გია სურგულაძე, გიორგი კივილაძე

შესავალი NoSQL მონაცემთა ბაზებში

„IT-კონსალტინგის ცენტრი“

საქართველოს ტექნიკური უნივერსიტეტი

გია სურგულაძე, გიორგი კივილაძე

შესავალი NoSQL მონაცემთა ბაზებში (MongoDB)

დამტკიცებულია:
სტუ-ს „IT-კონსალტინგის"
სამეცნიერო ცენტრის
სარედაქციო-საგამომცემლო
კოლეგიის მიერ

თბილისი
2017

უაკ 004.65

განხილულია მართვის საინფორმაციო სისტემებში განსაკუთრებით დიდი მონაცემების დამუშავების და შენახვის თანამედროვე მეთოდები და ინსტრუმენტული საშუალებები. კლასიკური რელაციური მონაცემთა ბაზების გვერდით წარმოდგენილია დღეისათვის ერთ-ერთი აქტუალური ტექნოლოგია – NoSQL, რომელიც, ხშირად, ახალი ტიპის არარელაციურ მონაცემთა ბაზების ოჯახის სახით განიხილება. აქვე, უფრო დეტალურად, მოცემულია მისი კონკრეტული რეალიზაციიის მაგალითი – MongoDB პაკეტი. შედარებულია რელაციური და არარელაციური ბაზების გამოყენების მახასიათებლები, მათი დადებითი და უარყოფითი მომენტები მართვის საინფორმაციო სისტემებში. განსაზღვრულია ასეთი მონაცემთა ბაზების ეფექტური გამოყენების სფეროები და პირობები. წიგნში განსაკუთრებული ყურადღება გამახვილებულია NoSQL ტიპის მონაცემთა ბაზებზე, მათ სტრუქტურაზე და კომპონენტებზე, მომხმარებლის სამუშაო გარემოზე და დიდი ბაზების (Big Data) ადმინისტრირების პრობლემებზე.

დამხმარე სახელმძღვანელო განკუთვნილია ინფორმატიკისა და მართვის საინფორმაციო სისტემების სპეციალობის სტუდენტებისა და მაგისტრანტებისთვის, ასევე, აღნიშნული საკითხებით დაინტერესებული მკითხველისთვის.

რეცენზენტები:

- პროფ. ე. თურქია (სტუ)
- ასოც.პროფ. დ. გულუა (თსუ)

რედკოლეგია:

ა. ფრანგიშვილი (თავმჯდომარე), მ. ახობაძე, გ. გოგიჩაიშვილი, ზ. ბოსიკაშვილი, ე. თურქია, რ. კაკუბავა, ნ. ლომინაძე, ჰ. მელაძე, თ. ობზაძე, რ. სამხარაძე, გ. სურგულაძე (რეადაქტორი), გ. ჩაჩანიძე, ა. ცინცაძე, ზ. წვერაიძე

ISBN 978-9941-0-9642-6

შინაარსი

შესავალი

მონაცემთა რელაციური ბაზების თეორია სათავეს 70-იანი წლებიდან იღებს, როდესაც გამოჩნდა ედგარ კოდის სტატიები მონაცემთა რელაციური მოდელების შესახებ [1,2]. მანამდე კი აშშ-ის და ევროპის განვითარებული ქვეყნების უნივერსიტეტებში ფართოდ გამოიყენებოდა და წარმოებაში ინერგებოდა არარელაციური ტიპის ბაზები, ე.წ. იერარქიული და ქსელური ბაზები [3,4,5]. მათზე საკმაო ლიტერატურაა შექმნილი, ამიტომ ჩვენ აქ მათ არ განვიხილავთ.

80-იანი წლებიდან დაიწყო სწრაფი განვითარება და 21-ე საუკუნიდან ინფორმაციული ტექნოლოგიების ბაზარზე უკვე გამეფდა რელაციური მონაცემთა ბაზის სისტემები: Oracle, Ms SQL Server, MySQL, Ms Access, Sybase, DB/2, PostgreSQL და სხვ. [5,6]. ასეთ სისტემებს თამამად შეიძლება ვუწოდოთ კლასიკური მონაცემთა ბაზები, რომელთა როლი და გამოყენების არეალი დღესაც დიდია.

რაც შეეხება სიახლეს ამ სფეროში და ჩვენი წიგნის ძირითად მიზანს, ესაა „ძველი“ ახალი ტიპის, ანუ არარელაციური მონაცემთა ბაზების, ე.წ. NoSQL ტიპის ბაზების საფუძვლების გაცნობა, მათი შედარება რელაციურ მონაცემთა ბაზებთან. ანალიზის საფუძველზე კი გარკვეული დასკვნის გაკეთება შეიძლება, თუ როდის უნდა გამოვიყენოთ რელაციური და როდის NoSQL ტიპის ბაზები, საინფორმაციო სისტემების ეფექტიანი მუშაობის ორგანიზების თვალსაზრისით.

წიგნის 1-ელ თავში განხილულია მონაცემთა კლასიკური და ახალი ბაზების თეორიული ასპექტები. მათი ძირითადი დადებითი და უარყოფითი მხარეები პრაგმატული თვალსაზრისით. მახასიათებლების შედარება. SQL-ენის ოჯახის წარმომადგენლად აქ განიხილება MySQL პაკეტი. გადმოცემულია აგრეთვე ობიექტ-ორიენტირებული, მულტიმედიური და ობიექტრელაციური მონაცემთა ბაზების დამახასიათებელი ნიშანთვისებები.

არარელაციური ბაზების იდეოლოგია განხილულია დოკუმენტ-ორიენტირებული, გრაფულ-ორიენტირებული, ჰიბრიდული და ახალიო ტიპის ბაზების მაგალითზე. როგორიცაა NewSQL მონაცემთა ბაზები. აქვე მოკლედ არის განხილული დიდ მონაცემთა (BigData) ახალი ტექნოლოგია Hadoop-ის სახით და მისი აქტუალობა დღეისათვის.

მე-2 თავი მთლიანად ეხება NoSQL ტიპის ბაზების კონცეფციის საფუძვლების გაცნობას. განხილულია მათი ფუნქციონირების ძირითადი ასპექტები და პრინციპები. კერძოდ, ნაშრომში დეტალურად არის წარმოდგენილი მონაცემთა ბაზების ვერტიკალური და ჰორიზონტალური ორგანიზაცია, მონაცემთა შენახვის ტიპები: „გასაღები-მნიშვნელობა“, „დოკუმენტ-ბაზირებული“, „სვეტებზე ბაზირებული“ და „გრაფზე ბაზირებული“.

განსაკუთრებული ყურადღება აქვს დათმობილი მონაცემთა მთლიანობის (Data integrity) და ტრანზაქციათა იზოლირების დონეების გაცნობას. მნიშვნელოვანია აგრეთვე ACID პრინციპებისა და CAP სამკუთხედის თეორემის განხილვა. ბოლოს, ამ თავში შემოთავაზებულია სისტემის განაწილებული გარემოს წარმოდგენა replication პროცესისა და sharding ტექნოლოგიის საფუძველზე, რაც მნიშვნელოვნად უწყობს ხელს მონაცემთა სინქრონიზაციას და მათი წვდომის ეფექტურობის ამაღლებას.

მე-3 თავი ეხება NoSQL ოჯახის ტიპურ წარმომადგენელს - MongoDB-ს, რომელიც დოკუმენტებზე ორიენტირებული ბაზების მართვის სისტემაა [7]. განხილულია მისი სამუშაო გარემო, გრაფიკული ინტერფეისი RoboMongo პლატფორმის სახით. აღწერილია მონაცემთა ბაზის აგებისა და მასთან მუშაობის კონრეტული მაგალითები MongoDB სისტემაში.

დანართში წარმოდგენილია MongoDB ბაზის აგების და გამოყენების ერთ-ერთი პოპულარული მაგალითი, რომელიც მკითხველს გაუადვილებს წიგნში გადმოცემული თეორიული მასალის პრაქტიკულ გათვითცნობიერებას.

I თავი
მონაცემთა ბაზების კლასიკური და ახალი ტექნოლოგიები

მონაცემთა ბაზების მართვის სისტემა (მბმს, Database Management System, DBMS) – არის პროგრამული უზრუნველყოფა (Software), რომლის დანიშნულებაცაა ამ ბაზის მონაცემთა გამოყენება და მოდიფიკაცია მრავალმომხმარებლურ რეჟიმში.

წინამდებარე თავში განვიხილავთ მონაცემთა რელაციურ ბაზას, როგორც კლასიკური ტიპის ბაზას, რომელზეც მოთხოვნილება საკმაოდ მაღალია. ობიექტურ ბაზებს, ობიექტ-რელაციურ ბაზებს და არარელაციურ NoSQL ბაზებს. ამ უკანასკნელის თვალსაზრისით ჩვენ შევეხებით დოკუმენტ-ორიენტირებულ და გრაფულ-ორიენტირებულ მონაცემთა ბაზებს.

1.1. რელაციური მონაცემთა ბაზა

რამდენიმე ათეული წელია, რაც მართვის ავტომატიზებულ სისტემებში ინფორმაციის შენახვისა და გადამუშავების ყველაზე ეფექტური და ფართოდ გამოყენებადი მონაცემთა ბაზები ეფუძნება ედგარ კოდის რელაციურ მოდელს, Oracle, Ms SQL Server, SyBase, MsAccess, MySQL და სხვ. [1-3]. ასეთი ბაზებია დანერგილი დღეს სახელმწიფო და კერძო სტრუქტურების უმრავლეს ორგანიზაციასა და დიდ კორპორაციაში.

მონაცემთა რელაციურ მოდელს საფუძვლად უდევს სიმრავლური მიმართების მათემატიკური ცნება. ამავე დროს, მიმართება მოცემულ მოდელში შეიძლება იყოს წარმოდგენილი ცხრილის სახით, სადაც ცხრილის სვეტები მიმართების თვისებებია ან ატრიბუტები. ავიღოთ n სიმრავლეთა $A_1, A_2, \ldots, An$ ერთობლიობა. R მიმართებას ამ სიმრავლეებზე ეწოდება დეკარტული ნამრავლის ქვესიმრავლე, რომელიც $(a_1, a_2, \ldots, a_n)$ სახისაა, სადაც $a_i \in A_i$, $i=(1,n)$.

ამავე დროს, მიმართების მათემატიკური განსაზღვრებიდან შეგვიძლია ჩავწეროთ:

$$R \subseteq A_1 \times A_2 \times \ldots \times A_n.$$

სიმრავლეთა $A_1, A_2, \ldots, An$ ერთობლიობა არის R მიმართების განსაზღვრის არე, ხოლო A_i სიმრავლეს უწოდებენ დომენს. R მიმართების ელემენტებს უწოდებენ კორტეჟებს ან ამონაკრებებს.

რელაციური მიმართების ცხრილი ორგანზომილებიანია, რომლის სტრიქონები შეესაბამება ატრიბუტების მნიშვნელობების კორტეჟს, ხოლო სვეტები - ატრიბუტებს. აქვე აღვნიშნოთ, რომ ატრიბუტი განსაზღვრულია დომენის სიმრავლეზე. დავუშვათ, C_i არის ატრიბუტი, მაშინ $C_1 \subseteq A_1$, $C_2 \subseteq A_2$, . . . , $C_n \subseteq n_1$. R მიმართების სიმძლავრე (კარდინალური რიცხვი) განისაზღვრება კორტეჟების რაოდენობით.

1.1 ნაზაზზე მოცემულია რელაციური მოდელის მაგალითი ცხრილური ფორმით.

აკადემიური–ჯგუფი

ჯგუფის ნომერი	სპეციალობა	სპეციალობის შიფრი	სტუდენტების რაოდენობა	სექტორი
108435	მას	2202	14	ქართული
608536	კაქს	2201	16	ქართული
108739	ამტს	2101	11	რუსული
108638	სსტ	1906	12	ინგლისური
608534	მას	2202	15	ქართული

ნახ.1.1. მონაცემთა რელაციური ცხრილი

რელაციური მიმართების სახელია **აკადემიური_ჯგუფი.** თუ მას აღვნიშნავთ R-ით, მაშინ გვექნება:

$$R \subseteq A_1 \times A_2 \times A_3 \times A_4 \times A_5.$$

სადაც სადაც A_1 დომენი არის ჯგუფის ნომერი, A_2 დომენი - სპეციალობა, A_3 - სპეციალობის შიფრი, A_4 - სტუდენტების რაოდენობა, A_5 სექტორი.

დომენები შეიცავს შემდეგ მნიშვნელობებს:

A_1 = {108435, 608536, 108739, 108638, 608534};
A_2= {მას, კქს, ამს, სსტ};
A_3= {2202, 2201, 2101, 1906};
A_4= {14, 16, 11, 12, 15};
A_5= {ქართული, ინგლისური, რუსული}.
და ა/შ.

მონაცემთა ბაზების თეორიაში განიხილავენ მონაცემთა მოდელირების სამ - კონცეპტუალურ, ლოგიკურ და ფიზიკურ დონეებს. აგრეთვე მონაცემთა სტრუქტურების ოპტიმიზაციის საკითხებს ნორმალურ ფორმათა თეორიის საფუძველზე [4].

მონაცემთა რელაციურ ბაზებზე და მათ მოდელებზე საკმაო ლიტერატურული წყაროებია ქართულ ენაზეც [2-6], ამიტომ აქ ჩვენ მათ დეტალურად არ განვიხილავთ, ვინაიდან წიგნის მიზანი უფრო ახალი, არარელაციური NoSQL ბაზის გაცნობაა.

ამჯერად ჩვენ განვიხილავთ მონაცემთა რელაციური ბაზების ოჯახის მთავარ ენას SQL-ს, მის ძირითად სტრუქტურასა და ოპერაციათა სინტაქსს.

შემდეგ, საილუსტრაციო მაგალითის სახით განვიხილავთ მონაცემთა რელაციური ბაზების მართვის ერთ-ერთ პოპულარულ სისტემას - MySQL -ს.

1.1.1. SQL ენა

განვიხილოთ SQL (Structured Query Language) მოთხოვნების სტრუქტურირებადი ენა, რომელსაც იყენებს თითქმის ყველა რელაციური მონაცემთა ბაზის მართვის სისტემა.

აქ უნდა ვახსენოთ ასევე ორი ენა - მონაცემთა აღწერის ენა (DDL – Data Definition Language) და მონაცემთა მანიპულირების ენა (DML – Data Manipulation Language). პირველი გამოიყენება მონაცემთა ბაზის ობიექტების გამოცხადების (შექმნის, მოდიფიკაციის) დროს, ხოლო მეორე - ამ ობიექტების დამუშავებისას [5].

SQL ენა – მოთხოვნების სტანდარტული ენაა მონაცემთა რელაციურ ბაზებში. იგი შეიქმნა ე. კოდის რელაციური ალგებრის ენის საფუძველზე. SQL ენის დანიშნულებაა ბაზის ცხრილებზე (Tables) ოპერაციების ჩატარება: შექმნა, წაშლა, მოდიფიკაცია, მონაცემთა ამორჩევა, ცვლილება, დამატება განსაზღვრული პირობებით. ცხრილებთან მუშაობისას იქმნება წარმოდგენები (Views), ანუ ესეც კომბინირებული ცხრილია არსებულების საფუძველზე, რომლებშიც ატრიბუტების (სვეტების) და კორტეჟების (სტრიქონების, ჩანაწერების) შედგენილობა დამოკიდებულია შემოსულ მოთხოვნაზე.

როგორც ცნობილია, ყველა რელაციურ ბაზის სისტემას გააჩნია მოთხოვნების აგების როგორც სკრიპტული ფორმა, ასევე გრაფიკული ინსტრუმენტული საშუალება. იგი მნიშვნელოვნად ამარტივებს მომხმარებელთა მუშაობას ბაზასთან, მოთხოვნების ფორმირების პროცესის თვალსაზრისით.

SQL ენის ოპერატორები პირობითად შეიძლება დავყოთ სამ ძირითად ჯგუფად:

- SELECT ოპერატორი;
- მონაცემთა მანიპულირების ოპერატორები;
- მონაცემთა აღწერის ოპერატორები.

➢ SELECT ოპერატორის სინტაქსი ასეთია:

SELECT [ALL| DISTINCT] <ველების სია>|*
FROM <ცხრილების სია>
[WHERE <პირობის-პრედიკატი ამორჩევისათვის ან შეერთებისათვის>]
[GROUP BY <შედეგის ველების სია>]
[HAVING <პირობის-პრედიკატი ჯგუფისთვის>]
[ORDER BY <ველების სია შედეგის მოწესრიგებისთვის>];

სადაც:

ALL - *ყველა სტრიქონია (დასაშვებია დუბლირება);*

DISTINCT - *უნიკალური სტრიქონები (არაა დუბლირება);*

WHERE – *შედარების პრედიკატები:* **(=, <>, >, >=, <, <=)**

GROUP BY – *მოიცემა ველების სია დაჯგუფების მიზნით;*

HAVING - *ახდენს ჩანაწერების ფილტრაციას ჯგუფში;*

ORDER BY - *ალაგებს შედეგს ველების სიის მიხედვით*

და ა.შ.

- მონაცემთა მანიპულირების ენაზე (DML) მოთხოვნები იწერება ბაზაში კორტეჟების დასამატებლად, წასაშლელად და შესაცვლელად. მათი სინტაქსი ასეთია:

INSERT INTO ცხრილის_სახელი [(<სვეტების სია>)]
VALUES (<მნიშვნელობათა სია>)

DELETE FROM <ცხრილის_სახელი> [WHERE <ამორჩევის_პირობა>]

თუ ამორჩევის პირობა არაა მითითებული, მაშინ ცხრილში წაიშლება ყველა ჩანაწერი.

მონაცემთა განახლების ოპერაცია *UPDATE* შედგება სამი ნაწილისგან:

- ***UPDATE წინადადება,*** მიუთითებს განსაახლებელ ცხრილს;
- ***SET წინადადება,*** იძლევა განსაახლებელ მონაცემებს;
- ***WHERE არააუცილებელი კრიტერიუმი,*** ზღუდავს ჩანაწერთა რაოდენობას, რომელზეც უნდა იმოქმედოს განახლების მოთხოვნამ.

- მონაცემთა სქემის აღწერის ენა (SDL, Schema Definition Language) ან (DDL) - არის SQL-ის ინსტრუქციები, რომლებიც უზრუნველყოფს მონაცემთა ბაზის სტრუქტურის ელემენტების შექმნას და მოდიფიცირებას.

- ცხრილის შექმნის ოპერატორს აქვს შემდეგი სინტაქსი:

CREATE TABLE <ცხრილის სახელი>
(<სვეტის სახელი> <მონაცემთა ტიპი> [NOT NULL]
[,<სვეტის სახელი> <მონაცემთა ტიპი> [NOT NULL]]…)

- ცხრილის წაშლის ინსტრუქცია:

DROP TABLE <ცხრილის სახელი>

- ცხრილის სტრუქტურის მოდიფიკაცია (ველების დამატება, წაშლა, მათი ტიპების შეცვლა):

ALTER TABLE <ცხრილის სახელი>
MODIFY | ADD| DROP <ველის სახელი> [<მონაცემთა ტიპი>]

- ცხრილის შექმნის შემდეგ შესაძლებელია ინდექსების შექმნა CONSTRAINT წინადადებით:

CREATE [UNIQUE] INDEX <ინდექსის სახელი>
ON <ცხრილის სახელი>
(<სვეტის სახელი> [ASC | DESC]
[, <სვეტის სახელი> [ASC | DESC]…)

ინდექსების შექმნა ცხრილის ერთი ან არამდენიმე ველისთვის უზრუნველყოფს მოთხოვნის შესაბამისი საძიებო ოპერაციების დაჩქარებას. ინდექსების წაშლა ხდება ინსტრუქციით:

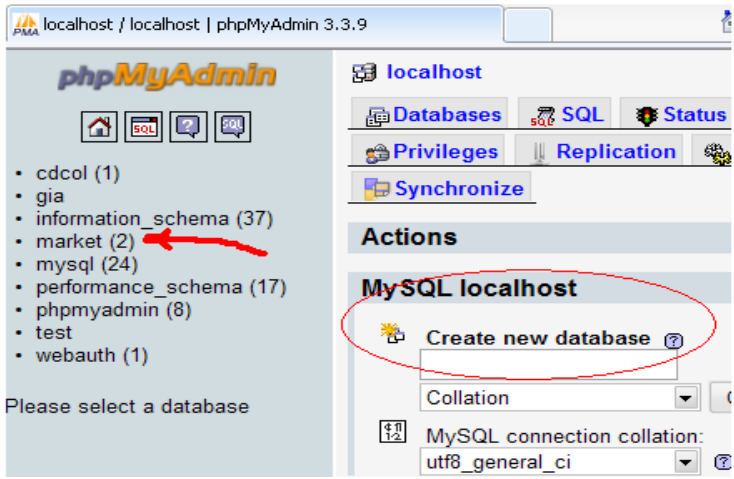
DROP INDEX <ინდექსის სახელი> ON<ცხრილის სახელი>

1.1.2. MySQL მონაცემთა ბაზის მართვის სისტემა

განვიხილოთ MySQL მონაცემთა ბაზების მართვის სისტემა. იგი, როგორც განაწილებული რელაციური ბაზა ერთ-ერთი აქტუალური და ფართოდ გამოყენებადი კლიენტ-სერვერული პაკეტია დღეისათვის, განსაკუთრებით php ვებ-ტექნოლოგიებში .

ვგულისხმობთ, რომ MySQL სისტემა დაინსტალირებულია კომპიუტერზე (თუ არა, მაშინ ის ჩამოწერილ უნდა იქნას http://www.mysql.com საიტიდან და დაინსტალირდეს).

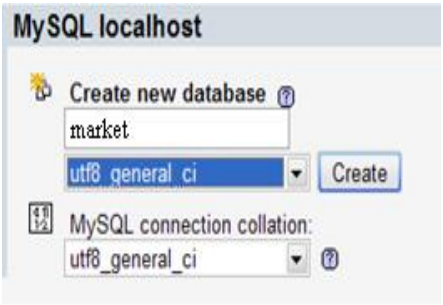
როგორც წესი, MySQL სისტემის გამოძახება ხდება რომელიმე ბრაუზერში, მაგალითად, Internet Explorer-ში url-მისამართად უნდა მიეთითოს: http://localhost/phpmyadmin/ სტრიქონი. 1.2 ნახაზზე გამოტანილია შედეგი.



ნახ.1.2

აქ ავირჩევთ ჩვენთვის საჭირო ბაზას, მაგალითად, market, რომელსაც აქვს 2 ცხრილი. თუ საჭიროა ახალი ბაზის შექმნა, მაშინ გამოვიყენებთ ველს: create new database. აქ ჩავწერთ ახალი ბაზის

სახელს, Collation ველ-ში ვირეჩევთ სტრიქონს utf8_general_ci, რაც უნიკოდის გამოყენების საშუალებას იძლევა (ნახ 1.3).



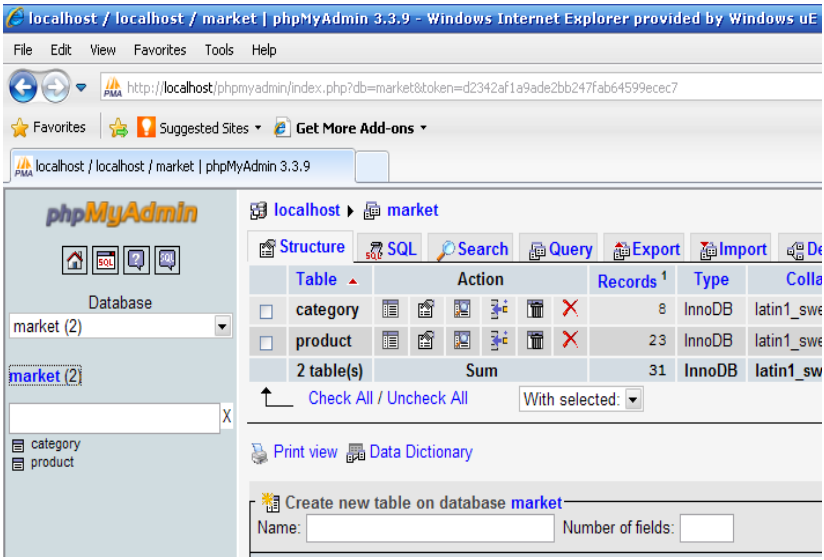
ნახ.1.3

შემდეგ უნდა შევქმნათ ბაზის ცხრილები, რისთვისაც ველში "Create new table on database": market - ჩავწერთ ცხრილის სახელს, მაგალითად, product ან category და დავიწყებთ ცხრილის ველების (სვეტების) შექმნას (მაგალითად, ნახ.1.4).

Field	Type	Length/Values[1]
pr_ID	INT	4
Name	VARCHAR	30
prize	DECIMAL	
cat_ID	INT	2

ნახ.1.4

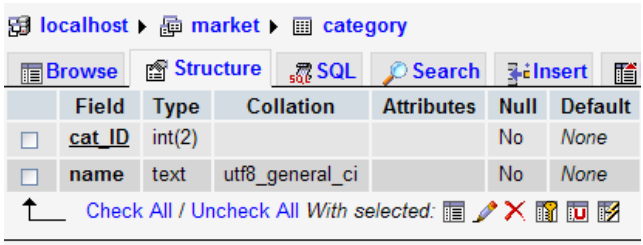
1.5 ნახაზზე ნაჩვენებია ცხრილები: category და product.



ნახ.1.5

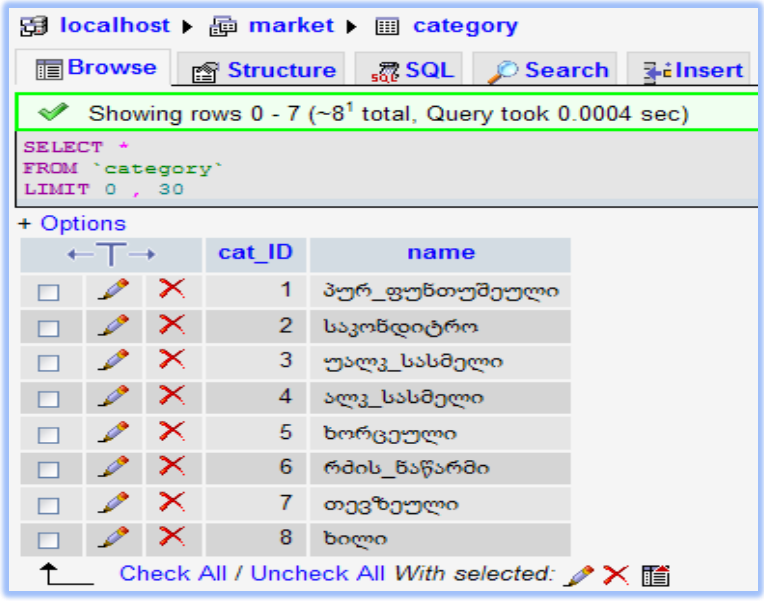
ავირჩიოთ category ცხრილი და მენიუში Structure.

1.6 ნახაზზე ნაჩვენებია შედეგი.



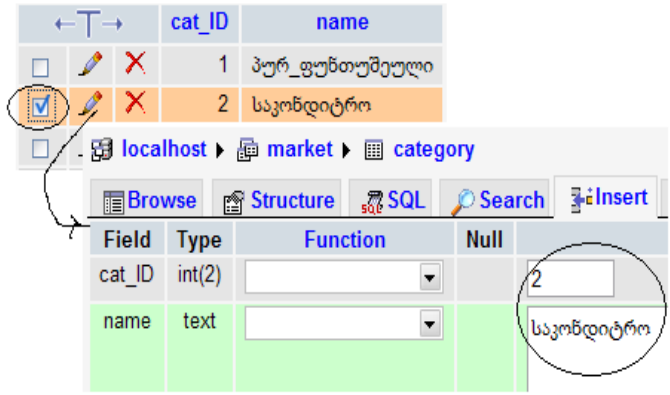
ნახ.1.6

cat_ID გასაღებური ველი (პირველადი ინდექსი) აქ გახაზულია. ახლა უნდა შევავსოთ სტრიქონები კონკრეტული მნიშვნელობებით. ამისათვის ავამოქმედებთ Browse გადამრთველს და გამოჩნდება 1.7 ნახაზზე მოცემული ფანჯარა.



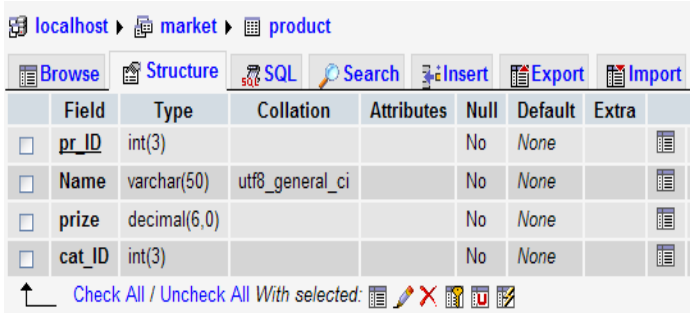
ნახ.1.7

„ჩეკბოქსის" ჩართვით და ფანქრის არჩევით შევალთ სტრიქონის რედაქტირების რეჟიმში და შევასრულებთ ჩვენთვის საჭირო ფუნქციას (ნახ.1.8).

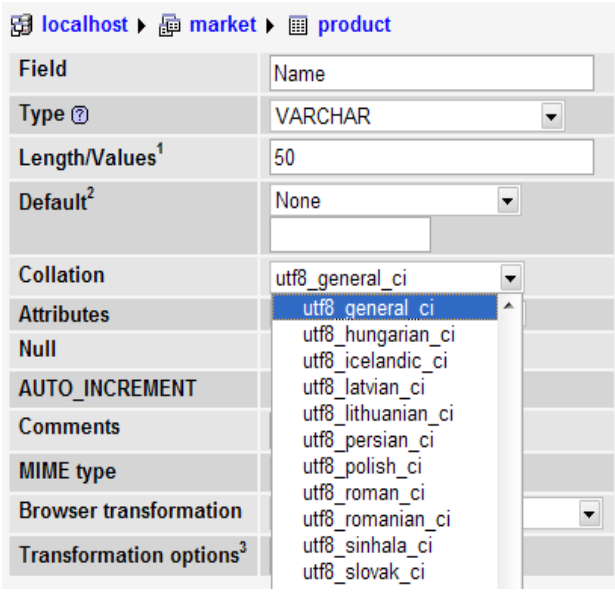


ნახ.1.8

ახლა გადავიდეთ product ცხრილზე. მისი ველების განსაზღვრით და სტრიქონების შეტანით შესაძლოა ქართული უნიკოდების მაგივრად ’????? ????’ სტრიქონის მიღება. ასეთი ველის სტრუქტურაში Collation სვეტში უნდა ჩავსვათ utf8_general_ci (ნახ.1.9-ა,ბ).

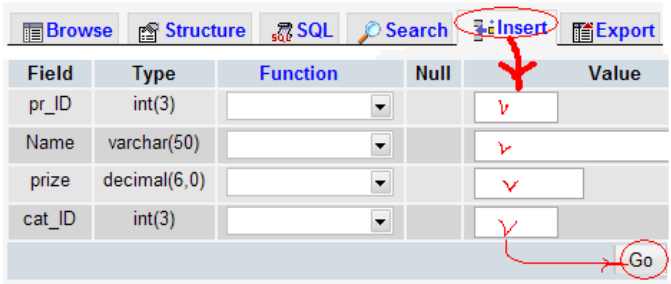


ნახ.1.9-ა



ნახ.1.9-ბ

შევავსოთ პროდუქციის ცხრილი მნიშვნელობებით Browse და Insert გადამრთველების გამოყენებით (ნახ.1.10).



ნახ.1.10

შევსებული ცხრილის ფრაგმენტი მოცემულია 1.11 ნახაზზე.

	pr_ID	Name	prize	cat_ID
	1	არაჟანი	4	6
	2	პური	1	1
	3	ძეხვი	11	5
	4	ხაჭო	3	6
	5	ფუნთუშა ქიშმიშით	1	1
	6	კოკა_კოლა	2	3
	7	ფანტა	2	3
	8	ღვინო ქინძმარაული	18	4
	9	ღვინო ხვანჭკარა	23	4
	10	ტირამისუ	7	2
	11	საქონლის ხორცი	18	5
	12	კონიაკი "ვარციხე"	20	4
	13	არაყი "გომი"	13	4
	14	ასატრინა	25	7
	15	ვაშლი "გოლდენი"	3	8
	16	კონსერვი "შპროტი"	5	7
	17	ფორთოხალი	3	8
	18	მინერალური "ბორჯომი"	2	3
	19	მინერალური "ნაბეღლავი"	1	3
	20	ორცხობილა "წუშის"	4	2
	21	წაყინი "ვანილის"	6	2
	22	მინერალური "ლიკანი"	2	3
	23	ღორის სამწვადე	23	5

Check All / Uncheck All With selected:

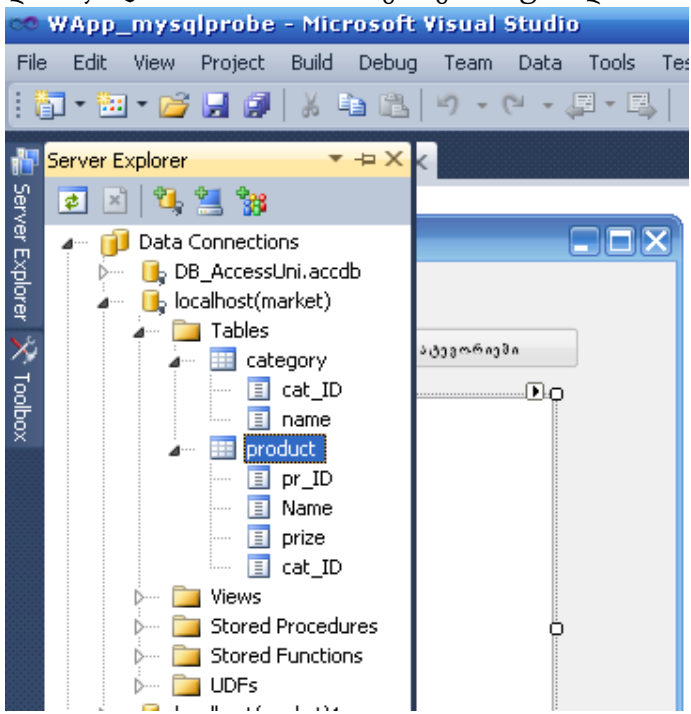
ნახ.1.11

ამგვარად, MySQL ბაზის ორი ცხრილი category და product მზადაა გამოსაყენებლად. დავხუროთ ეს ბაზა.

1.1.3. MySQL ბაზის დამუშავება C# ენის გამოყენებით

ახლა ავამუშავოთ Ms Visual Studio.NET პაკეტი და C# ენის WindowsForm რეჟიმში Form1-ზე გამოვიტანოთ MySQL ბაზის მონაცემები, სხვადასხვა ჭრილში, მოთხოვნების შესაბამისად.

1.12 ნახაზზე ნაჩვენებია Server Explorer-ის ფანჯარა, სადაც localhost(market) მიერთებულ ბაზაში ჩანს Tables: category და product თავიანთი ველებით (ატრიბუტებით). მთავარი მომენტია C# კოდიდან MySQL-ის market ბაზის მიერთება სამუშაოდ.



ნახ.1.12

პროგრამის კოდში უნდა მოთავსდეს MySQL ბაზის დასაკავშირებელი რამდენიმე სტრიქონი, რომელიც ქვემოთაა მოცემული.

```
MySql.Data.MySqlClient.MySqlConnection msqlConnection = null;
msqlConnection = new  MySql.Data.MySqlClient. MySqlConnection
     ("server=localhost;"+
      "User Id=user@localhost; database=market;
                                Persist Security Info=True");
```

აქ განსაზღვრულია კლიენტის ბაზის მისაერთებლად აუცილებელი მონაცემები, როგორიცაა localhost-სერვერი, მომხმარებლის User-იდენტიფიკატორი და ბაზის სახელი market.

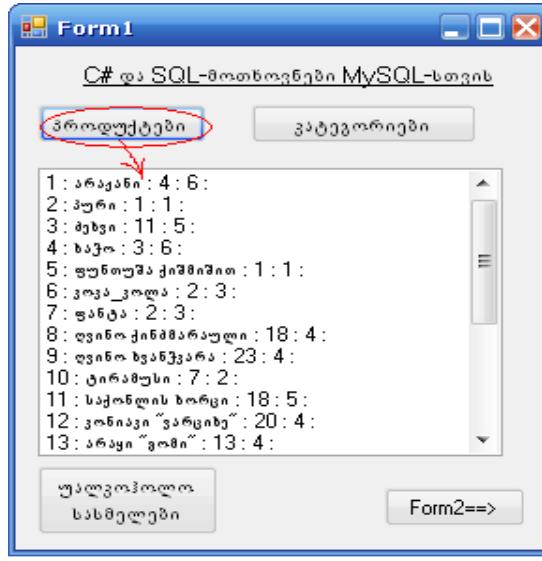
პროგრამის მთლიან კოდს შემდეგ განვიხილავთ. ახლა Form1-ფორმაზე მოთავსებული რამდენიმე ღილაკი განვიხილოთ.

ღილაკით "**პროდუქტები**" ListBox1-ში გამოიტანება ყველა პროდუქტის სია, იდენტიფიკატორით, დასახელებით, ფასით და კატეგორიის ნომრით. მისი SQL-მოთხოვნის "ამორჩევის" კოდს ექნება ასეთი სახე:

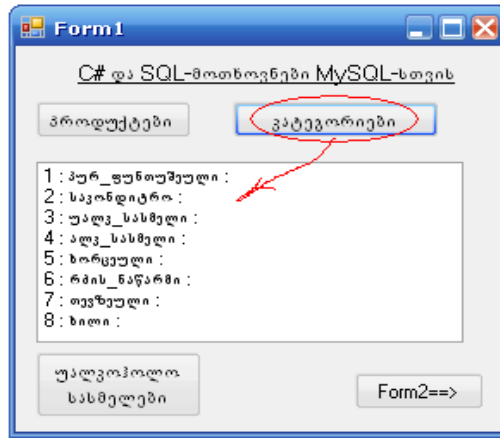
```
private void button1_Click(object sender, EventArgs e)
  {
     Amorcheva("select * from product order by pr_ID, Name",
                          "pr_ID", "Name", "prize", "cat_ID");
  }
```

შედეგები ასახულია 1.13 ნახაზზე მოცემულ ფანჯარაში.

ღილაკით "**კატეგორიები**" ListBox1-ში გამოიტანება ბაზაში არსებული პროდუქციის ყველა კატეგორიის დასახელება. როგორც 1.14 ნახაზიდან ჩანს, სახეზეა 8 კატეგორია.

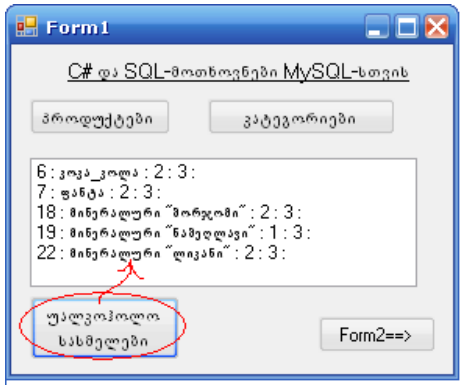


ნახ.1.13



ნახ.1.14

ღილაკით "**უალკოჰოლო სასმელები**" ListBox1-ში გამოიტანება იმ პროდუქტების სია, რომელთა ველში "კატეგორიის იდენტიფიკატორი" შეესაბამება უალკოფოლო სასმელებს, ანუ ჩვენ შემთხვევაში `cat_ID=3`. შედეგი ასახულია 1.15 ნახაზზე.



ნახ.1.15

C# პროგრამის კოდი მოცემულია 1.1 ლისტინგში.

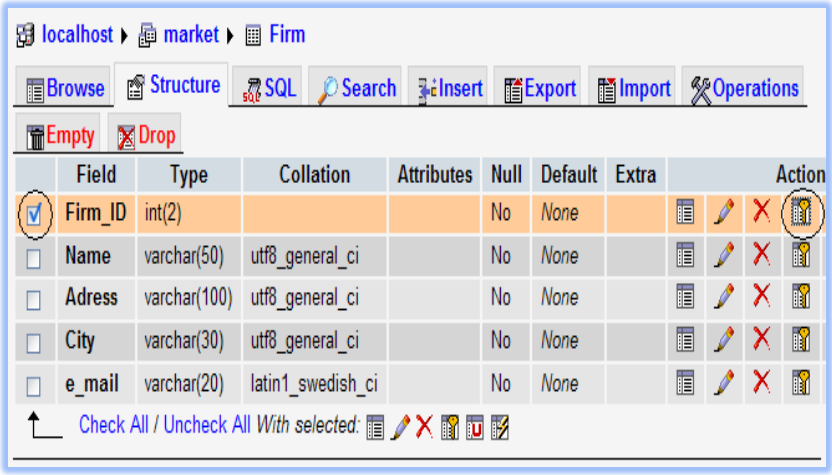
```
// ლისტინგი_1.1 ----------
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace WApp_mysqlprobe
{
 public partial class Form1 : Form
 {
   public Form1() { InitializeComponent(); }
   private void Amorcheva(string sqlbefehl,
                                  params string[] velebi)
    {
      int i;
      string striqoni;
MySql.Data.MySqlClient.MySqlConnection msqlConnection = null;
```

```
    msqlConnection = new
MySql.Data.MySqlClient.MySqlConnection("server=localhost;"+
       "User Id=user@localhost;database=market;Persist
                                          Security Info=True");
MySql.Data.MySqlClient.MySqlCommand msqlCommand = new
                        MySql.Data.MySqlClient.MySqlCommand();
   msqlCommand.Connection = msqlConnection;
    msqlCommand.CommandText = sqlbefehl;
 try
   {
      msqlConnection.Open();
     MySql.Data.MySqlClient.MySqlDataReader msqlReader =
                                  msqlCommand.ExecuteReader();
     listBox1.Items.Clear();
     while (msqlReader.Read())
      {
       striqoni = "";
       for (i = 0; i < velebi.Length; i++)
          striqoni += msqlReader[velebi[i]] + " : ";

       listBox1.Items.Add(striqoni);
      }
    }
  catch (Exception ex)
    {
       MessageBox.Show(ex.Message);
    }
  finally
      {
        msqlConnection.Close();
      }
  }
 private void button1_Click(object sender, EventArgs e)
  {
     Amorcheva("select * from product order by pr_ID, Name",
                          "pr_ID", "Name", "prize", "cat_ID");
  }
 private void button2_Click(object sender, EventArgs e)
  {
     Amorcheva("select * from category order by cat_ID",
                                              "cat_ID", "Name");
  }
private void button3_Click(object sender, EventArgs e)
  {
```

```
    Amorcheva("select * from product where cat_ID=3 order by
            pr_ID", "pr_ID", "Name", "prize", "cat_ID");
  }
 }
}
```

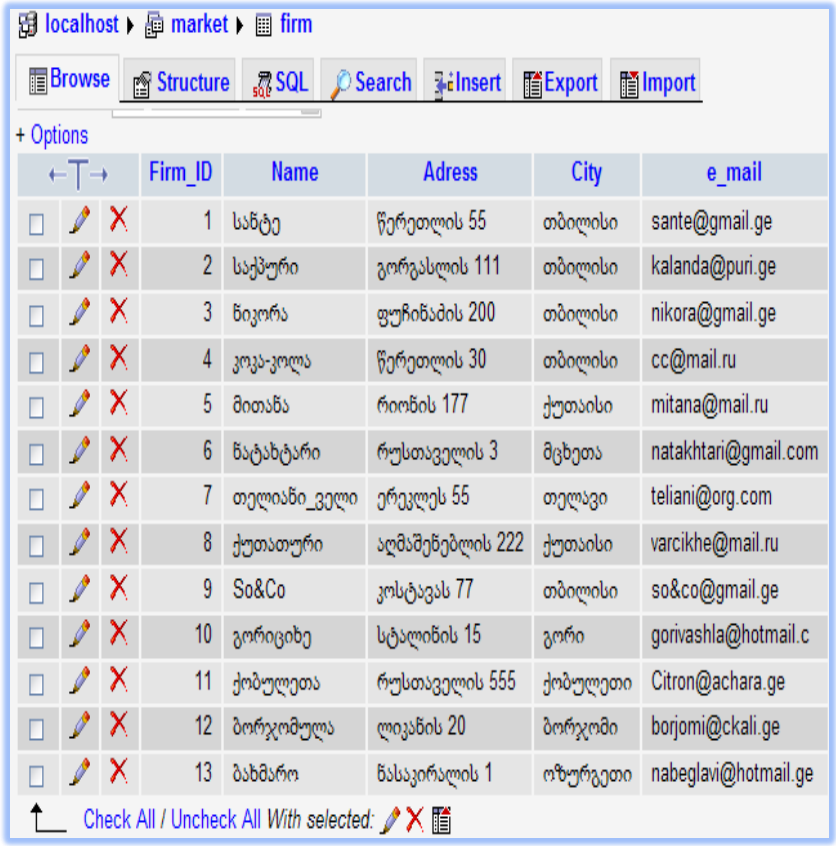
ამოცანა: განხილულ ბაზას დავამატოთ ახალი ცხრილი პროდუქციის მწარმოებელი ფირმებისათვის, სახელით Firm, რომელსაც ექნება ხუთი ველი: იდენტიფიკატორი, დასახელება, მისამართი, ქალაქი და ელ-ფოსტა (ნახ.1.16).



ნახ.1.16

Browse გადამრთველით გამოვიტანოთ ფირმების ცხრილი და შევავსოთ იგი (ნახ.1.17).

product ცხრილში საჭიროა დავამატოთ ველი Firm_ID, რომლითაც ის დაუკავშირდება ამ პროდუქციის მწარმოებელი ფირმის სტრიქონს. შევიტანოთ product ცხრილის Firm_ID ველის სვეტში შესაბამისი ფირმების იდენტიფიკატორები (ნახ. 1.18).



Firm_ID	Name	Adress	City	e_mail
1	სანტე	წერეთლის 55	თბილისი	sante@gmail.ge
2	სანპური	გორგასლის 111	თბილისი	kalanda@puri.ge
3	ნიკორა	ფუჩიწაძის 200	თბილისი	nikora@gmail.ge
4	კოკა-კოლა	წერეთლის 30	თბილისი	cc@mail.ru
5	მითანა	რიონის 177	ქუთაისი	mitana@mail.ru
6	ნატახტარი	რუსთაველის 3	მცხეთა	natakhtari@gmail.com
7	თელიანი_ველი	ერეკლეს 55	თელავი	teliani@org.com
8	ქუთათური	აღმაშენებლის 222	ქუთაისი	varcikhe@mail.ru
9	So&Co	კოსტავას 77	თბილისი	so&co@gmail.ge
10	გორიციხე	სტალინის 15	გორი	gorivashla@hotmail.c
11	ქობულეთა	რუსთაველის 555	ქობულეთი	Citron@achara.ge
12	ბორჯომული	ლიკანის 20	ბორჯომი	borjomi@ckali.ge
13	ბახმარო	წასაკირალის 1	ოზურგეთი	nabeglavi@hotmail.ge

ნახ.1.17

			pr_ID	Name	prize	cat_ID	Firm_ID
☐			1	არაჟანი	4	6	1
☐			2	პური	1	1	2
☐			3	ძეხვი	11	5	3
☐			4	ხაჭო	3	6	1
☐			5	ფუნთუშა ქიშმიშით	1	1	3
☐			6	კოკა_კოლა	2	3	4
☐			7	ფანტა	2	3	6
☐			8	ღვინო ქინძმარაული	18	4	7
☐			9	ღვინო ხვანჭკარა	23	4	7
☐			10	ტირამუსი	7	2	5
☐			11	საქონლის ხორცი	18	5	5
☑			12	კონიაკი "ვარციხე"	20	4	8
☐			13	არაყი "გომი"	13	4	8
☐			14	ასატრინა	25	7	9
☐			15	ვაშლი "გოლდენი"	3	8	10
☐			16	კონსერვი "შპროტი"	5	7	9
☐			17	ფორთოხალი	3	8	11
☐			18	მინერალური "ბორჯომი"	2	3	12
☐			19	მინერალური "ნაბეღლავი"	1	3	13
☐			20	ორცხობილა "ნუშის"	4	2	5
☐			21	ნაყინი "ვანილის"	6	2	1
☐			22	მინერალური "ლიკანი"	2	3	12
☐			23	ღორის სამწვადე	23	5	5

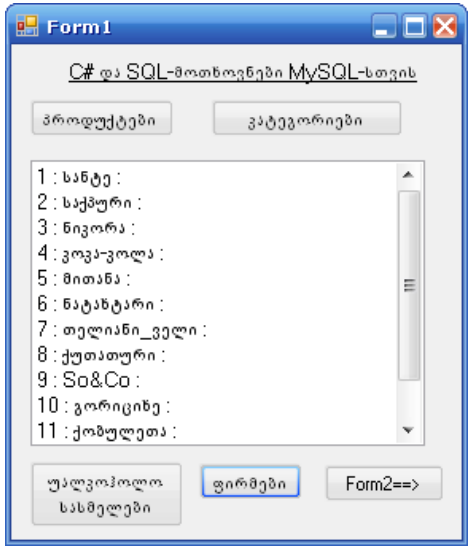
Check All / Uncheck All *With selected:*

ნახ.1.18

ღილაკით "**ფირმები**" ListBox1-ში გამოვიტანოთ პროდუქციის მწარმოებელი ფირმების სია. C#კოდის ფრაგმენტს ექნება შემდეგი სახე:

```
private void button5_Click(object sender, EventArgs e)
   {
     Amorcheva("select * from Firm order by Firm_ID",
                                        "Firm_ID", "Name");
   }
```

შედეგები მოცემულია 1.19 ნახაზზე.



ნახ.1.19

1.2. ობიექტური მონაცემთა ბაზა

1.2.1. ობიექტ-ორიენტირებული მონაცემთა ბაზის სისტემები

ობიექტ-ორიენტირებული მონაცემთა ბაზების (ოომბ) კონცეფცია წარმოიშვა რელაციური მონაცემთა ბაზების (რმბ) ტექნოლოგიის შემდგომ სრულყოფასთან დაკავშირებით. ასეთი იდეის უპირატესობა კი ის იყო, რომ ოომბ-ის პროექტი უფრო ახლოსაა საპრობლემო სფეროსთან. აქ გამოიყენება ოო-მიდგომის ელემენტები, როგორიცაა კლასები და ობიექტები, მეთოდები, კლასებს შორის იერარქია, მემკვიდრეობითობა და ა.შ. [5,8,9].

ამგვარად, ობიექტური ბაზები (object databases) არის მონაცემთა ბაზებისა და ობიექტ-ორიენტირებული დაპროგრამების ენების გაერთიანებით მიღებული ერთობლივი პროდუქტი.

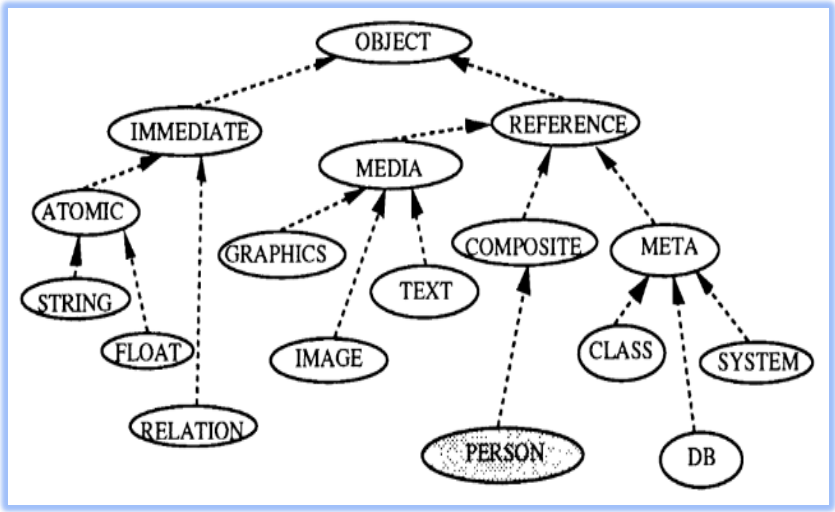
რეალური სამყაროს ნებისმიერი არსი (Entity) ოო-ენებსა და სისტემებში მოდელირდება ობიექტის სახით. ყოველი ობიექტი მისი შექმნის დროს ღებულობს სისტემის მიერ გენერირებულ უნიკალურ იდენტიფიკატორს, რომელიც კავშირშია ამ ობიექტთან მისი მთელი არსებობის პერიოდში და არ იცვლება ობიექტის მდგომარეობის შეცვლისას. ყოველ ობიექტს აქვს მდგომარეობა (ატრიბუტების მნიშვნელობათა ერთობლიობა) და ახასიათებს ქცევა - მეთოდების ერთობლიობა (პროგრამული კოდი), რომლითაც იგი ოპერირებს ობიექტის მდგომარეობაზე.

ობიექტის მდგომარეობა და ქცევა ინკაფსულირებულია ობიექტში. ობიექტებს შორის ურთიერთმოქმედება ხორციელდება შეტყობინებათა გადაცემით და შესაბამისი მეთოდების ამოქმედებით და შესრულებით. ობიექტთა სიმრავლე ერთიდაიმავე ატრიბუტებით და მეთოდებით ქმნის ამ ობიექტთა კლასს. ობიექტი უნდა მიეკუთვნებოდეს მხოლოდ ერთ კლასს. დასაშვებია ახალი კლასის შექმნა უკვე არსებულის საფუძველზე და მემკვიდრეობითობის პრინციპის რეალიზაცია.

მოთხოვნების დასამუშავებლად გამოიყენება OQL (Object Query Language), ხოლო მონაცემთა მანიპულირებისთვის ODL (Object Definition Language).

როგორც აღვნიშნეთ, ოო-მონაცემთა ბაზები 80-იანი წლებიდან ვითარდებოდა. მათ შორის იყო გამორჩეული პაკეტები, რომლებსაც უკეთესი მახასიათებლები და კომერციული თვალსაზრისით მეტი მომხმარებელი ჰყავდა. აქ შეიძლება ვახსენოთ მაგალითად, ფრანგული Altair კონსორციუმის O2 სისტემა, ამერიკული MCC კომპანიის ORION სისტემა, Htwlett-Packard-ის Iris სისტემა და ა.შ. [5, 10-13].

1.20 ნახაზზე მოცემულია ობიექტ-ორიენტირებული ბაზის კლასისთვის ობიექტის ზოგადი მოდელი (სტრუქტურა), რომელსაც იყენებს სისტემა მონაცემთა განსაზღვრის ენაში [13].



ნახ.1.20.

სისტემის მიერ განსაზღვრული კლასები ასახავს ობიექტური მოდელის სემანტიკას. OBJECT არის განსახილველი ობიექტის სტრუქტურის (ჩარჩოს) ფესვი. OBJECT იყოფა REFERENCE და

IMMEDIATE აღნიშვნებით მითითებით და მნიშვნელობით ობიექტებად. IMMEDIATE ობიექტები იყოფა ATOMIC და RELATION ობიექტებად. ATOMIC ობიექტები შედგება INTEGER, FLOAT, STRING და VSTRING ობიექტებისგან. RELATION ობიექტებს შეიძლება ჰქონდეს REFERENCE და IMMEDIATE ობიექტები ატრიბუტების მნიშვნელობების სახით. RELATION ობიექტები რეალიზდება ჩადგმული რელაციების სახით.

კომპლექსური ობიექტები, მაგალითად, ORION სისტემაში შესაძლებელია აისახოს RELATION ობიექტების გამოყენებით [13]. ასეთ შეზღუდვას უწოდებენ *კომპლექსური ობიექტის მთლიანობას*, რომელიც სრულდება „მთელი-ნაწილი“ დამოკიდებულებაში.

სისტემის მიერ განსაზღვრულ კლასებს აქვს სიტემის მიერ განსაზღვრული ატრიბუტები, რომლების მემკვიდრულ კავშირშია მომხმარებლის მიერ განსაზღვრულ ქვეკლასთან. IMMEDIATE ობიექტებს აქვს პროცედურული ატრიბუტები. ისინი შეიცავს სკანირების ფუნქციებს, როგორიცაა openscan, next, და closescan. გარდა ამისა, ATOMIC ობიეტებს აქვს აგრეგატული ფუნქციები, როგორიცაა count, average, sum, max და min. RELATION ობიექტებს აქვს ანგარიშის შესაძლებლობა როგორც აგრეგატული ფუნქციებს.

REFERENCE-ს აქვს თვლადი ატრიბუტები, როგორიცაა Class და Oid. მას აქვს აგრეთვე პროცედურული ატრიბუტები, როგორიცაა put, replace, delete და destroy დამატებით სკანირების ფუნქციებთან. REFERENCE-ს ობიექტები იყოფა COMPOSITE, MEDIA და META ნაწილებად. მომხმარებელი, როგორც წესი, განსაზღვრავს ობიექ-ტების დომენს (საპრობლემო სფეროს) როგორც COMPOSITE ქვეკლასს.

MEDIA-ს აქვს ქვეკლასები IMAGE, GRAPHICS და TEXT. მათ აქვს პოლიმორფული პროცედურული ატრიბუტები, როგორიცაა input, output, change-size და move, რომელთაც აქვს განსხვავებული რეალიზაციები, მაგრამ ერთი იგივე ინტერფეისი. მომხმარებელს შეუძლია მუშაობა მულტიმედიის ერთიან ინტერფეისზე.

META-ს აქვს ქვეკლასი CLASS., რომელიც ასახავს კლასის სტრუქტურის სემანტიკას. მას აქვს ატრიბუტები, პროცედურები, ლექსიკონური ინფორმაცია, DB-ს, როგორც META-ს ქვესიმრავლეს, აქვს ატრიბუტები მთლიანი ცოდნის ბაზის სამართავად. SYSTEM-ს აქვს ატრიბუტები სესიებისა და ტრანზაქციების სამართავად.

1.2.2. ობიექტ-ორიენტირებული მულტიმედიური მონაცემთა ბაზა

პირველი ნაშრომები მულტიმედიური მონაცემთა ბაზების შესახებ გამოჩნდა 80-ან წლებში, როდესაც მონაცემთა რელაციური მოდელების თემატიკა, როგორც ახალი მეცნიერული მიმართულება, ძალზე აქტუალური და დომინირებადი იყო [3, 14]. გასაოცარი არაა, რომ ამ დროს მულტიმედიური მონაცემთა ბაზების სისტემების პირველი პროექტები მოიაზრებოდა როგორც აუცილებლად ობიექტ-ორიენტირებული სისტემები [15]. მაგრამ, ამ პერიოდში ვერ მოხერხდა ასეთი სისტემების პრაქტიკული რეალიზაცია.

ამ პრობლემებზე მუშაობა და მნიშვნელოვანი მიღწევები მიღებულ იქნა 90-ანი წლებიდან, როდესაც ODMG (Object Data Management Group) ჯგუფმა წარმოადგინა რეალაციური ბაზებისთვის SQL-ის მსგავსი ეფექტური კონცეფცია [16].

ობიექტორიენტირებულ სისტემებში მულტიმედიური მონაცემთა ტიპების ასახვა ხდება უშუალოდ როგორც კლასები. ეგზემპლარები უნდა იყოს ინკაფსულირებული, ისე, რომ წვდომის განხორციელება შეიძლებოდეს მხოლოდ კლასის მეთოდებზე.

ეს კლასები შეიძლება იყოს ორგანიზებული განზოგადებული (გენერალიზებული) იერარქიით. ისინი გამორიცხავს საერთო მეთოდების განმეორებად განსაზღვრებებს და ნებას იძლევა მონაცემთა სპეციალური ტიპების შესატანად, რომლებისთვისაც დამატებითი მეთოდებია განსაზღვრული.

მონაცემთა ტიპები Text, Image, ასევე Graphics, Audio და Video განიხილება როგორც MediaObject-ის ქვეკლასები, და მემკვიდრეობით ღებულობს მის მეთოდებს. ისინი აფართოებენ ამ მეთოდებს საკუთარი ოპერაციებით, რომლებიც მორგებულია სპეციალურ მულტიმედიურ მონაცემთა ტიპებზე, მაგალითად, image-ს დროს height, ან Text-ის დროს length.

ORION სისტემა ან MIM (Multimedia Information Manager)-ით აღნიშნული კლასების კოლექცია წარმოადგენს კიდევ ერთ შრეს MediaObject-სა და Text ან Image ტიპებს შორის, რომლების შეიცავს კლასებს „წრფივი მედიაობიექტები“ და „სივრცითი მედიაობიექტები“. პირველი მოიცავს ტექსტებს და აუდიოს, ხოლო მეორე სურათებს, გრაფიკას და ვიდეოს.

მართლაც, ამ მეთოდებს, როგორიცაა length და height, შეუძლია ფაქტობრივად განსაზღვრულ იქნას ამ კლასებზე და შემდეგ მემკვიდრეობით იქნას გამოყენებული. სხვა მხრივ უფრო მნიშვნელოვნად ჩანს კლასიფიკაცია როგორც „დროზე დამოკიდებული“ და „დროზე დამოუკიდებელი“. ამიტომ გადაწყვეტა ასეთი შუალედური შრის შესახებ ჯერჯერობით არ განიხილება.

კლასების იერარქიის გენერალიზაცია და აგება საშუალებას იძლევა მოვახდინოთ არა მხოლოდ მულტიმედიური ობიექტების სასარგებლო ჩანერგვა, არამედ აგრეთვე ობიექტებისაც (Entities), რომლებსაც ისინი ასახავს ან აღწერს.

წინა პარაგრაფში ობიექტრელაციური მოდელების დისკუსიამ გვიჩვენა, რომ გარკვეულ შემთხვევებში სხვადასხვა is.presented_in დამოკიდებულება ამოდელირებს და ეს ძებნის დროს ყველა ცხადად უნდა იქნას გათვალისწინებული.

გენერალიზაცია, პირიქით ნებას იძლევა, რომ ამომრჩეველი, დეპუტატობის_კანდიდატი და საარჩევნო_უბნის_თანამშრომელი ზემნიშვნელობით (Superclass) გავაერთიანოთ, როგორიცაა „პიროვნება“ და მათი ეგზემპლარები სურათებით დავაკავშიროთ.

რომელი ზე-მნიშვნელობა მიესადაგება დასმულ მიზანს, დამოკიდებულია იმაზე, თუ რომელი ობიექტებია სურათებზე, ტექსტებში, გრაფიკებზე და ა.შ. უფრო რელევანტური (შესაბამისი) გამოსაყენებლად. ყველაზე ზოგად შემთხვევაში, შეძლებისდაგვარად, თუ მოცემულია პრესიდან სურათი (ფოტო), საჭიროა ისევ „Object“ კლასზე მიმართვა.

ობიექტორიენტირებულ მოდელს შეუძლია მე-3 ტიპის სქემის (იხ. წინა პარაგრაფი) უკეთესად წარმოდგენა, ვიდრე რელაციურ მოდელს. როგორი მდგომარეობაა სქემის სხვა ტიპებისთვის ? კლასის ეგზემპლარები გამოისახება კორტეჟების სახით ატრიბუტების მნიშვნელობებზე, რომელთა მნიშვნელობათა არეები დადგენილია კლასების განსაზღვრებაში.

კლასიკური რელაციური მოდელისგან განსხვავებით მნიშვნელობათა ეს არეები არ უნდა იყოს არჩეული ერთი წინასწარგანსაზღვრული სიმრავლიდან, არამედ შესაძლებელია ასევე იყოს ნებისმიერი თვითგანსაზღვრებადი კლასი.

ასე მაგალითად, employee კლასს შეუძლია თავისი ეგზემპლარებისთვის განსაზღვროს ატრიბუტი (ეგზემპლარის ცვლადი) Portrait, რომელთა მნიშვნელობები შეიძლება იყოს GreyscaleImage კლასის ეგზემპლარები. ეს შეესაბამება სქემის 1-ელ ტიპს რელაციური ბაზისთვის.

ობიექტორიენტირებული სისტემები არ მოითხოვს ნორმალიზაციას თავისი კლასებისა და ეგზემპლარებისთვის, ასე რომ ნებადართულია ატრიბუტები რამდენიმე მნიშვნელობით. აუცილებლობის შემთხვევაში საჭიროა ტიპების კონსტრუქტორის, როგორიცაა list ან set გამოყენება. შესაბამისად იგი მე-2 ტიპის სქემასთან შედარებით ოდნავ ჭარბია. Portrait ატრიბუტს შეუძლია აგრეთვე მარტივად ჰქონდეს რამდენიმე სურათი.

მრავალი ობიექტორიენტირებული სისტემა არ განსხვავდება სუფთად ობიექტის ატრიბუტებით და კომპონენტებით. ხშირად აგრეგაცია გამოისახება მარტივად ატრიბუტთა დახმარებით.

ODMG-მოდელი მკაფიოდ გამოყოფს ატრიბუტებს დამოკიდებულებებისგან და თუ ატრიბუტებს შეუძლია მხოლოდ მნიშვნელობების მიღება, ობიექტები უკავშირდება ერთმანეთს დამოკიდებულებებით.

ობიექტორიენტირებული სისტემების სუსტი ადგილია სიმრავლეებზე ორიენტაციის არარსებობა და მასთან დაკავშირებული ძებნა. ეს გასაგები იყო, ამიტომაც ცდილობდნენ ობიექტორიენტირებული ბაზების მართვის სისტემებისთვის მოთხოვნების ენის განსაზღვრას [17].

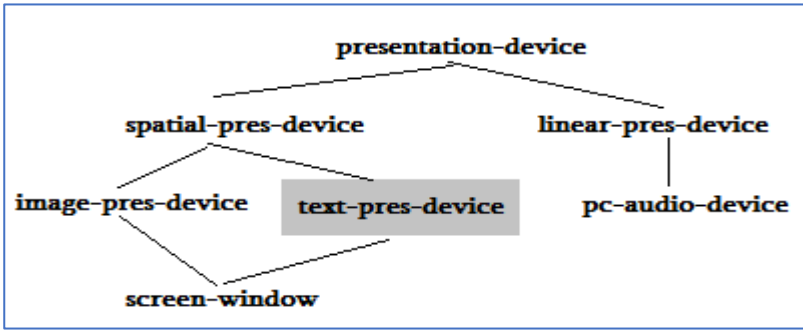
OMDG-მოდელში არსებობს ამისთვის OQL-ენა (Object Query Language), რომელიც ორიენტირებულია SQL-ზე და ზოგიერთ შემთხვევაში სრულად თავსებადიცაა. მას მოაქვს თან ყველა სასურველი თვისება, მაგრამ არ შეესაბამება ობიექტებზე პირველად წვდომას, რისთვისაც იყო შექმნილი ოო-ბაზების სისტემები: იგულისხმება ირიბი მიმართვა. ვინაიდან OQL ენა ძალზე რთულია, მისი გამოყენება ნაკლებად გვხვდება პროგრამულ პროდუქტებში.

ახლა განვიხილოთ ობიექტორიენტირებული მულტიმედიური მონაცემთა ბაზის სისტემების კონკრეტული მაგალითი ORION სისტემის გამოყენებით, რომელიც ასევე ცნობილია მულტიმედიური ინფორმაციული მენეჯერის (MIM – Multimedia Information Manager) სახელით [17, 18]. ესაა ობიექტ-ორიენტირებული კლასების ბიბლიოთეკა.

ასეთი ობიექტორიენტირებული მონაცემთა ბაზების სისტემა მომხმარებლისთვის აადვილებს მულტიმედიური ობიექტები დაიყვანონ სუბკლასებამდე და შემდეგ ისარგებლონ MIM ბიბლიოთეკის ოპერაციებით მემკვიდრეობითობის მექანიზმის გამოყენებით [17].

MIM-ის განსაკუთრებული თავისებურება მდგომარეობს იმაში, რომ ყველა მოწყობილობა წარმოიდგინება როგორც ობიექტები ORION-ში, კერძოდ, როგორც შეტანა/გამოტანის, ასევე

მეხსიერების მოწყობილობები. გამოტანის მოწყობილობისთვის მომზადებულია კლასების იერარქია, რომელიც 1.21 ნახაზზეა მოცემული.



ნახ.1.21. MIM-ში კლასთა იერარქია გამოსატანი მოწყობილობებისთვის

გრაფის წიბოები ასახავს მემკვიდრეობითობის კავშირებს, სადაც სუბკლასები მოთავსებულია კლასების ქვეშ. მონიშნული კლასები text-pres-device არის მაგალითი გაფართოებისა, რომელიც თვით მომხმარებელმა გააკეთა. ასეთი კლასი არ ეკუთვნის MIM-ბიბლიოთეკას.

კლასთა ეგზემპლარები ამ იერარქიაში არაა განაწილებული ცალსახად გამომტანი მოწყობილობებისთვის, არამედ ისინი სპეციფიცირებულია:

- სადაა მოწყობილობაზე ასახული (მაგალითად, ეკრანის რომელ ნაწილში);
- მედიალური ობიექტის რომელი ნაწილი (ამონაჭერი) არის ასახული.

შედეგად შესაძლებელია რამდენიმე ეგზემპლარის მიცემა, რომლებიც ერთიდაიმავე ფიზიკურ მოწყობილობას ასახავს. შეიძლება ასევე მათი ინტერპრეტაცია შეზღუდვებით როგორც „გამოცემის ფორმატი“ მედიური ობიექტებისთვის. მაგალითად,

კლასი spatial-pres-device ასე განმარტავს მისი ეგზემპლარის შემდეგ ატრიბუტებს:

upper-left-x,
upper-left-y,
width,
height.

ამ ატრიბუტების მნიშვნელობები შეესაბამება მედიური ობიექტის ამონაჭერს (მაგალითად, რასტრულ სურათს), რომელიც უნდა გამოიცეს სივრცით გამომტან მოწყობილობაზე.

ქვეკლასში screen-window სპეციფიცირდება ატრიბუტები:

win-upper-left,
win-upper-right,
win-width,
win-height

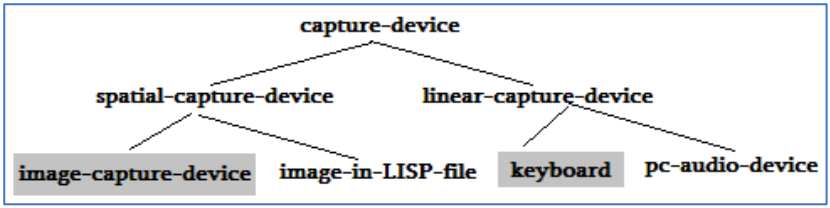
დამატებით ეკრანის ნაწილზე, რომელიც უნდა იქნას გამოყენებული ასახვისთვის.

screen-window -სთვის განსაზღვრულია ასევე მეთოდები, კერძოდ:

present,
capture,
present-pres,

რომლებიც გამოიყენება განსაზღვრული მულტიმედიური ობიექტების გამოსატანად და, აუცილებლობის შემთხვევაში, გამოიძახება წაკითხვის აღსადგენადაც.

ანალოგიურად არის MIM-ში განსაზღვრული კლასთა იერარქია შესატანი მოწყობილობებისთვის (ნახ.1.22). აქ მონიშნული კლასები, image-capture-device და keyboard აღწერეს მომხმარებელთა გაფართოებებს.



ნახ.1.22. MIM-ში კლასთა იერარქია შესატანი მოწყობილობებისთვის

აქაც, როგორც წინა შემთხვევაში, კლასის ეგზემპლარები მეტია, ვიდრე მხოლოდ სპეციფირებული მოწყობილობები. ისინი მიუთითებს ასევე თუ მულტიმედიური ობიექტის რომელი ნაწილი განიხილება და როგორაა მოწყობილობა დაკომპლექტებული. შედეგად შესაძლებელია ერთი ფიზიკური მოწყობილობისთვის კვლავ capture-device ტიპის რამდენიმე ეგზემპლარის მიცემა.

spatial-capture-device სუბკლასის ეგზემპლარები მიუთითებს შემდეგ ატრიბუტებს:

upper-left-x,
upper-left-y,
width,
height.

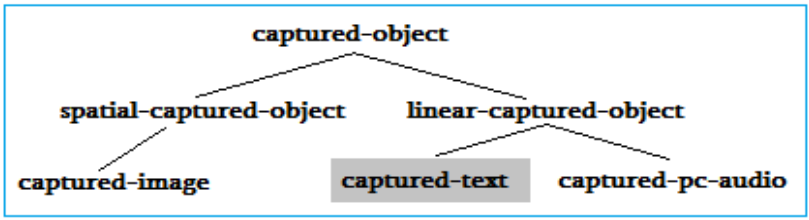
იგი სპეციფიცირებას უკეთებს სივრცითი მულტიმედიური ობიექტის ამონაჭერს, რომელიც მიიღება შემტანი მოწყობილობით ან ჩანაცვლებით. მომხმარებლის მიერ განსაზღვრულ image-capture-device კლასს შეუძლია

cam-width,
cam-height,
bits-per-pixel

ატრიბუტების და capture მეთოდის წარმოდგენა.

დამახსოვრებული მულტიმედიური ობიექტები ორგანიზებულია ასევე კლასთა იერარქიის სახით. ამ დროს კვლავ

განასხვავებენ სივრცით და წრფივ მულტიმედიურ ობიექტებს. რასტრული სურათები და გრაფიკები მიეკუთვნება სივრცითს, ხოლო ტექსტი და აუდიო კი წრფივს. ქვეკლასები ასახულია 1.23 ნახაზზე.



ნახ.1.23.. MIM-ში დამახსოვრებული მედიური ობიექტების კლასთა იერარქია

captured-object კლასებში ყველა ქვეკლასისთვის განსაზღვრულია შემდეგი ატრიბუტები:

storage-object – მიუთითებს storage-device კლასის ერთ ეგზემპლარზე, რომელიც ქვემოთ შემოიტანება.

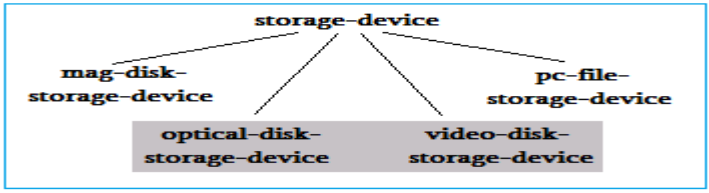
logical-measure - განსაზღვრავს საწყისი მონაცემების ელემენტარულ ერთეულებს მომხმარებლის თვალსაზრისით, რომლებიც არ უნდა ემთხვეოდეს სისტემოტექნიკურ ერთეულებს (ბიტი, ბაიტი ან მეხსიერების სიტყვა (memory word)). ასეთი ლოგიკური ზომის ერთეულებია, მაგალითად, წამები აუდიოს დროს, ან კადრები - ვიდეოს დროს.

დამოკიდებულება ფიზიკურ და ლოგიკურ ზომის ერთეულებს შორის აისახება phys-logic-ratio ატრიბუტში. ეს იქნება მაშინ ბაიტი/წამში აუდიოსთვის ან ბაიტი/კადრი ვიდეისთვის.

spatial-captured-object სუბკლასი იძლევა ატრიბუტებს width, height და row-major. უკანასკნელი გვიჩვენებს დამახსოვრება მოხდა სტრიქონულად თუ სვეტურად.

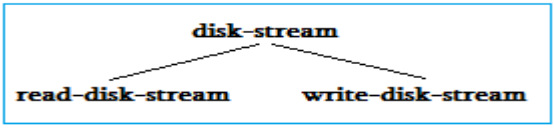
და ბოლოს, არსებობს ატრიბუტი bits-per-pixel, რომელიც ცხადჰყოფს, რომ საქმე ეხება ტიპურ სარეგისტრაციო მონაცემებს.

დამახსოვრებისთვის არსებობს მოწყობილობები, რომლებიც შესატან და გამოსატან მოწყობილობათა მსგავსად ნაწილობრივ გამოყენებაშია და storage-device კლასის ეგზემპლარების საშუალებით წარმოიდგინება (ნახ.1.24).



ნახ.1.24. MIM-ში შენახვის მოწყობილობების კლასთა იერარქია

და ბოლოს, კლასებში არსებობს კიდევ ორგანიზებული მონაცემთა ობიექტები, რომლებიც ასახავს ცალკეულ წაკითხვის ან ჩაწერის პროცესებს (ნახ.1.25).



ნახ.1.25. MIM-ში წაკითხვის და ჩაწერის პროცესების კლასთა იერარქია

მათი წარმოება ხდება დინამიკურად და შეტანის ან გამოტანის დამთავრების შემდეგ ისევ იშლება. ამავდროულად, disk-stream შეიცავს ორივე შემთხვევისთვის საჭირო storage-object ატრიბუტს, რომელიც მიმართავს storage-device კლასის წასაკითხ ან ხელახლაჩასაწერ ეგზემპლარს. read-disk-stream -ში არსებობს დამატებთ read-block-list ატრიბუტი, რომელც აღწერს პოზიციო-ნირების მარკირებას და აიდენტიფიცირებს მულტიმედიური ობიექტის მომდევნო წასაკითხ ბლოკს. მსგავს როლს ასრულებს write-block-list ატრიბუტი write-disk-stream -ში.

ამჯერად ილუსტრირებული გვაქვს აღნიშნული კლასების ურთიერთმოქმედება და მეთოდების ჩადგმული გამოძახება რასტრული სურათის გამოცემის მაგალითზე.

დავუშვათ, რომ არსებობს car კლასი (ავტომობილი), რომელიც თავისი ეგზემპლარებისთვის განსაზღვრავს ატრიბუტებს image (სურათი) ტიპით captured-image და output_ device (გამოსატანი მოწყობილობა) ტიპით image-pres-device.

ავტომანქანის აღწერას ეკუთვნის აგრეთვე გამოსატანი მოწყობილობა, რომელზეც მანქანის სურათი კარგად უნდა აისახოს. ამის გარდა ეს კლასი შეიცავს ასევე მეთოდს show_image (სურათის ჩვენება), რომელიც ახორციელებს დამახსოვრებული სურათის გამოცემას წინასწარგანსაზღვრულ გამოსატან მოწყობილობაზე.

ამის მისაღწევად, საჭიროა show-image მეთოდმა გააგზავნოს present შეტყობინება ობიექტისკენ, რომელიც წარმოადგენს გამოსატან მოწყობილობას და მას ერთ პარამეტრში დაუსახელოს გამოსატანი სურათი.

თუ გავითვალისწინებთ, რომ ასეთი გადაცემის ან გამოძახების ბრძანებები LISP-ენის სახის მეთოდოლოგიის სინტაქსით აიგება, მიმღები ობიექტი და პარამეტრი სპეციფიცირდება, მაშინ show-image -ს შესაბამისი ბრძანება შეიძლება ასე ჩაიწეროს:

(present output-device Image)

output-device ატრიბუტით იდენტიფიცირებული image-pres-device კლასის ეგზემპლარი ასრულებს ამის შემდეგ მის present მეთოდს.

მისი ატრიბუტები upper-lef-x, upper-lef-y, width და height უთითებს სურათის რომელი ნაწილი (ამონაჭერი) იხილება. ეს მართკუთხა ამონაჭერი გაითვლება წრფივ კოორდინატებში. ეს შესაძლებელია მხოლოდ captured-image ეგზემპლარზე წვდომის საშუალებით, რომელიც იდენტიფიცირებულია image პარამეტრით,

რადგან აქ row-major ატრიბუტი მხოლოდ გვეუბნება, რომ რეგისტრაციის მონაცემები ხელმისაწვდომია.

ამჯერად შენახული სურათი წასაკითხად იხსნება:

(open-for-read Image [start-offset])

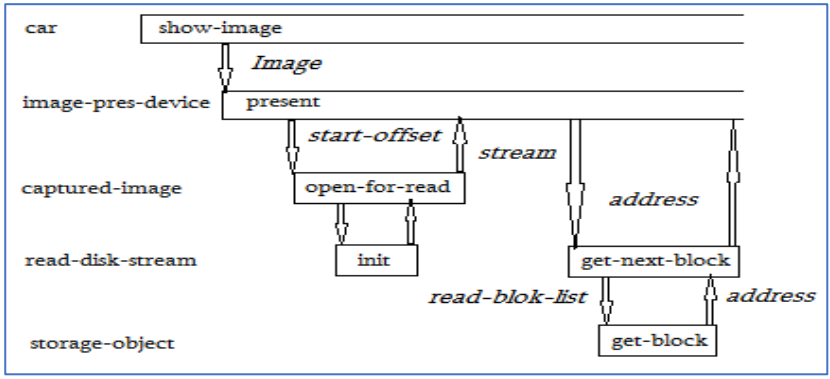
Image-ს საშუალებით დასახელებული capture-image ეგზემპლარი ასრულებს open-for-read მეთოდს. ამ დროს იგი აწარმოებს ახალ read-disk-stream ეგზემპლარს, რომლის სახელს (ობიექტის იდენტიფიკატორს) იგი image-pres-device -ს უკან უგზავნის. თუ შესაძლებელია, სპეციფიკაცია start-offset ემსახურება read-block-list -ის ინიციალიზაციას. გამოსატანი მოწყობილობა აგზავნის ამჯერად (present-ის შესრულების გაგრძელება) ამ ნაკადისთვის (straem) შეტყობინებას წაკითხვის შესახებ:

(get-next-block read-disk-stream)

შემდეგ ნაკადი (straem) თვითონ აგზავნის ისევ:

(get-block storage-object read-block-list)

ამ გამოძახებით მიიღება ბუფერის მისამართი, სადაც ინახება სასურველი ბლოკი. შემდეგ ნაკადი ზრდის მაჩვენებელს და აწვდის image-pres-device–ს უკან ბუფერის მისამართს, როგორც get-next-device მეთოდის შედეგს. 1.26 ნახაზი ასახავს ზემოაღწერილ პროცესს გრაფიკულად.



ნახ.1.26. MIM-მეთოდების ჩადგმული გამოძახება სურათის გამოცემის დროს

მარცხენა მხარეს შემოტანილია კლასის სახელები, რომელთაგან თითოეული ზუსტად ერთ ეგზემპლარში მონაწილეობს. მართკუთხედები აღნიშნავს მეთოდის შესრულებას, რომლებიც დახრილი (italic) სიმბოლოებით აღწერილი პარამეტრებით გამოიძახება და ასევე კურსივით აღწერილ მნიშვნელობებს აბრუნებს. შემდეგი მსვლელობისას გამომტანი მოწყობილობა გადასცემს წაკითხული ბლოკის შინაარსს ტექნიკურ უზრუნველყოფას და აცნობებს ნაკადს, რომ ბუფერის სახელები კვლავ ხელმისაწვდომია:

(free-block read-disk-stream)

სურათის ყველა ბლოკის წაკითხვის და გათავისუფლების შემდეგ (close-read read-disk-stream)-ით დაიხურება წაკითხვის პროცესი. ნაკადი შეიძლება კვლავ წაიშალოს.

captured-object კლასი უზრუნველყოფს თავისი ეგზემპლარებისთვის კიდევ სხვა მეთოდებს. make-captured-object-version -ის საშუალებით იწარმოება შენახული მულტიმედია ობიექტის ახალი ვერსია და ირიბად ასევე დაკავებული მეხსიერების მოწყობილობის ახალი ვერსია (storage-device). ძველი და ახალი ვერსიები თავიდან იკავებს დისკზე ერთიდაიმავე ბლოკებს.

დასასრულ, არსებებოს კიდევ delete-captured-object და delete-part-of-captured-object მეთოდები. უკანასკნელი ელოდება პარამეტრებს start-offset და delete-count, რომლებიც მიუთითებს თუ რომელი ბაიტ-პოზიციიდან და რამდენი ბაიტი უნდა იქნას წაშლილი. ეს ოპერაცია მოითხოვს სიფრთხილეს, ვინაიდან იგი იმავდროულად არ სრულდება სარეგისტრაციო მონაცემების ცვლილებით. ასეთი ოპერაცია არაა გათვლილი სისტემის საბოლოო მომხმარებელზე.

ამგვარად, ORION სისტემა MIM-თან ერთად ასახავს ყველაზე სრულყოფილ წინადადებას დღემდე არსებულ მულტიმედიურ მონაცემთა ბაზების მართვის სისტემებს შორის. იგი მომხმარებელს სთავაზობს დიდ მოქნილობას, რათა არსებული კლასების იერარქია

საკუთარი სუბკლასების სპეციალიზირებით გაფართოვდეს, რაც დამატებითი დამუშავების და წვდომის ოპერაციების გამოყენების შესაძლებლობას იძლევა.

ამგვარად, ობიექტრელაციური და ობიექტორიენტირებული მონაცემთა ბაზების მართვის სისტემების შედარება გვიჩვენებს, რომ ზოგადად არსებობს ერთიანობა მათში ახალი ტიპების განსაზღვრისა და მართვისათვის, ან ობიექტორიენტირებული სისტემის კონტექსტში (მაგალითად, ORION), ან რეალიურ სისტემაში (მაფალითად, SQL / MM).

1.3. ობიექტურ-რელაციური მონაცემთა ბაზა

ობიექტურ-რელაციური ბაზები (object-relational database) ეყრდნობა მონაცემთა რელაციური ბაზების ტექნოლოგიას, რომელსაც ემატება ობიექტ-ორიენტირებული ბაზების შესაძლებლობები [5,8].

დღეს ბევრს საუბრობენ ობიექტრელაციურ სისტემებზე, ამასთანავე მათი გაფართოებები არის ძალზე სასარგებლო მულტიმედიური ობიექტებისთვის. ასეთ სისტემების საფუძველი ყოველთვის რელაციური მოდელებია. მათ უდავოდ აქვს უდიდესი პრაქტიკული მნიშვნელობა.

ჩვენ ქვემოთ დავახასიათებთ ჯერ მათ იმ განსაკუთრებულ თვისებებს, რომლებიც მულტიმედიისთვისაა მთავარი, შემდეგ ნაჩვენები იქნება თუ როგორ პროექტდება სქემები, რომლებიც მულტიმედიური ობიექტების ჩადგმულ მონაცემთა ტიპებს გამოიყენებს. და ბოლოს, ნაჩვენები იქნება, თუ როგორ შეიძლება SQL-მოთხოვნების ენით, რომელიც ასეთი სისტემების მნიშვნელოვანი ენაა, განხორციელდეს მონაცემებთან მიმართვა ობიექტრელაციურ მონაცემთა ბაზებში.

1.3.1. ობიექტრელაციური მონაცემთა ბაზების სისტემა

ობიექტრელაციური მონაცემთა ბაზების სისტემა (ორმბს) წარმოადგენს მცდელობას, რელაციური მბს ისე განვითარდეს და ახალი კონცეფციებით გამდიდრდეს, რომ მან შეძლოს, მინიმუმ, ფუნქციონალობათა ერთი ნაწილის შემოთავაზება, რომელიც აქამდე ცნობილი იყო ობიექტორიენტირებული მბს-თვის. ესენია კერძოდ, ობიქტთა იდენტურობა, მომხმარებელთა მიერ განსაზღვრებადი მონაცემთა ტიპები და ტიპთა იერარქიები. თუმცა ძირითადი სტრუქტურა ყველა მონაცემთა ორგანიზაციის და მონაცემთა შენახვისა რჩება ისევ ცხრილები. ასე რომ, ყველა ეს ახალი გაფართოება უნდა დაექვემდებაროს, როგორც ადრე, ამ (წინა) კონცეფციას.

მულტიმედიური მონაცემებისთვის თანამედროვე სტანდარტებით განიხილება დიდი ობიექტები (large objects), ორი ფორმატით: სიმბოლოების ობიექტი (character large object - CLOB) და ბინარული ობიექტები (binary large object - BLOB).

არსებობს გარკვეული შეზღუდვები ამ ტიპის ატრიბუტებისთვის: ისინი არ შეიძლება იყოს განსაზღვრული როგორც პირველადი გასაღებური ატრიბუტები, UNIQUE ან მეირადი გასაღებური ატრიბუტები. აგრეთვე არ შეიძლება მათი გამოყენება GROUP BY ან ORDER BY კონსტრუქციებში.

განსაკუთრებით მნიშვნელოვანია სტრუქტურირებული, მომხარებალთა მიერ განსაზღვრული ტიპები (UDT's). ეს ტიპები შეიცავს ატრიბუტებს და მეთოდებს. ასევე ძველ ფუნქციებს და პროცედურებს, რომლებიც 1996 წლიდად SQL92-ის სტანდარტის გაფართოებით იქნა განსაზღვრული, შეეძლოთ ასეთი ტიპის პარამეტრებით სარგებლობა [19].

პროცედურები განსაზღვრული იყო ისე, რომ ისინი შედეგის მნიშვნელობას უკან არ აბრუნებდა, მაგრამ შემავალი და გამომავალი პარამეტრები შეიძლება ჰქონოდა.

ფუნქციები, პირიქით, აბრუნებს უკან მიღებულ შედეგებს და აქვს მხოლოდ შემავალი პარამეტრები.

ტიპები შეიძლება იყოს ორგანიზებული იერარქიებად. ამავე დროს, ტიპები მიეკუთვნება ეგზემპლარულს (instantiable), თუ მათ შეუძლია უშუალოდ შედეგის მნიშვნელობების მოცემა. წინააღმდეგ შემთხვევაში ტიპები აბსტრაქტული ან ვირტუალურია, რაც ნიშნავს, რომ მათ შეუძლია მხოლოდ ქვეტიპების მნიშვნელობებზე მითითება. ქვეტიპების განსაზღვრა ამით თავიდან აცილებულია, რადგანაც ტიპი თვითონ ასრულებს ამ ფუნქციას.

იერარქიულობის ეს პრინციპი არის ურთიერთშენაცვლების: სადაც ტიპის მნიშვნელობა შეიძლება მოთავსდეს, იქ შეიძლება ასევე ქვეტიპების მნიშვნელობათა არსებობა. იმისათვის, რომ ეს შესრულდეს, ტიპის ატრიბუტები და მეთოდები ასევე წვდომადი უნდა იყოს მისი ყველა ქვეტიპისთვის, ანუ მემკვიდრეობითობის თვისება უნდა იყოს რეალიზებული.

განვიხილოთ მაგალითი ტიპისა და ქვეტიპებისათვის.

```
create type Student
    under Person
     as (
          MatrNr  integer,
          Studienfach varchar(30),
    )
    instance method Durchschnittsnote()
           return real
           language SQL
           deterministic
           contains SQL
    . . . ;
```

მეთოდები მხოლოდ ცხადდება ტიპების სღწერაში, ანუ განისაზღვრება მათი სიგნატურები. რეალიზაცია შეიძლება მოგვიანებით განხორციელდეს (ბრძანება create method).

ჩვეულებისამებრ, განასხვავებენ ეგზემპლარულ მეთოდებს (instance method) კლასთა მეთოდებს (static method).

დამატებითი ინფორმაცია მეთოდისთვის შემდეგია:

- returns - განსაზღვრავს უკან დასაბრუნებელ მნიშვნელობის ტიპს. იგი აღწერილია create-method ბრძანებაში;
- language SQL – მიუთითებს, რომ მეთოდის ტანის რეალიზაცია მთლიანად ხორციელდება SQL -ში;
- deterministic – მიუთითებს, რომ უშუალოდ ერთმანეთის მომდევნო გამოძახებები ერთიდაიმავე პარამეტრების მნიშვნელობებით, ყოველთვის ერთიდაიგივე შედეგებს იძლევა;
- contains SQL – მიუთითებს, რომ მეთოდის რეალიზაცია შეიცავს SQL-ბრძანებას, რომელიც არ ეხება ეგზემპლარის ან კლასის მონაცემებს, არამედ მხოლოდ სხვა მონაცემებს ეხება. ალტერნატიული ინფორმაციაა no SQL, როცა არაა მოცემული SQL-ბრძანება; reads SQL data, როცა ეგზემოლარის ატრიბუტები მხოლოდ უნდა წაკითხულ იქნას; modifies SQL data, როცა ატრიბუტები ასევე უნდა შეიცვალოს;
- returns null on null input - მიუთითებს, რომ შედეგი არ ბრუნდება უკან, როცა შემავალი პარამეტრია Nullwert (zero values);

ერთი ტიპის ყოველი ატრიბუტისთვის ორი მეთოდი ავტომატურად გენერირდება:

დამკვირვებელი (Observer) პასუხისმგებელია წაკითხვის წვდომისთვის.

```
instance method MatrNr ()
    returns integer
    language SQL
    deterministic
    contains SQL
```

ვინაიდან მეთოდის გამოძახებისას პარამეტრების გარეშე ყოველთვის ხდება ფრჩხილების გამოტოვება, ამ მეთოდის ვიზუალური გამოძახება არ განსხვავდება ატრიბუტებთან ნორმალური წვდომისგან: Student1.MatrNr.

მუტატორი ახდენს ატრიბუტის მნიშვნელობის ცვლილებას (ჩანაცვლებას):

```
instance method MatrNr (new value)
    returns Student
    self as result
    language SQL
    deterministic
    contains SQL
    returns null on null input
```

შემდეგი მაგალითი უფრო გამოკვეთილად შეესაბამება მულტიმედიურ შემთხვევას:

```
create type ImageType
    under <Supertyp>
    as  (< Attributliste> )
    static method countImages ()
    returns integer
    language SQL
    deterministic
    contains SQL
instance method height ()
    returns integer
    language SQL
    deterministic
   rads SQL data
```

კიდევ ერთხელ, საბოლოოდ უნდა გაესვას ხაზი იმას, რომ კორტეჟები და ცხრილები შეუცვლელად თამაშობს მნიშვნელოვან როლს: იგი უშუალოდ დაკავშირებულია კორტეჟების შეტანასთან ცხრილებში, სხვა ობიექტები ან მნიშვნელობები არ იქმნება. ასევე ყოველთვის შესაძლებელია მოთხოვნები ცხრილებისადმი, რომლებიც გამოსასვლელზე იძლევა ისევ ცხრილებს.

1.3.2. ობიექტრელაციური ბაზის სტრუქტურები მულტიმედია ობიექტისთვის

ობიექტრელაციურ ბაზის სქემაში დასაშვებია მონაცემთა ახალი ტიპების გამოყენება მულტიმედიური ობიექტებისთვის, როგორც მნიშვნელობათა არეები (domains). ატრიბუტები შეიძლება იყოს ტიპებით text, image და სხვა. მაგალითად, ყოველი ამომრჩევლისთვის კორტეჟში უშუალოდ ჩაისმება პირადი ნომერი, გვარი, დაბადების თარიღი, და ა.შ., ასევე ფოტო (სურათი), თითების ანაბეჭდი, ხმა და მოხდება მისი შენახვა:

```
Voter ( Pers_N integer,
        PersName varchar (50),
        BirthsDate date,
        Adress varchar (100),
            . . .
        PersFoto image,
       PersFingerprints image,
       PersVoice sound)
```

დავარქვათ ამ სტრუქტურას, პირობითად, რელაციური სქემა (ტიპი_1), რათა შემდგომში შემოტანილი სხვა ტიპებისგან განვასხვავოთ.

ნორმალურ ფორმათა თეორიის საფუძველზე, სქემათა ოპტიმალური სტრუქტურის მისაღებად, ხშირად საჭიროა ერთი რელაციის დეკომპოზიცია ორ ან მეტ რელაციად, რომლებშიც მოხდება შესაბამისი კორტეჟების გადანაწილება, პირველადი და მეორეული გასაღებური ატრიბუტების განსაზღვრა და ა.შ.

ჩვენი მაგალითისთვის შესაძლებელია ამომრჩევლის პირადი ტექსტური მონაცემების ერთ რელაციაში შენახვა, ხოლო მულტიმედიური მონაცემებისა მეორეში. ქვემოთ ნაჩვენებია ეს შემთხვევა:

```
Person ( Pers_N integer,
```

PersLastName varchar (50),
PersFirstName varchar (30),
BirthsDate date,
Adress varchar (100),
. . .
)
Voters_MM_Daten(Pers_N integer,
PersFoto **image,**
PersFingerprints **image,**
PersVoice **sound**)

მიღებული სტრუქტურა არის რელაციური სქემა (ტიპი_2). ამ შემთხვევაში ამომრჩევლის ვინაობა (გვარი, სახელი,...) უკავშირდება მულტიმედიური მონაცემების რელაციას პირველადი გასაღებური ატრიბუტით Pres_N. ამ ორი რელაციის საფუძველზე მოთხოვნების დასამუშავებლად საჭირო იქნება „Join" გაერთიანების ოპერაციის გამოყენება.

არსებობს შემთხვევები, როდესაც რელაციებს შორის არსებობს m:n დამოკიდებულება. ამ დროს საჭიროა რელაციებსშორის კავშირების რეალიზაცია დამატებითი ინდექსების საფუძველზე, მათ შორის პირველადი და მეორეული გასაღებებითაც. განვიხილოთ მაგალითი, რომელიც ასახავს მაჟორიტარი დეპუტატის (Majority_Deputy) განაწილებას რეგიონის (Region) მხარის (Area) საარჩევნო უბნებზე (Polling_Districts). ერთი დეპუტატი კენჭს იყრის რამდენიმე საარჩევნო უბანზე, ასევე ერთ საარჩევნო უბანზე რამდენიმე კანდიდატია, ანუ საქმე გვაქვს m:n დამოიკდებულებასთან:

Majority_Deputy (majority_deputyID integer,
Deputy_Name varchar (50),
. . .
Deputy_Foto image

)

Polling_District (polling_districtID integer,
Polling_DistrictNr varchar (20),
polling_district_AddressName varchar (100),
areaID integer,
RegionID integer
)

Results_List (polling_districtID integer,
majority_deputyID integer,
number_votes integer,
Percentage double,
Position integer
)

მივიღეთ რელაცირი სქემა (ტიპი_3). ამგვარად, შეიძლება შევაჯამოთ შედეგები შემდეგი სახით:

- სამივე ტიპის რელაციური სქემა მულტიმედიურ ობიექტებსა და არსებს შორის შეიძლება აისახოს 1: 1, 1 : n, m : n დამოკიდებულებათა სახით, სპეციალური სემენტიკის გარეშე;
- მულტიმედიური ობიექტები შეიძლება გამოვლინდეს როგორც ატრიბუტები ან როგორც არსები;
- მიმართვა ზოგიერთ რელაციურ სქემაზე შეიძლება იყოს მოუხერხებელი და მოითხოვდეს რამდენიმე გაერთიანების ოპერაციის (Join) გამოყენებას.

1.3.3. მოთხოვნების დამუშავება ობიექტრელაციურ მონაცემთა ბაზებში

რელაციურ მონაცემთა ბაზების შესახებ, რომლებშიც ტრადიციული, ტექსტური ტიპის ატრიბუტების გარდა არის აგრეთვე მულტიმედიური ტიპებიც, როგორიცაა image, graphics და ა.შ., აქამდე ზედაპირულად იყო დახასიათებული. აქვე ნახსენები იყო მომხმარებლისთვის მოუხერხებელი გამოყენების საშუალება გაერთიანების ოპერაციის სახით.

იმისათის რომ აქ შედარებით ზუსტი სურათის გადმოცემა მოხერხდეს, საჭიროა წარმოდგენილ იქნას ახალ ტიპებზე განსაზღვრული ოპერაციების სავარაუდო ჩანერგვა (embedding) რელაციურ მოთხოვნის ენაში. ამას გვთავაზობს SQL ენა, რომელიც სტანდარტის სახით დამკვიდრდა მოთხოვნათა ენებისთვის.

განვიხილოთ მარტივი, ორატრიბუტიანი რელაციის მაგალითი:

```
Aerial_image ( Nr integer,
               Picture image )
```

კორტეჟების შესატანად ბაზაში საჭიროა SQL-ენის insert-ოპერატორის გამოყენება. ამ დროს ცალკეული ატრიბუტების მნიშვნელობები გადაეცემა როგორც კონსტანტები ან პროგრამის ცვლადები. მაგალითად,

```
insert into  Aerial_image
values (:nr, image(:pr, :cm ));
```

სადაც pr-ში ინახება სურათი, ხოლო cm-ში ფერთა ცხრილები.

გამოსახულებაში ორი წერტილის შემდეგ დგას პროგრამული ცვლადები, რითაც ისინი სინტაქსურად განსხვავდება რელაციების და ატრიბუტების სახელებისგან.

value წინადადების პირველი ჩანაწერი ეკუთვნის ატრიბუტის ნომერს და უნდა იყოს integer ტიპის. მეორე ჩანაწერი ეკუთვნის სურათს და ტიპი image. კომპილატორი ცნობს image ტიპის ოპერაციების პარამეტრების და შედეგის ტიპებს და შეუძლია

შესაბამისი შმოწმების ჩატარება. ეს პრინციპი ძალაშია შემდგომ განხილული ოპერატორებისთვისაც.

თუ კორტეჟი სურათის ატრიბუტებით ერთხელ უკვე შენახულია მონაცემთა ბაზაში, მაშინ შესაძლებელია მათი SQL-ის update ბრძანებით ცვლილება.

```
update  Aerial_image
set  Picture = Picture.replaceColormap(:yuv, 4096, 24, : cm )
where Nr = 1286;
```

შინაარსობრივი მონაცემების მისაღებად შესაძლებელია შემდეგი სახის ბრძანების ჩაწერა:

```
update Aerial_image
set Picture = Picture.newDescr (
      „მოედანს, რომლის ცენტრში ძეგლი და გარშემო\n
       ყვავილებია, უერთდება ხუთი ქუჩა“
 where Nr  = 1234;
```

განსაზღვრული კორტეჟების მოსაძებნად (განსაზღვრული სურათებისთვის) შესაძლებელია ჩვეულებრივი SQL გამოსახულებების გამოყენება. ატრიბუტთა მნიშვნელობების შედარება კონტანტებთან, რომელიც ამ დროს მთავარ როლს თამაშობს, დასაშვებია, უპირველეს ყოვლისა, მხოლოდ სტანდარტული ტიპებისთვის. მულტიმედიური ტიპებისთვის კი არსებობს სპეციალური შედარების ოპერაციები.

პროგრამაზე გადაცემის დროს image ტიპის ატრიბუტებს არ შეუძლია უშუალოდ პროგრამის ცვლადებზე მინიჭება, რადგან ცვლადთა ტიპები სხვადასხვა დანართში შეიძლება სხვადასხვა იყოს. პირიქით, კომპონენტების ამორჩევა და მასთან დაკავშირებული ტიპების შერჩევა (ადაპტაცია) ხდება ცხადად შესაბამისი image - ოპერაციებით:

```
select Picture.getPixrect(), Picture.getColormap()
into :pr,  :cm
```

```
from Aerial_image
where Nr = :k;
```

ამ მაგალითში k-ცვლადში მოცემულია სურათის ნომერი, რომელშიც სურათი (Picture) იქნება გამოძახებული მონაცემთა ბაზიდან Pixrect ფორმატში. შსაძლებელია ასევე სურათის ატრიბუტების თვისებათა შერჩევა, თუ იგი წინასწარ image-ოპერაციების გამოყენებით მონაცემთა ტიპებისთვის შეიქმნა, რომლებზეც განსაზღრულია შედარების ოპერაციები:

```
select Picture.height(),  Picture.width()
into :hoehe,  :breite
from  Aerial_image
where Picture.pixelcount( :dunkelbraun) < 1000
and Picture.noOfColors() > 4095;
```

შეიძლება ალტერნატიული ვარიანტის განხილვაც image - მონაცეთა ტიპის შედარების ოპერაციისთვის:

```
select Picture.description(), . . .
from Aerial_image
where Picture.contains("ცენტრში ძეგლი");
```

ეს უპირველეს ყოვლისა უჩვენებს მომხმარებლის ინტერფეისის სიმარტივეს. ზემოთ აღწერილი select-ბრძანება იძლევა სურათს ნომრით Nr1234, ვინაიდან ამ აღწერაში არის სიტყვები „ცენტრში ძეგლი“ ნახსენები, და იგი როგორც საძებნი გამოსახულება „ცენტრში ძეგლი“, ისე მოიაზრება.

შემდეგ, შენახვის და მოთხოვნის შემთხვევებში შეიძლება გამოიცეს შეცდომის შეტყობინება, როგორიცაა, მაგალითად, „ამ სიტყვას არ ვიცნობ“ ან „ეს ტექსტი არ მესმის“ და სხვა. ესაა ის ღირებულება, რომლითაც მარტივი ტექსტების შედარებისგან განსხვავებით მომხმარებლის ინტერფეისი საძებნი პროცედურის უკეთესი ხარისხით გამოირჩევა.

instert - ბრძანებაში დაუშვებელია კომბინაცია values-წინადადებისა (კონსტანტების ან პროგრამის ცვლადების მონაცემები) და ქვეარჩევის (subselect) (ატრიბუტების ახალი მნიშვნელობების შეკრება მონაცემთა ბაზისთან მიმართვის შესახებ). ასეთი პროცედურა უფრო მაშინაა საჭირო, როდესაც სურათის ნაწილი, ამოღებული მონაცემთა ბაზიდან, უნდა იქნას შენახული სხვა კორტეჟში.

ამ ახალი კორტეჟის ფორმატირებული ატრიბუტები მიიღება არა მონაცემთა ბაზიდან, არამედ პროგრამებიდან. მიზანშეწონილია კონსტანტების მიწოდება select-წინადადებაში, რაც თუმცა არც ძალიან გასაგებად გამოიყურება:

```
insert into Aerial_image
    select  :neueNr, Bild window (50,50,100,100)
    from Aerial_image
    where Nr = :alteNr;
```

update - ბრძანების სემანტიკა, სპეციალური set-წინადადება, არის მნიშვნელობის ჩანაცვლება (აღდგენა), და არა მნიშვნელობის მოდიფიკაცია. მნიშვნელობის მინიჭების კონსტრუქციის მარჯვენა ნაწილში შეიძლება ნებისმიერი სირთულის გამოსახულება იყოს, რომელსაც უნდა ჰქონდეს სწორად შერჩეული შედეგის ტიპი. მაგალითად:

```
update  Aerial_image
apply Picture.replaceColormap ( :cm);
where Nr = 1234;
```

SQL - ბრძანებები პროგრამული ენების სინტაქსის მსგავსად პროცედურების გამოძახებას ჰგავს, და არა ფუნქციურ გამოსახულებას, რომელთაც მნიშვნელობა გააჩნია. ამიტომაც ბრძანებები, როგორც მაგალითად ქვემოთაა ნაჩვენები, ძალზე მგრძნობიარეა მონაცემთა ახალი ტიპების მიმართ, რათა თავიდან იქნას აცილებული შრომატევადი დუბლირების პროცესები:

```
var := select . . . from . . . where . . . ;
writeScreen(select Picture.pixelmatrix ( ) );
```

მიზანშეწონილია, რომ მეთოდები აღიწეროს დეტალურად. ეს ნიშნავს იმას, რომ მოხდეს განსხვავება წასაკითხსა და ჩასაწერს შორის, და მულტიმედიური კონტექსტის თვალსაზრისით, უპირველეს ყოვლისა, განასხვავონ დროზე დამოკიდებული და დამოუკიდებელი პროცესები.

რელაციური მონაცემთა ბაზები, მოთხოვნების დამუშავების თვალსაზრისით, წლების განმავლობაში ასრულებს განსაკუთრებულ როლს, სხვა, ახალი მონაცემთა ბაზებისგან განსხვავებით. მათი გაფართოება ახალი ტიპის, მულტიმედიალური მონაცემებით ძალზე ეფექტურია და პრაქტიკულად ღირებული. მომხმარებელს, მათი გამოყენების მიზნით, სჭირდება ამ მიმართულებით დამატებითი ცოდნის მიღება.

1.4. დოკუმენტ-ორიენტირებული მონაცემთა ბაზა

დოკუმენტ-ორიენტირებული მონაცემთა ბაზა (Document-Oriented Database) არის მონაცემთა ბაზების მართვის სისტემა (მბმს), რომელიც გამოიყენება დოკუმენტების (მონაცემთა იერარქიული სტრუქტურების) შესანახად და რეალიზებულია NoSQL მიდგომის საშუალებით [7,8,20,24].

დოკუმენტ-ორიენტირებული მბმს-ას საფუძვლად უდევს დოკუმენტების საცავი (document Store), რომელსაც აქვს ხის სტრუქტურა. ხის სტრუქტურა იწყება ფესვური კვანძით და შეიძლება შეიცავდეს რამდენიმე შიგა კვანძს და ფოთლების კვანძს.

ფოთელბის კვანძი შეიცავს მონაცემებს, რომლებიც დოკუმენტის დამატების დროს შეიტანება ინდექსებში, რაც უზრუნველყოფს, რთული სტრუქტურების შემთხვევაშიც კი, მოიძებნოს გზა საჭირო მონაცემებისკენ [21].

მოთხოვნის საფუძველზე API (Application Programming Interface) ახორციელებს დოკუმენტების და მათი ნაწილების ძებნას.

დოკუმენტები შეიძლება იყოს ორგანიზებული (დაჯგუფებული) კოლექციებში. ეფექტური ინდექსირების მიზნით სასურველია კოლექციებში მსგავსი სტრუქტურების დოკუმენტების გაერთიანება.

დოკუმენტ-ორიენტირებული მონაცემთა ბაზები გამოიყენება შინაარსის მართვის სისტემებში (CMS -Content Management System), საგამომცემლო საქმეში, დოკუმენტების საძიებო სისტემებში და სხვ. ასეთი ბაზების მართვის სისტემების მაგალითებია: MongoDB, CouchDB, Couchbase, MarkLogic, eXist, IBM Lotus Notes და სხვ.[22-27].

დოკუმენტ-ორიენტირებული მონაცემთა ბაზების მთავარი ცნებაა „დოკუმენტი“, რომელიც განისაზღვრება როგორც მონაცემთა ინკაფსულაცია ინფორმაციის კოდირების სტანდარტული ფორმატებისა და მეთოდების გამოყენების საფუძველზე. ასეთი ფორმატებია: XML, JSON, BSON, YAML. ზოგ შემთხვევაში

შესაძლებელია PDF, Ms Office და მსგავსი დოკუმენტების ბინარული ფორმატით შენახვაც [8].

დოკუმენტი მონაცემთა ბაზაში მისამართდება უნიკალური გასაღების საშუალებით. ხშირად ეს გასაღები მარტივი სტრიქონია, რომელიც შეიძლება იყოს URI (Unified Resource Identifier) ან გზა (path) დოკუმენტამდე. ასეთი გასაღების ან მისი ინდექსის საშუალებით მოიძებნება დოკუმენტი ბაზაში და შესაძლებელია მისი სწრაფად ამოღება.

დოკუმენტური ბაზის დამახასიათებელია სიტყვა-გასაღების (მნიშვნელობის-გასაღების) მარტივად განსაზღვრა მოთხოვნილი დოკუმენტების მოსაძებნად. მონაცემთა ბაზას აქვს სპეციალური API ანუ მოთხოვნების ენა, რომელიც უზრუნველყოფს დოკუმენტების მიღებას მათი შინაარსის (content) მიხედვით.

API არის აპლიკაციების დაპროგრამების ინტერფეისი. იგი შეიცავს მზა კლასების, პროცედურების, ფუნქციების, სტრუქტურებისა და კონსტანტების ერთობლიობას, რომელსაც წარმოადგენს დანართი (ბიბლიოთეკა ან სერვისი) ან ოპერაციული სისტემა. იგი გამოიყენება პროგრამისტების მიერ აპლიკაციის შექმნისას.

მომდევნო თავებში ჩვენ დეტალურად შევეხებით NoSQL მონაცემთა ბაზების საკითხებს და კონკრეტულად, MongoDB პაკეტის შესაძლებლობებს.

1.5. გრაფულ-ორიენტირებული მონაცემთა ბაზა

გრაფული მონაცემთა ბაზის მართვის სისტემებისთვის დამახასიათებელია მონაცემთა გრაფული მოდელი, ანუ ინფორმაციის შენახვა ხდება არა „ცხრილებით“ (tables) და ატრიბუტებით (როგორც რელაციური ბაზებში), არამედ გრაფული სტრუქტურებით, ანუ კვანძებით (nodes) და მათ შორის კავშირებით (გრაფის წიბოები - edges). ასეთი კავშირები შეიძლება რამდენიმე დონეს მოიცავდეს (ღრმა კავშირები) და ისინი ძალზე აქტუალურია დიდი სოციალური პროექტების (ქსელების), ბიოინფორმატიკის, რთული მარშრუტების, სემანტიკური ქსელის (Web, HTTP გვერდებით) და სხვა სფეროს ამოცანების გადასაწყვეტად.

გრაფული მონაცემთა ბაზა არის ქსელური მოდელის (ან RDF-მოდელის) რეალიზაციის ნაირსახეობა [23,24]. მისი კონცეფცია ჯერ კიდევ 80-იან წლებში გამოჩნდა, ხოლო პირველი გრაფული რეალიზაცია 2007 წელს, Neo4j სისტემის სახით. დღეისთვის უკვე არსებობს რამდენიმე ათეული ასეთი ბაზებისა, მაგალითად: ArangoDB, OrientDB, MarkLogic, Oracle Spatial and Graph და სხვ.

RDF (Resource Description Framework) - რესურსის აღწერის გარემო შეიქმნა WWW კონსორციუმის მიერ როგორც მონაცემთა აღწერის მოდელი - მეტამონაცემებით.

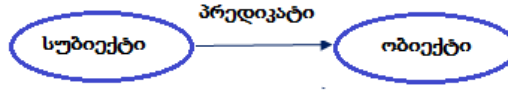
რესურსი RDF-ში შეიძლება იყოს ნებისმიერი არსი, როგორც ინფორმაციული (მაგალითად, ვებ-გვერდი, გამოსახულება), ასევე არაინფორმაციული (მაგალითად, ადამიანი, მანქანა ან აბსტრაქტული ცნება). რესურსის შესახებ გამონათქვამის მტკიცებას აქვს სამეულის (Triple Stores - სამადგილიანი შენახვა) სახე:

„სუბიექტი — პრედიკატი — ობიექტი“

მაგალითად, მტკიცება „დოლიძე არის პროფესორი“, RDF-ის ტერმინოლოგიით ჩაიწერება ასე:

სუბიექტი — „დოლიძე“, პრედიკატი — „აქვს თანამდებობა“, ობიექტი — „პროფესორი“.

გრაფიკულად იგი 1.27 ნახაზზეა მოცემული



ნახ.1.27. RDF-ის სამეული

ამგვარად, RDF-ის მტკიცებულებები (ფაქტები) ქმნის ორიენტირებულ გრაფს, რომელშიც კვანძებია სუბიექტები და ობიექტები, ხოლო წიბოები - მათ შორის მიმართებები. RDF არის მონაცემთა აბსტრაქტული მოდელი, ანუ იგი აღწერს მოცემულ სტრუქტურას, დამუშავების ხერხებს და მონაცემთა ინტერპრეტაციებს.

ქვემოთ მოცემულია NoSQL ტიპის ოჯახის ზოგიერთი პოპულარული მბმს, რომლებიც გრაფულ ან ჰიბრიდულ ბაზებს მიეკუთვნება [23].

AllegroGraph – დამუშავებულია W3C სტანდარტით Common Lisp ენაზე Triple Store (პრედიკატული სამეული) სახით მონაცემთა RDF მოდელისთვის, Windows, Linux და Mac OSX -თვის, კლიენტის ინტერფეისებით: Java, Python, Ruby, Perl, C#, Clojure და Common Lisp.

ArangoDB – დაწერილია C++ და JavaScript ენებზე მულტი-მოდელური ბაზის სახით. გამოიყენება როგორც key/value, document, და graph data ბაზები და აქვთ ერთი საერთო მოთხოვნების ენა [25]. 2011 წლამდე გამოდიოდა AvocadoDB სახელით.

DataStax Enterprise Graph (DSE) – აგებულია Java ენაზე Web-საიტებისა და მობილური ტექნიკისთვის. სერვერის მხარეს Backend-ის სახით იყენებს Apache Cassandra-ს. შეუძლია დაამუშაოს წამში პეტაბაიტი ინფორმაცია და ერთდროულად მოემსახუროს ათას მომხმარებელს. ბაზა განაწილებულია კვანძების კლასტერებში და აქვს მასშტაბირებადი არქიტექტურა [24,28]. მასში ჩადგმულია OLAP ანალიზის და გრაფში ძებნის მხარდაჭერა. აქვს უსაფრთხოების დამატებითი პარამეტრები კონფიდენციალური მონაცემებისთვის.

MarkLogic – მულტიმოდელური ბაზაა სემანტიკური გრაფით და RDF სამეულით. ინახავს დოკუმენტებს JSON (JavaScript Object Notation) და XML ფორმატებში [24,29]. აქვს ჩადგმული საძიებო სისტემა, ACID მახასიათებლების მქონე ტრანზაქციები – მაღალი წვდომადობა და ავარიული აღდგენის უნარი, გარანტირებული უსაფრთხოება, მოქნილობა და მასშტაბურობა.

Neo4j – კომპანია Neo Technology-ის პროდუქტი, დამუშავებულია java ენაზე და ერთ-ერთი ყველაზე პოპულარული გრაფული ბაზაა [24,30]. აპლიკაციების დაპროგრამების ინტერფეისი მონაცემთა ბაზისთვის რეალიზებულია მრავალი ენისთვის, მათ შორის: Java, Python, Ruby, PHP და სხვ.

OrientDB – გრაფული- და დოკუმენტ-ორიენტირებული ბაზის სისტემაა, დამუშავებულია Java ენაზე Orient Technologies LTD ფირმის მიერ Windows, Linux, Mac და სხვა ოპერაციული სისტემებისთვის [24,31]. მოთხოვნების ენისათვის აქვს SQL-ის მხარდაჭერა (ამიტომაც მას NewSQL ბაზასაც მიაკუთვნებენ). იგი არ იყენებს JOIN ოპერაციას. მის მაგივრად აქვს სუპერ-სწრაფი მუდმივი მიმთითებლები ჩანაწერებს შორის, რომლებიც გრაფული ბაზებისთვისაა დამახასიათებელი. ეს უზრუნველყოფს ჩანაწერების ცალკეული ან მთლიანი ხეების და გრაფების გადასინჯვას რამდენიმე მილიწამის ფარგლებში.

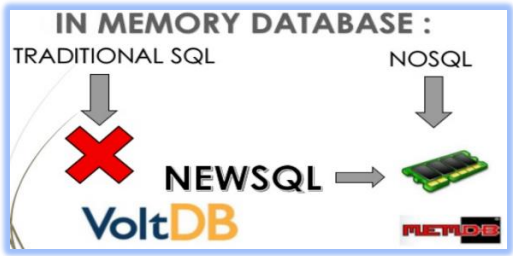
Stardog – არის კროსპლატფორმული, სემანტიკური გრაფული ბაზის სიტემა, რეალიზებული Java ენაზე, RDF-ის და OWL (Web Ontology Language)-ის მხარდაჭერით [32]. OWL ენით აღიწერება კლასები და მიმართებები მათ შორის, რომლებიც დამახასიათებელია ვებ-დოკუმენტებისა და აპლიკაციებისთვის [33].

1.6. NewSQL მონაცემთა ბაზები

NewSQL არის თანამედროვე რელაციური ბაზების მართვის სისტემების კლასი, რომელიც გაფართოებული ფუნქციონალობის საფუძველზე უზრუნველყოფს NoSQL ბაზების მსგავს მწარმოებლურობას ტრანზაქციების ოპერატიული დამუშავებისათვის [26, 34]. ამასთანავე იგი ინარჩუნებს ACID პრინციპებს.

განსაკუთრებით საყურადღებოა აქ მონაცემთა შენახვის პრინციპულად ახალი პლატფორმების შექმნა, რომლებიც ორიენტირებულია განაწილებული არქიტექტურის და მრავალნაკადურ სისტემებზე.

ასეთი მიდგომის ერთ-ერთი პოპულარული მონაცემთა ბაზაა MemSQL (რელაციური ბაზა ოპერატიული მეხსიერებით [In-Memory Storage] Linux ოპერაციული სისტემისთვის) [36] (ნახ.1.28). იგი იყენებს SQL ენას. კოდის გენერაცია სრულდება C++ ენაზე. ანუ MemSQL სერვერზე გაგზავნილი მოთხოვნები გარდაიქმნება C++ -ზე და კომპილირდება GCC-ს დახმარებით.



ნახ.1.28

MemSQL თავსებადია MySQL-თან. აპლიკაციები შეიძლება შეერთდეს MemSQL სისტემასთან ODBC/JDBC სტანდარტებით, ასევე დრაივერებით და MySQL-ის მომხმარებლებით.

გარდა ზემოთ აღნიშნულისა, ლიტერატურულ წყაროებში განიხილავენ NewSQL-ის ტიპის შემდეგ ბაზებს: NuoDB, VoltDB, OrientDB, Clustrix, ScaleDB, dbShards და სხვ [36,37] (ნახ.1.29).

ნახ.1.29

1.30 ნახაზზე მოცემულია მონაცემთა ტრადიციული SQL ბაზების, NoSQL და NewSQL ბაზების შედარება ოთხი ძირითადი თვისებით (Properties) [36].

ნახ.1.30

როგორც ვხედავთ, NewSQL მონაცემთა ბაზა აერთიანებს ტრადიციული (რელაციური) და NoSQL (არარელაციური) ბაზების საუკეთესო თვისებებს, ამიტომაც იგი ძალზე პერსპექტიულია სამომავლო პროექტებისათვის.

ერთ-ერთი საინტერესო გადაწყვეტა ამ თვალსაზრისით არის MySQL და NoSQL ბაზების ინტეგრაციის საკითხი, რომელსაც მომდევნო პარაგრაფში განვიხილავთ.

1.7. NoSQL მონაცემთა ბაზა MySQL ბაზასთან ერთად

ერთ-ერთი აქტუალური მიმართულება მონაცემთა რელაციური ბაზების მწარმოებლურობის (ეფექტიანობის) ამაღლების მიზნით არის NoSQL ბაზების პრინციპების ჩანერგვა რელაციურ სისტემებში. ამის კონკრეტული მაგალითია Oracle კორპორაციის მიერ C-ენაზე დაწერილი კროსპლატფორმული პროდუქტი InnoDB, რომელიც გამოიყენება MySQL მონაცემთა ბაზების მართვის სისტემაში (5.5 ვერსიიდან) [39].

InnoDB არის მონაცემთა საცავი (database engine, storage engine), ბაზების მართვის სისტემის პროგრამული კომპონენტი, რომელიც გამოიყენება მონაცემთა ბაზის შენახვის, განახლების და ინფორმაციის ძებნის (create, update, delete, read) მექანიზმების სამართავად.

ამგვარად, Oracle-მ მოახერხა MySQL-ის (v.5.6) InnoDB-ში NoSQL-ის შესაძლებლობების დამატებით 9-ჯერ აემაღლებინა ტრანზაქციათა დამუშავების ეფექტურობა [27].

აქ რეალიზებულია NoSQL-ის ინტერფეისები MySQL და MySQL Cluster მონაცემთა ბაზებთან. ისინი სრულიად უვლის გვერდს SQL შრეს, მის სინტაქსურ ანალიზს და ოპტიმიზაციას. მონაცემები შეიძლება უშუალოდ ჩაიწეროს MySQL-ის ცხრილებში 9-ჯერ უფრო სწრაფად, ACID პრინციპების დაცვით.

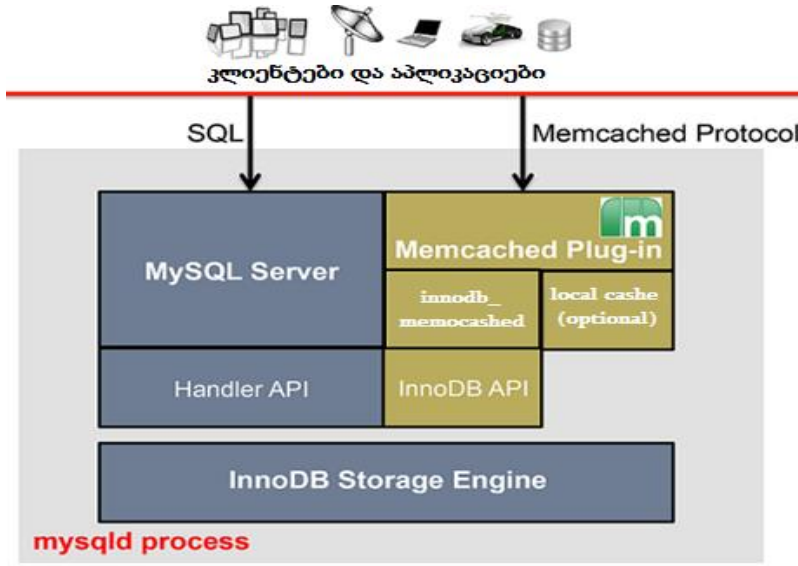
როგორაა NoSQL რეალიზებული MySQL-ში ?

MySQL 5.6 უზრუნველყოფს მარტივ, პირდაპირ *გასაღებ-მნიშვნელობა* ურთიერთქმედებას InnoDB-ს მონაცემებთან ცნობილი API-ის კეშ-მეხსიერების დახმარებით.

კეშ-მეხსიერება (Memcached) წარმოადგენს საერთო დანიშნულების განაწილებული მეხსიერების კეშირების სისტემას [40]. იგი ხშირად გამოიყენება მონაცემთა ბაზების მქონე დინამიკური ვებ-საიტების დასაჩქარებლად მონაცემებისა და ობიექტების კეშირების

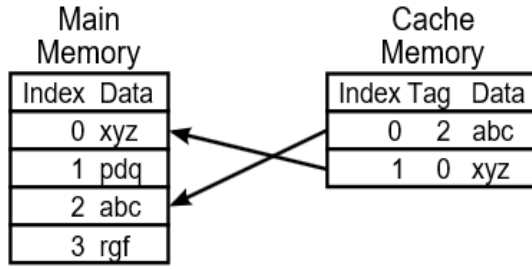
დახმარებით ოპერატიულ მეხსიერებაში. აქ მნიშვნელოვნად მცირდება, მაგალითად, აპლიკაციის ინტერფეისით ინფორმაციის წაკითხვა მონაცემთა გარე წყაროებიდან. API შესაძლებლობას იძლევა აგრეთვე Memcached-ისა და კლიენტების სტანდარტული ბიბლიოთეკით (C++, Java, Python, PHP და სხვა ენებისათვის) კეშირებულ იქნას მონაცემები ხელმისაწვდომი სერვერების ოპერატიულ მეხსიერებაში და შემდგომ მრავალჯერ იქნას გამოყენებული.

სისტემის რეალიზაცია მოცემულია 1.31 ნახაზზე [27].



ნახ.1.31

MySQL-ის გაფართოებულ ვერსიაში NoSQL შესაძლებლობებით კლიენტების და აპლიკაციების მოთხოვნები შედის სისტემაში SQL ენაზე (პირდაპირ MySQL Server-ზე) ან Memcached პროტოკოლით (InnoDB-ს გავლით). მეორე შემთხვევაში აპლიკაციებისთვის გარანტირებულია მაღალი სისწრაფის კითხვა-ჩაწერის ოპერაციების შესრულება, ვინაიდან მონაცემები ინახება კეშ-მეხსიერებაში (ნახ.1.32).



ნახ.1.32

კეშ-მეხსიერება (Cache memory) არის აპარატურული ან პროგრამული ნაწილის კომპონენტი, რომელიც იქმნება მონაცემთა დროებით შესანახად, მათი სწრაფად, მრავალჯერადი გამოყენების მიზნით. აქ ინახება წინასწარ გათვლილი ან ამორჩეული მონაცემთა ერთობლიობა, რომელთა ხშირად გამოყენების ალბათობა მაღალია. ძირითად მეხსიერებასთან შედარებით კეშ-მეხსიერება არაა დიდი ზომის.

1.8. MariaDB ბაზა MySQL-ის Open Source ალტერნატივა

მონაცემთა ბაზა MariaDB არის MySQL-ის ახალი ალტერნატიული free-ვარიანტი, რომელიც შექმნა მაიკლ ვიდენიუსმა 2009 წელს. იგი იყო ავტორი MySQL-ისაც (1995), რომელიც Oracle კორპორაციამ შეიძინა 2008 წელს და გახადა იგი კომერციული პროდუქტი [40]. (მ. ვიდენიუსის უფროსი ქალიშვილია მაია - MySQL, ხოლო უმცროსი - მარია).

MariaDB თავსებადია MySQL-თან, უზრუნველყოფს შესაბამისობას API-სთან და MySQL-ის ბრძანებებთან. მასში დამატებულია მონაცემთა საცავის (storage engine) ქვესისტემა XtraDB, რომელიც

ცვლის MySQL-ის InnoDB-ს [41-43]. ეს საცავი მაღალმწარმოებლურია InnoDB-სთან შედარებით, აქვს მეხსიერების მართვის და ინფორმაციის ნაკადების შეტანა-გამოტანის უფრო სრულყოფილი მექანიზმები, რეალიზებულია ტრანზაქციების ACID მოთხოვნები (Atomicity, Consistency, Isolation, Durability) და აქვს MVCC (MultiVersion Concurrency Control - მონაცემთა ბაზასთან წვდომის პარალელური უზრუნველყოფა) არქიტექტურა.

იმისათვის, რომ დავაყენოთ MariaDB მონაცემთა ბაზა, უნდა შესრულდეს შემდეგი პირობები:

1) გამართული უნდა იყოს ლინუქსის ოპერაციული სისტემა ფიზიკურ ან ვირტუალურ მანქანაზე (2 CORE , 2 GB RAM , 20 GB HDD). რაც შეეხება Linux-ის დისტრიბუციას, არჩევანი დიდია (მაგალითად, CentOS-ს, რომელიც დიდი პოპულარობით სარგებლობს) [44,45];

2) გამართულ ოპერაციულ სისტემას წვდომა უნდა ჰქონდეს ინტერნეტში (ბაზის დაყენების დროს).

მას შემდეგ რაც ოპერაციული სისტემა გამართულია და ინტერნეტში წვდომაც გვაქვს, შეგვიძლია დავიწყოთ მონაცემთა ბაზის დაყენება (აღნიშნული ინსტრუქცია გათვლილია „CentOS 6 64-bit“-სთვის) [41,42].

ოპერაციულ სისტემაში შევდივართ „root“ მომხმარებლით და ტერმინალში ვწერთ შემდეგ ბრძანებებს:

1) touch /etc/yum.repos.d/MariaDB.repo რაც შექმნის MariaDB.repo ფაილს /etc/yum.remos.d/ დირექტორიაში;

2) vi/etc/yum.repos.d/MariaDB.repo vi ედიტორით გავხსნათ MariaDB.repo ფაილი კლავიატურაზე „i“ ღილაკის გამოყენებით გადავიდეთ „insert“ რეჟიმში და ფაილში ჩავწეროთ შემდეგი :

- [mariadb]
- name = MariaDB
- baseurl = http://yum.mariadb.org/5.5/centos6-amd64
- gpgkey=https://yum.mariadb.org/RPM-GPG-KEY-MariaDB

- gpgcheck=1

კლავიატურაზე „Esc“ ღილაკის გამოყენებით გადავიდეთ ბრძანების რეჟიმში ვწერთ „wq“ და ვაწვებით „Enter“ ღილაკს, რის შემდეგადაც ჩვენი შეყვანილი ინფორმაცია შეინახება „MariaDB.repo“ ფაილში.

3) yum -y install MariaDB MariaDB-server (დაიწყება ბაზის ინსტალაცია).

4) /etc/init.d/mysql start (მონაცემთა ბაზის გაშვება)

5) როდესაც მონაცემთა ბაზა გაეშვება „mysql“ ბრძანებით შეგვიძლია შევიდეთ მონაცემთა ბაზის ტერმინალში სადაც გაუშვებს mysql ის ბრძანებებს:

- show databases;
- quit;

6) მას შემდეგ, რაც მონაცემთა ბაზას წარმატებით დავუკავშირდით და ყველაფერმა იმუშავა, აუცილებელია უსაფრთხოების პარამეტრების გამართვა, რისთვისაც ვუშვებთ შემდეგ ბრძანებას და დაკვირვებით გავივლით შემდგომ ეტაპებს:

mysql_secure_installation

```
/usr/bin/mysql_secure_installation: line 379: find_mysql_client:
command not found
NOTE: RUNNING ALL PARTS OF THIS SCRIPT IS RECOMMENDED
FOR ALL MariaDB
SERVERS IN PRODUCTION USE!  PLEASE READ EACH STEP
CAREFULLY!
In order to log into MariaDB to secure it, we'll need the current
password for the root user.  If you've just installed MariaDB, and
you haven't set the root password yet, the password will be blank,
so you should just press enter here.
Enter current password for root (enter for none):
OK, successfully used password, moving on...
Setting the root password ensures that nobody can log into the MariaDB
root user without the proper authorisation.
Set root password? [Y/n] Y
New password:
Re-enter new password:
```

Password updated successfully!
Reloading privilege tables..
... Success!
Remove anonymous users? [Y/n] y
... Success!
Normally, root should only be allowed to connect from 'localhost'. This ensures that someone cannot guess at the root password from the network.
Disallow root login remotely? [Y/n] y
... Success!
By default, MariaDB comes with a database named 'test' that anyone can
access. This is also intended only for testing, and should be removed before moving into a production environment.
Remove test database and access to it? [Y/n] y
- Dropping test database...
... Success!
- Removing privileges on test database...
... Success!
Reloading the privilege tables will ensure that all changes made so far will take effect immediately.
Reload privilege tables now? [Y/n] y
... Success!
Cleaning up...
All done! If you've completed all of the above steps, your MariaDB installation should now be secure.
Thanks for using MariaDB!

7) /etc/init.d/mysql restart (კონფიგურაციის გავლის შემდეგ აუცილებელია მონაცემთა ბაზის გადატვირთვა).

8) chkconfig mysql on (სისტემის ჩატვირთვისას მონაცემთა ბაზა ავტომატურად რომ გაეშვას)

9) mysql -u root –p (მონაცემთა ბაზის ტერმინალში შესასვლელად -u პარამეტრით გადავცემთ მომხმარებლის სახელს, ხოლო -p პარამეტრით პაროლს).

1.9. MongoDB ბაზა და მისი ინსტალაცია

MongoDB არის დოკუმენტზე ორიენტირებული NoSQL მონაცემთა ბაზა [46]. მისი ინსტალიაციის მიზნით კომპიუტერზე აუცილებელია იგივე წინაპირობები და ეტაპები, რაც MariaDB-სთვის:

1) სისტემაში შევდივართ „root“ მომხმარებლით
2) touch /etc/yum.repos.d/mongodb.repo
3) vi /etc/yum.repos.d/mongodb.repo vi ედიტორის საშუალებით mongodb.repo ფაილში ჩავწრეოთ შემდეგი :

- [mongodb]
- name=MongoDB Repository
- baseurl=http://downloads-distro.mongodb.org/repo/redhat/os/x86_64/
- gpgcheck=0
- enabled=1

4) yum –y install mongo-10gen mongo-10gen-server (მონაცემთა ბაზის დაყენება)
5) service mongod start (მონაცემთა ბაზის სერვისის გაშვება)
6) service mongod status (მონაცემთა ბაზის სტატუსის შემოწმება)
7) mongostat (მონაცემთა ბაზაში მიმდინარე პროცესების ნახვა)

ამით MongoDB-ს დაყენება დამთავრებულია. იმისათვის, რომ მონაცემთა ბაზაში მუშაობა შევძლოთ, ოპერაციული სისტემის ტერმინალზე უნდა ავკრიფოთ mongo და შევიდეთ მონაცემთა ბაზის ტერმინალში, სადაც უშუალოდ mongo-ს ბრძანებების გაშვებას შევძლებთ. mongo (მონაცემთა ბაზის კლიენტი) ბრძანების გაშვებისას ოპერაციული სისტემა ავტომატურად მიმართავს „localhost:27017“ და ცდის ბაზასთან დაკავშირებას.

მომდევნო თავებში დეტალურად გავეცნობით ამ ბაზის ფუნქციონირების პრინციპებს და მუშაობის შესაძლებლობებს.

1.10. Hadoop – „დიდ მონაცემთა“ ახალი ტექნოლოგია

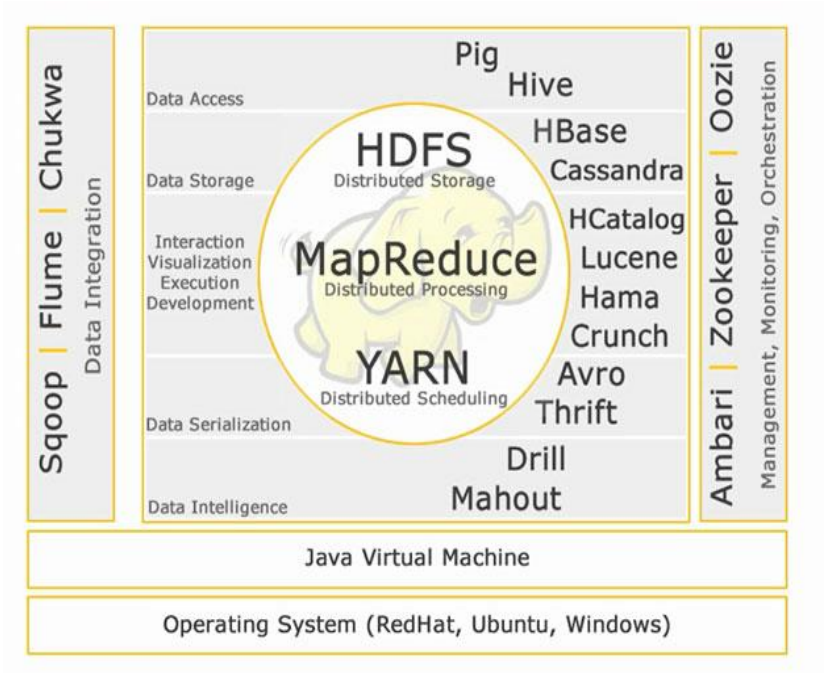
ჩვენ ვცხოვრობთ ინფორმაციის ეპოქაში. 2013 წლისთვის ციფრული სამყაროს ზომა 4.4 ზეტაბაიტი იყო, 2020 წლისთვის კი ნავარაუდებია ინფორმაციის მოცულობის ათმაგი ზრდა, 44 ზეტაბაიტამდე (44 მილიარდი ტერაბაიტი). [51]

ამ მოცულობის ინფორმაციიდან გარკვეული ნაწილი ისეთ კორპორაციებზე მოდის, როგორიცაა ნიუ იორკის საფონდო ბირჟა – 4-5 ტერაბაიტი დღეში [52], Facebook.com – ჯამში 240 მილიარდი ფოტო, თვეში 7 პეტაბაიტი ზრდის მახასიათებლით [53], Ancestry.com – 10 პეტაბაიტი მოცულობის გენეალოგიის ბაზა [54].

ეს ადამიანის შექმნილი მონაცემებია, მაგრამ ბოლო ათწლეულში ტენდენცია შეიცვალა, დღეს უკვე ინფორმაციის უდიდეს ნაწილს ადამიანების ნაცვლად კომპიუტერული ტექნიკა აგენერირებს.

ბოლო წლებში აქტუალური გახდა ტერმინი IOT (Internet Of Things), რაც თავის თავში მოიცავს ყველა იმ აპარატს და კომპიუტერულ ტექნიკას, რაც მიმდინარე დროში დიდი რაოდენობით მონაცემებს აგენერირებს. მაგალითად, მანქანის GPS სისტემები, სხვადასხვა სენსორები და ყველა ის ტექნიკა, რაც ძირითადი ფუნქციონირების პარალელურად წარმოშობს დიდი რაოდენობის დამხმარე ინფორმაციას (metadata).

ამ რაოდენობის მონაცემების დამუშავებას სრულიად განსხვავებული სისტემა სჭირდება არა მხოლოდ რელაციური და არარელაციური ბაზების დონეზე, არამედ იმ სერვერული არქიტექტურის დონეზე, სადაც ვაყენებთ მონაცემთა ბაზებს. დღეისათვის საუკეთესო გამოსავალი დააპროექტა Apache Software Foundation-მა, სახელით Hadoop. Hadoop არის უფასო, ჯავაზე დაფუძნებული პლატფორმა, რომელიც შექმნილია დიდი ზომის მონაცემთა ნაკადის დასამუშავებლად (ნახ.1.33) [37].



ნახ.1.33

Hadoop ეკოსისტემაში სხვადასხვა პროდუქტებია გაერთიანებული, ბირთვად კი სამი ძირითადი კომპონენტი აქვს:

- HDFS (Hadoop Distributed File System) - განაწილებული ფაილური სისტემა მონაცემების შესანახად;
- Map Reduse - მთავარი კომპონენტი განაწილებული გამოთვლების ჩასატარებლად;
- YARN (Yet Another Resource Negotiator) - განაწილებული გარემოს მართვა.

1.11. დასკვნა

ბოლო ათწლეულში მონაცემთა ბაზების მართვის სისტემების განვითარება ხორციელდებოდა რამდენიმე მიმართულებით.

მთავარი იყო *რელაციური ბაზების*, როგორც უმრავლესი მართვის საინფორმაციო სისტემების ძირითადი კომპონენტის სრულყოფა მწარმოებლურობის თვალსაზრისით. ვითარდებოდა მოთხოვნების სწრაფად დამუშავების ოპტიმიზაციის მეთოდები და ალგორითმები.

მეორე მიმართულება გახდა NoSQL („არარელაციური“ ან „არა მხოლოდ რელაციური“) ტიპის ბაზების მართვის სისტემების შექმნა და სრულყოფა, განსაკუთრებით „დიდ მონაცემთა“ დასამუშავებლად, სადაც რელაციური ბაზები არაეფექტურია, მათი Join და სხვა რელაციური ოპერაციებით ცხრილების დამუშავების პროცესის დიდი დროის გამო. ასეთია, მაგალითად, დიდ კორპორაციულ საცავებში დოკუმენტებთან მუშაობის სისტემები.

მესამე მიმართულება ეხება გრაფული ბაზების შექმნას და განვითარებას. მათთვის დამახასიათებელია ინფორმაციის შენახვა გრაფული სტრუქტურებით და აქტუალურია დიდი სოციალური ქსელების, ბიოინფორმატიკის და სხვა სფეროების ამოცანების გადასაწყვეტად.

ბოლოს, შეიძლება ვახსენოთ ჰიბრიდული, NewSQL მონაცემთა ბაზების მართვის სისტემები, სადაც ინტეგრირდება რელაციური, NoSQL-ის და გრაფული ბაზების კონცეფციები.

ყველა მიმართულება აქტუალურია და სწრაფად ვითარდება მსოფლიოს წამყვან უნივერსიტეტებსა და პროგრამული სისტემების ფირმებში. რომელი ტიპის ბაზას აირჩევს მომხმარებელი, დამოკიდებულია კონკრეტული მართვის საინფორმაციო სისტემის მიზნებსა და ფუნქციებზე.

II თავი
NoSQL მონაცემთა ბაზების ძირითადი პრინციპები MongoDB-ს მაგალითზე

როგორც შესავალში აღვნიშნეთ, ტერმინი „ახალი არარელაციური“ შემთხვევითი არაა, რადგან არსებობდა ასევე „ძველი არარელაციური“ მონაცემთა მოდელები და ბაზებიც [2-4]. ისინი ედგარ კოდის რელაციური ბაზის იდეის რეალიზაციამდე გამოიყენებოდა [1]. ესენი იყო იერარქიული და ქსელური ბაზები (მაგალითად, IMS/2. OKA, CODASYL, ADABAS და სხვ.). დღეს ისინი ისტორიულ მუზეუმში იმყოფება, ვინაიდან რელაციური ბაზის მართვის სისტემები უკონკურენტოა (გამოყენების უმრავლეს სფეროებში).

მაგრამ უნაკლო არაფერია, მონაცემთა რელაციურ ბაზებსაც გააჩნია სუსტი მხარეები. ამიტომ წლების განმავლობაში იქმნებოდა რელაციური ბაზების გაფართოებული მოდელები, და ბოლოს, გაჩნდა NoSQL ახალი „არარელაციური“ (ან „არა მხოლოდ რელაციური“) ბაზების კონცეფციაც.

სწორედ ამ საკითხებს შევეხებით წიგნის ამ ნაწილში.

2.1. რელაციური და NoSQL ბაზების შედარება

რელაციური და არარელაციური მოდელები საკმაოდ განსხვავდება ერთმანეთისგან. რელაციურ მოდელისთვის არსებობს სქემა და მონაცემები განთავსებულია მის დაკავშირებულ ცხრილებში (Tables). ცხრილი შეიცავს სტრიქონებს (Rows) და სვეტებს (Columns). ცხრილები ერთმანეთს უკავშირდება პირველადი (PrimaryKey) და მეორადი (ForeignKey) გასაღებების საშუალებით. როდესაც მოთხოვნის საფუძველზე იძებნება რაიმე ინფორმაცია, შესაბამისი ჩანაწერები მიიღება რამდენიმე ცხრილის

შეერთების (join), პროექციის, შეზღუდვის, უნიკალურობის (სიმრავლის) და/ან სხვა ოპერაციების მიმდევრობით (ან პარალელური) შესრულების საფუძველზე. მსგავსია ჩაწერის და მოდიფიკაციის პროცედურებიც, რომლებიც უნდა მოხდეს რამდენიმე ცხრილში ერთდროულად (ბაზის მთლიანობის შესანარჩუნებლად).

NoSQL ბაზებს, რელაციურთან შედარებით, აქვს სრულიად განსხვავებული მოდელი. მაგალითად: დოკუმენტზე ორიენტირებული NoSQL ბაზები მონაცემებს ინახავს JSON (JavaScript Object Notation) ფორმატში [24]. თითოეული JSON დოკუმენტი შეიძლება განვიხილოთ როგორც ობიექტი, რომელსაც მიიღებს აპლიკაცია. ის შეიძლება შეიცავდეს რელაციური მოდელის რამდენიმე ცხრილის გადაბმით მიღებულ ინფორმაციას ერთ დოკუმენტში/ობიექტში, რაც უზრუნველყოფს ჩაწერა/წაკითხვის ოპერაციების წარმადობის გაუმჯობესებას.

მაგალითად, MongoDB არის open-source მონაცემთა ბაზის სისტემა. იგი ინახავს მონაცმებს JSON ტიპის დოკუმენტებში, რომელთა სტრუქტურაც შეიძლება იცვლებოდეს [47,55]. ინფორმაცია ინახება ერთად. MongoDB იყენებს დინამიურ სქემებს, რაც ნიშნავს, რომ შესაძლებელია ჩანაწერების შექმნა სტრუქტურის (ველების, მნიშვნელობათა ტიპების) წინასწარი განსაზღვრის გარეშე. შემდგომ შეიძლება ჩანაწერების სტრუქტურის (ანუ დოკუმენტების) მარტივად შეცვლა ახალი ველის დამატებით ან არსებულის წაშლით.

ეს მოდელი გვაძლევს საშუალებას წარმოვადგინოთ იერარქიული კავშირები და სხვა უფრო რთული სტრუქტურები შედარებით მარტივად. დოკუმენტებს კოლექციაში არ სჭირდება იდენტური ველები და მონაცემთა დენორმალიზაცია არის აქ ჩვეულებრივი მოვლენა. MongoDB ბაზის სისტემა შეიქმნა მაღალი წვდომადობისა და მასშტაბირების რეალიზაციის მიზნით, ფლობს რეპლიკაციას და ავტომატურ სეგმენტაციას (auto-sharding).

MySQL და MongoDB ბაზებში მრავალი საერთო ტერმინი და ცნებაა (ცხრ.2.1) [47].

ტერმინების შედარება ბაზებში **ცხრ.2.1**

MySQL	MongoDB
Table	Collection
Row	Document
Column	Field
Joins	Embedded documents, linking

Collection არის mongo დოკუმენტების ჯგუფი. იგი RDBMS ცხრილების ეკვივალენტია.

Document – ცხრილის ძირითადი ნაწილია. დოკუმენტები არის დინამიური. MongoDB და SQL ბაზები გვთავაზობს ფუნქციების მდიდარ კომპლექსს (ცხრ.2.2) [47].

ფუნქციები და შესაძლებლობები **ცხრ.2.2**

დასახელება	MySQL	MongoDB
Rich Data Model	No	Yes
Dynamic Schema	No	Yes
Typed Data	Yes	Yes
Data Locality	No	Yes
Field Updates	Yes	Yes
Easy for Programmers	No	Yes
Complex Transactions	Yes	No
Auditing	Yes	Yes
Auto-Sharding	No	Yes

MongoDB ბაზას აქვს მაღალფუნქციონალური მოთხოვნების ენა (query language), რომელსაც შეუძლია აგრეთვე ტექსტებთან და სივრცით მონაცემებთან (გეოსისტემები) მუშაობა. აღნიშნული ფუნქციების გამოყენება შესაძლებელია მონაცემთა მრავალი სახის ტიპებთან, ვიდრე რელაციურ ბაზებში.

ქვემოთ მოყვანილია რამდენიმე მარტივი მოთხოვნის ფორმირების მაგალითი MySQL და MongoDB ბაზებთან მუშაობის დროს:

- „შევიტანოთ products მონაცემთა ბაზაში (იხ. ნახ.1.4) ახალი პროდუქტის შესახებ ინფორმაცია (სტრიქონი), რომელშიც გვაქვს ოთხი ველი: პროდუქტის იდენტიფიკატორი (pr_ID), დასახელება (Name), ფასი (price) და პროდუქტის კატეგორიის იდენტიფი-კატორი (cat_ID)“.
  - MySQL-ში:

```
INSERT INTO products (pr_ID, Name, price, cat_ID)
VALUES (101, 'არაჟანი', 5.80, 6)
```

  - MongoDB-ში:

```
db.products.insert ({
    pr_ID: 101,
    Name: 'არაჟანი',
    price: 5.80,
    cat_ID: 6
})
```

- „ვნახოთ (ავირჩიოთ) ყველა პროდუქტის მონაცემები“:
  - MySQL-ში:

```
SELECT * FROM products
```

  - MongoDB-ში:

```
db.products.find()
```

- „შევცვალოთ (გავზარდოთ) მე-6 კატეგორიის („რძის ნაწარმი“) პროდუქტების ფასი 20 % -ით“:
  - MySQL-ში:

```
UPDATE products SET price = price*0.2
WHERE cat_ID = 6
```

- MongoDB-ში:

```
db.products.update(
   { cat_ID: 6 },
   { $set: { price: price*0.2 } }
   { multi: true }
)
```

➢ „წავშალოთ products ბაზაში ალკოჰოლური სასმელების მონაცემები (ალკ_სასმელის იდენტიფიკატორია cat_ID = 4)“.

- MySQL-ში:

```
DELETE FROM products
WHERE cat_ID = 4
```

- MongoDB-ში:

```
db.products.remove (
   ( cat_ID:  4  }
)
```

და ა.შ.

MongoDB ორგანიზაციებს აძლევს საშუალებას უფრო სწრაფად შექმნას აპლიკაციები, დაამუშავოს განსხვავებულ ტიპთა დიდი მოცულობის ჩანაწერები, აპლიკაციათა მასშტაბირების მართვა განახორციელოს უფრო ეფექტურად. ამავდროულად, მონაცემთა ბაზის დამუშავება (დეველოპმენტი) და განახლება საკმაოდ გამარტივებულია.

მონაცემთა ობიექტ-რელაციური მოდელის (სქემის) ცვლილება, მაგალითად, MySQL-ში, მოითხოვს განახლების შედარებით დიდ დროს, როდესაც MongoDB-ს მონაცემთა მოქნილი მოდელის გამო ასეთი პროცედურები სწრაფად ხორციელდება.

მომდევნო თავში ჩვენ უფრო დეტალურად განვიხილავთ MongoDB ბაზის სისტემაში მუშაობის საკითხებს.

2.2. მონაცემების ასახვა და მათი დამუშავება

რელაციურ მოდელში მონაცემები აისახება (Data Representation,) ერთი სტრუქტურით (ERM – Entity-Relationship Model), აპლიკაციაში კი გამოიყენება სრულიად განსხვავებული სტრუქტურით. ამგვარად, მონაცემთა სტრუქტურა, რომელიც წარმოდგენილია ბაზაში, აბსოლუტურად განსხვავდება ოპერატიულ მეხსიერებაში მისი შესაბამისი სტრუქტურისაგან, რომელსაც შემდგომ იყენებს აპლიკაცია (დანართი).

პროგრამული აპლიკაცია მონაცემთა დამუშავების პროცესში საკმაოდ დიდ დროს და რესურსს უთმობს შემდეგ ოპერაციებს:

- მონაცემთა წაკითხვა რელაციური ბაზის სხვადასხვა ცხრილებიდან (Tables);
- ცხრილების გაერთიანება (Join) აპლიკაციისათვის საჭირო ობიექტში;
- მიღებული ობიექტის დამუშავება (პროექცია, შეზღუდვა და სხვა რელაციური ოპერაციები) და მისი გამოყენება;
- საშედეგო ობიექტის დაშლა და ცვლილებების განთავსება შესაბამის ცხრილებში (ბაზის მთლიანობის შენარჩუნებით).

რაც შეეხება NoSQL ტიპის დოკუმენტურ მონაცემთა ბაზას, როგორიცაა MongoDB, იგი მონაცემებს ინახავს JSON (JavaScript Object Notation) ფორმატში [24]. უფრო ზუსტად, BSON-ში, რომელიც არის JSON -ის ბინარული წარმოდგენა. დოკუმენტების მონაცემები MongoDB-ში აისახება წყვილით: „ველი-მნიშვნელობა“. მაგალითად, ველის დასახელება, მოთავსებული ორმაგ ბრჭყალებში, შემდეგ „ : “ და ბოლოს, მნიშვნელობა ორმაგ ბრჭყალებში. მნიშვნელობა შეიძლება იყოს ისევ დოკუმენტი, მასივები და დოკუმენტების მასივი. თითოეული წყვილი გამოიყოფა მძიმით. დოკუმენტები თავსდება ფიგურულ ფრჩხილებში “{ }“, მასივები კი - კვადრატულ ფრჩხილებში “[ ]“. ქვემოთ მოცემულია დოკუმენტის სტრუქტურის მქონე მონაცემთა ბაზა, რომელიც MongoDB – თვისაა დაწერილი,

```
"category":
 {
    "cat_ID" : "<integer val>",
    "name" : "<string val>"
 }

"products":
   {
     "pr_ID" : "<integer val>",
     "Name" : "<string val>",
     "price"  :  "<number val>",
     "cat_ID" : ["<integer val>"]
   }
```

განვიხილოთ მოთხოვნების ფორმირება ჩვენ ბაზებთან, MySQL და MongoDB შემთხვევაში:

1) „ვიპოვოთ ბაზაში არსებული თითოეული კატეგორიის დასახელების ყველა პროდუქტის დასახელება“.

- MySQL-ში:

```
SELECT MAX(price) FROM products;  // მაქსიმუმი
SELECT MIN(price) FROM products;  // მინიმუმი
```

- MongoDB-ში:

```
db.products.find().sort({price:-1}).limit(1)      //      მაქსიმუმი
db.products.find().sort({price:+1}).limit(1)      //      მინიმუმი
```

2) „ვიპოვოთ პროდუქტის საერთო რაოდენობა კონკრეტული კატეგორიისთვის (cat_ID = 4)“.

- MySQL-ში:

```
SELECT  count ( pr_ID)
FROM    products p, category c
WHERE  c.cat_ID = p.cat_ID and c.cat = 4;
```

- MongoDB-ში:

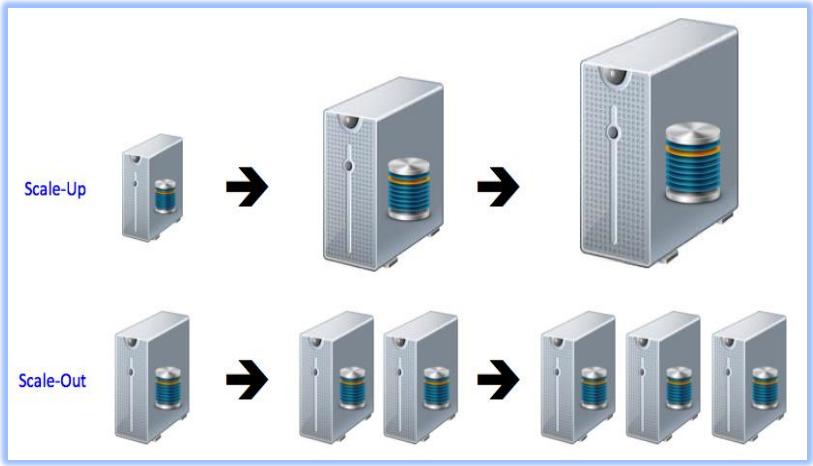
```
db.products.count ( {cat_ID = 4 } )
```

2.3. ვერტიკალური და ჰორიზონტალური სკალები

ინფორმაციული და კომუნიკაციური ტექნოლოგიების გამოყენების სფეროში საგრძნობლად გაიზარდა დიდი მოცულობის მონაცემებისა და პარალელური მოთხოვნების დამუშავების რაოდენობა, რაც ხშირად საჭიროს ხდის არსებული სისტემების არქიტექტურისა და კონფიგურაციების სრულიად შეცვლას.

ტრადიციულ რელაციურ ბაზებში, თუ იზრდება მოთხოვნების დატვირთვა, ვაძლიერებთ სერვერს (vertical scale). გარკვეული ზღვრის შემდეგ სერვერის განვითარება საკმაოდ ძვირი ჯდება, თანაც ნაკლებად ეფექტურია.

სპეციალისტები მივიდნენ იმ დასკვნამდე, რომ მცირე რაოდენობის მძლავრ სერვერებს სჯობს დიდი რაოდენობის იაფფასიანი სერვერები (horizontal scale), რომლებიც იმუშავებს როგორც ერთი კლასტერი (ნახ.2.1).



ნახ.2.1

2.4. მონაცემეთა შენახვის ტიპები

NoSQL ბაზებში მონაცემების შენახვის 4 ძირითადი ტიპი არსებობს:

1. **Key-Value Store** – აქვს დიდი ჰეშ ცხრილი გასაღებებისა და მათი მნიშვნელობებისათვის. მონაცემებზე წვდომა გვაქვს პირველადი გასაღების (Primary Key) გამოყენებით. (მაგალითად, Riak, Amazon S3 [Dynamo], Redis…);

2. **Document-based Store** - ინახავს იარლიყიანი ელემენტე-ბისგან (tagged elements) შემდგარ დოკუმენტებს. ამ ტიპის მონაცემთა ბაზებს JOIN ოპერატორის მხარდაჭერა არ აქვთ. ცხრილების გადაბმის/გაერთიანების ლოგიკა აპლიკაციის მხარეს არის დასაწერი. სამაგიეროდ, ობიექტზე ორიენტირებული აპლიკაციისთვის ძალიან მარტივია მთელი საჭირო ინფორმაციის ერთი დოკუმენტიდან ამოღება. (მაგალითად, MongoDB, CouchDB…);

3. **Column-based Store** - მეხსიერების თითოეული ბლოკი ინახავს მხოლოდ ერთი სვეტის ინფორმაციას, რაც ბევრად ამარტივებს აგრეგატული ფუნქციების (MIN, SUM, AVG, COUNT…) შესრულებას. ამ ტიპის მონაცემთა ბაზები კვლავ ცხრილურ სტრუქტურას იყენებენ, მაგრამ JOIN ოპერატორის მხარდაჭერა არ აქვთ. ცხრილების გადაბმის/გაერთიანების ლოგიკა აპლიკაციის მხარეს არის დასაწერი. (მაგალითად, HBase, Cassandra);

4. **Graph-based**-ქსელური ტიპის მონაცემთა ბაზა, რომელიც იყენებს წიბოებსა და კვანძებს მონაცემების შესანახად და წარმოსადგენად. მაგალითად, Neo4.

დეტალურად განვიხილოთ მონაცემთა შენახვის თითოეული ტიპი.

2.4.1. Key-Value Store

Key-Value ყველაზე მარტივი სტრუქტურაა: გვაქვს უნიკალური პარამეტრი key და შესაბამისი მნიშვნელობა - value (ცხრ.2.3).

ცხრ.2.3

გავრცელებული სისტემები	Tokyo Cabinet/Tyrant, Redis, Voldemort, Oracle BDB, Amazon SimpleDB, Riak
ტიპური გამოყენება	Content caching (განაწილებულ სერვერებზე დიდი ოდენობის ინფორმაციის დამუშავებაზე ორიენტირება), ლოგ–სერვერები...
მოდელი	Key-Value წყვილების კოლექცია
უპირატესობა	სწრაფი ძებნა
სისუსტე	შენახულ მონაცემებს არ აქვს სქემა

სიმარტივისთვის რომ წარმოვიდგინოთ, შესაბამის ლოგიკას რელაციურ ბაზაში ორი სვეტით ავაწყობდით, – Primary key და მასთან დაკავშირებული სვეტი მონაცემებისთვის. განსხვავება ის იქნებოდა, რომ რელაციურ ბაზაში წინასწარ უნდა მიგვეთითებინა მონაცემების სვეტის ტიპი, NoSQL ბაზებში კი ეს მნიშვნელობა შეიძლება იყოს რიცხვი, ტექსტი, სურათი და ამასთან ერთად, არ არის აუცილებელი, რომ ყველა სტრიქონში ერთსა და იმავე ტიპის ინფორმაციას ვინახავდეთ.

მონაცემების ამგვარი შენახვა გვაძლევს იმის საშუალებას, რომ ერთი ცხრილის სხვადასხვა ჩანაწერები კლასტერის სხვადასხვა კვანძებზე მოვათავსოთ. როგორც წესი, ეს გადანაწილება გასაღების მნიშვნელობის მიხედვით ხდება.

Key-Value ტიპის ბაზებში რთულია ატომარობისა და კონსისტენტურობის დაცვა – სანამ ერთი მომხმარებელი ჩანაწერს ანახლებს, სხვა მომხმარებელმა შეიძლება წასაკითხად მიაკითხოს იმავე ჩანაწერს. ამგვარ შემთხვევებში გვაქვს ორი ვარიანტი – მივაწოდოთ მომხმარებელს ჩანაწერის ბოლო განახლებული ვერსია, ან მივაწოდოთ ყველა არსებული ვერსია და თავად მივცეთ

იმის საშუალება, რომ გაარკვიოს, რომელი ვერსიის გამოყენება ურჩევნია.

ამგვარად, აპლიკაციის მხარეს მეტნაკლებად შესაძლებელი ხდება კონსისტენტურობის დაცვაც, მაგრამ თუ დასმული ამოცანისთვის კრიტიკულია ბაზის ტრანზაქციულობა, მაშინ key-value strorage საიმედო გადაწყვეტილება ვერ იქნება.

როგორც წესი, key-value მეთოდი იყენებს ჰეშ ცხრილებს, რომლებშიც ინფორმაცია ლოგიკურ გაერთიანებებად (bucket) არის დაყოფილი. რეალურად, გასაღები არის ჩვენს გასაღებს + Bucket მნიშვნელობების კონკატენაციის ჰეში – hash (Bucket+ Key)

CAP თეორემას თუ მივუბრუნდებით, ცხადი ხდება, რომ key-value მეთოდი იდეალურია Availability და Partition მხარდასაჭერად. მაგრამ საგრძნობლად მოიკოჭლებს Consistency-ის კუთხით.

Key-value storage-ს საკმაოდ კარგი გამოყენებაა სესიის ინფორმაციის შენახვა, როგორც key - სესიის იდენტიფიკატორი და value - სესიასთან დაკავშირებული ინფორმაციის ნაკრები.

ასევე, საინტერესო მაგალითია მომხმარებლის პირადი ინფორმაციის შენახვა, სადაც key გასაღები არის მომხმარებლის უნიკალური იდენტიფიკატორი, value მნიშვნელობა კი მთელი ის ინფორმაცია, რაც ახასიათებს მომხმარებელს.

2.4 ცხრილში Key-value პრინციპით შეტანილია საქართველოს ბანკის ფილიალები დასახლებების მიხედვით.

ცხრ..2.4

Key	Value
“ლანჩხუთი”	{“ჟორდანიას ქუჩა #101"
“ქობულეთი”	{“ნინოშვილის ქუჩა #1"}
“თბილისი”	{“ვეკუას ქუჩა #1თბილისი","პუშკინის ქუჩა #3თბილისი","კოსტავას ქუჩა #24თბილისი","თაბუკაშვილის ქუჩა #38თბილისი"}

უნდა გავითვალისწინოთ, რომ თუ გასაღებად ტექსტურ მნიშვნელობებს ავირჩევთ, დროთა განმავლობაში გაგვიჭირდება უნიკალურობის დაცვა.

Key-value ტიპის ბაზა მონაცემების დასამუშავებლად მომხმარებელს შემდეგ ფუნქციებს სთავაზობს:

- Get(key), აბრუნებს პარამეტრად მიღებული key გასაღების შესაბამის value მნიშვნელობას;
- Put(key, value), აკავშირებს მნიშვნელობას გასაღებთან;
- Multi-get(key1, key2, .., keyN), აბრუნებს მიღებული გასაღებების შესაბამისი მნიშვნელობების მიმდევრობას;
- Delete(key), მონაცემთა ბაზიდან ამოშლის შესაბამის ჩანაწერს.

გასაღები-მნიშვნელობა მოდელს თუ რელაციურ მოდელს შევადარებთ, შემდეგ შესაბამისობებს მივიღებთ:

- Table -> bucket
- Row -> key-value
- Rowid -> key

2.4.2. Document-based Store

მონაცემების წარმოდგენის დოკუმენტზე ორიენტირებული ტიპები, key-value ტიპის მსგავსად, ინფორმაციას ინახავს. განსხვავება იმაშია, რომ დოკუმენტებად შენახულ ინფორმაციას გარკვეული სტრუქტურა და წარმოდგენის განსხვავებული სახე გააჩნია (ცხრ.2.5).

ცხრ.2.5

გავრცელებული სისტემები	CouchDB, MongoDb
ტიპური გამოყენება	ლოგ–სერვერები, ვებ აპლიკაციები (Key-value–ს დახვეწილი ვერსია)
მოდელი	Key-Value წყვილების კოლექციათა კოლექციები (ჩადგმული კოლექციები)

უპირატესობა	ფუნქციონირებს არასრული ინფორმაციის შემთხვევაშიც
ნაკლოვანება	ტრანზაქციების წარმადობა; არ აქვს სტანდარტული სინტაქსი მოთხოვნების ჩამოსაყალიბებლად.

Document-based მონაცემთა ბაზები ინფორმაციას List<Object> სიის სახით ინახავს. ამგვარ სიებში შესაძლებელია ჩაიწეროს მრავალი სხვადასხვა სახის მონაცემი.

მაგალითად, Document-based მონაცემთა ბაზები შეიძლება შევადაროთ ფოლდერს, რომელშიც Ms Word-ის მრავალი ფაილია მოთავსებული, თითოეულ ფაილში კი სხვადასხვა სტრუქტურა და შიგთავსია. თითოეული ფოლდერი ჩვენთვის დოკუმენტების კოლექცია იქნება (collection).

დოკუმენტებზე ორიენტირებულ მოდელს თუ რელაციურ მოდელს შევადარებთ, შემდეგ შესაბამისობებს მივიღებთ:

Table -> collection
Row -> BSON document
Column -> BSON Field
Rowed -> _id
Index -> Index
Join -> ჩადგმული დოკუმენტი (Embedded Document)
Partition -> Shard
Partition Key -> Shard Key

Document-based მოდელის გამოყენების საინტერესო მაგალითია CMS ძრავებსა [62] და ყველა იმ დავალებაში, სადაც გვიწევს, ბრაუზერს ყოველ მიმართვაზე თავიდან დავაგენერირებინოთ კოდი. ცხადია, ბევრად უკეთეს წარმადობას მივიღებთ, თუ უკვე დაგენერირებულ სკრიპტს შევინახავთ და ბრაუზერს არ მოუწევს ყოველ ჯერზე ხელახალი გენერირება.

2.4.3. Column-based Store

Column-based Store პრინციპი შემუშავებულ იქნა მრავალ მანქანაზე გადანაწილებული დიდი რაოდენობის მონაცემების დასამუშავებლად (ცხრ.2.6).

ცხრ.2.6

გავრცელებული სისტემები	Cassandra, HBase, Riak
ტიპური გამოყენება	განაწილებული ფაილური სისტემები Hadoop Distributed File System (HDFS)
მოდელი	სვეტი –> სვეტების გაერთიანება
უპირატესობა	სწრაფი ძიება, განაწილებული გარემოს საუკეთესო მხარდაჭერა
ნაკლოვანება	დაბალი დონის API

2.5. მონაცემთა მთლიანობა

მონაცემთა ბაზების მართვის სისტემები უზრუნველყოფს ინსტრუმენტებს მონაცემთა მთლიანობის (Data Integrity) შესანარჩუნებლად.

საბაზისო წესების მნიშვნელოვან სიმრავლეს, რომელიც უზრუნველყოფს ბაზის მონაცემთა მთლიანობას და არაწინააღმდეგობრივობას, კავშირების (მინიშნებების) დონეზე, მთლიანობის შეზღუდვები ეწოდება (Referential Integrity Constraints) [2,3].

მონაცემთა თანმიმდევრულობისა და მთლიანობის დამცავი პრინციპები რეალიზებულია შემდეგი ჩაშენებული წესებით:

1. ახალი სტრიქონის დამატება მშობელ-ცხრილში ყოველთვის ნებადართულია;
2. ახალი სტრიქონის დამატება შვილ-ცხრილში ნებადართულია მხოლოდ იმ შემთხვევაში თუ აქ შესაბამისი გარე გასაღები არსებობს მშობელ-ცხრილისკენ;

3. მშობელი-ცხრილიდან სტრიქონის ამოშლა ნებადართულია მხოლოდ იმ შემთხვევაში თუ მას არ გააჩნია შვილი სტრიქონები შვილ-ცხრილში;

4. შვილი-ცხრილიდან სტრიქონის ამოშლა ყოველთვის არის ნებადართული;

5. პირველადი გასაღების განახლება მშობელ-ცხრილში ნებადართულია მხოლოდ მაშინ, თუ არ არსებობს შვილი სტრიქონები;

6. შვილი სტრიქონის გარე გასაღების (Foerign key) განახლება ნებადართულია იმ შემთხვევაში, თუ ამ გარე გასაღების შესაბამისი პირველადი გასაღების (Primary key) ახალი მნიშვნელობა არსებობს მშობელ-ცხრილში.

ხშირ შემთხვევებში ბიზნეს-წესების უზრუნველსაყოფად საჭირო ხდება დამატებითი შეზღუდვების შემოტანა [57].

მონაცემთა ბაზების მართვის სისტემები იძლევა კიდევ ერთ შესაძლებლობას მონაცემთა მთლიანობის უზრუნველსაყოფად. ამ შესაძლებლობას ტრანზაქცია (transaction) ეწოდება.

ტრანზაქცია არის მონაცემთა ბაზის ერთმანეთთან დაკავშირებული ცვლილებების დაჯგუფების სპეციალური მექანიზმი, რომელიც გამოიყენება იმ შემთხვევაში, როდესაც უნდა განხორციელდეს ყველა ცვლილება, ან საერთოდ არცერთი ცვლილება არ უნდა შესრულდეს.

მონაცემთა მართვის სისტემა ნებას აძლევს მომხმარებელს (ან პროგრამისტს) განსაზღვროს ტრანზაქციის საზღვრები. მონაცემთა ბაზის ყველა ცვლილება, რომელიც გათვალისწინებულია ტრანზაქციის საზღვრების შიგნით ან უნდა განხორციელდეს წარმატებულად, ან ტრანზაქცია მთლიანად უნდა დაბრუნდეს უკან. როდესაც ხდება ტრანზაქციის უკან დაბრუნება, ყველა სვეტის მნიშვნელობები უბრუნდება იმ მნიშვნელობებს, რომლებიც მათ ჰქონდათ ტრანზაქციის დაწყებამდე.

ტრანზაქციების განხორციელება ეფუძნება მონაცემთა წინასწარ რეგისტრაციას. ტრანზაქციის დასაწყისში მონაცემთა

ბაზებში შესატანი ახალი მნიშვნელობები (ანუ ცვლილებები) და ბაზაში არსებული საწყისი მნიშვნელობები (ანუ ის მნიშვნელობები, რომლებიც უნდა შეიცვალოს ახალი მნიშვნელობებით) ჩაიწერება სარეგისტრაციო ჟურნალში და არა მონაცემთა ბაზაში. როდესაც ტრანზაქცია მთლიანად წარმატებულად შესრულდება, მაშინ ის ფიქსირდება (ანუ მონაცემთა მართვის სისტემა აფიქსირებს წარმატებულ ტრანზაქციას). ამ ფაზაში ის ცვლილებები, რომლებიც იყო ჩაწერილი სარეგისტრაციო ჟურნალში, გადადის მონაცემთა ბაზაში და შესაბამისი ცვლილებები ხილვადი ხდება სხვა მომხმარებლებისათვის. მეორეს მხრივ, თუ ტრანზაქციის რომელიმე ნაწილი ვერ განხორციელდა (ანუ მისი განხორციელება ჩავარდა) ნებისმიერი მიზეზის გამო, მაშინ ცვლილებები ბრუნდება უკან, ანუ არცერთი ცვლილება, ჩაწერილი სარეგისტრაციო ჟურნალში, არ გადადის მონაცემთა ბაზაში.

წინასწარი რეგისტრაცია ასევე სასარგებლოა მონაცემთა ბაზის აღდგენისას ავარიული სიტუაციის შემდეგ. სარეგისტრაციო ჟურნალი (log) მოიცავს მონაცემთა ბაზაში განხორციელებულ ყველა ცვლილებას, იმ ინფორმაციის ჩათვლით დაფიქსირდა თუ უკან დაბრუნდა თითოეული ტრანზაქცია. მონაცემთა ბაზის აღსადგენად ადმინისტრატორს შეუძლია აღადგინოს მონაცემთა ბაზის წინამორბედი *სარეზერვო ასლი* და გაიმეოროს დაფიქსირებული ტრანზაქციები. ამას ეწოდება მონაცემთა ბაზის აღდგენა დაფიქსირებული ტრანზაქციების გამეორებით (roll forward recovery).

ზოგიერთი მონაცემთა ბაზის მართვის სისტემა იყენებს წინასწარ რეგისტრაციას (ცვლილებების წინსწრებით რეგისტრაციას), მაგრამ ამასთან ერთად ახორციელებს მონაცემთა ბაზის ფაქტობრივ ცვლილებას ტრანზაქციის ფორმალურ დაფიქსირებამდე. ასეთ სისტემებში ავარიული სიტუაციის შემდეგ აღდგენა შესაძლებელია განხორციელდეს მონაცემთა ბაზების მართვის

სისტემის გადატვირთვით და სარეგისტრაციო ჟურნალში არსებული ყველა იმ ტრანზაქციების გაუქმებით, რომლებიც არ დაფიქსირდა. ასეთ მიდგომას დაუფიქსირებელი ტრანზაქციების გაუქმებით აღდგენას უწოდებენ (rollback recovery).

2.6. ტრანზაქციის იზოლირების დონეები

როდესაც მონაცემთა ბაზას ერთდროულად მიმართავს მრავალი მომხმარებელი, არსებობს იმის შესაძლებლობა, რომ ერთი პიროვნების მიერ შესრულებული ცვლილებები ზემოქმედებას მოახდენს სხვა პიროვნების მუშაობაზე [63]. მაგალითად წარმოვიდგინოთ, რომ ორი პიროვნება ერთდროულად იმყოფება ავიაბილეთების შეკვეთათა ცხელი ხაზის სისტემაში (on-line flight reservation system), ორივე ხედავს, რომ ფანჯარასთან მდებარე ადგილი მე-18 რიგში ჯერ კიდევ თავისუფალია, და ორივე უკვეთავს ამ ადგილს თითქმის ერთდროულად. შესაბამისი კონტროლის გარეშე ორივე პიროვნება დარწმუნებული იქნება, რომ მათი ადგილი შეკვეთილია, მაგრამ ერთერთი მათგანი იმედგაცრუებული დარჩება. მოყვანილი მაგალითი წარმოადგენს დამთხვევათა პრობლემების ერთ-ერთ სახეს, რომელსაც დაკარგულ განახლებათა პრობლემა ეწოდება (the lost update problem).

გარდა აღნიშნული პრობლემისა არსებობს სხვა პოტენციალური პრობლემებიც. მაგალითად, შეცდომითი წაკითხვა (მცდარი მონაცემების წაკითხვა - **dirty reads**) ხდება მაშინ, როდესაც ერთი ტრანზაქცია კითხულობს მეორე დასაფიქსირებელი ტრანზაქციით შეცვლილ მონაცემებს და ეს მეორე ტრანზაქცია მოგვიანებით უკან ბრუნდება ყველა შესაბამისი ცვლილებების გაუქმებით.

მაგალითად, სესია1–მა დაიწყო ტრანზაქცია და შეცვალა მონაცემები. სესია2–მა წაიკითხა უკვე შეცვლილი მონაცემები, რომელიც სესია1–ს ჯერ დამახსოვრებული (commited) არ ჰქონდა. რეალურად, სესია2–მა წაიკითხა ინფორმაცია, რომელიც რაიმე

პრობლემის შემთხვევაში შეიძლება უკან დაბრუნდეს (roll back). ამგვარ შემთხვევაში სესია2–ს ექნება დამახინჯებული მონაცემები (dirty data).

განსხვავებული სახის პრობლემაა არაგანმეორებადი წაკითხვა (**nonrepeatable read**). ეს პრობლემა წამოიჭრება მაშინ, როცა ტრანზაქცია რამდენიმეჯერ კითხულობს ერთსა და იმავე მონაცემებს. პირველი ტრანზაქციის შესრულების დროს თუ მეორე ტრანზაქციამ მონაცემები შეცვალა, მაშინ პირველი ტრანზაქცია მონაცემების განმეორებით წაკითხვისას სხვა მნიშვნელობებს მიიღებს.

მაგალითად, სესია1–მა დაიწყო ტრანზაქცია და მიიღო გარკვეული სტრიქონი. სესია2–მა განაახლა ზუსტად იგივე სტრიქონი. სესია1–მა ხელმეორედ გაუშვა წინა ტრანზაქცია, მაგრამ შედეგი უკვე განსხვავებული მიიღო.

მსგავს პრობლემას წარმოადგენს მოჩვენებითი წაკითხვა (phantom read). ამ პრობლემას ადგილი აქვს მაშინ, როცა პირველი ტრანზაქცია ცხრილიდან ირჩევს სტრიქონებს, მეორე ტრანზაქცია კი ამავე ცხრილში სვავს სტრიქონებს პირველი ტრანზაქციის მუშაობის დამთავრებამდე. შედეგად, პირველი ტრანზაქციის მიერ წასაკითხი სტრიქონები არაკორექტული იქნება. თუ თავისი მუშაობის დროს ერთი ტრანზაქცია კითხულობს ჩანაწერების რომელიმე სიმრავლეს ორჯერ, მაშინ მან შესაძლებელია წაიკითხოს ახალი ჩანაწერები მეორე წაკითხვისას. ეს შესაძლებელია მოხდეს იმ შემთხვევაში, თუ პარალელურად მონაცემთა ბაზაში სხვა ტრანზაქციას ახალი ჩანაწერები შეაქვს.

ამ ტიპის ყველა პრობლემის გადაჭრა ხორციელდება ჩაკეტვის მექანიზმებით (locking mechanisms), რომლებიც უზრუნველყოფს მომხმარებელთა და ტრანზაქციების ერთმანეთისაგან იზოლირებას.

ადრე პროგრამისტებს საჭირო დაცვის განსახორციელებლად ჩაკეტვების მართვის საკუთარი სისტემების შემოტანა სჭირდებოდათ, მაგრამ დღეს ჩაკეტვების მართვას მთლიანად მონაცემთა ბაზების მართვის სიტემა ახორციელებს.

დაკარგული განახლებების პრობლემას თანამედროვე მონაცემთა ბაზების მართვის სიტემები წაკითხვისა და ჩაწერის ჩაკეტვების მართვით ახორციელებს. სხვა შესაძლო პრობლემების გადასაწყვეტად ჩვენ უბრალოდ განვსაზღვრავთ საჭირო იზოლაციის დონეს იმ ოთხი სტანდარტული დონიდან, რომლებსაც ადგენს 1992 წლის SQL სტანდარტი. დაცვის ოთხი სხვადასხვა სტანდარტის არსებობა დაკავშირებულია იმ ფაქტთან, რომ რაც უფრო ძლიერია დაცვა (მაღალია იზოლაციის დონე) მით უფრო მცირეა მოქმედების სისწრაფე [63].

ტრანზაქციათა იზოლაციის დონეებია:

- Read uncommitted დონე არ უზრუნველყოფს დაცვას იმ დამთხვევათა პრობლემებისაგან, რომლებიც განვიხილეთ, მაგრამ უზრუნველყოფს მოქმედების მაქსიმალურ სისწრაფეს, რადგან იზოლაციის ამ დონეში არ ხდება მოთხოვნილი მონაცემების ბლოკირება (lock). ცხადია, ამ დონის გამოყენებისას მზად უნდა ვიყოთ, რომ თუ ერთი პროგრამიდან შეტანილი ცვლილებები უკან დაბრუნდება, update-სა და rollback-ს შორის შუალედში ყველა სხვა მოთხოვნას dirty data-ს ანუ დამახინჯებულ მონაცემებს გავატანთ. ამ დონეს გამოვიყენებთ ისეთი მოთხოვნებისთვის, რომელთა სისწორეც არ არის სასიცოცხლოდ აუცილებელი, ან თუ ვიცით, რომ სანამ ჩვენ ვკითხულობთ, მესამე მხარე ცვლილებას არ შეიტანს;
- Read committed დონე გარანტირებულად არიდებს თავს შეცდომითი წაკითხვების პრობლემას, რადგანაც იზოლაციის ამ დონეზე ნებადართულია მხოლოდ დაფიქსირებული ტრანზაქციებით შეცვლილი მონაცემების წაკითხვა;
- Repeatable read დონე იცილებს ასევე არაგანმეორებადი წაკითხვების პრობლემასაც;

- Serializable დონე უზრუნველყოფს ერთდროული ტრანზაქციების სრულ განცალკევებას, რაც ხორციელდება ტრანზაქციაში მონაწილე ყველა სტრიქონის ჩაკეტვით. ფანტომების წარმოქმნის პრობლემა წყდება გასაღების დიაპაზონის დაბლოკვით. იზოლირების ამ დონის დაყენების შემთხვევაში მცირდება პარალელიზმის ხარისხი და იზრდება სისტემაზე დატვირთვა.

ჩვეულებრივ ყველაზე უფრო ხშირად გამოიყენებენ read committed იზოლაციის დონეს [64].

ცხრ.2.7

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
READ UNCOMMITTED	შესაძლებელია	შესაძლებელია	შესაძლებელია
READ COMMITTED	--	შესაძლებელია	შესაძლებელია
REPEATABLE READ	--	--	შესაძლებელია
SERIALIZABLE	--	--	--

იზოლაციის ეს დონეები შემუშავებულია 1992 წლის SQL სტანდარტით, მაგრამ რეალურად, სხვადასხვა მონაცემთა ბაზებში ტრანზაქციის იზოლირების დონეებს სხვადასხვა სახე აქვს. მაგალითად, Oracle მონაცემების წაკითხვისას lock-ს საერთოდ არ იყენებს და dirty data–ს პრობლემა latch-ებით აქვს გადაწყვეტილი.

Lock-ის შემთხვევაში რიგში ვდგებით, რომ მივიღოთ მონაცემებზე წვდომა, latch–ის დროს კი პერიოდულად ვაკითხავთ მონაცემებს და ვამოწმებთ, თუ არის თავისუფალი.

Oracle-ის ტრანზაქციის იზოლირების დონეებია [64]:

- Read Committed: ტრანზაქციის იზოლაციის ეს დონე არის default (დუმილით) მნიშვნელობა;
- Serializable: ტრანზაქციის იზოლირების ამ დონეზე არ გვაქვს დამახინჯებული მონაცემები (dirty data);

- Read Only: მხოლოდ ის მონაცემები არის ხელმისაწვდომი, რომლებიც ტრანზაქციის დაწყების მომენტში უკვე დასრულებული (commited) იყო.

DB2-ის ტრანზაქციის იზოლირების დონეებია [65]:

- Uncommitted Read: lock-ები არ გვაქვს, სამაგიეროდ მზად უნდა ვიყოთ დამახინჯებული ინფორმაციისთვის;
- Cursor Stability: გვაქვს მცირე ხანგრძლივობის ბევრი lock. ეს დონე არის DB2-სთვის default მნიშვნელობა. კითხვის დროს, ამ შემთხვევაში, ბლოკი ედება მხოლოდ მიმდინარე სტრიქონს. თუ B და C აპლიკაციები ერთსა და იმავე სტრიქონს კითხულობს, გამოიყენება shared lock, რაც იმის საშუალებას იძლევა, რომ ერთი და იმავე მონაცემის წასაღებად მისული აპლიკაციები ერთმანეთს არ დაელოდოს;
- Read Stability: გვაქვს დიდი ხანგრძლივობის ცოტა lock. წინა დონისგან განსხვავებით, RS იზოლაციის დონე ერთდროულად ბლოკავს ყველა მოთხოვნილ სტრიქონს;
- Repeatable Read: გვაქვს დიდი ხანგრძლივობის მქონე ბევრი lock. RR ყველაზე მკაცრი დონეა და მოთხოვნილ სტრიქონებთან ერთად ბლოკავს შედეგთან დაკავშირებულ ყველა ჩანაწერსაც.

Oracle, SQL Server და DB2-ში იზოლაციის დონე შეიძლება მიეთითოს კონკრეტული სესიის ან ტრანზაქციისთვის (DB2-ს დამატებით აქვს აპლიკაციის დონეც).

NoSQL სისტემებისთვის, კონკრეტულად კი MongoDB-სთვის ტრანზაქციების იზოლირების მსგავსი მეთოდები არ გვაქვს. ნაშრომის შემდგომ თავებში დეტალურად განვიხილავთ იმ მეთოდებს, რომლებსაც NoSQL სისტემები იყენებს სტანდარტული პრობლემების ასაცილებლად.

2.7. ACID პრინციპები

1970-იანი წლების ბოლოს ACID (Atomicity, Consistency, Isolation, Durability) ტესტის სახელით ჩამოყალიბდა ის კრიტერიუმები, რაც უმთავრესია ტრანზაქციის საიმედოობისთვის [58]:

- Atomicity (ატომარობა) – ტრანზაქციის პროცესი ან მთლიანად სრულდება (უწყვეტად) ან საერთოდ არ სრულდება. თუ ტრანზაქციის პროცესი ნაწილობრივ შესრულდა, მაშინ ატომარობა დარღვეულია;
- Consistency (მთლიანობა) – ტრანზაქციის პროცესი სრულდება მაშინ, როცა ბაზა მთლიანია. ტრანზაქციის პროცესის დასრულებისას ბაზა ისევ მთლიანობის მდგომარეობაში რჩება (მთლიანობა ნიშნავს, რომ თითოეული სტრიქონი და მნიშვნელობა უნდა შეესაბამებოდეს რეალურ სიტუაციას და უნდა სრულდებოდეს ყველა შეზღუდვა. მაგალითად, თუ წერია შეკვეთები და არ წერია საქონელი, მთლიანობის პრინციპი დარღვეულია);
- Isolation (იზოლირება) – ტრანზაქციის თითოეული პროცესი უნდა იყოს იზოლირებული, რაც ნიშნავს, რომ ყველა ტრანზაქციის პროცესი უნდა ხორციელდებოდეს ერთმანეთისაგან დამოუკიდებლად. ახალმა პროცესმა არ უნდა შეუშალოს ხელი უკვე დაწყებული ტრანზაქციის პროცესის დასრულებას. ეს პრინციპი განსაკუთრებით მნიშვნელოვანია, როცა ბაზასთან მუშაობს რამდენიმე მომხმარებელი;
- Durability (მდგრადობა) – გულისხმობს ტრანზაქციის პროცესის შესრულებას სისტემური შეფერხებებისაგან დამოუკიდებლად. მონაცემთა ბაზების მართვის სისტემა ისე უნდა იყოს დაპროექტებული, ახალი ტრანზაქციის შესრულებისას შეფერხებების არსებობის შემთხვევაში, შესაძლებელი გახდეს ბოლოს, სრულად შესრულებული ტრანზაქციის პროცესის შემდეგ არსებული მდგომარეობის აღდგენა.

რელაციურ ბაზაში ნებისმიერი ოპერაცია ხორციელდება პრინციპით: *ყველაფერი ან არაფერი.*

არარელაციურ ბაზებში ACID გარანტია არ გვაქვს, მაგრამ შეგვიძლია სხვადასხვა ლოგიკით მივაღწიოთ ტრანზაქციების საიმედოობას. ამგვარი გვერდის ავლის ერთ–ერთი საუკეთესო მაგალითი ორფაზიანი განხორციელებაა (Two Phase Commits).

მართალია ACID გარანტია არ გვაქვს, მაგრამ სამაგიეროდ გვაქვს საშუალება, ავირჩიოთ CAP თვისებები, მაგალითად, 100%–იანი წვდომა.

2.8 CAP სამკუთხედის თეორემა

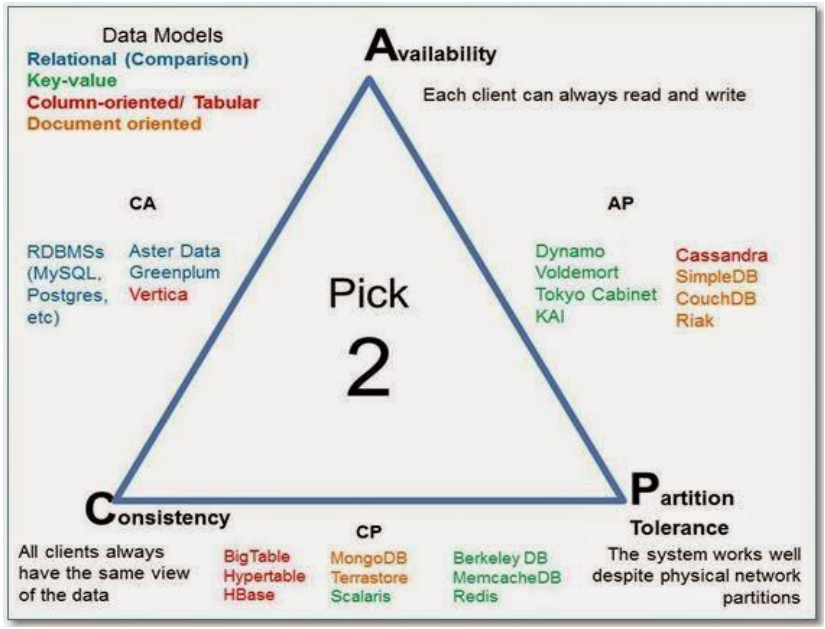
მონაცემთა ბაზებთან მუშაობისას შესაძლოა გვქონდეს "network partition"-ები, რომლებიც გულისხმობს კლასტერში კვანძებს (node) შორის კომუნიკაციის დაკარგვას და შესაბამისად ინფორმაციის არასინქრონულობას [6,7] (ნახ.2.2).

CAP თეორემა (Consistency, Availability, Partition_tolerance) ან ბრიუერის (*Brewer Eric*) თეორემა არის ევრისტიკული მტკიცება იმის შესახებ, რომ განაწილებული გამოთვლების ნებისმიერ რეალიზაციაში შესაძლებელია სამი (C,A,P) თვისებიდან მხოლოდ ორის უზრუნველყოფა (ნახ.2.2) [8].

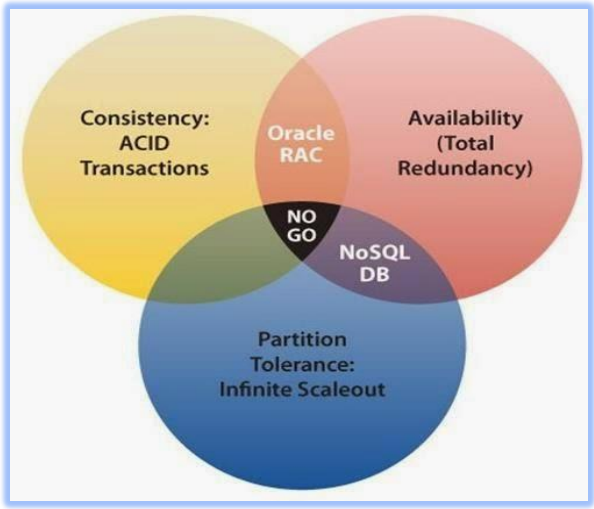
Consistency (მთლიანობა) – განაწილებული კომპიუტერული სისტემის ყველა კვანძში (node – კლასტერში) მონაცემები შეთანხმებულია (არაწინააღმდეგობრივია დროის მოცემულ მომენტში);

Availability (წვდომადობა) – ნებისმიერი მოთხოვნა განაწილებულ სისტემასთან სრულდება კორექტული პასუხით (თუ რომელიმე კვანძი გათიშულია, მას ცვლის სხვა);

Partition Tolerance (განაწილების მდგრადობა) – კლასტერი განაგრძობს ფუნქციონალობას მაშინაც კი როდესაც კვანძებს შორის არის კომუნიკაციის პრობლემა.


Data Models
Relational (Comparison)
Key-value
Column-oriented/ Tabular
Document oriented
Availability
Each client can always read and write
CA
AP
RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica
Pick
2
Dynamo
Voldemort
Tokyo Cabinet
KAI
Cassandra
SimpleDB
CouchDB
Riak
Consistency
Partition Tolerance
CP
All clients always have the same view of the data
BigTable
Hypertable
HBase
MongoDB
Terrastore
Scalaris
Berkeley DB
MemcacheDB
Redis
The system works well despite physical network partitions

ნახ.2.2


Consistency: ACID Transactions
Oracle RAC
Availability (Total Redundancy)
NO GO
NoSQL DB
Partition Tolerance: Infinite Scaleout

ნახ.2.3

მოვიყვანოთ დეტალური მაგალითები CAP სამკუთხედზე (ნახ.2.3). როდესაც კომუნიკაციის პრობლემაა, არჩევანი გვაქვს:

1. ან დავუშვათ, რომ არასინქრონირებული მონაცემები იყოს ბაზაში (დავივიწყოთ Consistency);

2. ჩავთვალოთ რომ კლასტერი დაზიანდა, ანუ დავივიწყოთ Availability.

CAP თეორემის თვალსაზრისით, ამ სამიდან ერთდროულად მხოლოდ რომელიმე ორი თვისება შეიძლება განხორციელდეს. განაწილებული სისტემები იყოფა სამ კლასად:

CA (მთლიანობა და წვდომადობა) – ერთი და იგივე მონაცემებია ყველა კლასტერში. თუ კომუნიკაციის რაიმე პრობლემა დაფიქსირდა კვანძებს შორის (partition), მონაცემები იქნება არასინქრონირებული მანამ, სანამ პრობლემა არ გადაიჭრება, ანუ კლასტერებში განსხვავებული მონაცემები იარსებებს;

CP (მთლიანობა და განაწილების მდგრადობა) – მონაცემები არაწინააღმდეგობრივია კლასტერებში. მაგალითად, დაფიქსირდა კომუნიკაციის პრობლემა. ამ დროს არასინქრონული რომ არ იყოს მონაცემები, შესაბამისი კვანძი გახდება გაუქმებული (ანუ წაკითხვა/ჩაწერა იქნება გაჩერებული შესაბამის node-ში). აქ თავს ვიცავთ იმისგან, რომ ბაზაში სხვადასხვა მონაცემები არ გვეწეროს;

AP (წვდომადობა და განაწილების მდგრადობა) – არაა გარანტირებული მონაცემთა მთლიანობა. მაგალითად, დაფიქსირდა პრობლემა კლასტერებს შორის. აქ დაზიანებული node-ები არაა გაუქმებული. ისინი დამოუკიდებლად განაგრძობს მუშაობას. როდესაც კომუნიკაციის პრობლემა აღდგება, დაიწყება მონაცემების სინქრონიზაცია, თუმცა გარანტია არ გვაქვს რომ ყველა მონაცემი იგივე იქნება კვანძებს შორის.

კალიფორნიის უნივერსიტეტის პროფესორის ერიკ ბრიუერის CAP თეორემა, რომელიც მან 2000 წელს შემოგვთავაზა, პოპულარული გახდა განაწილებული კომპიუტერული სისტემების

სფეროში [8]. NoSQL ბაზების კონცეფცია, რომლითაც იქმნება განაწილებული არატრანზაქციული მონაცემთა ბაზების სისტემები, ხშირად იყენებს სწორედ ამ პრინციპს მონაცემთა მთლიანობის (consistency) დარღვევის გარდაუვალობის დასაბუთების მიზნით. უმრავლესი NoSQL სისტემები არ იძლევა მონაცემთა ბაზის მთლიანობის გარანტიას. ამიტომაც AP-სისტემების აგების ამოცანა მდგომარეობს მონაცემთა მთლიანობის გარკვეული, პრაქტიკულად მიზანშეწონილი დონის უზრუნველყოფის განხორციელებაში. ასეთი AP-სისტემები ლიტერატურაში მოიხსენიება ტერმინით „სუსტი მთლიანობის“ (weak consistent) სისტემები [2].

2.8.1 მთლიანობის მოდელები CAP თეორემაში

როდესაც ვლაპარაკობთ მთლიანობაზე, განვიხილავთ სცენარს, როდესაც სხვადასხვა კვანძებზე გვაქვს ერთი და იგივე მონაცემების დამოუკიდებელი კოპიოები. კონფლიქტური სიტუაცია დგება, როდესაც კონკრეტული კვანძის მომხმარებელი სერვერს სთხოვს გარკვეული ინფორმაციის ცვლილებას. ანუ, სანამ ცვლილება სხვა კვანძებზეც აისახება, დროის გარკვეულ მონაკვეთში ახალი მომხმარებლები სხვადასხვა კვანძებიდან სხვადასხვა მონაცემებს (მონაცემების სხვადასხვა ვერსიებს) მიიღებენ.

სწორედ აქ შემოდის განაწილებულ გარემოში მთლიანობის სხვადასხვა მოდელები:

- ძლიერი მთლიანობა (Strong Consistency) : W + R > N
- სუსტი მთლიანობა (Weak / Eventual Consistency) : W + R <= N
- კითხვაზე ოპტიმიზირებული : R = 1, W = N
- ჩაწერაზე ოპტიმიზირებული : W = 1, R = N,

სადაც R აღნიშნავს იმ კვანძების მაქსიმალურ რაოდენობას, რომლებიც პასუხისმგებლები არიან წაკითხვის მოთხოვნების დამუშავებაზე.

W აღნიშნავს იმ კვანძების მაქსიმალურ რაოდენობას, რომლებიც პასუხისმგებლები არიან ჩაწერის მოთხოვნების დამუშავებაზე.

N კი იმ კვანძების რაოდენობაა, სადაც ხდება მონაცემების დუბლირება.

მაგალითად, განვიხილოთ კლასტერი 5 კვანძით, R, W და N მნიშვნელობების ცვლილებით შეგვიძლია ვაკონტროლოთ მონაცემთა მთლიანობის დონეები.

თუ კლასტერს ავაწყობთ R = 5 და N = 5 მნიშვნელობებით, მივაღწევთ მთლიანობის მაქსიმალურ დონეს, - ყველა კვანძზე გვექნება დუბლირებული მონაცემები, სამაგიეროდ, ჩვენი სისტემა აღარ იქნება საიმედო განაწილებული გარემოს კუთხით, რადგან არცერთი ჩაწერა არ ჩაითვლება წარმატებულად რომელიმე კვანძის ჩავარდნის შემთხვევაში.

შეგვიძლია იგივე კლასტერი ავაწყოთ R = 1 და N = 1 მნიშვნელობებით. ამ შემთხვევაში მონაცემების ცვლილების ოპერაცია წარმატებულად ჩაითვლება მინიმუმ ერთ კვანძზე წარმატებული ჩაწერის შემდეგ. სამაგიეროდ, ზოგიერთ კვანძს არ ექნება ბოლოს შეცვლილი მონაცემების მიმდინარე ვერსია.

მთლიანობის მსგავსად, შეგვიძლია წვდომადობასაც ვაკონტროლოთ წვდომის მაქსიმალური დასაშვები დროის ცვლილებით.

2.8.2. AP სისტემები - ძლიერი მთლიანობის მოდელი

როგორც ზემოთ აღვნიშნეთ, ძლიერ მთლიანობას ადგილი აქვს მაშინ, როდესაც ინფორმაციის დაცვის მიზნით, მონაცემებს ყველა კვანძზე პარალელურ რეჟიმში ვანახლებთ. ამით ვიღებთ იმის გარანტიას, რომ დროის ნებისმიერ მომენტში შევძლებთ მონაცემების მიმდინარე ვერსიის წაკითხვას, იმის მიუხედავად, თუ რომელ კვანძზე ჩაიწერა მონაცემი თავდაპირველად. ამ გარანტიას ძლიერი მთლიანობის მოდელი იმის ხარჯზე გვაძლევს, რომ სანამ

რომელიმე მომხმარებლის მიერ შეტანილი ცვლილება ყველა კლიენტისთვის ხილვადი გახდება, ახალმა ცვლილებამ ყველა კვანძზე უნდა გაიაროს ვალიდაცია. ცვლილების შემტან კლიენტს კი მოთხოვნაზე პასუხი მხოლოდ მას შემდეგ დაუბრუნდება, რაც ეს ცვლილება ყველა კვანძზე აისახება.

ძლიერი მთლიანობის საჭიროება ნათლად ჩანს საბანკო სისტემებში. როდესაც გადაგვაქვს თანხა ერთი ანგარიშიდან მეორეზე, ტრანზაქციის დაწყებამდეც და დასრულების შემდეგაც ჯამური ბალანსი უცვლელი უნდა დარჩეს [2]. ასეთივე მოთხოვნები ექნება ყველა იმ სისტემას, სადაც ინფორმაციასთან წვდომა რეალურ დროში გვჭირდება.

2.8.3. AP სისტემები - სუსტი მთლიანობის მოდელი

ძლიერი მთლიანობა (strong consistency) და მაღალი წვდომადობა (high availability) სისტემის სასურველი თვისებებია, მაგრამ CAP თეორემა გვიჩვენებს, რომ კვანძებს შორის არასაიმედო კავშირის შემთხვევაში ერთდროულად ორივეს მიღწევა შეუძლებელია და მიგვანიშნებს „სუსტი მთლიანობის“ მოდელისკენ, რაც ცხოვრებისეულად უფრო რეალურია.

სუსტი მთლიანობის მოდელში, არაა აუცილებელი, რომ მონაცემები ყველა კვანძზე რეალურ დროში განახლდეს, მაგრამ დარწმუნებით შეგვიძლია ვთქვათ, რომ საბოლოოდ ინფორმაცია ყველა კვანძზე განახლდება. წაკითხვისას კი კვანძი დაგვიბრუნებს მონაცემის იმ ვერსიას, რომელსაც პირველად იპოვის.[2]

სუსტი მთლიანობის გამოყენება გარკვეული კლასის ამოცანებში იმდენად ზრდის წარმადობასა და ჰორიზონტალურ სკალას, რომ ეს მოდელი შესაბამისი პრობლემების გადასაწყვეტად შეუცვლელია. ამგვარი ამოცანებია გამომწერების სია Twitter-ში, მეგობრების სია Facebook-ში და სამეცნიერო პროგრამის log ჩანაწერები.

სუსტი მთლიანობის მოდელი წარმატებული იქნება ყველა იმ კლასის ამოცანაში, სადაც ზუსტი, რეალური დროის პასუხის დაბრუნებაზე უფრო მნიშვნელოვანი უბრალოდ პასუხის დაბრუნებაა.

თუ მონაცემთა ბაზის მოდელი და სქემა სწორად არის შერჩეული, მაშინ, მიუხედავად მონაცემთა ბაზის ზომებისა და მასში არსებული ჩანაწერების რაოდენობისა, მონაცემებთან წვდომა თითქმის მყისიერია. მილიონობით ჩანაწერში ჩვენთვის საინტერესო ჩანაწერის ზუსტი და მყისიერი მოძებნა ამაოგნებელ შთაბეჭდილებას ახდენს.

და ისევ, ისეთ ამოცანებში, სადაც შეხება გვაქვს ფინანსურ ტრანზაქციებსა და რეალურ დროში დასამუშავებელ მონაცემებთან, მოძველებულმა ინფორმაციამ შესაძლოა წარმოშვას მნიშვნელოვანი რისკები. ამიტომ, ამ კლასის ამოცანებში სუსტი მთლიანობის მოდელის გამოყენება არ არის რეკომენდირებული.

2.9. განაწილებული გარემო – replication

რეპლიკაცია არის პროცესი, რომლის დროსაც მონაცემები სინქრონიზირდება ერთი, მთავარი სერვერიდან რამდენიმე სათადარიგო სერვერზე. რეპლიკაციის დროს მთავარ (primary) სერვერზე ჩაწერისა და კითხვის ოპერაციების განხორციელება შეგვიძლია, სათადარიგო სერვერებიდან კი მხოლოდ ვკითხულობთ. რეპლიკაციას რამდენიმე ძირითადი დადებითი მხარე გააჩნია:

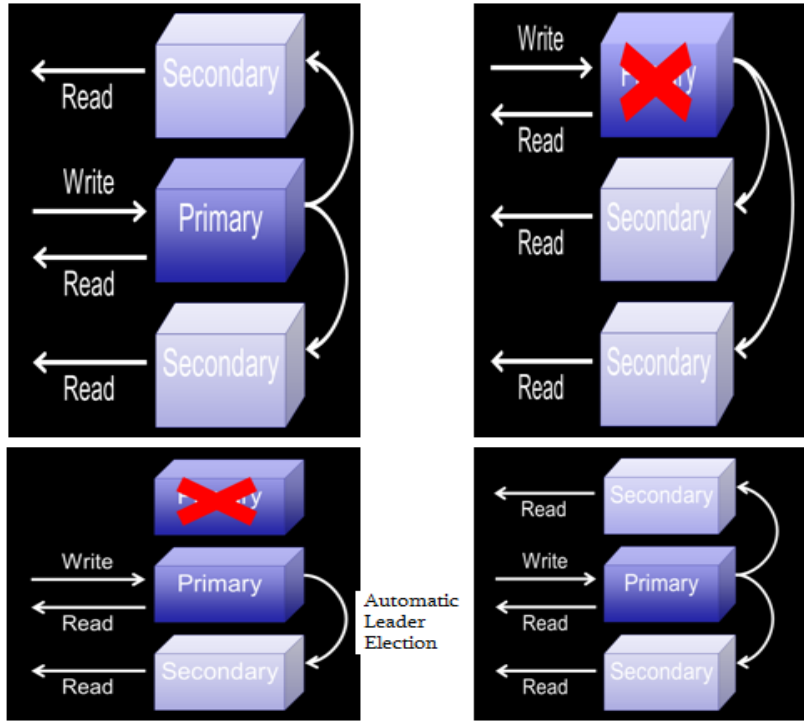
- მონაცემების კითხვის გაუმჯობესება – რამდენ კოპიო სერვერსაც დავაკონფიგურირებთ, იმდენი დამოუკიდებელი წყაროდან შეგვიძლია ინფორმაციის პარალელურ რეჟიმში წაკითხვა;
- Disaster Recovery – პრობლემის ან გეგმიური სამუშაოების დროს შეგვიძლია რომელიმე სათადარიგო (standby) სერვერი ვაქციოთ ძირითად (primary) სერვერად;
- აღარ არის საჭირო backup-ებისა და ინდექსების რეორგანიზაციის გამო მონაცემთა ბაზის გაჩერება ან შენელება;
- რეპლიკა სეტები აპლიკაციის დონეზე არ ჩანან.

მთავარ სერვერთან კავშირის დაკარგვის ან გათიშვის შემთხვევაში სათადარიგო სერვერები აწყობს არჩევნებს და ირჩევს ახალ primary სერვერს.

რეპლიკა სეტების არჩევნებში მონაწილეობს წინასწარ განსაზღვრული სერვერები – არბიტრები. არბიტრი არ შეიცავს მომხმარებელთა ინფორმაციას, მისი ერთადერთი დანიშნულება არჩევნებში ხმის მიცემაა.

თუ Replica set-ში გვაქვს ლუწი რაოდენობის კვანძი, მაშინ არბიტრის დამატებით ვიღებთ კენტი რაოდენობის წევრებს, რითაც ვაღწევთ ხმათა უმრავლესობას არჩევნებში.

ყველაზე ხშირად რეპლიკაციას სამი რეპლიკა სეტისგან აწყობენ ხოლმე (ნახ.2.4).



ნახ.2.4

რეპლიკაციის აწყობა დიდ სირთულეებთან არ არის დაკავშირებული, მთავარ სერვერზე ვქმნით რეპლიკაციის გარემოს:

mongod --port "PORT" --dbpath "ბაზის მისამართი" --replSet "რეპლიკა სეტის ინსტანსის სახელი";

ამის შემდეგ ვუკავშირდებით უკვე შექმნილ გარემოს და ვუშვებთ რეპლიკაციის პროცესს : rs.initiate().

რეპლიკა სეტის კონფიგურაციის შემოწმება შეგვიძლია ბრძანებებით rs.conf() და rs.status().

ახალი წევრების დასამატებლად ვუშვებთ

rs.add(HOST_NAME:PORT)

სტრუქტურის ბრძანებას.

თუ არაფერს არ შევცვლით, კითხვისათვის კლიენტები ძირითად სერვერს მიმართავენ, მაგრამ აპლიკაციიდან შეგვიძლია ავირჩიოთ „Read Preference“ და კითხვა განვახორციელოთ მეორადიდან.

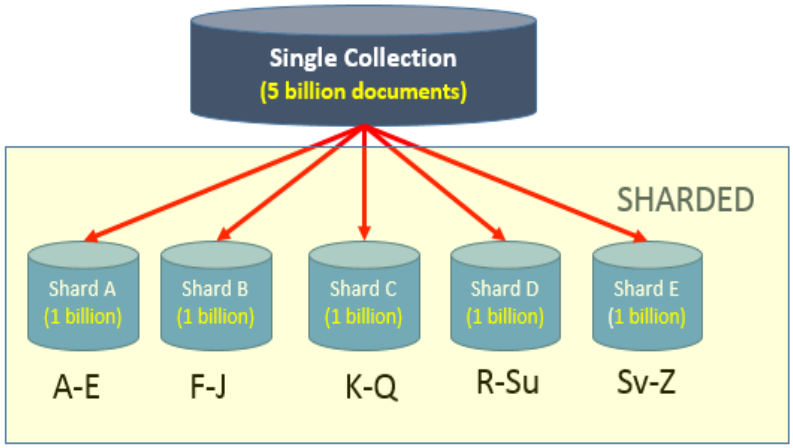
Read Preference პარამეტრი საგრძნობლად აუმჯობესებს წარმადობას, მაგრამ ცხადია, პრობლემა ხდება კონზისტენტურობა,– სანამ ძირითად(primary) სერვერზე განხორციელებული ცვლილება სათადარიგო(secondary) ბაზამდე მიაღწევს, კლიენტები ძველ ინფორმაციას კითხულობენ. ეს პრობლემა გადაჭრილია majority პარამეტრის შემოღებით. ამ პარამეტრით ვუთუოთებთ, მინიმუმ რამდენ კვანძზე უნდა ჩაიწეროს ინფორმაცია, რომ ტრანზაქცია წარმატებულად ჩაითვალოს. ძირითად სერვერზე ინფორმაციის რამდენიმე ვერსია გვექნება, მაგრამ მომხმარებლებს მივაწვდით მხოლოდ იმ ვერსიას, რომელიც დასრულეულია, ანუ სათადარიგო სერვერების საჭირო რაოდენობაზე უკვე ჩაწერილია.

2.10. განაწილებული გარემო – sharding

Sharding ტექნოლოგია მონაცემთა ბაზის დანაწევრების (database partitioning) ერთერთი სახეა. Shard ტერმინი ერთი მთლიანის პატარა ნაწილს ნიშნავს,– შარდინგის მეშვეობით დიდი ზომის უმართავ ბაზას ვყოფთ პატარა, სწრაფ და ადვილად მართვად ნაწილებად.

ტექნიკურად არც შარდინგის აწყობაა რთული, მთავარი სირთულე shard key გასაღების სწორად არჩევაა. ამ გასაღების მიხედვით ნაწილდება დოკუმენტები შესაბამის Shard-ებზე.

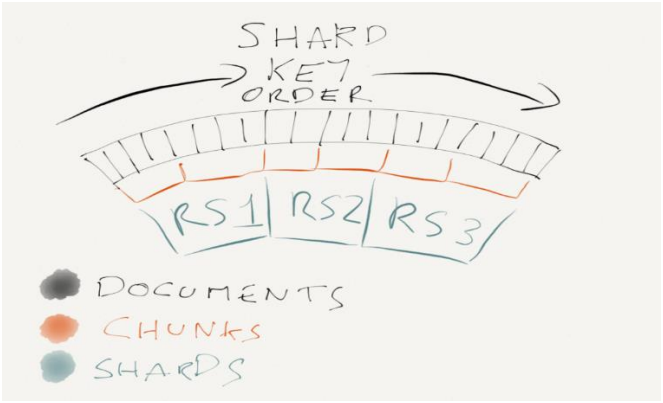
მაგალითისთვის შეგვიძლია მოვიყვანოთ 5 ბილიონი ჩანაწერის მქონე კოლექცია, რომელიც გადანაწილებულია 5 Shard-ზე, გასაღებად კი აღებულია ტექსტური ველი. გასაღების ამგვარი არჩევის შემთხვევაში მარტივად შეგვიძლია ახალი Shard-ის დამატება რომელიმე ძველ Shard-ზე ინფორმაციის გადანაწილებით (ნახ.2.5).



ნახ.2.5

ჩანკების (chunk) დასაყოფად შეგვიძლია გამოვიყენოთ Range based partitioning ან hash based partitioning.

დოკუმენტები ერთიანდება ფიქსირებული ზომის chunk-ებში (default ზომა – 64 მბ). თავად chunk-ები კი shard-ებში ერთიანდება (ნახ.2.6).



ნახ.2.6

მონაცემების გაზრდის შემდეგ ახალი shard სერვერის დამატებას რომ დავაპირებთ, ერთი ან რამდენიმე არსებული shard-იდან ახალ სერვერზე უნდა გადმოვიტანოთ ახალი დაყოფის შესაბამისი chunk-ები.

mongoDB chunk-ების ბალანსირების პროცესს ავტომატურად, კლიენტებისგან დამოუკიდებლად ასრულებს.

ასევე, მომხმარებლისგან დამოუკიდებლად წყდება, თუ რომელი shard-ებია პასუხისმგებელი კლიენტის მოთხოვნაზე. ამისთვის არსებობს მოთხოვნების მარშრუტიზატორი (query router), რომელიც კლიენტისგან იღებს მოთხოვნას, წინასწარ შენახული metadata (მეტა ინფორმაციის) მიხედვით არჩევს შესაბამის shard სერვერებს და კლიენტს საბოლოოდ აწყობის შედეგს უბრუნებს.

2.11. განაწილებული გარემო – replication + sharding

საერთოდ, მონაცემთა ბაზების წარმადობა დამოკიდებულია სამ ძირითად მახასიათებელზე : პროცესორი, ოპერატიული მეხსიერება და მყარი დისკი.

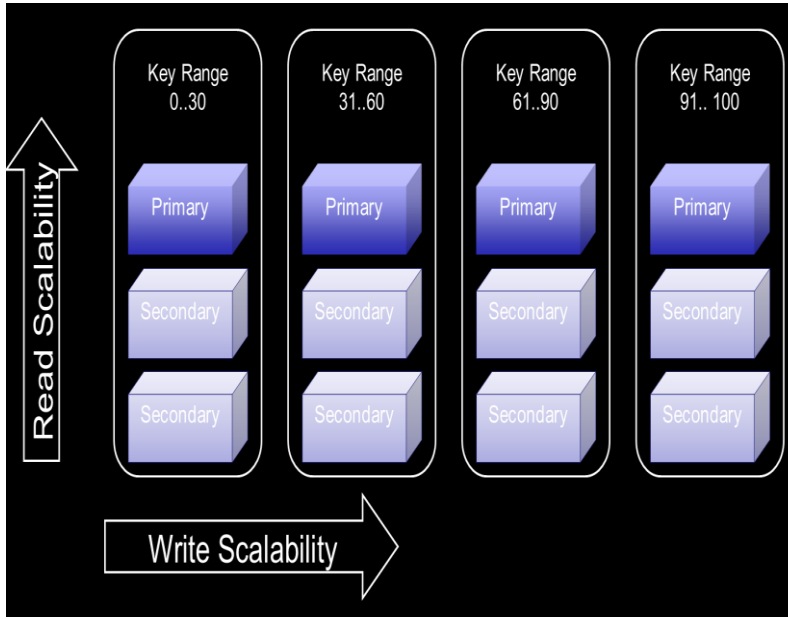
შარდინგისა და რეპლიკაციის დროს კი მოთხოვნის ზრდასთან ერთად შეგვიძლია ასობით და ათასობით ახალი მანქანის დამატება და თითოეული ახალი მანქანა გაგვიუმჯობესებს სამივე ძირითად მახასიათებელს (ნახ.2.7).

ხშირად, ადმინისტრატორები მონაცემთა ბაზებს საჭიროზე მეტ ოპერატიულსა და პროცესორებს უყოფენ, ამ დროს bottleneck (ბოთლის შევიწროებული ყელი) პრობლემას ვაწყდებით დისკების სიმცირეში. სერვერი დიდ დროს კარგავს დისკიდან ინფორმაციის ამოკითხვაზე.

ეს პრობლემა განაწილებულ გარემოში მუშაობისას მინიმუმამდეა დაყვანილი :

- გვჭირდება სწრაფი კითხვა? – ვამატებთ ახალ რეპლიკა სეტს, ანუ ბაზის ახალ კოპიოს;

- გვჭირდება სწრაფი ჩაწერა/წაკითხვა? – შარდინგ ტექნოლოგიით უფრო მცირე ნაწილებად ვყოფთ მონაცემთა ბაზას, შესაბამისად, უფრო მეტი დისკი წერს და კითხულობს ერთდროულად.



ნახ..2.7

III თავი
მონაცემთა ბაზა MongoDB

3.1. მონაცემთა ბაზის აგება MongoDB გარემოში

MongoDB-ში ბაზას ცხადად არ ვქმნით, ვირჩევთ მიმდინარე ბაზას (რომელიც ახალი ბაზის შექმნის დროს რეალურად არც არსებობს) და არჩეულ ბაზაში უბრალოდ ვამატებთ ახალ ჩანაწერებს. თვითონ მონაცემთა ბაზა პირველივე მონაცემის ჩაწერის პარალელურად, ავტომატურად იქმნება.

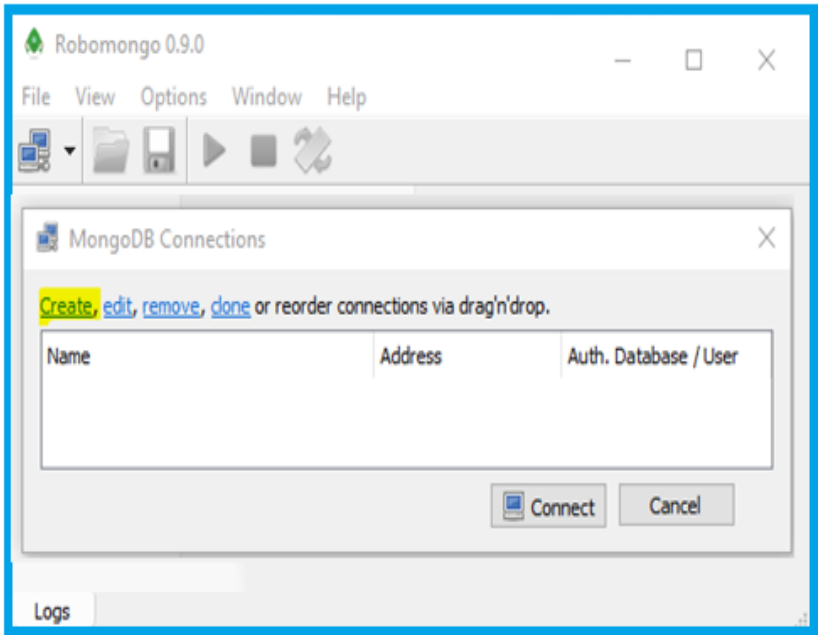
1-ელ დანართში მოყვანილია საცდელი ბაზის შექმნის მაგალითი MongoDB სისტემისთვის [67]. კონკრეტული ოპერაციებისა და მაგალითების განხილვამდე შევქმნათ moviesDB მონაცემთა ბაზა დანართის მიხედვით.

moviesDB მონაცემთა ბაზაში მხოლოდ ერთი კოლექცია გვაქვს - movies, თუმცა, ცხადია, რეალურ სისტემაში ამ კოლექციის გარდა შესაძლოა, გვქონოდა music, games, users, actors … სხვადასხვა კოლექციები. ამ კოლექციაში თავმოყრილია მთელი ის ინფორმაცია, რაც ფილმის შესახებ ინახება.

3.2. RoboMongo პლატფორმა

RoboMongo პლატფორმა არის ათზე მეტი ალტერნატიული ინსტრუმენტიდან ერთ-ერთი ყველაზე გავრცელებული GUI (Graphical User Interface) გრაფიკული ინტერფეისი MongoDB მონაცემთა ბაზასთან სამუშაოდ [68].

RoboMongo პლატფორმა მოხერხებულობითა და სიმარტივით გამოირჩევა. თავდაპირველად ვამატებთ ახალ კავშირს MongoDB სისტემასთან (ნახ. 3.1).



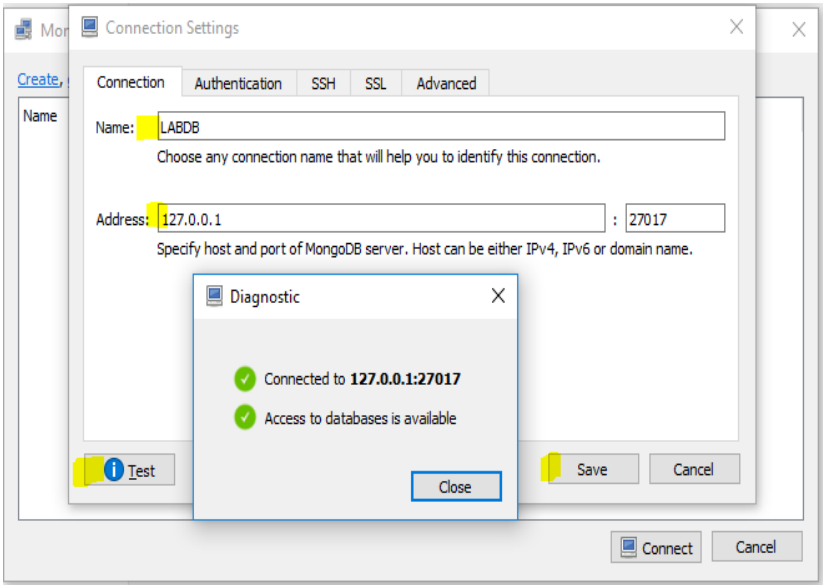
ნახ. 3.1

დიალოგის ფანჯარაში (ნახ. 3.2) შევავსებთ საჭირო ველებს:

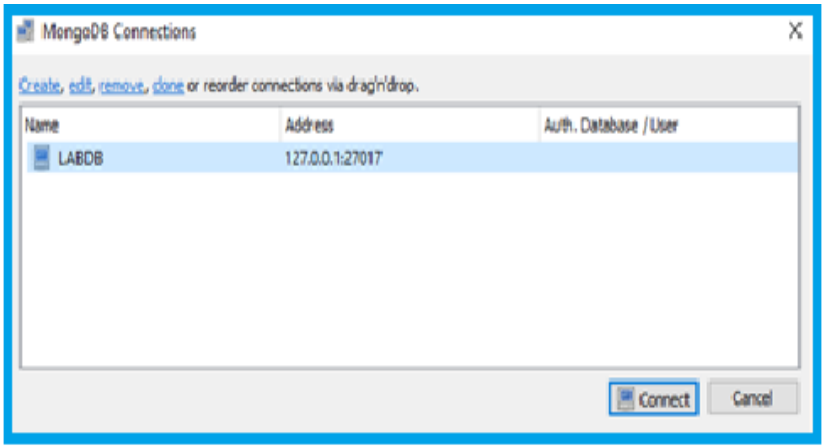
- Name - კავშირის სახელი (შეგვიძლია ნებისმიერი სახელის დარქმევა);
- Address - სერვერის IP მისამართი და პორტი, რომელზეც ელოდება კლიენტებისგან მოსულ მოთხოვნებს. ჩვენ შემთხვევაში ჩავწერეთ 127.0.0.1 - რადგან სერვერი ჩვენივე კომპიუტერზე გვაქვს დაყენებული. პორტი კი უცვლელად დავტოვეთ (default – 27017).

კავშირის შექმნამდე (Save) შეგვიძლია შევამოწმოთ მისი ვალიდურობა Test ღილაკზე დაჭერით.

შემდეგ, 3.3 ნახაზზე ვხსნით ახლად შექმნილ კავშირს.

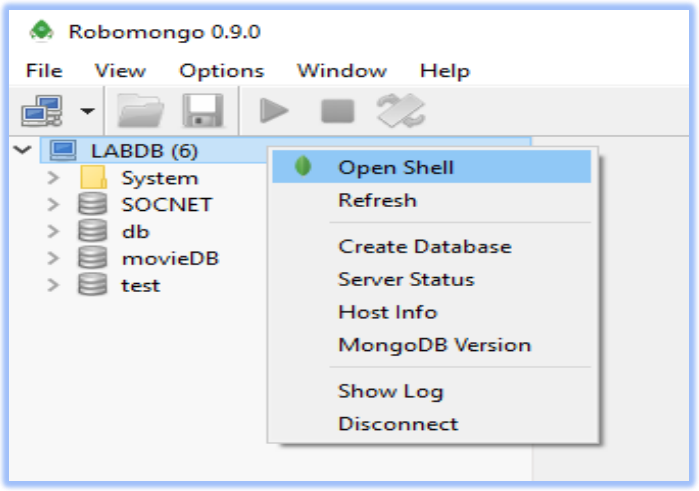


ნახ. 3.2



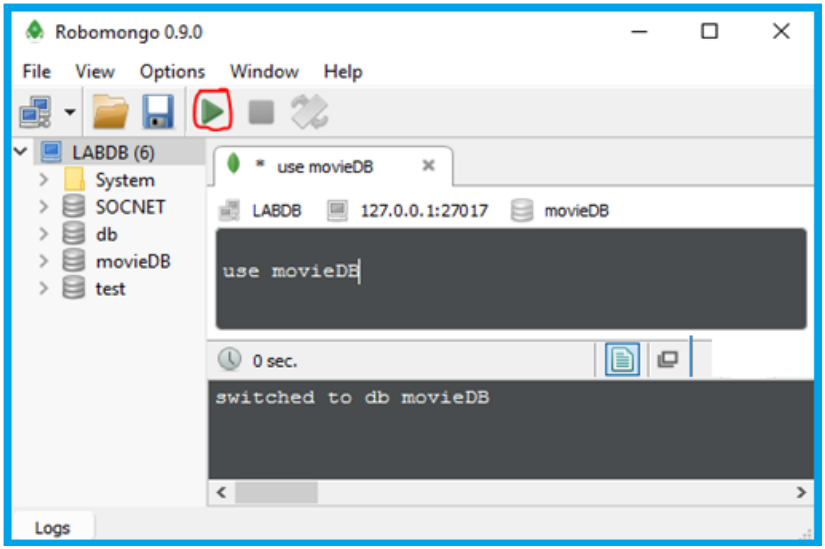
ნახ. 3.3

3.4 ნახაზზე ჩანს უკვე დაკავშირებულ სერვერთან ბრძანებების რეჟიმში გადასვლის მეთოდი.



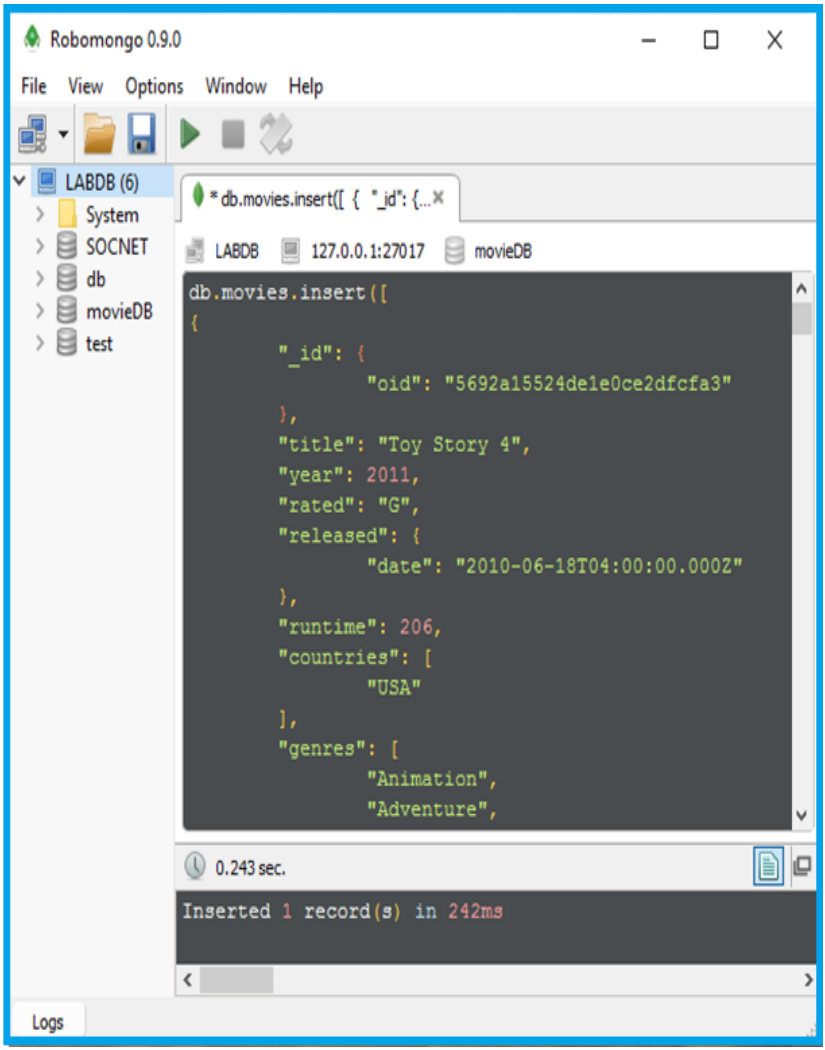
ნახ. 3.4

შემდეგ 3.5 ნახაზზე მოცემულია პირველი ბრძანების - ბაზასთან დაკავშირების მაგალითი



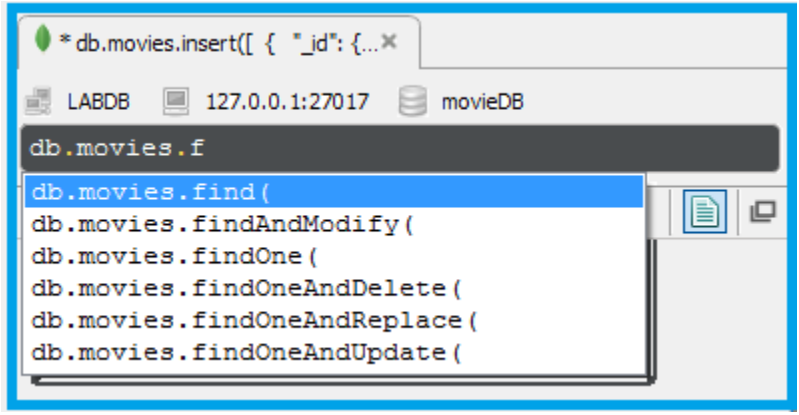
ნახ. 3.5

შემდეგ ეტაპზე დავამატოთ 1-ელ დანართში ნაჩვენები მონაცემები movies კოლექციაში (ნახ.3.6).



ნახ. 3.6

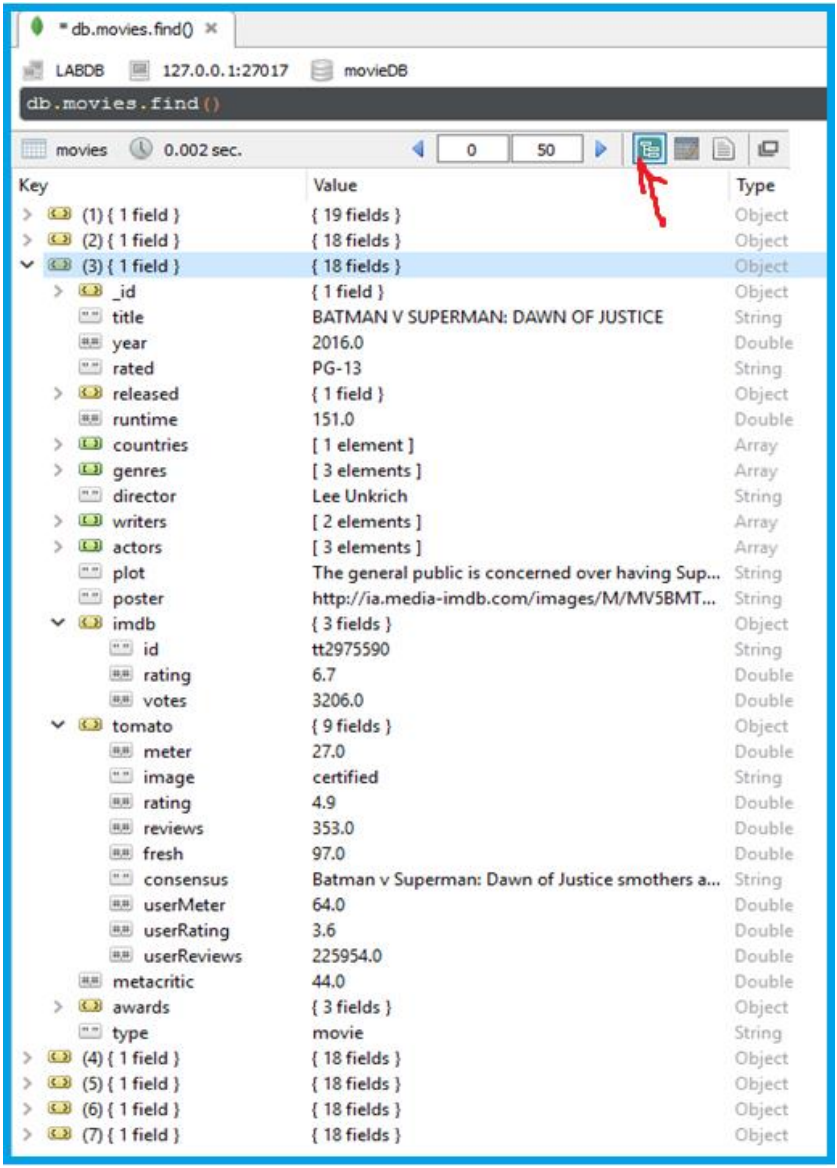
შემდეგ, 3.7 ნახაზზე ჩანს RoboMongo გარემოს intellisense შესაძლებლობა - შემოგვთავაზოს შესაძლო ვარიანტები კოდის წერის დროს.



ნახ. 3.7

db.movies.find() ბრძანება გამოიტანს მიმდინარე ბაზის (movieDB) movies კოლექციის ყველა ჩანაწერს. შედეგის გამოტანა შესაძლებელია სამი სხვადასხვა ფორმატით:

- View results in tree mode - ხისებრი სტრუქტურა (ნახ. 3.8);
- View results in table mode - ცხრილის სტრუქტურა (ნახ. 3.9);
- View results in text mode – JSON სტრუქტურა (ნახ. 3.10).



ნახ.3.8. ხისებრი სტრუქტურა

ნახ. 3.9. ცხრილის სტრუქტურა

ნახ. 3.10. JSON სტრუქტურა

3.3. ბრძანებებისა და ოპერაციების მაგალითები MongoDB ბაზაში

3.3.1. ინფორმაციის ამოღების მაგალითები

მას შემდეგ, რაც წარმატებით შევქმენით და შევავსეთ movieDB მონაცემთა ბაზა დანართში მოცემული საცდელი მონაცემებით, მოვიყვანოთ სხვადასხვა ოპერაციების მაგალითები ამ სისტემის გამოყენებით.

- მიმდინარე ბაზის არჩევა
  use movieDB
- ძიება ფილმის სათაურის მიხედვით
  db.movies.find({"title":'Deadpool'})
- ძიება ფილმის გამოშვების თარიღის მიხედვით
  db.movies.find({ year: { $gt: 2010 } })
  db.movies.find({ year: { $gt: 2010, $lt: 2012 } })

სადაც $gt და $lt შედარების ოპერატორებია (Greater Then, Less, Then)

- ფილმის ძიება IMDB რეიტინგის მიხედვით (ამ მაგალითში ჩანს, როგორ უნდა მივწვდეთ ჩადგმული დოკუმენტის ველებს)

```
db.movies.find(
 {
        "imdb.rating": {$gt:8}
 }
)
```

- იმ ფილმების ძიება სადაც ფავორიტი მსახიობებიდან ერთ-ერთი მაინც თამაშობს

```
db.movies.find(
  {
    actors: { $in: [ "Tom Hanks", "Gina Carano" ] }
  }
)
```

- ძიება უნიკალური ნომრის მიხედვით

```
db.movies.find(
 {
   "_id.oid" : "5692a15524de1e0ce2dfcfa3"
 }
)
```

- ძიება ჩადგმული დოკუმენტების მასივში:

შემდეგი ფრაგმენტი დაგვიბრუნებს ყველა იმ ფილმს, რომელზეც დატოვებულია კონკრეტული მომხმარებლის კომენტარი

```
db.movies.find(
 {
   reviews: {
         $elemMatch: {
            name: "parvesh"
         }
   }
 }
)
```

- სორტირება

```
db.movies.find().sort( { year: 1 } )
```

- რაოდენობის შეზღუდვა

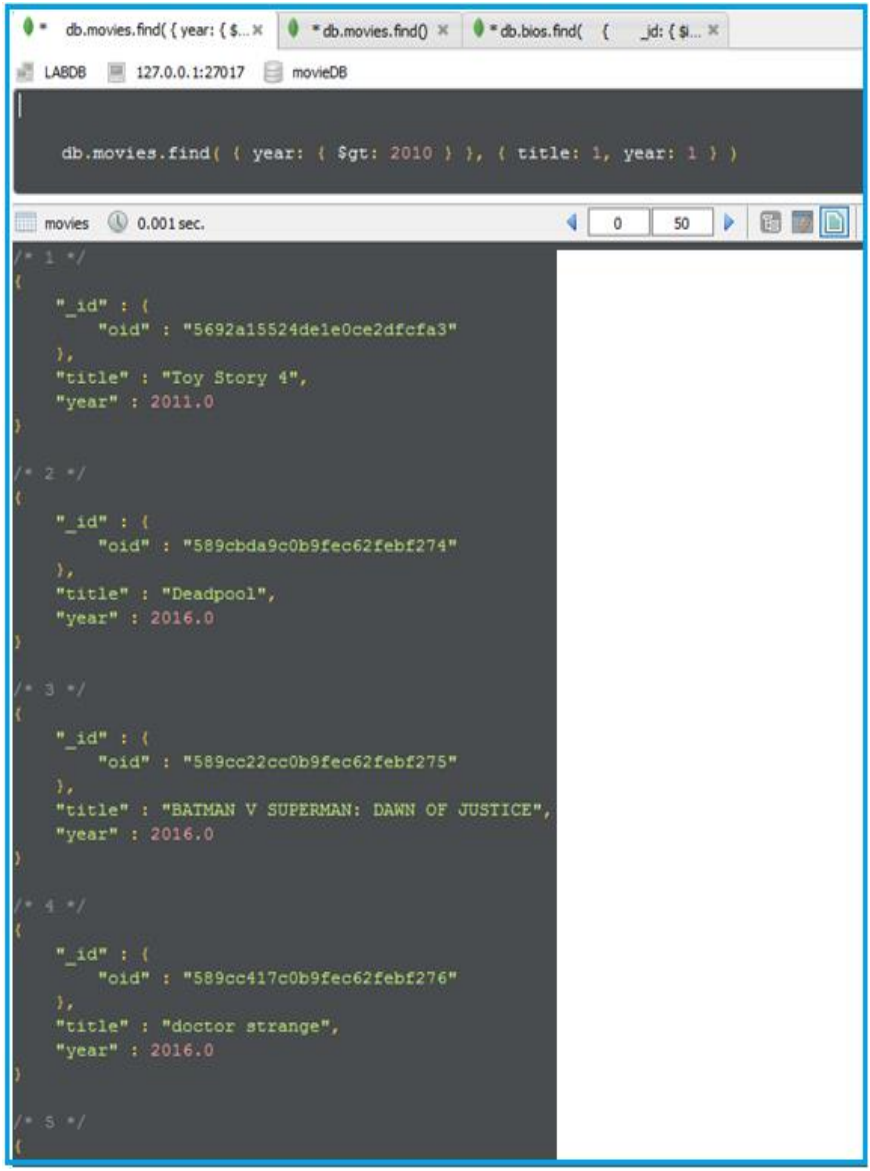
```
db.movies.find().sort( { year: 1 } ).limit(1)
```

ამ ბრძანებით შეგვიძლია ამოვიღოთ ინფორმაცია ყველაზე ძველ ფილმზე

- სასურველი ველების პროექცია

```
db.movies.find( { year: { $gt: 2010 } }, { title: 1, year: 1 } )
                        { title: 1, year: 1 }
```

ამ სახის ფორმატით შევძლებთ მხოლოდ სასურველი ველების დაბრუნებას (ნახ:3.11)


db.movies.find({ year: { $... × *db.movies.find() × *db.bios.find({ _id: { $i... ×
LABDB 127.0.0.1:27017 movieDB
db.movies.find({ year: { $gt: 2010 } }, { title: 1, year: 1 })
movies 0.001 sec. 0 50
/* 1 */
{
"_id" : {
"oid" : "5692a15524de1e0ce2dfcfa3"
},
"title" : "Toy Story 4",
"year" : 2011.0
}
/* 2 */
{
"_id" : {
"oid" : "589cbda9c0b9fec62febf274"
},
"title" : "Deadpool",
"year" : 2016.0
}
/* 3 */
{
"_id" : {
"oid" : "589cc22cc0b9fec62febf275"
},
"title" : "BATMAN V SUPERMAN: DAWN OF JUSTICE",
"year" : 2016.0
}
/* 4 */
{
"_id" : {
"oid" : "589cc417c0b9fec62febf276"
},
"title" : "doctor strange",
"year" : 2016.0
}
/* 5 */
{

ნახ. 3.11

ანალოგიურად, შეგვიძლია ამოვიღოთ მხოლოდ სასურველი ველები ჩადგმული დოკუმენტების ძებნითაც:

```
db.movies.find( { "imdb.rating": {$gt:8}}, { title: 1, "imdb.rating": 1} )
{ title: 0, "imdb.rating": 0}
```

ამ შემთხვევაში მივიღებდით ყველა ველს, ამ ორის გარდა:

```
/* 1 */
{
    "_id" : {
        "oid" : "5692a15524de1e0ce2dfcfa3"
    },
    "title" : "Toy Story 4",
    "imdb" : {
        "rating" : 8.4
    }
}
/* 2 */
{
    "_id" : {
        "oid" : "589cbda9c0b9fec62febf274"
    },
    "title" : "Deadpool",
    "imdb" : {
        "rating" : 8.1
    }
}
/* 3 */
{
    "_id" : {
        "oid" : "589cc846c0b9fec62febf278"
    },
    "title" : "zootopia",
    "imdb" : {
        "rating" : 8.1
    }
}
```

3.3.2. სხვადასხვა ოპერაციების მაგალითები წიგნების კოლექციისათვის

- ერთმანეთში ჩადგმული დოკუმენტები

```
var book = {
        title: 'MongoDB: The Definitive Guide',
        authors: [
            { lastName: 'Chodorow', firstName: 'Kristina' },
            { lastName: 'Dirolf', firstName: 'Michael' }
        ],
        tags: ['NoSQL', 'Database', 'BigData', 'Programming'],
        pages: 195,
        published: 2010
};
```

- **Insert**

```
db.books.save(book);
// primary key '_id'
// generated by client driver
// e.g. 4fba97070f318c1e73763350
book._id;
```

- **Update**

```
db.books.update({title: /Good Parts/},
                        {$inc: {pages: 3}});
db.books.update({title: /in Action/},
            {$set: {publisher: 'Manning'}},
            false, true);
db.books.update({},
      {$addToSet: {tags: 'Programming'}},
      false, true);
```

- **Delete:**

```
db.books.remove({_id: mybook._id});  // წაშლა კონკრეტული
                                     // იდენტიფიკატორის  მიხედვით
db.books.remove({tags: 'Cooking'});
db.books.remove();            // კოლექციის გასუფთავება
```

- **მარტივი მოთხოვნები**

```
db.books.find();  // შედეგში აისახება ყველა წიგნისგან შემდგარი სია
db.books.count();  // შედეგად დაბრუნდება დოკუმენტების საერთო
                   // რაოდენობა
db.books.find().count();  // შედეგად დაბრუნდება დოკუმენტების
                          // საერთო რაოდენობა
 db.books.findOne({  // უნიკალური იდენტიფიკატორის მიხედვით ძებნა
        _id: ObjectId("4fba97190f318c1e73763353")
});
   // ძიება სხვადასხვა პარამეტრების მიხედვით
db.books.find({ title: 'JavaScript Patterns' });
db.books.find({ title: /^MongoDB/ });
db.books.find({ title: /^MongoDB/,  pages: {$gt: 200} });
// ძიება ჩადგმულ დოკუმენტებიან სტრუქტურაში
db.books.find({
        'authors.lastName': 'Katz'});
db.books.find({
        'authors.lastName':
           {$in: ['Katz', 'McCaw']} });
 db.books.find({
        $or: [
           {'authors.lastName': 'Katz'},
           {'authors.lastName': 'McCaw'}
        ]});
// პროექცია, სორტირება, ზღვარი
db.books.
```

```
        find({/* all */},
        {title: 1, pages: 1}).
        sort({title: 1}).
        limit(4);
```

- **ბრძანებები:**

```
db.runCommand({count: 'books',
        query: {published: 2012}});
db.runCommand({distinct: 'books', key:'tags'});
db.runCommand({group: {
        ns: 'books',
        key: { published: true },
        $reduce: function (obj, prev) {
                prev.pages += obj.pages;
        },
        initial: { pages: 0 }
}});

db.runCommand({ dropDatabase: 1 });
db.runCommand({ getLastError: 1 });
db.runCommand({ serverStatus: 1 });
db.runCommand({ shutdown: 1 });
```

- **Indexes**

```
db.books.ensureIndex({"title": 1}, {unique: true});
db.books.ensureIndex({"authors.lastName": 1});
db.books.ensureIndex({"tags": 1});
db.books.getIndexes();
db.books.dropIndex('title_1');
```

- **Import / Export**

```
mongoexport -d test -c books > mongo.books.txt
mongo test --eval "db.books.remove()"
 mongoimport -d test -c books --file books.txt
mongoexport -d test -c books --jsonArray > books.json
mongoimport -d test -c books --jsonArray < books.json
```

3.4. ოთხი რეპლიკა სეტის აწყობა შარდინგ ტექნოლოგიით

მაგალითი: ამოცანის არსი არის 4 რეპლიკა სეტის აწყობა, რომლებიც Sharding ტექნოლოგიით ინაწილებს მონაცემებს. სასწავლო რეჟიმში ყველა სერვისი ერთ კომპიუტერზე იყო გაშვებული და განსხვავებულ პორტებზე მუშაობდა. რეალურ შემთხვევაში Mongo-ს სერვისები იქნება გაშვებული სხვადასხვა სერვერზე [66].

A რეპლიკა სეტი:

```
mkdir  a0    192.168.0.10
mkdir  a1    192.168.0.11
mkdir  a2    192.168.0.12
```

B რეპლიკა სეტი:

```
mkdir  b0    192.168.1.10
mkdir  b1    192.168.1.11
mkdir  b2    192.168.1.12
```

C რეპლიკა სეტი:

```
mkdir  c0    192.168.2.10
mkdir  c1    192.168.2.11
mkdir  c2    192.168.2.12
```

D რეპლიკა სეტი:

```
mkdir  d0    192.168.3.10
mkdir  d1    192.168.3.11
mkdir  d2    192.168.3.12
```

```
// mkdir-ში ვგულისხმობთ შესაბამის სერვერებზე
// რეპლიკა სეტის, საქაღალდეების, datafile-ებისა
// მთლიანად მონაცემთა ბაზისთვის საჭირო დირექტორიების შექმნას

mkdir  cfg0    192.168.9.10    // კონფიგ–სერვერების datafile-ები
mkdir  cfg1    192.168.9.11
mkdir  cfg2    192.168.9.12
# config servers    კონფიგ–სერვერების შექმნა
```

```
mongod  --configsvr  --dbpath cfg0  --port 26050  --fork  --logpath log.cfg0
--logappend
mongod  --configsvr  --dbpath cfg1  --port 26051  --fork  --logpath log.cfg1
--logappend
mongod  --configsvr  --dbpath cfg2  --port 26052  --fork  --logpath log.cfg2
--logappend
```

რეპლიკა სეტის მოსაწყობად შესაბამის A,B,C,D[0,1,2] სერვერებზე სათითაოდ ვუშვებთ შესაბამის სერვისებს:

```
# “shard servers” (mongod data servers)
# note: don’t use  small files  such a small oplogsize in production;
# “shard servers” (mongod data servers)
# note: don’t use  small files  such a small oplogsize in production;
mongod --shardsvr --replSet a --dbpath a0 --logpath log.a0 --port 27000 -
-fork --logappend
 --smallfiles --oplogSize 50
mongod --shardsvr --replSet a --dbpath a1 --logpath log.a1 --port 27001 -
-fork --logappend
--smallfiles --oplogSize 50
mongod --shardsvr --replSet a --dbpath a2 --logpath log.a2 --port 27002 -
-fork --logappend
--smallfiles --oplogSize 50
mongod --shardsvr --replSet b --dbpath b0 --logpath log.b0 --port 27100 -
-fork --logappend
--smallfiles --oplogSize 50
mongod --shardsvr --replSet b --dbpath b1 --logpath log.b1 --port 27101 -
-fork --logappend
--smallfiles --oplogSize 50
mongod --shardsvr --replSet b --dbpath b2 --logpath log.b2 --port 27102 -
-fork --logappend
--smallfiles --oplogSize 50
mongod --shardsvr --replSet c --dbpath c0 --logpath log.c0 --port 27200 -
-fork --logappend
--smallfiles --oplogSize 50
```

```
mongod --shardsvr --replSet c --dbpath c1 --logpath log.c1 --port 27201 --fork --logappend
--smallfiles --oplogSize 50
mongod --shardsvr --replSet c --dbpath c0 --logpath log.c0 --port 27200 --fork --logappend
--smallfiles --oplogSize 50
mongod --shardsvr --replSet c --dbpath c1 --logpath log.c1 --port 27201 --fork --logappend
--smallfiles --oplogSize 50
mongod --shardsvr --replSet c --dbpath c2 --logpath log.c2 --port 27202 --fork --logappend
 --smallfiles --oplogSize 50
mongod --shardsvr --replSet d --dbpath d0 --logpath log.d0 --port 27300 --fork --logappend
--smallfiles --oplogSize 50
mongod --shardsvr --replSet d --dbpath d1 --logpath log.d1 --port 27301 --fork --logappend
--smallfiles --oplogSize 50
mongod --shardsvr --replSet d --dbpath d2 --logpath log.d2 --port 27302 --fork --logappend
--smallfiles --oplogSize 50
```

პარამეტრები --smallfiles და --oplogSize 50 მხოლოდ სასწავლო/სატესტო რეჟიმისთვის არის რეკომენდებული:

```
# mongos processes Default port 27017
mongos --configdb mongo.db:26050,mongo.db:2651,mongo.db:26052 --fork --logappend --logpath
log.mongos0
mongos --configdb mongo.db:26050,mongo.db:2651,mongo.db:26052 --fork --logappend --logpath
log.mongos1 --port 26061
mongos --configdb mongo.db:26050,mongo.db:2651,mongo.db:26052 --fork --logappend --logpath
log.mongos2 --port 26062
```

```
mongos --configdb mongo.db:26050,mongo.db:2651,mongo.db:26052 --
fork --logappend --logpath
log.mongos3 --port 26063
```

MongoDB–ს აქვს მრავალსერვერიანი სისტემებისთვის რეპლიკა სეტისა და შარდინგის სკრიპტების დაგენერირების შესაძლებლობაც.

რეპლიკა სეტი:

```
mongo --nodb
var rst = new ReplSetTest({name:"testSet",
nodes:{node1:{smallfiles:"",oplogsize:40,nopreallocc:null},node2:{smallfiles
:"",oplogSize:40,nopreallo
c:null}, arb:{smallfiles:"",oplogSize:40, nopreallocc:null}}});
rst.startSet();
```

შარდინგი:

```
mongo --nodb

> config = { d0 : { smallfiles : "", noprealloc : "", nopreallocj : ""}, d1 : {
smallfiles : "", noprealloc : "",
nopreallocj : "" }, d2 : { smallfiles : "", noprealloc : "", nopreallocj : ""}};
> cluster = new ShardingTest( { shards : config } );
```

ზემოთ მოყვანილი სკრიპტები ერთმანეთთან აკავშირებს სერვერებს.

რეალურად, რეპლიკა სეტს კი შემდეგი ბრძანებებით ვააქტიურებთ და ვამოწმებთ:

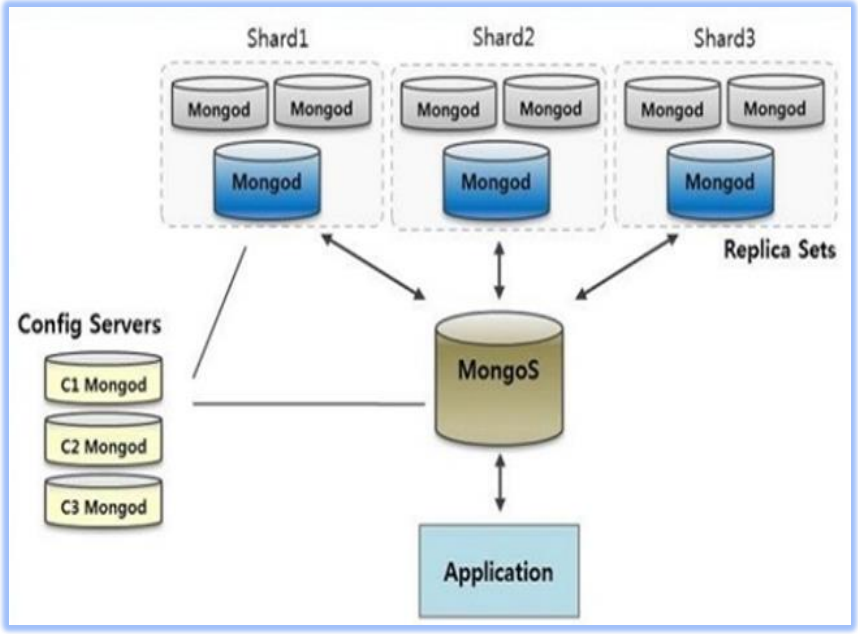
```
mongo --port 27000
rs.status()
rs.initiate()
rs.status()
rs.add(“Mongo.db:27001”)
rs.add(“Mongo.db:27002”)
rs.conf()
connect mongos
mongo
sh.addshard(“a/Mongo.db:27000”)
sh.status()
```

ინფორმაციის მიღება განაწილებული გარემოდან:

განვიხილოთ მაგალითი, თუ როგორ მოქმედებს Sharding-ი MongoDB-ში: მომხმარებელი ითხოვს კონკრეტულ ინფორმაციას.

პირველ რიგში, მოთხოვნა შემოვა Mongos მოთხოვნების გამანაწილებელ და მთავარ კონტროლერ სერვისთან, რომელიც კონფიგურაციის სერვერიდან მიიღებს ინფორმაციას, თუ რომელ Shard-ს უნდა მიმართოს ამ კონკრეტული მოთხოვნის შესასრულებლად და გადასცემს მოთხოვნას შესაბამის Shard-ს, მისგან მიღბულ ინფორმაციას კი დაუბრუნებს კლიენტს (ნახ.3.12). ამ ოპერაციის შესასრულებლად სხვა Shard-ების რესურსი არ გამოიყენება.

შესაბამისად, ვიღებთ ოპერაციების გადანაწილებას ჰორიზონტალურად, რაც, თავის მხრივ, უზრუნველყოფს სწრაფქმედებას.



ნახ.3.12

ლიტერატურა:

1. Codd E.F. Further normalization of the database relational model. In Data Base Systems, Courant Computer Science Symposia 6. Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 65-98.

2. ჩოგოვაძე გ., სურგულაძე გ., ქაჩიბაია ვ. მონაცემთა ბაზების მართვის სისტემები. სტუ. თბილისი. 1988

3. Чоговадзе Г., Сургуладзе Г., Качибая В. Теория реляционных зависимостей и проектирование логической схемы баз данных. Монография. „Мецниереба“, Тбилиси, 1988

4. ჩოგოვაძე გ., სურგულაძე გ., შონია ო. მონაცემთა და ცოდნის ბაზების აგების საფუძლები. სტუ. თბილისი. 1996

5. მეიერ-ვეგენერი კ., სურგულაძე გ., ბასილაძე გ. საინფორმაციო სისტემების აგება მულტიმედიური მონაცემთა ბაზებით. სტუ. თბილისი. 2014

6. Wedekind H., Surguladze G. Technology of Designing of Distributed Systems on the Basis of Objectoriented Programming. ISSN 021-7164, GTU, Tbilissi. 1996. pp.96-100.

7. ქოროღლიშვილი ვ. NoSQL'n CAP. Just Development. Computer sciences. 2014. http://vakhokor.blogspot.com/2014/10/nosql-n-cap.html

8. Гранков М.В., Жуков А.И. Системы управления Базами данных. Донской гос.техн. Университет. Ростов-на-Дону. 2013

9. Кузнецов С. Объектно-ориентированные базы данных - основные концепции, организация и управление: краткий обзор. http://citforum.ru/database/articles/art_24.shtml

10. ODBMS.ORG :: Object Database (ODBMS) | Object-Oriented Database (OODBMS) | Free Resource Portal. ODBMS (2013-08-31). Retrieved on 2013-09-18. Archived July 25, 2014, at the Wayback Machine.

11. Lecluse Ch., Richard Ph., Velez F. O2, an Object-Oriented Data Model. Proc.ACM SIGMOD Int.Conf.Manag.Data. Chicago, Ill, USA, June 1-3, 1988, ACM SIGMOD Record.- 17, N 3.- 1988.- 424-433

12. Kim W. Object-Oriented Databases: Definition and Research Directions. IEEE Trans. Data and Knowledge Eng.- 2, N 3.- 1990.- 327-341

13. Ishikawa H. Object-Oriented Database System: Design and Implementation for Advanced Applications. Springer-Verlag. Tokyo. 1993.

14. Fagin R.. A Normal Form for Relational Databases That Is Based on Domains and Keys. IBM Research Laboratory. ACM Transactions on Database Systems, Vol. 6, No. 3, September. 1981, pp. 387-415.

15. Mattos N., Meyer-Wegener K., Mitschang B. A Grand Tour of Concepts for Object-oriented from a Database Point of View. Data & Knowledge Engineering 9 (1992/93). pp. 321-352.

16. Cattell R.G., Barry D.K. The Object Database Standard: ODMG 3.0. San Francisco. Morgan Kaufmann Publ. 2000.

17. Bancilhon F. Query Language for Object-Oriented Database Systems: Analysis and a Proposal. In: Härder (Hrsg.), Datenbanksysteme für Büro,Technik und Wissenschaft, Proc.GI/SI-Fachtagung (Zürich, März 1989), InformatikßFachberichte, Nr.204, Berlin, Springer Verlag, 1989, pp. 1-18.

18. Kim W., Garza J.F., Ballou N., Woelk D. Architecture of the ORION Next-Generation Database System. 1990. http://dl.acm.org /citation.cfm?id=627402

19. Eisenberg A. New Standard for Stored Procedures in SQL. ACM SIGMOD Record 25. 12, 1996.

20. NoSQL. https://en.wikipedia.org/wiki/NoSQL

21. Document-Oriented Database. https://en.wikipedia.org/wiki/-Document-oriented_database

22. Document-oriented Database. Clusterpoint. Retrieved on 2015-10-08. https://www.clusterpoint.com/

23. https://en.wikipedia.org/wiki/Graph_database

24. NoSQL For Dummies®. Published by: John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030-5774, www.wiley.com Copyright © 2015, New Jersey

25. www.arangodb.com

26. https://en.wikipedia.org/wiki/NewSQL

27. Morgan A., Lord M. NoSQL with MySQL 2014. http://www.dr-dobbs.com/database/nosql-with-mysql/240167115

28. https://docs.datastax.com/en/latest-dse/datastax_enterprise/-graph/dseGraphAbout.html

29. MarkLogic. https://en.wikipedia.org/wiki/MarkLogic

30. Neo4j . https://en.wikipedia.org/wiki/Neo4j

31. OrientDB. https://en.wikipedia.org/wiki/OrientDB

32. StardogDB. https://en.wikipedia.org/wiki/Stardog

33. https://en.wikipedia.org/wiki/Web_Ontology_Language

34. Stonebraker M. New SQL: An Alternative to NoSQL and Old SQL for New OLTP Apps. June 2011. http://cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/fulltext

35. Craig S. Mullins. What Is a NewSQL Database System? 2015. http://www.dbta.com/Columns/DBA-Corner/What-Is-a-NewSQL-Data-base-System-104489.aspx

36. Choudhary S. NewSQL: The Best of Both "OldSQL" and "NoSQL"http://www.slideshare.net/sushantbchoudhary/newsql-the-best-o. 2014

37. Glushkov I. NewSQL overview. http://www.slideshare.net/-IvanGlushkov/newsql-overview. 2015.

38. Aslett M. MySQL vs. NoSQL and NewSQL - survey results. 2012. http://www.slideshare.net/mattaslett/mysql-vs-nosql-and-newsql-survey-results-13073043.

39. InnoDB - Storage Engine. https://en.wikipedia.org/wiki/InnoDB

40. Memcached. https://en.wikipedia.org/wiki/Memcached

41. MariaDB. https://en.wikipedia.org/wiki/MariaDB

42. Web-site of MariaDB. https://mariadb.com/kb/en/

43. Percona XtraDB. https://en.wikipedia.org/wiki/XtraDB

44. Nemeth E., Snyder G., Hein T.R., Whaley B. UNIX and LINUX System Administration Handbook fourth edition. Prentice Hall. 2010

45. Cobbaut P. Linux Fundamentals. 2015. http://linux-training.be-/linuxfun.pdf

46. mongoDB manual – http://docs.mongodb.org/manual/

47. https://www.mongodb.com/compare/mongodb-mysql

48. CouchDB Overview. http://couchdb.apache.org/

49. eXist. https://en.wikipedia.org/wiki/EXist

50. IBM Notes. https://en.wikipedia.org/wiki/IBM_Notes

51. The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things - http://bit.ly/digital_universe.

52. Groenfeldt T. (2013). At NYSE, The Data Deluge Overwhelms Traditional Databases. http://www.forbes.com/sites/tomgroenfeldt/2013-/02/14/at-nyse-the-data-deluge-overwhelms-traditional-databases/#644-feb072eb7

53. Miller R. Facebook Builds Exabyte Data Centers for Cold Storage. http://bit.ly/facebook_exabyte

54. Ancestry.com. Company Facts. www.ancestry.com/corporate/-about-ancestry/company-facts

55. სურგულაძე გ., კივილაძე გ., კახელი ბ. NoSQL მონაცემთა ბაზები: განვითარების პერსპექტივები და პრობლემები მართვის საინფორმაციო სისტემებში. სტუ, შრ.კრ. „მას“, N 2(22). გვ.230-239.

56. Rouse M. Scale-out Storage: Vertical vs. horizontal scalability. http://whatis.techtarget.com/definition/scale-out-storage

57. სურგულაძე გ., ჰ.ვედეკინდი, ნ.თოფურია. განაწილებული ოფის-სისტემების მონაცემთა ბაზების დაპროექტება/რეალიზაცია UML-ტექნოლოგიით. სტუ. თბილისი, -237 გვ., 2006. www.gtu.ge/ka-tedrebi/kat94/pdf/OrmErm-31.pdf

58. Wedekind H., Surguladze G. Technology of Designing of Distributed Systems on the Basis of Objectoriented Programming. ISSN 021-7164, GTU, Tbilissi. 1996. pp.96-100.

59. Brewer, Eric A. Towards robust distributed systems (англ.) // Proceedings of the XIX annual ACM symposium on Principles of distributed computing. – Portland, OR: ACM, 2000. –Vol.19, no.7. – DOI:10.1145/343477.343502.

60. CAP theorem. https://en.wikipedia.org/wiki/CAP_theorem.

61. Grigorik I. Weak Consistency and CAP Implications. Igvita, 24 June, 2010). Google-Co. https://www.igvita.com/2010/06/24/weak-consistency-and-cap-implications/

62. Dave Hillis, Ingeniux Blog, 2015, https://www.ingeniux.com/company/blog/why-nosql-is-the-future-of-web-content-management

63. მიდოდაშვილი პ., შესავალი: მონაცემთა ბაზები და მათი მართვის სისტემები; http://www.pacoge.net/Physics/Sesavali_monacemTa%20bazebi%20da%20maTi%20marTvis%20sistemebi.pdf

64. Tom Kyte, Oracle Technology Network, http://www.oracle.-com/technetwork/issue-archive/2010/10-jan/o65asktom-082389.html

65. IBM Knowledge Center, http://www.ibm.com/support/knowledgecenter/SSEPGG_10.1.0/com.ibm.db2.luw.admin.perf.doc/doc/c0004121.html

66. სურგულაძე გ., კივილაძე გ., მაღალი წვდომადობის მქონე მონაცემთა საცავის დაპროექტება არარელაციური მონაცემთა ბაზების გამოყენებით. https://www.slideshare.net/Giorgiiviladze/nosql-50798944

67. Parvesh Tandon, RestFul API using node.js and mongodb. https://github.com/mistertandon/node-express-hbs

68. AlternativeTo – Crowdsourced software recommendations http://alternativeto.net/software/robomongo/

დანართი 1

მონაცემთა ბაზის აგების ექსპერიმენტული მაგალითი MongoDB სისტემაში

დანართში მოყვანილია საცდელი ბაზის შექმნის მაგალითი MongoDB სისტემისთვის [67].

moviesDB (ფილმების) მონაცემთა ბაზაში მხოლოდ ერთი კოლექცია გვაქვს - movies, თუმცა, ცხადია, რეალურ სისტემაში ამ კოლექციის გარდა შესაძლოა, გვქონოდა music, games, users, actors და სხვა კოლექცია. ამ კოლექციაში თავმოყრილია მთელი ის ინფორმაცია, რაც ფილმის გარშემო ინახება:

- "_id" - უნიკალური იდენტიფიკატორი (ჩადგმული დოკუმენტით)
- "title" - ფილმის სათაური
- "year" - ფილმის გამოშვების წელი
- "rated" - ფილმის შეფასება
- "released" - საიტზე ფილმის დამატების თარიღი (ჩადგმული დოკუმენტით)
- "runtime" - ფილმის ხანგრძლივობა წუთებში
- "countries" - მწარმოებელი ქვეყნების სია
- "genres" - ფილმის ჟანრების სია
- "director" - ფილმის რეჟისორი
- "writers" - სცენარისტები
- "actors" - მსახიობების სია
- "plot" - ფილმის მოკლე შინაარსი
- "poster" - ფილმის პოსტერის ბმული
- "imdb" - ჩადგმული დოკუმენტი IMDB კინოკრიტიკოსების რეიტინგის შესანახად
- "tomato" - ჩადგმული დოკუმენტი Rotten Tomatoes მაყურებლების რეიტინგის შესანახად
- "metacritic" – Metacritic რეიტინგის შესანახად

- "awards" - ჩადგმული დოკუმენტი ჯილდოების შესანახად
- "type" - სრულმეტრაჟიანი/მოკლემეტრაჟიანი/სერიალი...
- "reviews" - ჩადგმული დოკუმენტების მასივი საიტის მომხმარებლების კომენტარების შესანახად.

MongoDB-ში ბაზას ცხადად არ ვქმნით, ვირჩევთ მიმდინარე ბაზას (რომელიც ახალი ბაზის შექმნის დროს რეალურად არც არსებობს) და არჩეულ ბაზაში უბრალოდ ვამატებთ ახალ ჩანაწერებს. თვითონ მონაცემთა ბაზა პირველივე მონაცემის ჩაწერის პარალელურად, ავტომატურად იქმნება.

```
use movieDB

db.movies.insert([
{
	"_id": {
		"oid": "5692a15524de1e0ce2dfcfa3"
	},
	"title": "Toy Story 4",
	"year": 2011,
	"rated": "G",
	"released": {
		"date": "2010-06-18T04:00:00.000Z"
	},
	"runtime": 206,
	"countries": [
		"USA"
	],
	"genres": [
		"Animation",
		"Adventure",
```

```
            "Comedy"
        ],
        "director": "Lee Unkrich",
        "writers": [
            "John Lasseter",
            "Andrew Stanton",
            "Lee Unkrich",
            "Michael Arndt"
        ],
        "actors": [
            "Tom Hanks",
            "Tim Allen",
            "Joan Cusack",
            "Ned Beatty"
        ],
        "plot": "ყველა ბავშვს სჯერა, რომ როდესაც ის თავის სათამაშოებს მარტო ტოვებს, ისინი ცოცხლდება და თავის საქმეებს აკეთებს. ეს ანიმაციური ფილმი მათ თავიანთ აზრებში კიდევ ერთხელ დაარწმუნებს.",
        "poster": "http://ia.media-imdb.com/images/M/MV5BMTgxOTY4Mjc0MF5BMl5BanBnXkFtZTcwNTA4MDQyMw@@._V1_SX300.jpg",
        "imdb": {
            "id": "tt0435761",
            "rating": 8.4,
            "votes": 500084
        },
        "tomato": {
            "meter": 99,
            "image": "certified",
            "rating": 8.9,
```

```
                "reviews": 287,
                "fresh": 283,
                "consensus": "Deftly blending comedy, adventure, and honest emotion, Toy Story 3 is a rare second sequel that really works.",
                "userMeter": 89,
                "userRating": 4.3,
                "userReviews": 602138
        },
        "metacritic": 92,
        "awards": {
                "wins": 56,
                "nominations": 86,
                "text": "Won 2 Oscars. Another 56 wins \u0026 86 nominations."
        },
        "type": "movie",
        "reviews": [
                {
                        "date": {
                                "date": "2017-02-13T04:00:00.000Z"
                        },
                        "name": "parvesh",
                        "rating": 8.9,
                        "comment": "My first review for Toy Story 3, hoping it will execute while trying for the very first time."
                },
                {
                        "date": {
                                "date": "2017-02-13T04:00:00.000Z"
                        },
                        "name": "Prabhash",
```

```
                        "rating": 8.9,
                        "comment": "My first review for Toy Story 3, hoping it will execute while trying for the very first time."
                },
                {
                        "date": {
                                "date": "2017-02-11T04:00:00.000Z"
                        },
                        "name": "praveen",
                        "rating": 6.7,
                        "comment": "My first review for Toy Story 3, hoping it will execute while trying for the very first time."
                }
        ]
},{
        "_id": {
                "oid": "589cbda9c0b9fec62febf274"
        },
        "title": "Deadpool",
        "year": 2016,
        "rated": "R",
        "released": {
                "date": "2016-06-18T04:00:00.000Z"
        },
        "runtime": 108,
        "countries": [
                "USA"
        ],
        "genres": [
                "Comics character",
                "Adventure",
```

```
                "Action"
        ],
        "director": "Tim Miller",
        "writers": [
                "Rhett Reese",
                "Paul Wernick"
        ],
        "actors": [
                "Ryan Reynolds",
                "Morena Baccarin",
                "Ed Skrein",
                "T.J. Miller",
                "Gina Carano",
                "Leslie Uggams",
                "Stefan Kapičić",
                "Brianna Hildebrand"
        ],
        "plot": "ყოფილი სპეცდანიშნულების რაზმის წევრი მონაწილეობას იღებს ექსპერიმენტში, რომელიც მას შესაძლებლობას აძლევს მიიღოს წარმოუდგენელი ძალა, სისწრაფე და უნარი სწრაფი განკურნებისა. ",
        "poster": "http://ia.media-imdb.com/images/M/MV5BMTgxOTY-4Mjc0MF5BMl5BanBnXkFtZTcwNTA4MDQyMw@@._V1_SX300.jpg",
        "imdb": {
                "id": "tt1431045",
                "rating": 8.1,
                "votes": 585141
        },
        "tomato": {
                "meter": 99,
                "image": "certified",
```

```
                "rating": 6.9,
                "reviews": 287,
                "fresh": 241,
                "consensus": "Fast, funny, and gleefully profane, the fourth-wall-busting Deadpool.",
                "userMeter": 90,
                "userRating": 4.3,
                "userReviews": 181719
        },
        "metacritic": 92,
        "awards": {
                "wins": 5,
                "nominations": 12,
                "text": "wo Golden Globe Award nominations for Best Motion Picture – Musical or Comedy and Best Actor – Motion Picture Musical or Comedy."
        },
        "type": "movie"
},{
        "_id": {
                "oid": "589cc22cc0b9fec62febf275"
        },
        "title": "BATMAN V SUPERMAN: DAWN OF JUSTICE",
        "year": 2016,
        "rated": "PG-13",
        "released": {
                "date": "2016-03-19T04:00:00.000Z"
        },
        "runtime": 151,
        "countries": [
                "USA"
```

```
],
"genres": [
        "Action",
        "Adventure",
        "Sci-Fi"
],
"director": "Lee Unkrich",
"writers": [
        "Chris Terrio",
        "David S. Goyer"
],
"actors": [
        "Amy Adams",
        "Henry Cavill",
        "Ben Affleck"
],
"plot": "იმის შიშით, რომ მეტროპოლისის სუპერგმირის ქმედებები ისევ დარჩება უკონტროლოდ, გოთემ-სითის რაინდი გამოუცხადებს სუპერმენს ომს. სანამ ისინი არჩევენ საქმეებს და აყენებენ კაცობრიობას არჩევანის წინაშე: თუ რომელი სუპერგმირი უფრო სჭირდებათ მათ, ჩნდება ახალი, გაცილებით საშიში საფრთხე",
"poster": "http://ia.media-imdb.com/images/M/MV5BMTgxOTY-4Mjc0MF5BMl5BanBnXkFtZTcwNTA4MDQyMw@@._V1_SX300.jpg",
"imdb": {
        "id": "tt2975590",
        "rating": 6.7,
        "votes": 3206
},
"tomato": {
        "meter": 27,
```

```
                "image": "certified",
                "rating": 4.9,
                "reviews": 353,
                "fresh": 97,
                "consensus": "Batman v Superman: Dawn of Justice smothers a potentially powerful story -- and some of Americas most iconic superheroes -- in a grim whirlwind of effects-driven action.",
                "userMeter": 64,
                "userRating": 3.6,
                "userReviews": 225954
        },
        "metacritic": 44,
        "awards": {
                "wins": 6,
                "nominations": 26,
                "text": "Actor of the Year, Most Original Poster, Best Body of Work"
        },
        "type": "movie"
},{
        "_id": {
                "oid": "589cc417c0b9fec62febf276"
        },
        "title": "doctor strange",
        "year": 2016,
        "rated": "PG-13",
        "released": {
                "date": "2016-11-04T04:00:00.000Z"
        },
        "runtime": 115,
        "countries": [
```

```
            "USA"
        ],
        "genres": [
            "Sci-Fi",
            "Fantasy",
            "Adventure",
            "Action"
        ],
        "director": "Scott Derrickson",
        "writers": [
            "Jon Spaihts",
            "Scott Derrickson"
        ],
        "actors": [
            "Benedict Cumberbatch",
            "Chiwetel Ejiofor",
            "Rachel McAdams"
        ],
        "plot": "მას შემდეგ, რაც მისი კარიერა განადგურდა, ბრწყინვალე მაგრამ ქედმაღალი ექიმი ახალ გამოწვევას იღებს ცხოვრებაში, როდესაც ჯადოქარი მას მფარველობაში აიყვანს და წვრთნის, რათა სამყარო დაიცვას ბოროტებისაგან. ",
        "poster": "http://ia.media-imdb.com/images/M/MV5BMTgxOTY-4Mjc0MF5BMl5BanBnXkFtZTcwNTA4MDQyMw@@._V1_SX300.jpg",
        "imdb": {
            "id": "tt1211837",
            "rating": 7.8,
            "votes": 191181
        },
        "tomato": {
            "meter": 27,
```

```
                "image": "certified",
                "rating": 4.9,
                "reviews": 353,
                "fresh": 97,
                "consensus": "Batman v Superman: Dawn of Justice smothers a potentially powerful story -- and some of Americas most iconic superheroes -- in a grim whirlwind of effects-driven action.",
                "userMeter": 64,
                "userRating": 3.6,
                "userReviews": 225954
        },
        "metacritic": 44,
        "awards": {
                "wins": 6,
                "nominations": 38,
                "text": "Oscar, Best Visual Effects"
        },
        "type": "movie"
},{
        "_id": {
                "oid": "589cc696c0b9fec62febf277"
        },
        "title": "kung fu panda 3",
        "year": 2016,
        "rated": "PG",
        "released": {
                "date": "2016-01-29T04:00:00.000Z"
        },
        "runtime": 95,
        "countries": [
                "USA"
```

```
],
"genres": [
        " Animation",
        "Action",
        "Adventure",
        "Comedy",
        "Family"
],
"director": " Alessandro Carloni",
"writers": [
        " Jonathan Aibel",
        "Glenn Berger"
],
"actors": [
        " Jack Black",
        "Dustin Hoffman",
        "Bryan Cranston"
],
```


"plot": ”კუნგ ფუ პანდა 3” ისევ დიდ თავგადასავალს გვპირდება. ამჯერად პო და მისი ისევ გამოჩენილი ღვიძლი მამა ლი მიემგზავრებიან საიდუმლო პანდების ადგილსამყოფელში, სადაც ისინი ბევრ წინააღმდეგობას აწყდებიან. მათ მთავარ საშიშროებას ცბიერი და ზებუნებრივით დაჯილდოვებული კაი წარმოადგენს. პანდა პოს რთული მისია აკისრია – მან სხვა პანდებს საბრძოლო ხელოვნება უნდა ასწავლოს. არც თუ ისე მარტივი დავალება აქვთ ფუმფულა პანდებს",


```
"poster": "http://ia.media-imdb.com/images/M/MV5BMTgxOTY-
4Mjc0MF5BMl5BanBnXkFtZTcwNTA4MDQyMw@@._V1_SX300.jpg",
"imdb": {
        "id": "tt2267968",
        "rating": 7.2,
```

```
                "votes": 83809
        },
        "tomato": {
                "meter": 87,
                "image": "certified",
                "rating": 6.8,
                "reviews": 153,
                "fresh": 133,
                "consensus": "Kung Fu Panda 3 boasts the requisite visual splendor, but like its rotund protagonist, this sequels narrative is also surprisingly nimble, adding up to animated fun for the whole family.",
                "userMeter": 79,
                "userRating": 3.9,
                "userReviews": 98794
        },
        "metacritic": 44,
        "awards": {
                "wins": 0,
                "nominations": 6,
                "text": "Best Animated Feature, Most Wanted Pet"
        },
        "type": "movie"
},{
        "_id": {
                "oid": "589cc846c0b9fec62febf278"
        },
        "title": "zootopia",
        "year": 2016,
        "rated": "PG",
        "released": {
                "date": "2016-04-04T04:00:00.000Z"
```

```
},
"runtime": 108,
"countries": [
        "USA"
],
"genres": [
        "Animation",
        "Adventure",
        "Comedy",
        "Crime",
        "Family",
        "Mystery"
],
"director": "Byron Howard",
"writers": [
        "Byron Howard",
        "Rich Moore"
],
"actors": [
        "Ginnifer Goodwin",
        "Jason Bateman",
        "Idris Elba"
],
"plot": " მოგესალმებით ცხოველთა ქალაქში - თანამედროვე ქალაქი, დასახლებული სხვადასხვა ცხოველებით, დიდი სპილოები პატარა თაგვები. ქალაქი იყოფა რაიონებად და აქ ასევე არის ელიტარული უბანი საჰარის ტერიტორიაზე და ასევე არასტუმართმოყვარე ტუნდრათაუნი. ქალაქში არის ახალი პოლიციელი, მხიარული კურდღელი ჯუდი ჰოპსი, რომელიც სამუშაოს პირველი დღიდანვე აცნობიერებს თუ რა ძნელია იყო პატარა და ფუმფულა დიდ და ძლიერ პოლიციელებს შორის. ჯუდი
```

ცდილობს პირველი შესაძლებლობისთანავე წარმოაჩინოს საკუთარი თავი იმის მიუხედავად, რომ მისი მეწყვილე არის მელია ნიკ უაიდლი. მათ ერთად მოუწევთ რთული საქმის გამოძიება, რის გამოაშკარავებამაც შეიძლება საფრთხე შეუქმნას მთელი ქალაქის მაცხოვრებლებს. ",

"poster": "http://ia.media-imdb.com/images/M/MV5BMTgxOTY-4Mjc0MF5BMl5BanBnXkFtZTcwNTA4MDQyMw@@._V1_SX300.jpg",

"imdb": {

"id": "tt2948356",

"rating": 8.1,

"votes": 262258

},

"tomato": {

"meter": 98,

"image": "certified",

"rating": 8.1,

"reviews": 241,

"fresh": 236,

"consensus": "Kung Fu Panda 3 boasts the requisite visual splendor, but like its rotund protagonist, this sequels narrative is also surprisingly nimble, adding up to animated fun for the whole family.",

"userMeter": 92,

"userRating": 4.4,

"userReviews": 95658

},

"metacritic": 44,

"awards": {

"wins": 26,

"nominations": 52,

"text": "Best Animated Feature Film of the Year, Best Motion Picture - Animated"

},

"type": "movie"

```
},{
        "_id": {
                "oid": "589d733a296ba85b1bc3bee6"
        },
        "title": "John Carter",
        "year": 2012,
        "rated": "PG-13",
        "released": {
                "date": "2012-03-09T04:00:00.000Z"
        },
        "runtime": 132,
        "countries": [
                "USA"
        ],
        "genres": [
                "Action",
                "Adventure",
                "Sci-Fi"
        ],
        "director": "Andrew Stanton",
        "writers": [
                "John Lasseter",
                "Andrew Stanton",
                "Lee Unkrich",
                "Michael Arndt"
        ],
        "actors": [
                "Andrew Stanton",
                "Mark Andrews"
        ],
        "plot": "ამერიკის სამოქალაქო ომის ვეტერანი ჯონ კარტერი,
მარსზე აღმოჩნდება, სადაც ტყვედ ხვდება მტრულად განწყობილ
ოთხმეტრიან აბორიგენებთან. კარტერი არამარტო თავად უნდა
გადარჩეს, არამედ პრინცესა დეა ტორისიც უნდა გადაარჩინოს. ",
```

"poster": "http://ia.media-imdb.com/images/M/MV5BMTgxOTY-4Mjc0MF5BMl5BanBnXkFtZTcwNTA4MDQyMw@@._V1_SX300.jpg",
"imdb": {
"id": "tt0401729",
"rating": 6.6,
"votes": 217518
},
"tomato": {
"meter": 51,
"image": "certified",
"rating": 5.7,
"reviews": 219,
"fresh": 111,
"consensus": "While John Carter looks terrific and delivers its share of pulpy thrills, it also suffers from uneven pacing and occasionally incomprehensible plotting and characterization.",
"userMeter": 60,
"userRating": 3.5,
"userReviews": 113966
},
"metacritic": 92,
"awards": {
"wins": 2,
"nominations": 7,
"text": "Top Box Office Films, Best Original Score for a Fantasy/Science Fiction/Horror Film"
},
"type": "movie"
}
])

გადაეცა წარმოებას 20.03.2017 წ. ხელმოწერილია დასაბეჭდად 30.03.2017 წ. ოფსეტური ქაღალდის ზომა 60X84 1/16. პირობითი ნაბეჭდი თაბახი 9. ტირაჟი 100 ეგზ.

Verba volant, scripta manent

სტუ-ს „IT კონსალტინგის ცენტრი“ (თბილისი, მ.კოსტავას 77)