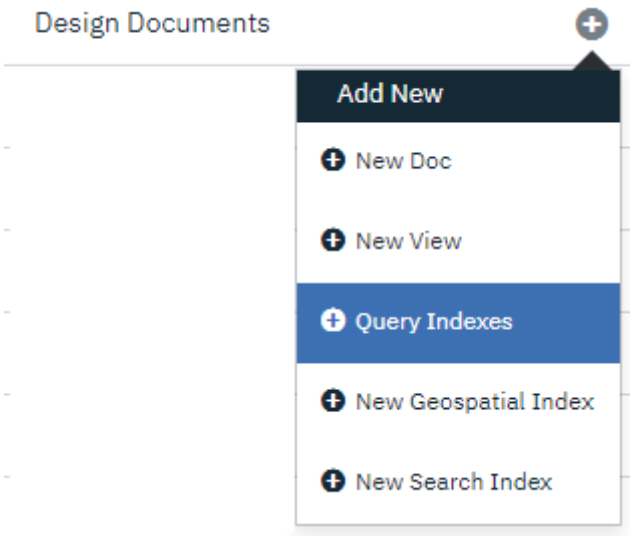
Задача 5: *Допълнете Задача 4 така, че имената на студентите да са сортирани по име от А към Я.*

Сортирането се реализира чрез поле “sort”. За да сортираме студентите по име възходящо, стойността на полето трябва да е “asc” (виж Фиг. 13.8)

```
{
   "selector": {
       ...
   },
   "sort": [{"name": "asc"}]
   "fields": [
     "course","name"
   ],
   "execution_stats": true
}
```

Фиг. 13.8 Mango заявка – Задача 5

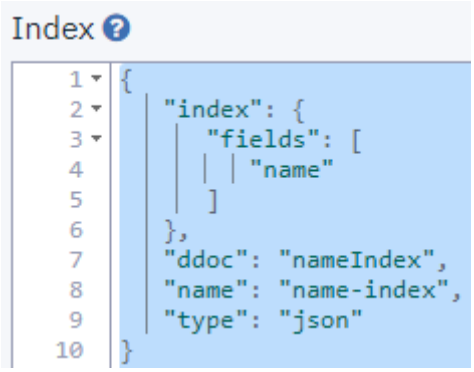
При стартиране на заявката ще получите грешка – липсва индекс за поле “name”. Тъй като сортирането е бавна операция, особено при големи по размер бази данни е необходимо да индексираме базата преди да изпълним заявката. За целта ще създадем необходимия индексен файл. Чрез интерфейса на IBM Cloudant изберете базата данни “students2”. От менюто изберете “Design Documents +”, а след това “+ Query Indexes”, както е показано на Фиг. 13.9.



Фиг. 13.9 Създаване на индексен файл

Ще получите достъп до документ-шаблон, на базата на който се създава индексния файл. Трябва да попълните стойностите на следните полета (виж Фиг. 13.10):

- "**fields**" – масив, който съдържа полетата от документа, които се използват за индекси ("name").
- "**ddoc**" – име на дизайн документа, който ще съдържа индекса. Ако го пропуснете файлът ще се именува автоматично.
- "**name**" – име на индекса.



Фиг. 13.10 Примерно съдържание на шаблона за създаване на индекс

За по-голяма ефективност индексите могат да бъдат групирани в един или няколко дизайн документи. Трябва да имате предвид, че промяната на един индекс в дизайн документ ще обезсили действието на всички други индекси в същия документ. По добрата практика е да създавате по един дизайн документ за всеки индекс. При конкретния пример дизайн документа, който се създава, е с име "_disign/nameIndex". Неговото съдържание се генерира автоматично (виж Фиг. 13.11).

Необходимите индексни файлове могат да се генерират и програмно. За целта се използва функция **_index**. Данните с шаблона за индекса (виж Фиг. 13.10) се изпращат чрез HTTP POST заявка към сървъра. Преди да изпратите _index заявка създайте JSON обект с шаблона:

```
var query = {
  "index": {
    "fields": [
      "name"
    ]
  },
  "ddoc": "nameIndex",
  "name" : "name-index",
  "type" : "json"
};
```

Използвайте функция **ajax** от библиотека jQuery. Поле "data" трябва да съдържа данните от шаблона като низ. За целта използвайте метод **stringify**:

```
data: JSON.stringify(query)
```

```
{
  "_id": "_design/nameIndex",
  "_rev": "1-8fef59ad59a510b2a2867a0ed5bcd960",
  "language": "query",
  "views": {
    "name-index": {
      "map": {
        "fields": {
          "name": "asc"
        },
        "partial_filter_selector": {}
      },
      "reduce": "_count",
      "options": {
        "def": {
          "fields": [
            "name"
          ]
        }
      }
    }
  }
}
```


Фиг. 13.11 Съсържание на индексен файл nameIndex

След като имате индексен файл за желаното поле, можете да използвате сортиране на документите. Сървърът автоматично ще избере необходимия индексен файл. Ако имате няколко вида индекси за едно и също поле трябва да укажете с кой точно индекс искате да работите. За целта използвайте поле “use-index” в Mango заявката. Стойността на полето е масив. Първият елемент на масива е името на индексния файл, а на втория - името на индекса. При конкретния пример, трябва да въведете следното съдържание:

```
"use_index": [
  "nameIndex",
  "name-index"
]
```

Резултатът, който се получава, при използване на желаното филтриране, е следния:

course	name
2	{ "firstname": "Надя", "family...
1	{ "firstname": "Стоян", "famil...

Задача 6: *Напишете HTML5 приложение, което позволява изпращане на Mango заявки към IBM Cloudant база данни "students2".*

Ще използваме програмния код на приложението от Задача 2, Упражнение № 12. На базата на това приложение (CLOUDANT_EX_1) създайте нов HTML5 NetBeans проект с име CLOUDANT_EX_2. Необходимо е да промените програмния интерфейс като предоставите област за въвеждане на текст (**textarea**). В тази област клиентите на приложението ще въвеждат Mango заявки. Следва списък на корекциите, които трябва да направите:

1. Модул index.html

В този модул ще променим съдържанието само на секция "query-box" като въведем поле за въвеждане на тест чрез HTML етикет textarea:

```
<section data-position="query-box">
    <div>
        <textarea id="query" placeholder="Вашата заявка..."
                  rows="10" cols="35"></textarea>
    </div>
    <button id="submit">Изпрати заявката</button>
</section>
```

2. Модул mystyle.css

В този модул ще зададем центриране на съдържанието в "query-box" и ще създадем ново CSS правило за етикет <textarea> с идентификатор "query":

```
[data-position="query-box"] {
    overflow: hidden;
    text-align: center;
}
textarea#query {
    padding: 5px 10px;
    border: 2px solid #aaa;
    font-family: monospace;
    color: rgb(0,0,150);
}
```

3. Модул display.js

Тъй като трябва да визуализираме отговора на сървъра след всяка Mango заявка трябва да променим съдържанието на функция **showInfo** от модул display.js. Вместо да използваме функция **append** за добавяне на съдържание в "info-box" ще използваме функция **html**:

```
function showInfo(data) {
    $('[data-position=info-box]').html(data);
    view('info-box');
}
```

4. Модул mycode.js

След като получим отговор е желателно да можем да се върнем към въвеждане на нова заявка или редактиране на текущо въведена заявка. За целта ще предоставим възможност на клиентите да се върнат към секция въвеждане на заявка чрез щракване с мишката в полето с отговора. За целта ще прехванем събитие "click" за DOM обект с идентификатор "info-box". При генериране на това събитие следва визуализиране на секция "query-box" чрез функция view:

```
$('[data-position="info-box"]').click(function() {
    view('query-box');
});
```

5. Модул database.js

В този модул ще има най-много промени. Първо, трябва да променим използваните константи, тъй като работи с друга база данни и заявката има друг формат. Използваните константи са показани на Фиг. 13.12.

```
var PORT = 443;
var TIMEOUT = 4000;
var PROTOCOL = 'https://';
var SERVER_NAME = 'dc3b328b-545c-4c02-9457-531fdc110a3e-
bluemix.cloudantnosqldb.appdomain.cloud';
var DATABASE = 'students2';
var KEY = 'apikey-fe27795948f14d4fb4c463eff484fb35';
var PASSWORD = '6aed1b959f1091ae9b824613590aeeda08398b6f';
```

Фиг. 13.12 Използвани константи – Задача 6

Програмния код от функция **addResultToPage** е кратък:

```
function addResultToPage(result) {
    var resAsString = JSON.stringify(result, null, " ");
    var resAsHtml = resAsString.replace(/\n/g,"<br/>");
    showInfo(resAsHtml);
}
```

Фиг. 13.13 Функция addresultToPage – Задача 6

Първо, чрез функция **stringify** конвертиране отговора от JSON обект до низ. Задали сме да няма филтриране на полета (null стойност за втория аргумент), а за по-добра четимост вместо табулатор сме използвали специалния HTML код “emsp;” чрез който се въвежда интервал с ширина, равна на ширината на буква ‘m’. След това чрез функция **replace** и използване на регулярен израз сме заместили всички кодове за нов ред “\n” с “
”.

```
function sendQuery() {
  var hash = btoa(KEY + ':' + PASSWORD);
  getResult(hash)
    .then(function (result) {
      addResultToPage(result);
    }).catch(
      function (error) {
        var status = error.status;
        if (status === 0) {
          showError('Невъзможна е връзката с базата данни!', 3);
        }
        else if (status === 401) {
          showError('Грешка при авторизация!', 3);
        }
        else {
          showError('Грешен синтаксис на заявката!', 3);
        }
      }
  );
}
```

Фиг. 13.14 Функция sendQuery – Задача 6

Трябва да променим и кода на функция **sendQuery**, както е показано на Фиг. 13.14. При успешно обслужена заявка (**then**) веднага се вика функция **addResultToPage** на която се предава обекта “result”. При грешка (**catch**) се анализира статус кода на грешката. При липса на мрежова свързаност този код е 0. При грешна двойка “ключ-парола” стойността на статус кода е 401. Приемаме, че в противен случай (код 400), че най-вероятната грешка е грешен синтаксис на заявката.

Остана да променим кода на функция **getResult** (виж Фиг. 13.15)

```
function getResult(hash) {
    var url = PROTOCOL + SERVER_NAME + '/' + DATABASE + '/_find';
    var query = $("textarea").val();
    if (query === '') {
        showError('Моля, въведете вашата Mango заявка!', 3);
        return;
    }
    var promise = new Promise(function (resolve, reject) {
        $.ajax({
            type: "POST",
            url: url,
            cache: false,
            contentType: "application/json",
            dataType: "json",
            data: query,
            timeout: TIMEOUT,
            beforeSend: function (req) {
                req.setRequestHeader('Accept', 'application/json');
                req.setRequestHeader('Authorization', 'Basic ' + hash);
            },
            success: function (data) {
                resolve(data);
            },
            error: function (data) {
                reject(data);
            }
        });
    });
    return promise;
}
```

Фиг. 13.15 Функция getResult – Задача 6

Формирайте променлива “url” така, че да се адресира ресурс **_find**. След това проверете стойността на DOM обект “textarea”. За целта може да използвате функция **val** от библиотека jQuery. Ако няма въведена информация се извежда съобщение за грешка. Ако има въведена информация се преминава към изпращане на заявка до ресурс _find. Типът на заявката трябва да бъде POST. Данните (низ), които се предават към ресурса, е стойността на поле “data”. За да разберем кога тази асинхронна заявка е обслужена ще използваме обект-обещание (promise).

На Фиг. 13.16 е показан резултат, получен от работа с потребителския интерфейс на приложението. Изпраща се заявка, която визуализира номера на курса и имената на студентите, които са от 1-ви курс.

Фиг. 13.16 Програмен интерфейс на приложението – Задача 6

Литература

1. Росен Иванов, JavaScript: Въведение в програмирането, Издателство "Фабер", 2018.
2. Филип Петров, Цветанка Георгиева-Трифонова, Нерелационни бази от данни. Практическо ръководство, УИ „Св. Климент Охридски”, 2019.
3. Axel Rauschmayer, JavaScript for impatient programmers, ISBN 978-1-09-121009-7, 2019.
4. Chris Anderson, Jan Lehnardt, Noah Slater, CouchDB: The Definitive Guide, O’Reilly Media, Inc., ISBN: 978-0-596-15589-6, 2010.
5. Christopher Bienko, Marina Greenstein, Stephen E Holt, Richard T Phillips, IBM Cloudant: Database as a Service Fundamentals, IBM RedPaper, 2015.
6. Christopher D. Bienko, Marina Greenstein, Stephen E Holt, Richard T Phillips, IBM Cloudant Database as a Service Advanced Topics, IBM RedPaper, 2015.
7. Copeland, R., MongoDB Applied Design Patterns: Practical Use Cases with the Leading NoSQL Database", ISBN: 978-1449340049, O'Reilly Media, Inc.", 2013.
8. Dayley, B., Dayley, B., Dayley, C., Node. js, MongoDB and Angular Web Development: The definitive guide to using the MEAN stack to build web applications. Addison-Wesley Professional., ISBN: 978-0134655536, 2017.
9. Guy Harrison. Next-Generation Databases: NoSQL and Big Data, Apress, 2015.
10. Richard Clark, Oli Studholme, Christopher Murphy and Divya Manian, Beginning HTML5 and CSS 3, Apress, ISBN 978-1-4302-2874-5, 2012.
11. Sadalage, P., M. Fowler, NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence, Pearson Education, 2012.
12. Satheesh, M., D'mello, B. J., Krol, J, Web development with MongoDB and NodeJs. Packt Publishing Ltd., ISBN: 981-1-79528-752-7, 2015.
13. Urma, R. G., Fusco, M., & Mycroft, A., Java 8 in action, Manning publications, 2014.

Нерелационни бази данни

Ръководство за лабораторни упражнения

Автор

Росен Стефанов Иванов

Рецензент

проф. дмн Стоян Недков Капралов

Националност българска

Първо издание

Предпечатна подготовка

Росен Стефанов Иванов

http://kst.tugab.bg/nosql