გულნარა ჯანელიძე, ბადრი მეფარიშვილი,
ლელა წითაშვილი

მონაცემთა ბაზების მართვის თანამედროვე სისტემები (Sql, NoSql)

(დამხმარე სახელმძღვანელო)

დამტკიცებულია:
სტუ-ს „IT-კონსალტინგის სამეცნიერო ცენტრის" რედკოლეგიის მიერ

თბილისი
2024

უაკ 004.5

განხილულია MS SQL Server და MongoDBCompass გარემოში მონაცემთა ბაზების, კოლექციების შექმნისა და მოთხოვნების დამუშავების ტექნოლოგია.

წარმოდგენილია რელაციური (SQL) და არარელაციური (NoSQL) მონაცემთა ბაზების ძირითადი მახასიათებლები, ასევე რელაციურ მონაცემთა ბაზებთან და დოკუმენტზე ორიენტირებულ მონაცემთა ბაზებთან მუშაობის ინსტრუმენტები და მეთოდები. წარმოდგენილი სისტემები არა მხოლოდ მართავენ სტრუქტურირებულ და არასტრუქტურირებულ მონაცემებს, არამედ გვთავაზობენ კომპლექსურ, ინტეგრირებულ პროგრამულ უზრუნველყოფას ექსპლოატაციისა და ანალიზისთვის.

დამხმარე სახელმძღვანელო განკუთვნილია ინფორმატიკის სფეროს სტუდენ-ტებისთვის.

რეცენზენტები:

რომან სამხარაძე – პროფესორი, ტექნიკის მეცნიერებათა დოქტორი (სტუ)
ლია გაჩეჩილაძე – ასოც. პროფესორი, აკად. დოქტორი (სტუ)

პროფ. გ. სურგულაძის რედაქციით

რედკოლეგია:

ა. ფრანგიშვილი (თავმჯდომარე), ზ. აზმაიფარაშვილი, ნ. ამილახვარი, მ. ახობაძე, ზ. ბოსიკაშვილი, ზ. გასიტაშვილიო, მ. თევდორაძე, თ. კაიშაური, რ. კაკუბავა, ვ. კვარაცხელია, თ. ლომინაძე, ნ. ლომინაძე, ჰ. მელაძე, ნ. ოთხოზორია, ლ. პეტრიაშვილი, თ. ჟვანია, გ. სურგულაძე, ი. ქართველიშვილი, მ. ჩხაიძე, ზ. წვერაიძე, ო.შონია

© სტუ-ს "IT-კონსალტინგის სამეცნიერო ცენტრი", 2024

ISBN ISBN 978-9941-8-6858-0

ყველა უფლება დაცულია. ამ წიგნის ნებისმიერი ნაწილის (ტექსტი, ფოტო, ილუსტრაცია თუ სხვა) არც ერთი ფორმითა და საშუალებით (ელექტრონული თუ მექანიკური) არ შეიძლება გამომცემლის წერილობითი ნებართვის გარეშე. საავტორო უფლების დარღვევა კანონით ისჯება

შინაარსი

შესავალი....4
SQL და NoSQL მონაცემთა ბაზების ძირითადი მახასიათებლები 5
I თავი. SQL მონაცემთა ბაზებთან მუშაობა 14
1.1. SQL Server Management Studio 14
1.2. მუშაობა SQL Server Management Studio-ში 21
1.3. მონაცემთა განსაზღვრის ენა (Data Definition Language – DDL)....39
1.4. მონაცემთა მანიპულირების ენა (DML). 65
1.5. JOIN, INSERT, UPDATE, DELETE ბრძანებები....85
1.6. წარმოდგენები....98
1.7. შენახვადი პროცედურები. მომხმარებლის მიერ შემუშავებული ფუნქციები....105
1.8. პარალელიზმის მართვა 113
1.9. ტრიგერებთან მუშაობა TRANSACT SQL-ში 118
1.20. კურსორები....123
1.21. სარეზერვო ასლების შექმნა. სარეზერვო ასლიდან აღდგენა....128
1.22. მონაცემთა იმპორტი და ექსპორტი....130
1.23. მონაცემთა ბაზების უსაფრთხოების სისტემა....131
II თავი. NoSql მონაცემთა ბაზებთან მუშაობა (Mongo DB) 142
2.1. NoSql მონაცემთა ბაზის სტრუქტურა....142
2.2. კოლექციის შექმნა და წაშლა. დოკუმენტის შექმნა და წაშლა....150
2.3. მონაცემთა დამატება....153
2.4. ამორჩევა და ფილტრაცია 156
2.5. პროექცია 159
2.6. მოთხოვნები ჩაშენებულ ობიექტებთან 160
2.7. კურსორი 162
2.8. ინდექსები 165
2.9. აგრეგატული ფუნქციები....167
2.10. ამორჩევის ოპერატორები....169
2.11. ლოგიკური ოპერატორები....170
2.12. ძებნა მასივების მიხედვით....171
2.13. მონაცემთა განახლება....172
2.14. კოლექციების მართვა 177
ლიტერატურა....179

შესავალი

SQL მონაცემთა ბაზა არის რელაციური მონაცემთა ბაზა, რომელშიც მონაცემები ორგანიზაბულია ცხრილების სახით სვეტებად და სტრიქონებად. SQL ნიშნავს სტრუქტურირებული მოთხოვნების ენას, რომელიც გამოიყენება მოთხოვნებისთვის და მონაცემთა მართვისთვის რელაციურ მონაცემთა ბაზებში.

NoSQL მონაცემთა ბაზა არის არარელაციური მონაცემთა ბაზა, რომელიც ინახავს მონაცემებს განსხვავებულ ფორმატში. NoSQL მონაცემთა ბაზები არის სხვადასხვა ტიპის, მონაცემთა მოდელის მიხედვით. ძირითადი ტიპებია: გასაღები-მნიშვნელობა ტიპის მონაცემები, სადაც მონაცემები ინახება არასტრუქტურირებული ფორმატით უნიკალური გასაღებით მონაცემების ამოსაღებად. მაგალითად: Redis და DynamoDB. დოკუმენტების მონაცემთა ბაზა, სადაც მონაცემები ინახება დოკუმენტის ფორმატში, როგორიცაა JSON. მაგალითად: MongoDB და CouchDB.; გრაფული მონაცემთა ბაზები, სადაც მონაცემები ინახება კვანძებსა და წიბოებში. მაგალითად: Neo4j და JanusGraph; სვეტისებრი მონაცემთა ბაზები, სადაც მონაცემები ინახება სვეტებში და არა სტრიქონებში. მაგალითად: Cassandra და HBase.

პოპულარული SQL მონაცემთა ბაზები:

- Oracle Database გვთავაზობს ფუნქციებს, როგორიცაა ACID (atomicity, consistency, isolation, durability) შესაბამისობა, SQL მხარდაჭერა და დიდი მოცულობის მონაცემების დამუშავების შესაძლებლობა;
- Microsoft SQL Server გვთავაზობს ფუნქციებს, როგორიცაა ACID შესაბამისობა, SQL მხარდაჭერა და ინტეგრაცია Microsoft-ის სხვა პროდუქტებთან, როგორიცაა Excel და SharePoint;
- PostgreSQL: მძლავრი ღია კოდის რელაციური მონაცემთა ბაზების მართვის სისტემა, რომელიც ხშირად გამოიყენება ვებ აპლიკაციებისთვის. PostgreSQL უზრუნველყოფს ფუნქციებს, როგორიცაა ACID შესაბამისობა, SQL მხარდაჭერა და გაფართოვების შესაძლებლობა მომხმარებლის მიერ განსაზღვრული ფუნქციებისა და შენახვადი პროცედურების მეშვეობით;
- MySQL: ღია კოდის რელაციური მონაცემთა ბაზების მართვის სისტემა, რომელიც ჩვეულებრივ გამოიყენება ვებ აპლიკაციებში. MySQL გთავაზობთ ფუნქციებს, როგორიცაა ACID შესაბამისობა, SQL მხარდაჭერა და მაღალ წარმადობას დიდი მოცულობის წაკითხვის ინტენსიური დატვირთვისთვის. Oracle Corporation ახლა ფლობს MySQL-ს.

პოპულარული NoSQL მონაცემთა ბაზები:

- დოკუმენტების საცავი: მაგალითები მოიცავს MongoDB, Couchbase და Apache CouchDB. ისინი ინახავენ ნახევრად სტრუქტურირებულ ან არასტრუქტურირებულ მონაცემებს დოკუმენტზე ორიენტირებულ ფორმატში, სადაც თითოეული დოკუმენტი შეიცავს გასაღები-მნიშვნელობის წყვილებს ან გასაღები - მასივის წყვილებს;

- გრაფული საცავი: მაგალითები მოიცავს Neo4j, JanusGraph და Amazon Neptune. ისინი აქტიურად იყენებენ გრაფულ მონაცემთა ბაზებს შენახვისა და გრაფული მოთხოვნების შესასრულებლად. მონაცემთა ელემენტები წარმოდგენილია კვანძების, წიბოების და თვისებების სახით. მათ შორის დამოკიდებულებები განისაზღვრება გრაფული ალგორითმების მეშვეობით.
- საცავი „გასაღები-მნიშვნელობა“. მაგალითებია Redis, Amazon DynamoDB და Riak. ისინი აქტიურად ინახავენ მარტივ მონაცემებს გასაღების მნიშვნელობის ფორმატში, რაც მონაცემთა მნიშვნელობების ამოღების საშუალებას იძლევა უნიკალური გასაღების გამოყენებით.

SQL და NoSQL მონაცემთა ბაზები გვთავაზობს სხვადასხვა მიდგომებს და შესაძლებლობებს მონაცემთა მართვისთვის, თითოეულს აქვს საკუთარი ძლიერი და სუსტი მხარეები. საბოლოო ჯამში, არჩევანი SQL და NoSQL მონაცემთა ბაზებს შორის დამოკიდებულია გამოყენების შემთხვევებზე და ბიზნეს მიზნებზე.

SQL და NoSQL მონაცემთა ბაზების ძირითადი მახასიათებლები

SQL და NoSQL არის ორი პოპულარული მონაცემთა ბაზის მოდელი, რომლებიც გამოიყენება სხვადასხვა პრობლემების გადასაჭრელად. იმის განსაზღვრისთვის, თუ რომელი უნდა გამოვიყენოთ მოცემული ამოცანის გადასაწყვეტად, საჭიროა მათი დადებითი და უარყოფითი მხარეების განსაზღვრა.

SQL (Structured Query Language) არის სტრუქტურირებული მოთხოვნების ენა, რომელიც გამოიყენება რელაციური მონაცემთა ბაზების მართვისა და მანიპულირებისთვის. SQL მონაცემთა ბაზები გამოიყენება იქ, სადაც აუცილებელია სტრუქტურირებული ხასიათის მონაცემების შენახვა და მართვა, მაგალითად, ინფორმაცია პროდუქტების, მომხმარებლებისა და შეკვეთების შესახებ.

NoSQL (არა მხოლოდ SQL) არის ფართო ტერმინი, რომელიც ეხება მონაცემთა ბაზის არარელაციურ მოდელებს, რომლებიც იყენებენ სხვადასხვა სტრუქტურებს მონაცემთა შესანახად: დოკუმენტი, გასაღები-მნიშვნელობა, სვეტოვანი და გრაფიკული მონაცემთა ბაზები. NoSQL მონაცემთა ბაზები გამოიყენება, როდესაც საჭიროა არასტრუქტურირებული მონაცემების შენახვა, როგორიცაა მაგალითად, დიდი რაოდენობით ტექსტური მონაცემები, სურათები და ვიდეო.

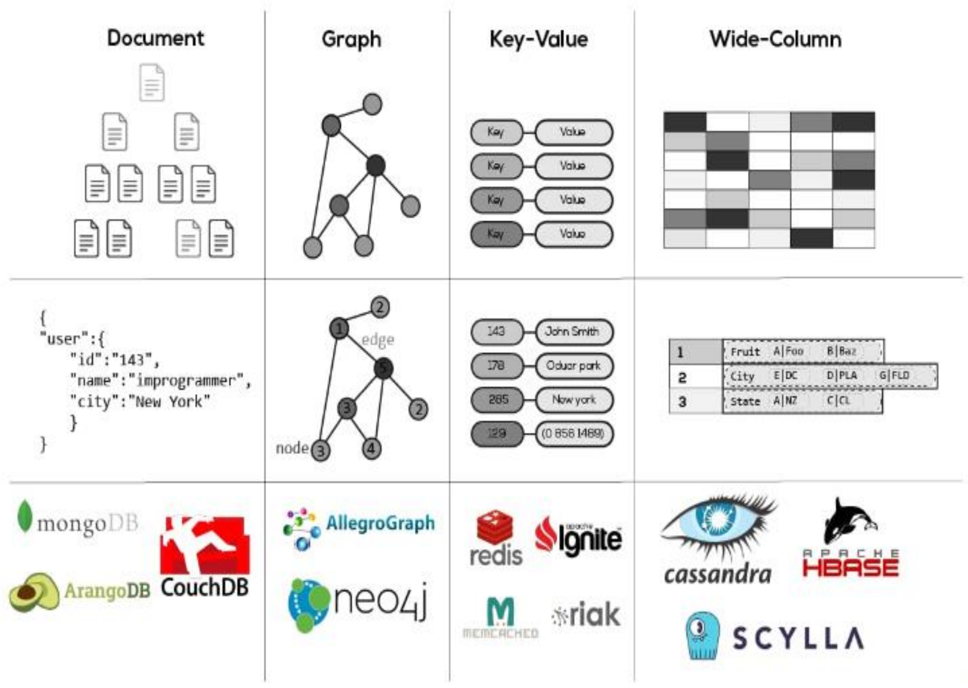
SQL-სა და NoSQL-ს ორი განსხვავებული მიდგომა აქვთ მონაცემთა ბაზებში ინფორმაციის შენახვისა და ორგანიზების თვალსაზრისით. თითოეულ მიდგომას აქვს მონაცემთა ტიპებისა და სტრუქტურების უნიკალური ნაკრები, რომლებიც გამოიყენება მონაცემთა შესანახად და დასამუშავებლად.

SQL მონაცემთა ბაზები იყენებენ რელაციურ მონაცემთა მოდელს, სადაც ინფორმაცია ინახება ცხრილებში, რომლებიც ერთმანეთთან არის დაკავშირებული. ცხრილს აქვს სვეტების ნაკრები, რომელთაგან თითოეული შეესაბამება მონაცემთა

კონკრეტულ ტიპს. მაგალითად, მთელი მონაცემები, სტრიქონები, თარიღები, დრო, ნულოვანი მნიშვნელობები, ლოგიკური მნიშვნელობები და ა.შ. ეს ნიშნავს, რომ მონაცემთა ნაკრები შეიძლება დაიყოს ცალკეულ ველებად თითოეული ტიპის ინფორმაციისთვის. SQL მონაცემთა ბაზები იყენებენ SQL მოთხოვნის ენას, რომელიც საშუალებას აძლევს მომხმარებლებს შექმნან ცხრილები, დაამატოთ, შეცვალონ და წაშალონ მონაცემები და მოიძიონ ინფორმაცია მონაცემთა ბაზიდან.

NoSQL მონაცემთა ბაზებს აქვთ უფრო მოქნილი მონაცემთა მოდელი, რომელიც არ საჭიროებს ცხრილებს და კავშირებს, როგორც SQL მონაცემთა ბაზებში. როგორც წესი, NoSQL მონაცემთა ბაზებში მონაცემები ინახება დოკუმენტებში, კოლექციებში ან გრაფებში. დოკუმენტი არის სტრუქტურირებული კონტეინერი მონაცემთა შესანახად გასაღები-მნიშვნელობის წყვილების ფორმატში, სადაც წყვილებს შეიძლება ჰქონდეთ მონაცემთა სხვადასხვა ტიპები. კოლექცია არის დოკუმენტების ჯგუფი, რომლებიც დაკავშირებულია ერთმანეთთან. გრაფი არის წვეროების ერთობლიობა და მათ შორის კავშირები. NoSQL მონაცემთა ბაზები იყენებს მოთხოვნების სპეციალურ ენებს, რომლებიც მომხმარებლებს საშუალებას აძლევს ამოიღონ და იმანიპულირონ მონაცემებით.

NoSQL მონაცემთა ბაზების ტიპები მოცემულია სურათზე:



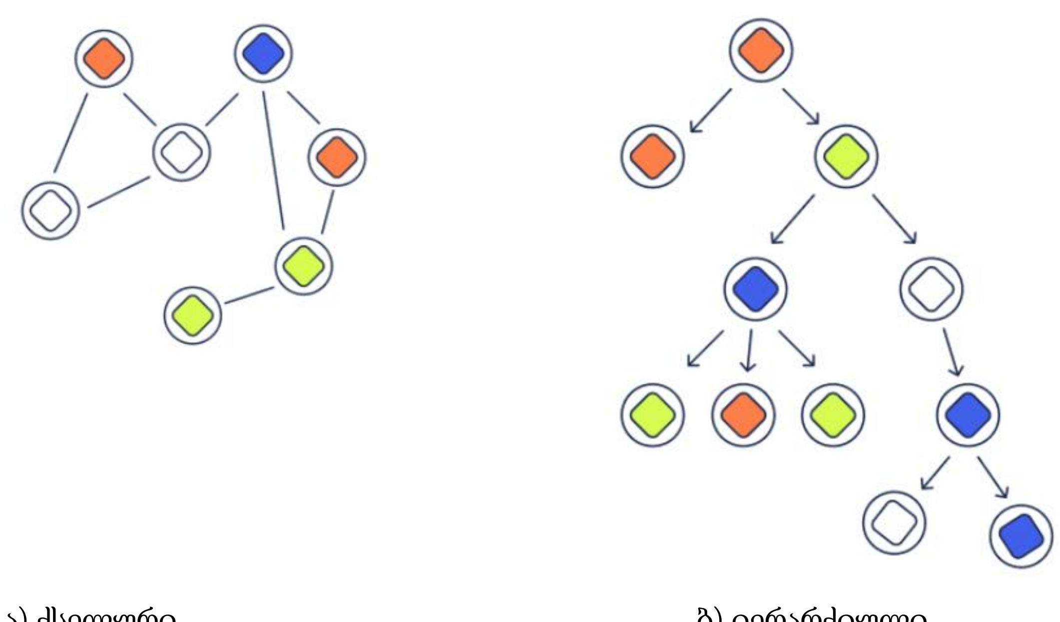
SQL და NoSQL მონაცემთა ბაზები იყენებენ მონაცემთა შენახვის სხვადასხვა მეთოდს და აქვთ შესრულების უნიკალური მახასიათებლები.

SQL მონაცემთა ბაზები, როგორც წესი, იყენებენ რელაციურ მოდელს, სადაც მონაცემები ინახება ცხრილებში, რომლებიც დაკავშირებულია ერთმანეთთან. ცხრილებში მონაცემები სტრუქტურირებულია მკაცრი წესების მიხედვით და შეიძლება დაკავშირება გარე გასაღების გამოყენებით. მონაცემთა ეს სტრუქტურა

შესაბამისს ხდის SQL მონაცემთა ბაზებს სტრუქტურირებული მონაცემების შესანახად და დასამუშავებლად. თუმცა, ეს მოდელი არ არის შესაფერისი არასტრუქტურირებული მონაცემების შესანახად, როგორიცაა სურათები, აუდიო ფაილები, ვიდეო კადრები და ა.შ.

NoSQL მონაცემთა ბაზები იყენებს არარელაციური მონაცემთა შენახვის მოდელს, სადაც მონაცემები შეიძლება ინახებოდეს გასაღების მნიშვნელობის, დოკუმენტების, გრაფიკების და ა.შ. NoSQL მონაცემთა ბაზებს არ აქვთ მკაცრი წესები მონაცემთა ორგანიზებისთვის, რაც მათ უფრო მოქნილს ხდის მონაცემთა სხვადასხვა ტიპების და სტრუქტურების შესანახად. თუმცა, ამ მოქნილობამ ზოგიერთ შემთხვევაში შეიძლება ზიანი მიაყენოს მონაცემთა დამუშავებას.

სურათზე წარმოდგენილია არარელაციური სტრუქტურა:



ა) ქსელური ბ) იერარქიული

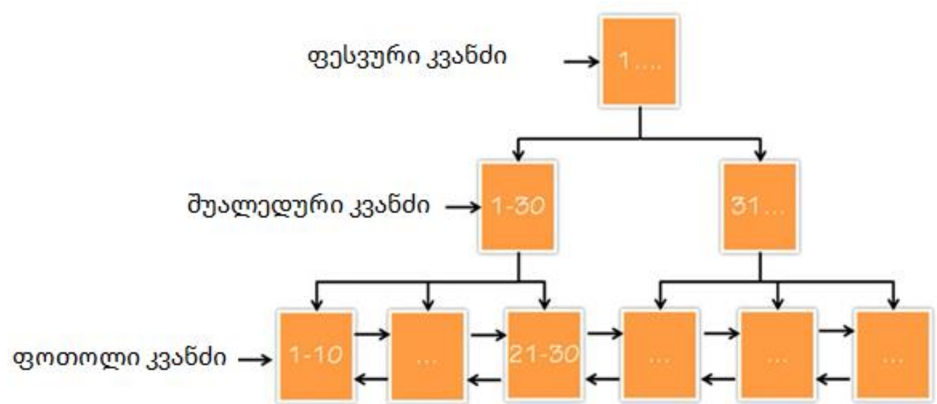
შესრულების თვალსაზრისით, SQL მონაცემთა ბაზებს, როგორც წესი, აქვთ მაღალი წარმადობა მაღალ სტრუქტურირებულ მონაცემებთან და ინფორმაციის დიდ მოცულობასთან მუშაობისას, რადგან მათ შეუძლიათ გამოიყენონ ოპტიმიზებული ინდექსები და მონაცემთა სწრაფი დახარისხება/ფილტრაციის ალგორითმები. თუმცა, არასტრუქტურირებულ მონაცემებთან მუშაობისას, SQL მონაცემთა ბაზების მწარმოებლურობა შეიძლება შემცირდეს.

NoSQL მონაცემთა ბაზებს, როგორც წესი, შეუძლიათ დაამუშავონ დიდი რაოდენობით არასტრუქტურირებული მონაცემები მაღალი წარმადობით, რადგან მათ არ აქვთ მკაცრი სტრუქტურირების წესები. თუმცა, რთულ მოთხოვნებთან მუშაობისას ან დაკავშირებული მონაცემებზე მოთხოვნების შესრულებისას შეიძლება წარმოიშვას მუშაობის პრობლემები.

ინდექსირება არის მნიშვნელოვანი გზა მონაცემთა ბაზების მუშაობის გასაუმჯობესებლად მოთხოვნების დამუშავებისას. SQL და NoSQL მოდელები იყენებენ მონაცემთა ინდექსირების სხვადასხვა ტექნიკას, რათა უზრუნველყონ მონაცემების სწრაფი წვდომა მოთხოვნების შესრულებისას.

SQL მონაცემთა ბაზები იყენებს B-Tree ინდექსს, რომელიც წარმოადგენს ხის სტრუქტურას, სადაც თითოეულ კვანძს აქვს გასაღების ნაკრები დალაგებული აღმავალი ან დაღმავალი თანმიმდევრობით. ამ ტიპის ინდექსი საშუალებას იძლევა სწრაფად შევასრულოთ SELECT, JOIN, WHERE და ORDER BY მოთხოვნები, ასევე უზრუნველყოფს სწრაფ წვდომას მონაცემებზე ჩასმის, განახლებისა და წაშლის ოპერაციების შესრულებისას. SQL მონაცემთა ბაზები ასევე იყენებენ მრავალსვეტიან ინდექსებს იმ მოთხოვნების შესრულების ოპტიმიზაციისთვის, რომლებიც იყენებენ მრავალ სვეტს.

B-Tree კლასტერული ინდექსი მოცემულია სურათზე:



NoSQL მონაცემთა ბაზები იყენებს სხვადასხვა ტიპის ინდექსებს, კონკრეტული მოდელის მიხედვით. MongoDB, რომელიც დოკუმენტზე ორიენტირებული მონაცემთა ბაზაა, იყენებს გასაღებზე დაფუძნებულ ინდექსირებას, რაც საშუალებას გვაძლევს სწრაფად მოვძებნოთ მნიშვნელობები დოკუმენტის ველებში. გასაღები - მნიშვნელობის მონაცემთა ბაზებში ცხადად უნდა მიეთითოს რომელი გასაღებები უნდა იყოს ინდექსირებული. ამ ტიპის ინდექსირება საშუალებას გვაძლევს სწრაფად მივიღოთ მნიშვნელობები გასაღებით, მაგრამ არ იძლევა რთული მოთხოვნების შესრულების საშუალებას, რადგან შემოიფარგლება მხოლოდ გასაღებებით ძებნით. გრაფული მონაცემთა ბაზები, როგორიცაა Neo4j, იყენებს გრაფის ინდექსებს, რომლებიც უზრუნველყოფენ სწრაფ წვდომას გრაფის კვანძებს შორის კავშირებზე.

ზოგადად, SQL მონაცემთა ბაზებს აქვთ უფრო განვითარებული ინდექსირება, ვიდრე NoSQL მონაცემთა ბაზებს რაც უზრუნველყოფს მაღალ წარმადობას რთული მოთხოვნების გაშვებისას, მაგრამ შეიძლება შენელდეს მარტივი მოთხოვნების გაშვებისას. NoSQL მონაცემთა ბაზებში, ინდექსირება დამოკიდებულია კონკრეტულ

მოდელზე და მოითხოვს უფრო დეტალურ დაგეგმვასა და დარეგულირებას საჭირო წარმადობის მისაღწევად.

მონაცემთა ბაზების სანდოობა და სტაბილურობა ერთ-ერთი ყველაზე მნიშვნელოვანი ასპექტია მათი შემუშავებისა და ფუნქციონირებისას. მონაცემთა ბაზებს შეიძლება შეექმნათ სხვადასხვა პრობლემები, როგორიცაა შეფერხებები, ამიტომ აუცილებელია სტაბილური და საიმედო სისტემა, რომელიც პრობლემების შემთხვევაში სწრაფად აღდგენის საშუალებას მოგვცემს.

SQL და NoSQL მონაცემთა ბაზებს აქვთ სხვადასხვა მექანიზმები საიმედობისა და სტაბილურობის უზრუნველსაყოფად, მაგრამ მათი მეთოდები დამოკიდებულია კონკრეტულ გადაწყვეტაზე და მის მოთხოვნებზე საიმედობისა და უსაფრთხოების მიმართ.

SQL მონაცემთა ბაზები იყენებენ ტრანზაქციულ მოდელს, რომელიც საშუალებას გვაძლევს შევინარჩუნოთ მონაცემთა მთლიანობა და უზრუნველვყოთ ACID (Atomicity, Consistency, Isolation, Durability). SQL მონაცემთა ბაზებს ასევე შეუძლიათ გამოიყენონ სარეზერვო და აღდგენის პროცესები მონაცემთა მთლიანობის უზრუნველსაყოფად და შეფერხების შემთხვევაში დანაკარგების შესამცირებლად.

NoSQL მონაცემთა ბაზები, რომლებიც იშვიათად იყენებენ ტრანზაქციულ მოდელებს, ჩვეულებრივ იყენებენ განაწილებულ არქიტექტურას საიმედოობისა და მდგრადობის გასაუმჯობესებლად. კლასტერირება და მონაცემთა რეპლიკაცია ხელს უწყობს მონაცემთა დაკარგვის ალბათობის მინიმუმამდე შემცირებას და უზრუნველყოფს მონაცემთა ხელმისაწვდომობას კვანძის მწყობრიდან გამოსვლის შემთხვევაში.

შედარებისთვის, SQL მონაცემთა ბაზებს აქვთ მრავალი უპირატესობა NoSQL-თან შედარებით. მაგალითად, SQL-ში შესაძლებელია მარტივად დაწესებულ იქნას შეზღუდვები მონაცემების წვდომაზე სხვადასხვა მომხმარებლისთვის და ასევე გამოყენებულ იქნას სხვადასხვა აუთენტიფიკაციის მექანიზმები მონაცემთა უსაფრთხოების უზრუნველსაყოფად. ამავდროულად, SQL მონაცემთა ბაზებს აქვთ უკეთესი ტრანზაქციის მხარდაჭერა, რაც საშუალებას იძლევა ავტომატურად დააბრუნოთ ცვლილებები პრობლემური ტრანზაქციების აღმოჩენისას და, ამრიგად, შეამციროთ უსაფრთხოების შესაძლო პრობლემები.

მეორეს მხრივ, NoSQL მონაცემთა ბაზებს აქვთ უსაფრთხოების გარკვეული უპირატესობა. მაგალითად, MongoDB და Couchbase იყენებენ დოკუმენტზე ორიენტირებულ მოდელს, რაც მათ უფრო დაუცველს ხდის SQL ინექციის შეტევებს. NoSQL მონაცემთა ბაზები ასევე იყენებენ მონაცემთა დაშიფვრის მეთოდებს, რაც უზრუნველყოფს უსაფრთხოების მაღალ დონეს. თუმცა, NoSQL მონაცემთა ბაზებში დეველოპერებმა თავად უნდა დანერგონ უსაფრთხოების მექანიზმები, რამაც შეიძლება გამოიწვიოს შესაძლო პრობლემები, თუ დეველოპერი არ არის გამოცდილი ან არ იჩენს სიფრთხილეს დამუშავებისას.

SQL და NoSQL მონაცემთა ბაზებს აქვთ მოქნილობისა და მასშტაბურობის სხვადასხვა დონე, რაც დამოკიდებულია კონკრეტულ გადაწყვეტაზე და მის მასშტაბურობასა და მოქნილობის მოთხოვნებზე.

NoSQL მონაცემთა ბაზების ერთ-ერთი უპირატესობა არის მოქნილობა და მასშტაბურობა. მათი დეცენტრალიზებული არქიტექტურის წყალობით, მათ შეუძლიათ ადვილად ჰორიზონტალურად მასშტაბირება ახალი სერვერების დამატებით და მათ შორის დატვირთვის განაწილებით. ასევე, NoSQL მონაცემთა ბაზებს არ აქვთ მკაცრი მოთხოვნები მონაცემთა სტრუქტურისთვის და ეს საშუალებას გაძლევთ სწრაფად შეცვალოთ მონაცემთა სქემები და დაამატოთ ახალი ველები სქემის ახალ ვერსიაზე გადასვლის გარეშე. თუმცა, ამან შეიძლება გამოიწვიოს მოთხოვნების დამუშავების სირთულეები, თუ მონაცემთა სტრუქტურა არ არის კარგად გააზრებული და ძალიან ხშირად იცვლება.

SQL მონაცემთა ბაზები, თავის მხრივ, შეიძლება უფრო შეზღუდული იყოს მონაცემთა სტრუქტურის მოქნილობაში და მასშტაბურობაში. ისინი საჭიროებენ ცხრილის სტრუქტურებისა და მონაცემთა ტიპების წინასწარ განსაზღვრას, რამაც შეიძლება გაართულოს სქემის შეცვლის პროცესი მომავალში. თუმცა, SQL მონაცემთა ბაზებში ქეშირებას შეუძლია გააუმჯობესოს შესრულება და უფრო ზუსტად აკონტროლოს ცვლილებები, რაც გვეხმარება შეკითხვის დამუშავების ოპტიმიზაციაში.

ამრიგად, NoSQL მონაცემთა ბაზები უფრო მოქნილი და მასშტაბურია, რაც განსაკუთრებით მნიშვნელოვანია დიდი პროექტებისთვის დიდი რაოდენობით მონაცემებით, მაგრამ ისინი ასევე ნაკლებად პროგნოზირებადი და ნაკლებად ორგანიზებულია, რამაც შეიძლება გამოიწვიოს მართვის სირთულეები. SQL მონაცემთა ბაზებს აქვთ მეტი სტრუქტურა და უკეთესად შეესაბამება ACID ეტიკეტს, მაგრამ ისინი უფრო შეზღუდულია მონაცემთა სტრუქტურის მოქნილობაში და მასშტაბურობაში, რაც შეიძლება იყოს პრობლემა დიდი პროექტებისთვის.

ტრანზაქციები არის ოპერაციები, რომლებიც შესრულებულია მონაცემთა ბაზაში და მხარს უჭერს ACID (Atomicity, Consistency, Isolation, Durability) თვისებებს.

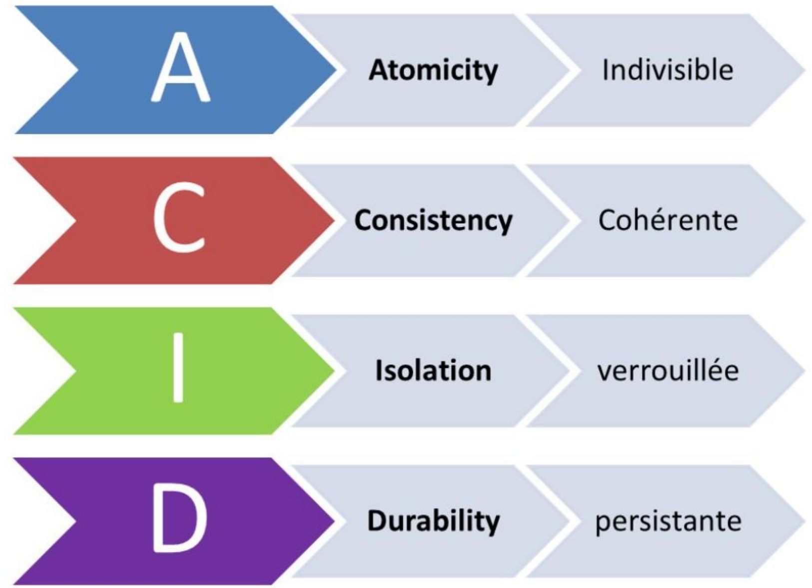
ატომარულობა (Atomicity) არის თვისება, რომელიც იძლევა გარანტიას, რომ ტრანზაქცია შესრულდება სრულად ან საერთოდ არ შესრულდება. თუ ტრანზაქცია მთლიანად ვერ დასრულდება, ის უკან დაბრუნდება და მონაცემთა ბაზა დაუბრუნდება იმ მდგომარეობას, რომელშიც იყო ტრანზაქციის დაწყებამდე.

თანმიმდევრულობა (Consistency) არის თვისება, რაც ნიშნავს, რომ ტრანზაქციამ უნდა გადაიყვანოს მონაცემთა ბაზა ერთი თანმიმდევრული მდგომარეობიდან მეორე თანმიმდევრულ მდგომარეობაში. ტრანზაქციის შედეგად მონაცემთა ბაზა არ შეიძლება დაზიანდეს და მონაცემები უნდა შეესაბამებოდეს ყველა შეზღუდვას და მთლიანობის წესს.

იზოლირება (Isolation) არის თვისება, რომელიც ხელს უწყობს არაპროგნოზირებადი შედეგების თავიდან აცილებას, როდესაც რამდენიმე ტრანზაქცია ერთდროულად მიმართავს მონაცემებს. იზოლაციის თვისება უზრუნველყოფს, რომ თითოეული ტრანზაქცია შესრულდება სხვებისგან დამოუკიდებლად, მაშინაც კი, თუ ისინი მუშაობენ ერთსადაიმავე მონაცემებზე.

გამძლეობა (Durability) არის თვისება, რომელიც უზრუნველყოფს, რომ ტრანზაქციის დასრულების შემდეგ მონაცემები შეინახება მონაცემთა ბაზაში და არ დაიკარგება ან დაზიანდება სისტემის შეფერხების შემთხვევაში. ტრანზაქცია უნდა შესრულდეს ისე, რომ მონაცემთა ბაზაში შეტანილი ცვლილებები შეინახოს მაშინაც კი, თუ სისტემა ავარიულია და საჭიროებს გადატვირთვას.

ეს თვისებები აუცილებელია მონაცემთა მთლიანობისა და თანმიმდევრულობის უზრუნველსაყოფად არა მხოლოდ მონაცემთა ბაზის ნორმალური მუშაობის დროს, არამედ სისტემის გაუმართაობის დროს.



SQL მონაცემთა ბაზები, როგორც წესი, იყენებენ ACID თვისებების კომპლექტს, რაც ნიშნავს, რომ ტრანზაქციები ერთად იმართება და ნებისმიერი ტრანზაქციის დაბრუნება მოხდება მონაცემთა ყველა შეერთებაში. ეს ნიშნავს, რომ თუ ტრანზაქცია ვერ მოხერხდება, ყველა ცვლილება უკან დაბრუნდება და მონაცემთა ბაზა დაუბრუნდება წინა მდგომარეობას. SQL მონაცემთა ბაზებში ტრანზაქციების მხარდაჭერა შესაძლებელია სპეციალური ენობრივი კონსტრუქციების მექანიზმების გამოყენებით (მაგალითად, SQL-ს აქვს BEGIN TRANSACTION, COMMIT და ROLLBACK ოპერატორები)

მეორეს მხრივ, NoSQL მონაცემთა ბაზები, როგორიცაა MongoDB ან Cassandra, როგორც წესი, არ იყენებენ ტრანზაქციებს, რადგან ისინი ფოკუსირებულია მონაცემთა დიდ მასშტაბებზე და დამუშავების სიჩქარეზე, ვიდრე ტრანზაქციის მხარდაჭერაზე. ამის ნაცვლად, NoSQL მონაცემთა ბაზები იყენებენ CAP თეორემას, რომელიც უზრუნველყოფს არჩევანი გაკეთდეს მონაცემთა თანმიმდევრულობას,

წვდომადობასა და დაყოფისადმი მდგრადობას შორის. ეს ნიშნავს, რომ ისინი, როგორც წესი, აძლევენ მომხმარებელს არჩევანს თანმიმდევრულობასა და ხელმისაწვდომობას შორის, და რომ ჩვენს აპლიკაციებს შეუძლიათ აირჩიონ ის, რაც მათ უფრო სჭირდებათ მოცემულ სიტუაციაში.

ამრიგად, SQL და NoSQL მონაცემთა ბაზებს აქვთ ტრანზაქციების განხორციელების სხვადასხვა მეთოდი. SQL მონაცემთა ბაზები აგებულია ტრანზაქციის მხარდაჭერის მოდელზე და მოიცავს ACID თვისებებს, ხოლო NoSQL მონაცემთა ბაზები ყოველთვის არ უჭერს მხარს ტრანზაქციებს. ამის ნაცვლად, NoSQL მონაცემთა ბაზები იყენებს CAP თეორემას, რათა დაეხმაროს მომხმარებლებს გადაწყვიტონ თანმიმდევრულობა და მონაცემთა ხელმისაწვდომობა.

SQL მონაცემთა ბაზების დასაყენებლად და სამართავად, ყველაზე გავრცელებული ინსტრუმენტებია SQL Management Studio (SQL Server-ისთვის), phpMyAdmin (MySQL-ისთვის) და pgAdmin (PostgreSQL-ისთვის). ისინი ადმინისტრატორებს საშუალებას აძლევს მართონ მონაცემთა ბაზა, შექმნან და შეცვალონ ცხრილები, ინდექსები, წარმოდგენები, პროცედურები და ტრიგერები, ასევე დააყენონ წვდომის უფლებები და დააკონფიგურირონ სხვადასხვა პარამეტრები.

NoSQL მონაცემთა ბაზების დასაყენებლად და სამართავად, ყველაზე გავრცელებული ინსტრუმენტებია MongoDB Compass, Cassandra Query Language Shell (CQLSH) და Neo4j Browser. ეს ხელსაწყოები ადმინისტრატორებს საშუალებას აძლევს მართონ მონაცემთა ბაზა, შექმნან და შეცვალონ კოლექციები (MongoDB-ის შემთხვევაში) ან ცხრილები (Cassandra-სა და Neo4j-ის შემთხვევაში), აწარმოონ მოთხოვნები და მართონ სხვადასხვა კონფიგურაციის პარამეტრები.

ძირითადი განსხვავებები SQL და NoSQL მონაცემთა ბაზების დაყენებასა და მართვაში დაკავშირებულია მათ განსხვავებულ სტრუქტურასთან. SQL მონაცემთა ბაზებს ხშირად სჭირდებათ მონაცემთა უფრო მკაცრი სქემები, რამაც შეიძლება შეზღუდოს ზოგიერთი ოპერაციების მოქნილობა, მაგალითად, ცხრილში ახალი სვეტების დამატება. ამის საპირისპიროდ, NoSQL მონაცემთა ბაზები, როგორც წესი, უზრუნველყოფს უფრო მოქნილ სქემებს, რაც საშუალებას გვაძლევს სწრაფად შევქმნათ და შევცვალოთ კოლექციები ან ცხრილები, გავხადოთ ისინი უფრო მასშტაბური და შესაფერი ზოგიერთი ტიპის აპლიკაციისთვის.

ძირითადი განსხვავება SQL და NoSQL ბაზებს შორის შეიძლება ჩამოყალიბდეს შემდეგი სახით:

მონაცემთა სქემა: SQL მონაცემთა ბაზას აქვს მონაცემთა მკაცრი სქემა, რომელიც განსაზღვრავს მონაცემთა ტიპებს და კავშირებს ცხრილებს შორის. NoSQL მონაცემთა ბაზებს არ აქვთ მონაცემთა მკაცრი სქემა.

მასშტაბურობა: SQL მონაცემთა ბაზებს აქვთ მასშტაბურობის შეზღუდვები, რამაც შეიძლება გახადოს ისინი არაეფექტური მონაცემთა დიდი მოცულობის

მართვაში. NoSQL მონაცემთა ბაზები ძალიან მასშტაბირებადია, რის გამოც მათ შეუძლიათ დიდი რაოდენობით მონაცემების დამუშავება.

მოთხოვნების მოქნილობა: SQL-ს აქვს ძალიან მძლავრი მოთხოვნების ენა, რაც მას საუკეთესო არჩევანს ხდის რთული მოთხოვნებისთვის, რომელიც მოიცავს ცხრილების დიდ რაოდენობას. მეორეს მხრივ, NoSQL-ს აქვს მარტივი მოთხოვნების ენა, რომელიც კარგად შეეფერება დიდი რაოდენობით მონაცემებთან დაკავშირებულ მოთხოვნებს.

დამუშავების სიჩქარე: მიუხედავად იმისა, რომ SQL ზოგადად უფრო ნელია ვიდრე NoSQL, მისი მძლავრი მოთხოვნის ენა საშუალებას იძლევა სწრაფად დამუშავებულ იქნას რთული მოთხოვნები. მეორეს მხრივ, NoSQL ძალიან სწრაფად მუშაობს არასტრუქტურირებული მონაცემებით დიდ მოცულობაში.

SQL - ის ძირითადი თავისებურებებია:

SQL მონაცემთა ბაზებს აქვთ მკაცრად განსაზღვრული მონაცემთა შენახვის ფორმატი, რაც მათ ყველაზე შესაფერის არჩევანს ხდის რთული დაკავშირებული მონაცემების წარმოსადგენად;

SQL-ს აქვს მოთხოვნების მძლავრი ენა, რომელიც საშუალებას იძლევა გავუმკლავდეთ კომპლექსურ მოთხოვნებს;

SQL მონაცემთა ბაზები მოითხოვს დიდ დანახარჯებს მომსახურებასა და გამართვაზე.

NoSQL - ის ძირითადი თავისებურებებია:

NoSQL მონაცემთა ბაზები ინახავს მონაცემებს დოკუმენტების სახით, რაც მათ საუკეთესო არჩევანს ხდის არასტრუქტურირებული მონაცემების შესანახად, როგორიცაა სოციალური მედია და ბლოგის მონაცემები;

NoSQL-ს აქვს მარტივი მოთხოვნის ენა, რომელიც საშუალებას იძლევა სწრაფად დავამუშავოთ მოთხოვნები არასტრუქტურირებული მონაცემების უზარმაზარ მოცულობებზე;

NoSQL მონაცემთა ბაზები სწრაფად მასშტაბირებისა და გაფართოვების საშუალებას იძლევა.

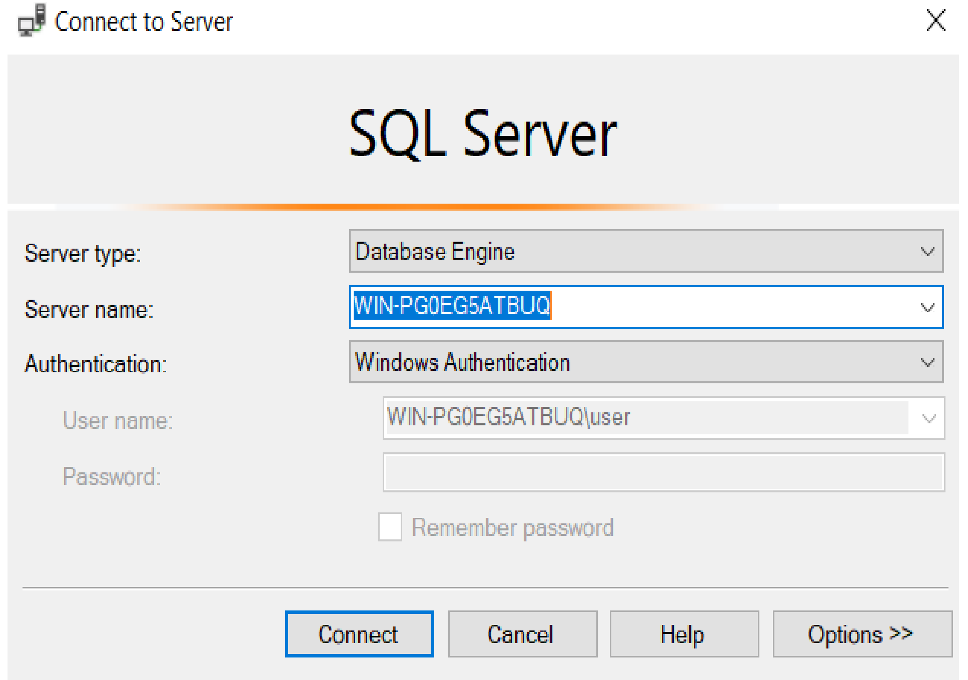
I თავი. SQL მონაცემთა ბაზებთან მუშაობა

1.1. SQL Server Management Studio

უტილიტა *Management Studio* განკუთვნილია შემდეგი ქმედებებისთვის:

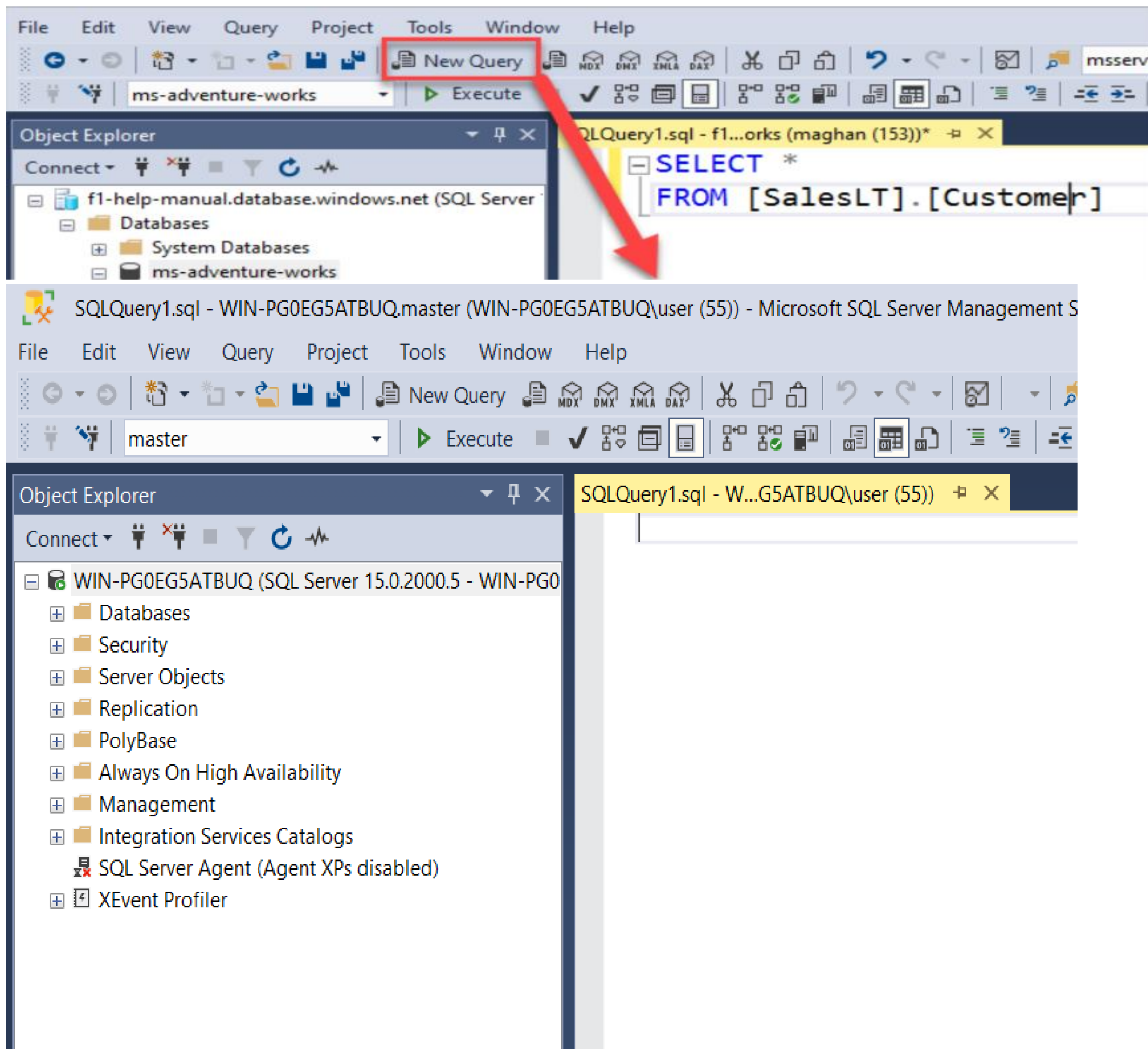
– MS SQL Server-ის აწყობათა მართვა;
– უსაფრთხოების სისტემის კონფიგურაცია;
– მონაცემთა ბაზების სტრუქტურებთან მუშაობა;
– დავალებათა განრიგის მიხედვით შესრულების მართვა;
– მიმდინარე აქტიურობის (მომხმარებელთა, ბლოკირებული ობიექტების, წარმადობის შესახებ ინფორმაციის) ჩვენება.

მუშაობის დაწყების წინ აუცილებელია სერვერთან მიერთება შემდეგი ინფორმაციის მითითებით: *Server Type*; *SQL Server*; *Authentication Type*.



მოთხოვნების რედაქტორი (Query Editor)

მონაცემთა ბაზაში ახალი მოთხოვნის ჩაწერისათვის სრულდება ბრძანება **New Query**, რომელიც განთავსებულია *Management Studio*-ს ინსტრუმენტების პანელზე. შედეგად გაიხსნება ახალი ჩანართი, რომელშიც შეიძლება კოდის ჩაწერა.



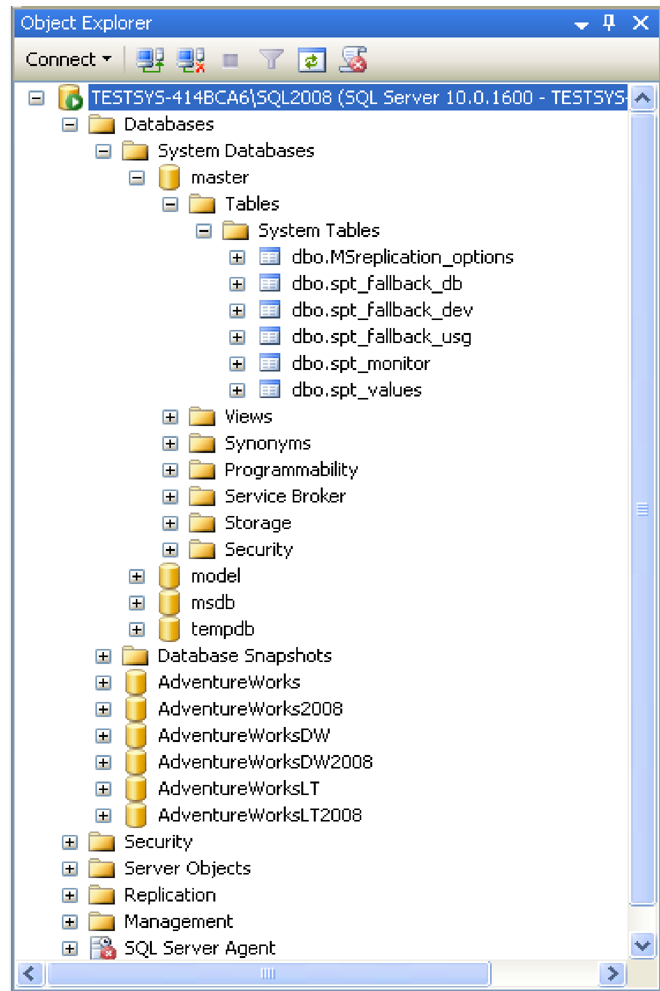
მოთხოვნის შესრულებისათვის აუცილებელია ბრძანების **Query – Execute (F5)** შესრულება, ხოლო მხოლოდ სინტაქსური ჩანაწერის სისწორის შესამოწმებლად კი ბრძანების **Query – Parse (Ctrl+F5)** შესრულება, რომლის დროსაც არ ხდება მოთხოვნის შესრულება.

Management Studio რამდენიმე ფანჯრის გახსნისა და ერთდროულად რამდენიმე მონაცემთა ბაზასთან მუშაობის საშუალებას იძლევა. ამასთან, თითოეულ ფანჯარაში დგება MS SQL Server-თან საკუთარი მიერთება, რომელიც აღწერილია *SQL Server Configuration Manage*-ში. ახალი ჩართვის შექმნისათვის გამოიყენება ბრძანება **File – New - Database Engine Query.**

მიმდინარე ჩართვის (შეერთების) მოთხოვნის შემცველობის გარე მატარებელზე შენახვა შესაძლებელია ბრძანებით **File – Save**.

Object Explorer

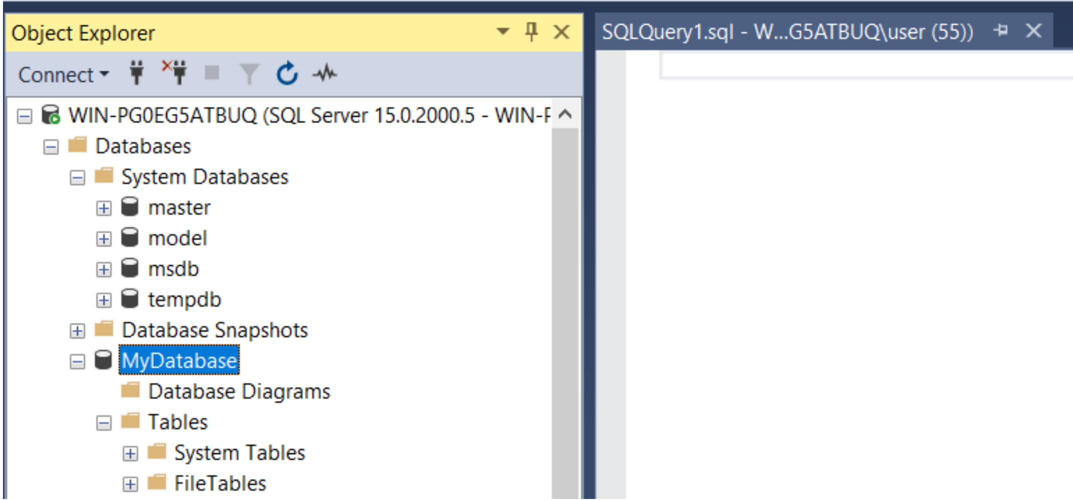
მონაცემთა ბაზაში ნავიგაციის (მისაწვდომი ობიექტების დათვალიერება, ცხრილების შემცველობის ნახვისათვის მოთხოვნათა შესრულება, ობიექტებისათვის სკრიპტების შექმნა და სხვ.) განხორციელებისათვის გამოიყენება Object Explorer.



შაბლონური მონაცემთა ბაზების (Sample Database) სია

ამ სიიდან შერჩეული მონაცემთა ბაზა გამოიყენება მოთხოვნათა რედაქტორში როგორც მონაცემთა ბაზა უსიტყვოდ. ამდენად, დიდი მნიშვნელობა ენიჭება საჭირო მონაცემთა ბაზის არჩევას, რაც შესაძლებელია ზემოხსენებული სიის მეშვეობით ან SQL ბრძანებით, მაგალითად:

```
USE MyDatabase
```



Reporting Services Configuration – გამოიყენება ანგარიშების სერვისის კონფიგურაციისათვის.

Bulk Copy Program – ბრძანებითი სტრიქონის უტილიტა განკუთვნილია დიდი მოცულობის ფორმატირებული მონაცემების გადასაცემად MS SQL სერვერისაკენ და მისგან.

SQL Server Profiler – ყველა ბრძანების რეალურ დროში შესრულების მიდევნების საშუალებას იძლევა.

sqlcmd – ბრძანებითი სტრიქონის უტილიტა SQL-სკრიპტების შესრულების საშუალებას იძლევა. იგი ბევრად უფრო ეფექტურია, ვიდრე *Management Studio*, როცა მომხმარებლის გრაფიკული ინტერფეისი არ არის საჭირო.

SQL Server Integration Services (SSIS) – ნებისმიერი წყაროდან ახდენს მონაცემთა ამოღებას OLE DB (Object Linking and Embedding Database) მექანიზმის ან .NET მონაცემთა პროვაიდერების მეშვეობით და ათავსებს მათ MS SQL სერვერის ცხრილებში.

SQL Server Business Intelligence Development Studio – რომელიც წარმოადგენს Visual Studio-ს განსაკუთრებულ ვერსიას, ქმნის პაკეტებს სერვისებისათვის *Integration Services*, *Reporting Services* და მუშაობს *Analysis Services* პროექტებთან.

MS SQL Server-ის კონფიგურაცია

MSSQLServer სამსახურის მუშაობის კონფიგურირება შეიძლება შესრულდეს სპეციალური შენახვადი პროცედურით, რომელიც სრულდება *Management Studio* უტილიტაში ან გრაფიკული ხერხით ამავე უტილიტის საშუალებებით.

Management Studio-ს დახმარებით ამ სამსახურის პარამეტრების ცვლილებისათვის აუცილებელია საჭირო სერვერის შერჩევა და კონტექსტურ

მენიუში **Properties** ბრძანების შესრულება და გამოჩენილ დიალოგურ ფანჯარაში სერვერის კონფიგურირების შესრულება.

General ჩანართში აისახება სისტემის შესახებ ძირითადი ცნობები: ოპერაციული სისტემის ვერსია. მეხსიერების მოცულობა, პროცესორების რაოდენობა და სხვ., აგრეთვე სერვერის სამსახურების გაშვების პარამეტრები. ***Memory*** ჩანართი MS SQL Server მოქმედებისათვის გამოყოფილი მეხსიერების მართვის (მეხსიერების დინამიური მართვა ან ფიქსირებული ზომის დაყენება) საშუალებას იძლევა. ***Processors*** ჩანართის საშუალებით ხდება იმ პროცესორების მართვა, რომელზედაც შეიძლება შესრულდეს SQL მოთხოვნები. ***Security*** ჩანართის მეშვეობით განისაზღვრება მომხმარებელთა აუტენტიფიკაციის ტიპი, აგრეთვე სერვერთან წვდომის აუდიტის პარამეტრები. ***Connections*** ჩანართი სერვერთან კლიენტური შეერთებების კონფიგურირების საშუალებას იძლევა. თუ პარამეტრი 0-ის ტოლია, მაშინ ნებადართულია მომხმარებელთა მაქსიმალური რაოდენობის – 32767 ჩართვა. ***Database Settings*** ჩანართის დახმარებით ხდება ახლად შექმნილი მონაცემთა ბაზების აწყობაზე მითითება: ინდექსების და სარეზერვო კოპირებების მოწყობილობებთან მუშაობის პარამეტრები, მონაცემთა ბაზების აღდგენის დრო. ***Advanced*** ჩანართი შეიცავს ზოგიერთ ცნობებს სერვერის ზოგადი დაყენების შესახებ. ***Permissions*** ჩანართი შესვლისა და როლების სახელების, აგრეთვე MS SQL Server-ში მოქმედებათა შესრულების უფლებების მართვის საშუალებას იძლევა.

სისტემური მონაცემთა ბაზები

MS SQL სერვერს აქვს ოთხი სისტემური მონაცემთა ბაზა: master; msdb; model; tempdb.

master შეიცავს სერვერის მთელ სისტემურ ინფორმაციას, აგრეთვე, ინფორმაციას სხვა მონაცემთა ბაზებისა და მათი პირველადი ფაილების ადგილმდებარეობის შესახებ. master მონაცემთა ბაზა შემდეგი ფაილებისაგან შედგება: Master.mdf (მონაცემების ფაილი) და Mastlog.ldf (ტრანზაქციების ჟურნალის ფაილი). ისინი ინახება \Data კატალოგში (C:\Program Files\Microsoft SQL Server\MSSQL.3\MSSQL\Data\). დაუშვებელია master სისტემურ მონაცემთა ბაზაში მომხმარებლის ობიექტების შექმნა.

model გამოიყენება როგორც შაბლონი ნებისმიერი ახალი მონაცემთა ბაზის შექმნის დროს. model მონაცემთა ბაზა მოთავსებულია \Data კატალოგში და ორი ფაილისგან შედგება: model.mdf (მონაცემების ფაილი) და model.ldf (ტრანზაქციების ჟურნალი).

msdb მონაცემთა ბაზა გამოიყენება *SQL Server Agent* სერვისის მიერ მოვლენების (alerts), ამოცანებისა (jobs) და ოპერატორების (operators) რეგისტრირების დაგეგმვისათვის. msdb ინახავს მთელ ინფორმაციას, რომელიც ეხება ადმინისტრირების ავტომატიზებასა და სერვერის მართვას. იგი შედგება ორი

ფაილისგან: msdbdata.mdf (მონაცემების ფაილი) და msdblog.ldf (ტრანზაქციების ჟურნალი).

tempdb შეიცავს დროებით ობიექტებს, როგორიცაა ცხრილები, შენახვადი პროცედურები, ცვლადები, კურსორები და ა.შ. მასში, ინახება, როგორც სისტემური, ისე მომხმარებლის მიერ შექმნილი ობიექტები. სერვერის ყოველი გაშვების დროს tempdb მონაცემთა ბაზა ხელახლა იქმნება და ყოველთვის იშლება სერვერის გაჩერების დროს.

შესაქმნელი დროებითი ობიექტები შეიძლება იყოს როგორც ლოკალური, ისე გლობალური. ლოკალური ობიექტი მისაწვდომია მხოლოდ მისი შემქმნელი მომხმარებლისთვის, გლობალური. კი - ყველა მომხმარებლისათვის. ლოკალური ობიექტები მოქმედებენ მხოლოდ სეანსის, შენახვადი პროცედურის, ტრიგერის ან ბრძანებების პაკეტის ფარგლებში. დროებითი ობიექტის შემქმნელი სტრუქტურიდან გამოსვლისას ეს ობიექტი მაშინვე წაიშლება, როგორც კი მომხმარებელი დაასრულებს სერვერთან მუშაობას. **tempdb** ორი ფაილისაგან შედგება: tempdb.mdf (მონაცემების ფაილი) და templogldf (ტრანზაქციების ჟურნალი).

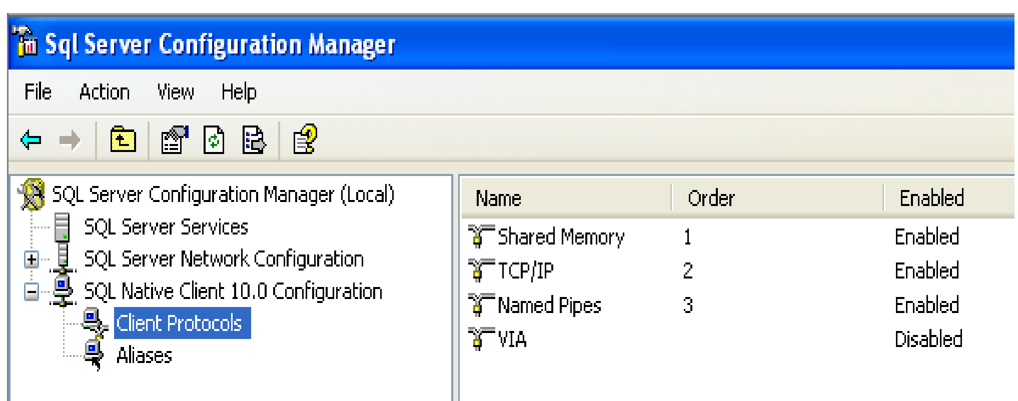
SQL Server Configuration უტილიტას შესწავლა.

1. სერვერზე გაუშვით უტილიტა *SQL Server Configuration Manager* და მისი მეშვეობით განვსაზღვრეთ სერვერზე გაშვებული სერვისების სია. ჩაწერეთ ეს სია ანგარიშში.

2. სერვერზე *Services* უტილიტის საშუალებით განსაზღვრეთ MS SQL სერვერის სერვისების გაშვების პარამეტრები და ჩაწერეთ ისინი ანგარიშში.

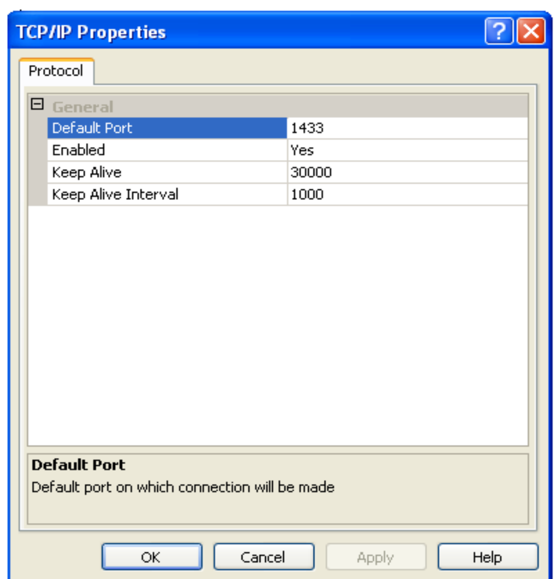
3. განსაზღვრეთ რომელი ქსელური ბიბლიოთეკების საშუალებით არის შესაძლებელი MS SQL სერვერთან შეერთება და რომელი ბიბლიოთეკები არიან აქტიური მოცემული მომენტისათვის. ჩაწერეთ ეს ინფორმაცია ანგარიშში.

4. SQL Server Configuration Manager–ის დახმარებით განსაზღვრეთ რომელი ქსელური ბიბლიოთეკების საშუალებით არის შესაძლებელი კლიენტის MS SQL სერვერთან შეერთება. ჩაწერეთ სია ანგარიშში.



სერვერის ფსევდონიმის შექმნა კლიენტური შეერთებისათვის

SQL Server Configuration Manager უტილიტას *SQL Native Client Configuration – Client Protocols* კვანძში უნდა დაემატოს ქსელური ბიბლიოთეკების კონფიგურაცია, მიეთითოს სერვერის ლოგიკური სახელი, სიიდან შეირჩეს აუცილებელი ბიბლიოთეკა და განისაზღვროს პარამეტრები. ქვემოთ მოყვანილია TCP/IP პროტოკოლის თვისებები.



მითითებანი შესრულებაზე:

1. განსაზღვრეთ და ჩაწერეთ ანგარიშში კომპიუტერის ქსელური სახელი, რომელზეც დაყენებულია MS SQL სერვერი.
2. კონტექსტურ მენიუში კვანძისათვის *SQL Native Client Configuration – Aliases* შეასრულეთ ბრძანება **New alias…** *MyServ*-ში დააყენეთ *Alias Name შეარჩიეთ პროტოკოლი* TCP/IP, ველში *Server* მიუთითეთ ადრე განსაზღვრული სახელი.

SQL სერვერთან შეერთების დაყენება

კლიენტურ კომპიუტერზე (სამუშაო სადგური) გაუშვით *SQL Server Management Studio* და სიიდან შეარჩიეთ სერვერის ლოგიკური სახელი *MyServ*, რომელიც წინა დავალებაში იყო განსაზღვრული *SQL Server Configuration Manager* უტილიტის მეშვეობით. თუ საჭირო სერვერი არ არის სერვერების სიაში, მაშინ ირჩევთ **<Browse for more…>** და შემდეგ იპოვით საჭირო სერვერს, რომელთანაც შეიძლება შეერთება.

1. მიუერთდით *MyServ* სერვერს აუტენტიფიკაციის საშუალებათა გამოყენებით: სააღრიცხვო ჩანაწერი «**sa**», პაროლი არ არის.
2. ახალი მოთხოვნის ჩაწერისათვის აუცილებელია *SQL Server Management Studio* ინსტრუმენტების პანელზე განთავსებული ბრძანების **New Query** შესრულება. შედეგად ჩნდება ახალი ჩანართი შესაბამისი შესაძლებლობებით.
3. ბრძანების **SELECT @@version** დახმარებით განსაზღვრეთ და ანგარიშში

ჩაწერეთ ინფორმაცია MS SQL სერვერის გამოყენებული ვერსიისა და ოპერაციული სისტემის შესახებ.

4. *Object Explorer* პანელის მეშვეობით განსაზღვრეთ სისტემის მიერ მხარდაჭერილი მონაცემთა ბაზების სახელები და სერვერის სისტემური მონაცემთა ბაზები. შეიტანეთ ეს ინფორმაცია ანგარიშში.

MS SQL სერვერის კონფიგურაციის პარამეტრების შესწავლა

1. *Management Studio*–ს მეშვეობით სერვისების პარამეტრების ცვლილებისათვის აუცილებელია *Object Explorer*–ში საჭირო სერვერის პოვნა, ხოლო კონტექსტურ მენიუში ბრძანების **Properties** შერჩევა**.** შედეგად ჩნდება დიალოგური ფანჯარა, რომელშიც შესაძლებელია ყველა აუცილებელი პარამეტრის აწყობა.
2. ასახეთ *MyServ* სერვერის პარამეტრების სია. თითოეულ ჩანართში აისახება ინფორმაცია, რომელიც წარმოადგენს შესაბამისი თვისებების აწყობის საფუძველს.
3. განსაზღვრეთ და ჩაწერეთ სერვერის ფესვის კატალოგის ანგარიშში სისტემაში პროცესორების რაოდენობა, მომხმარებელთა აუტენტიფიკაციის ტიპი და სერვერის მიერ მხარდაჭერილი მომხმარებლების მაქსიმალური რაოდენობა.
4. შეისწავლეთ MS SQL სერვერის დანარჩენი თვისებები, რომლებიც ხელმისაწვდომნი არიან ამ დიალოგში.

1.2. მუშაობა SQL Server Management Studio-ში

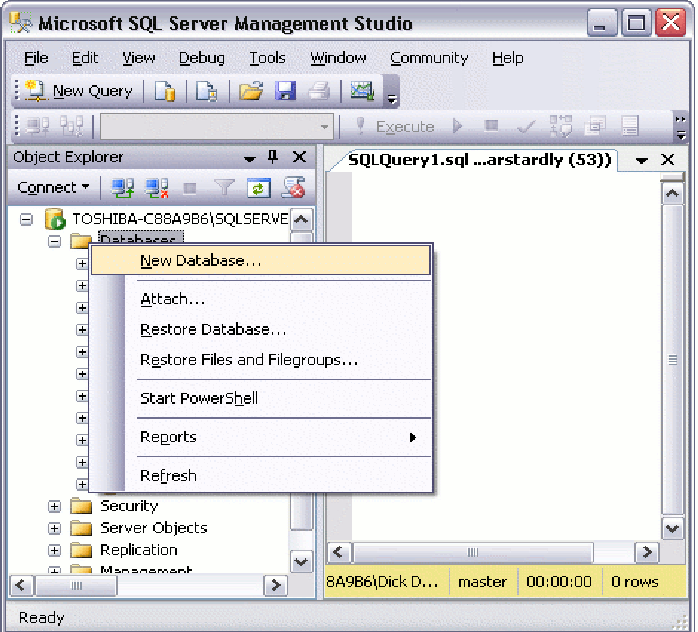
SQL Server Management Studio (SSMS) წარმოადგენს ერთ-ერთ მთავარ ინსტრუმენტს მონაცემთა ბაზებთან სამუშაოდ.

მარცხენა პანელი შეიცავს Object Explorer-ს, რომელიც უზრუნველყოფს მონაცემთა ბაზების ობიექტებს შორის ნავიგაციას. მარჯვენა პანელი შესაბამისი მონაცემთა ბაზის მოთხოვნათა დაწერისა და შედეგების ნახვის საშუალებას იძლევა.

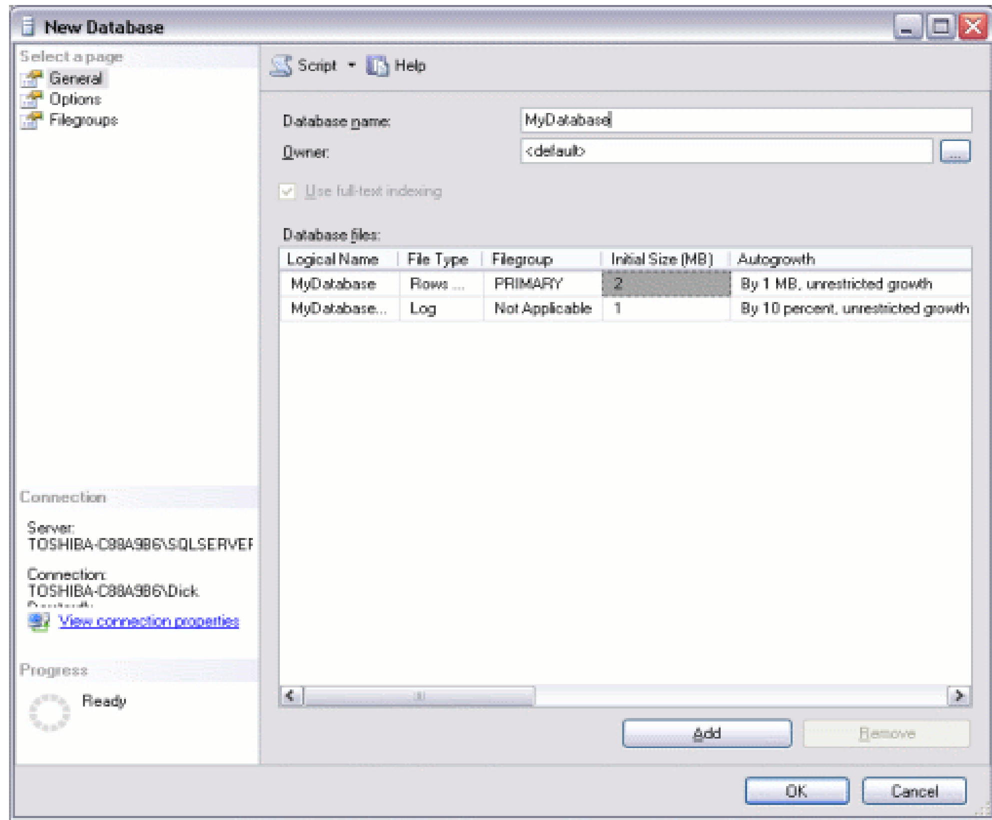
1. **მონაცემთა ბაზის შექმნა**

SSMS-ის გამოყენებით ახალი მონაცემთა ბაზის შექმნისათვის საჭიროა შემდეგი ბიჯების გაკეთება:

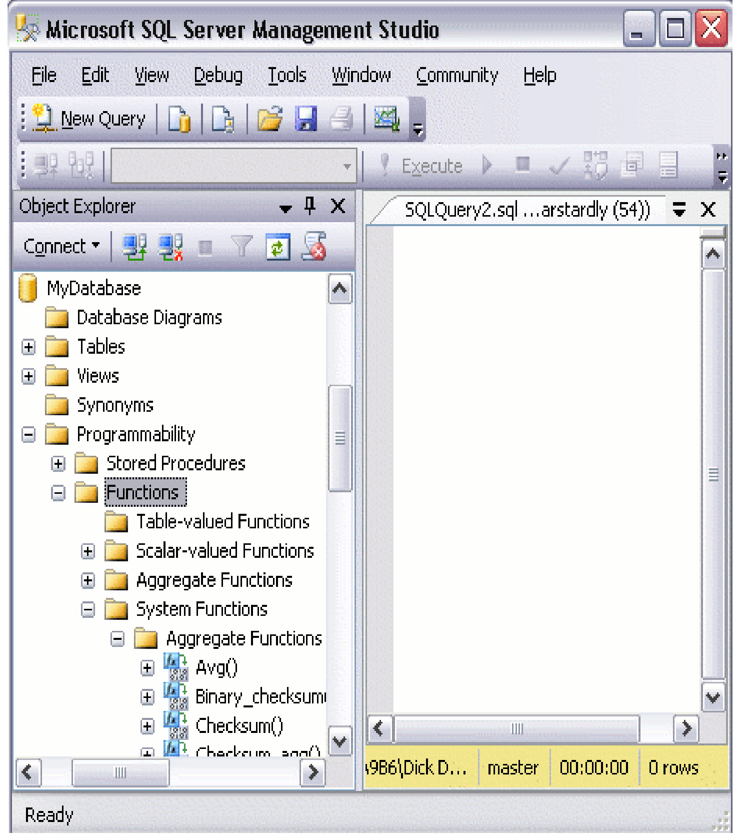
1. მარჯვენა click იკონაზე "Databases" და ვირჩევთ "New Database..."



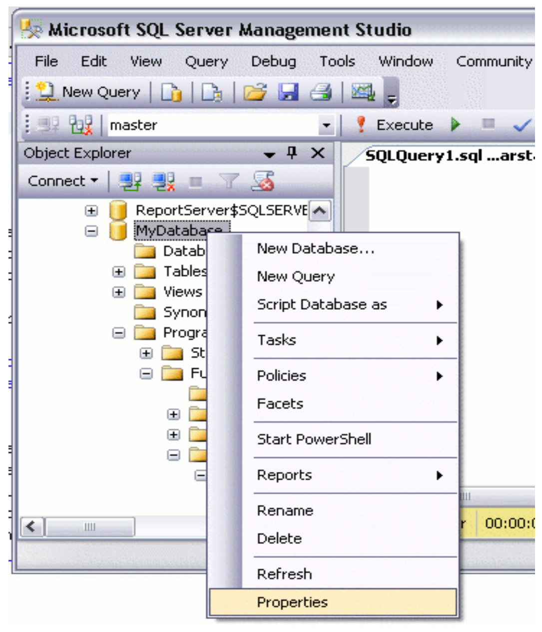
2. მონაცემთა ბაზას ვარქმევთ სახელს და click "OK"–ზე.

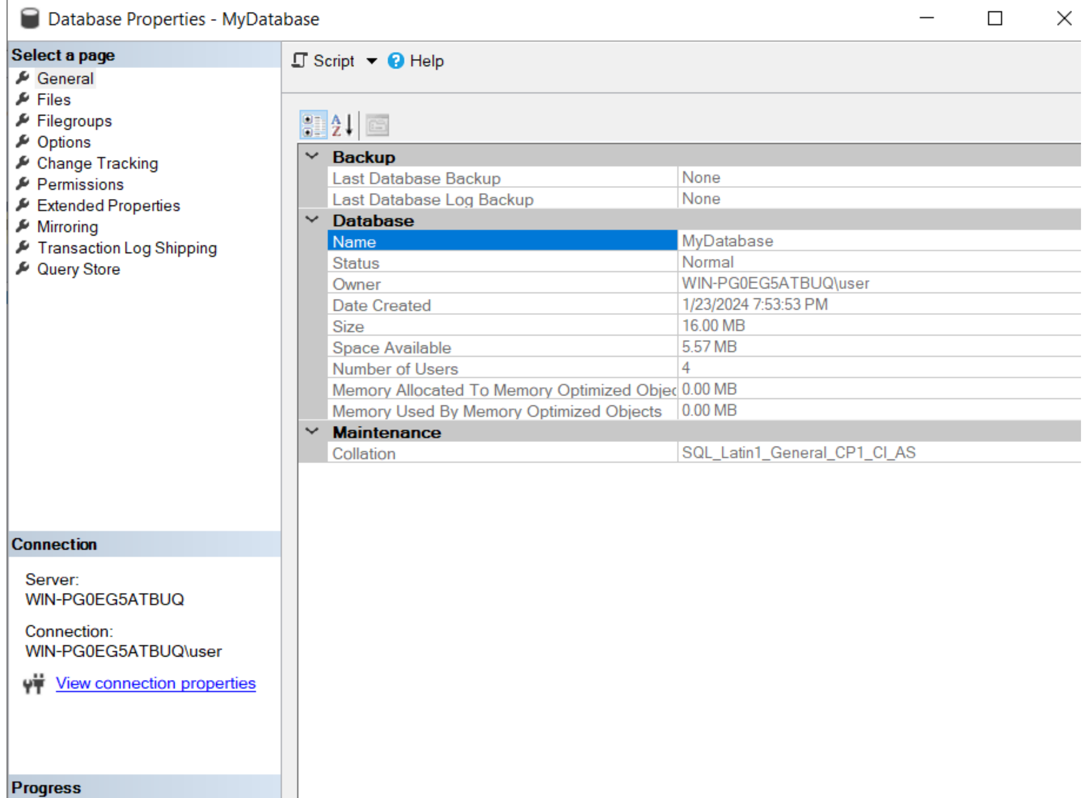


თქვენი მონაცემთა ბაზა განთავსდება "Model" სისტემურ მონაცემთა ბაზაში, რომელიც ახალი ბაზის შექმნის დროს გამოიყენება როგორც ნიმუში. თუ გავხსნით მარცხენა პანელს და გავშლით ხეს, ვნახავთ, რომ თქვენი მონაცემთა ბაზა (მაგალითად, MyDatabase) უკვე შეიცავს გარკვეულ ობიექტებს.



მონაცემთა ბაზა შეიძლება შეიქმნას უსიტყვო (default) ოპციების გამოყენებით, კერძოდ, "Data File" და "Transaction Log" ავტომატურად იქმნება სერვერზე მისი სტანდარტული ადგილმდებარეობის (default location) მიხედვით. თუმცა სურვილის შემთხვევაში შესაძლებელია მათი განსხვავებული ადგილსამყოფელისა თუ სხვა პარამეტრების განსაზღვრაც. მონაცემთა ბაზის თვისებების ცვლილებებისათვის უბრალოდ საჭიროა right click ბაზაზე და "Properties"-ის არჩევა.

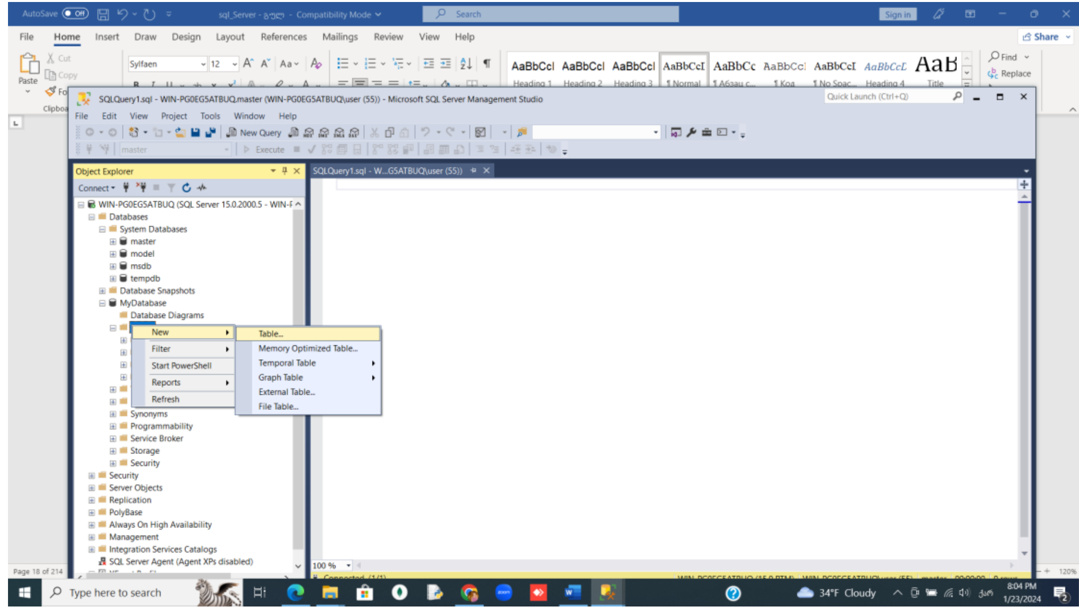




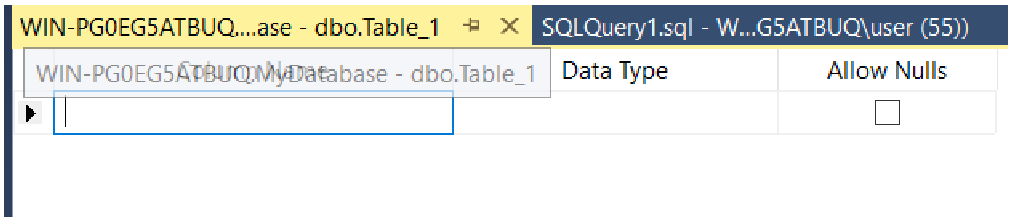
1. **ცხრილების შექმნა**

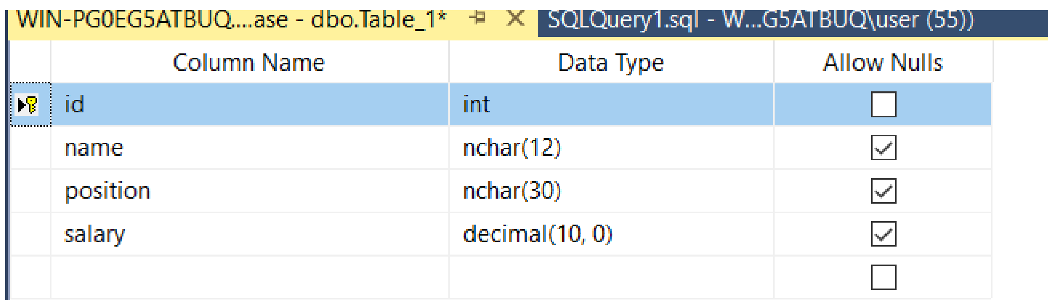
ცხრილის შექმნისათვის საჭიროა შემდეგი ბიჯების გაკეთება.

– მარჯვენა click იკონაზე"Tables" და ვირჩევთ "New Table..."–ს.



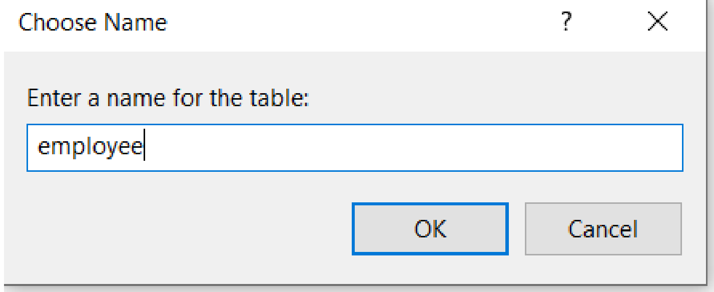
– ვიდრე ეკრანი გახსნილია, სრულად უნდა შეივსოს სვეტები "Column Name", "Data Type" და "Allow Nulls".



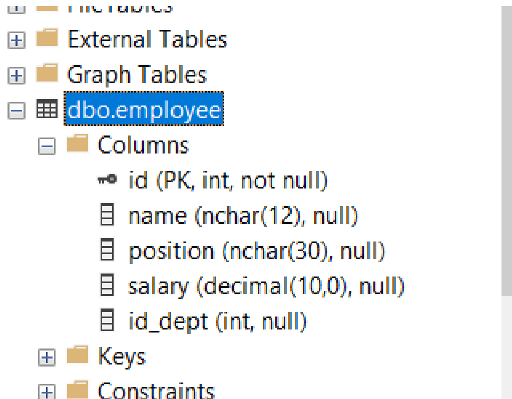


– შევინახოთ შექმნილი ცხრილი ქმედებით *File > Save Table_1*

– ვაქმევთ სახელს შექმნილს ცხრილს :



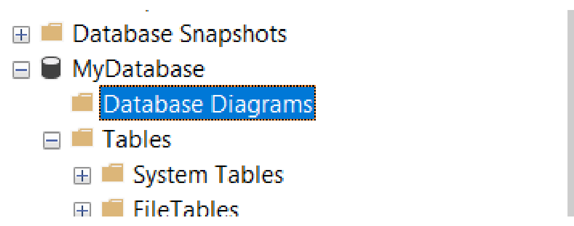
შექმნილი ცხრილი მასში შემავალი სვეტებით აისახება მონაცემთა ბაზის სექციაში "Tables".

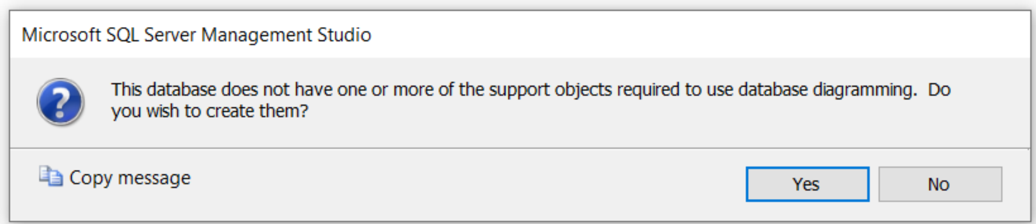


3. დიაგრამის შექმნა

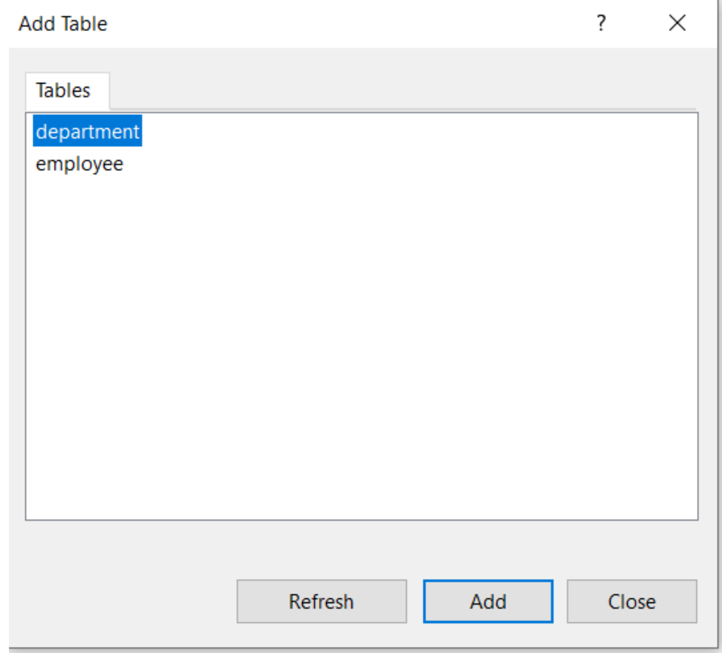
მონაცემთა ბაზის დიაგრამის შექმნისათვის საჭიროა შემდეგი მოქმედებების შესრულება:

1. **Database Diagrams-**ზე მარჯვენა click. ვირჩევთ **New Database Diagram.**

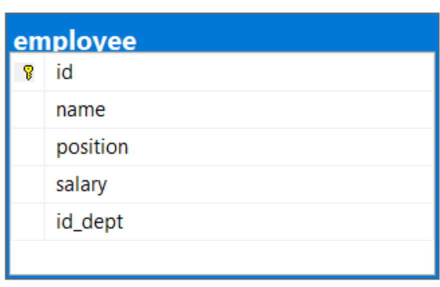


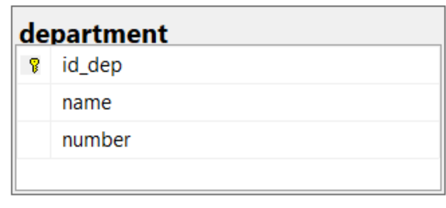


2. მოვნიშნავთ ცხრილს და ვაჭერთ **Add:**



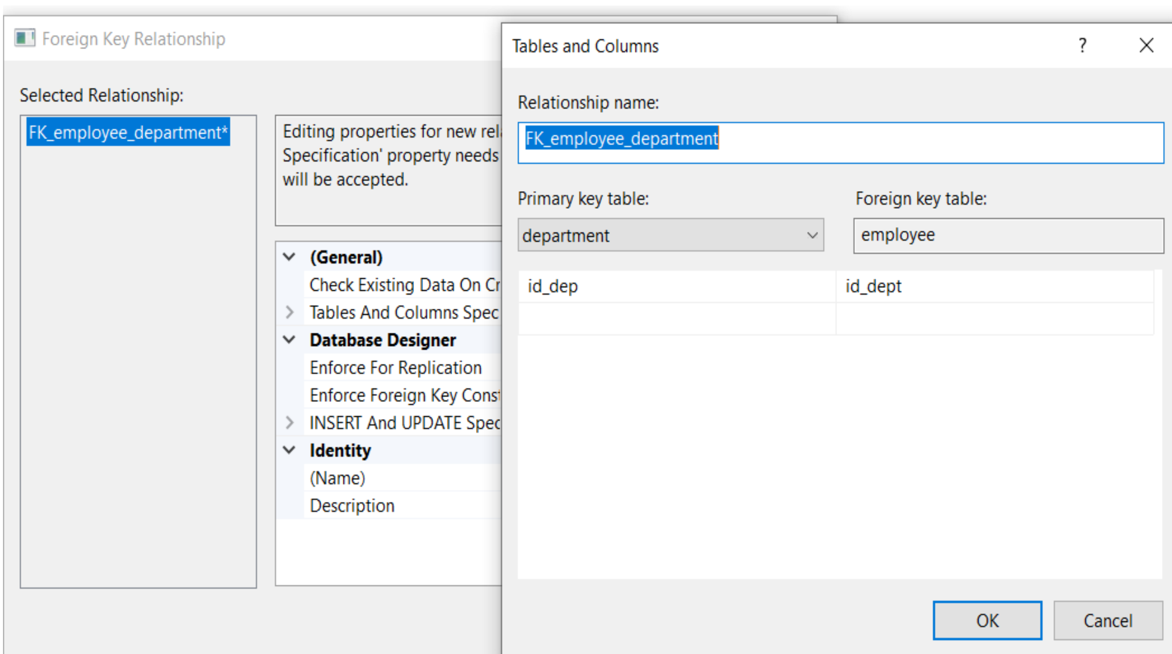
3. ცხრილები განთავსდება ფანჯარაში:



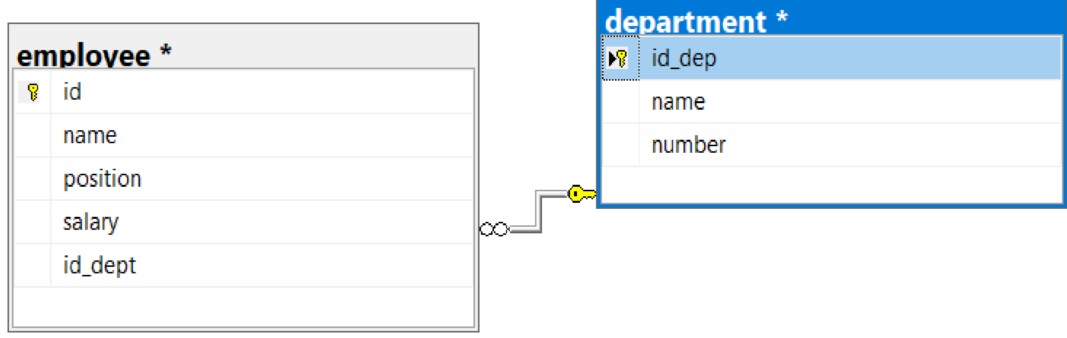


4. ცხრილების კავშირების ასახვისათვის ვამყარებთ კავშირებს პირველად და შესაბამის გარე გასაღებურ ველებს შორის.

5. ჩნდება დიალოგური ფანჯარა **Foreign Key Relationships,** სადაც ხდება კავშირების თვისებების აწყობა.



6. ვხურავთ ფანჯარას. ვაჭერთ ღილაკს Yes, ვინახავთ დიაგრამას.

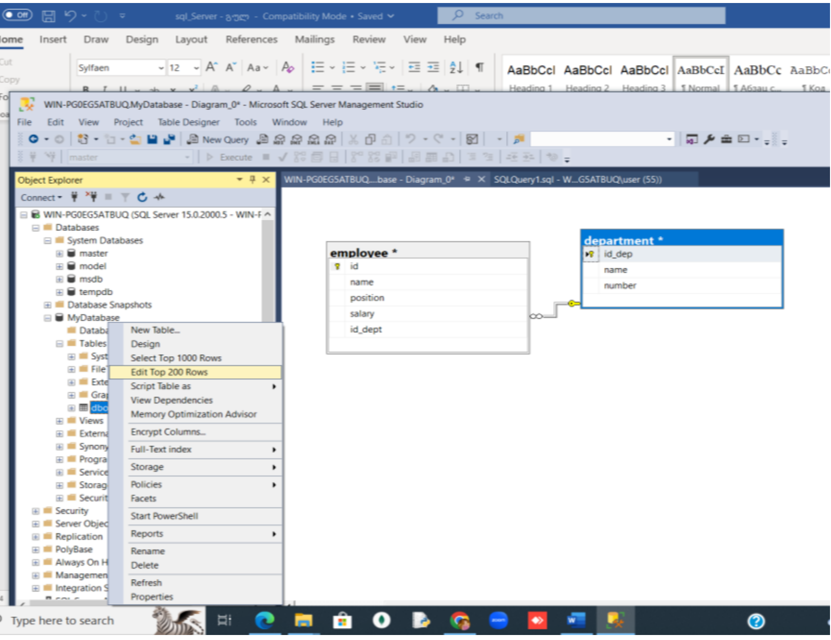


ზემოაღნიშნული თანამიმდევრობით ერთი მონაცემთა ბაზისათვის შესაძლებელია მრავალი დიაგრამის შექმნა.

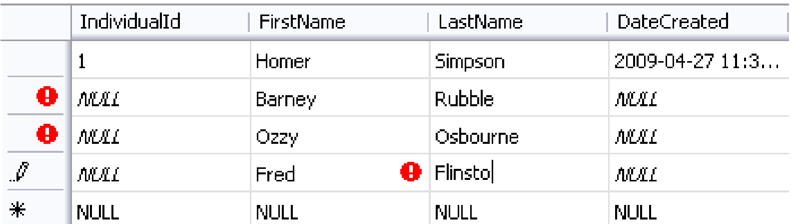
სტრიქონების რედაქტირება

ცხრილში მონაცემთა დამატებისათვის შეგვიძლია გამოვიყენოთ ოპცია "Edit Top 200 Rows".

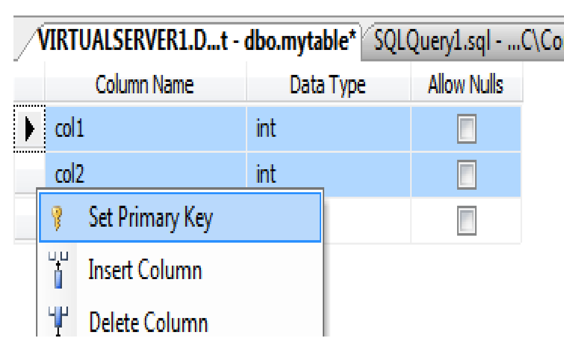
1. ამ ოპციის გამოსაყენებლად მარჯვენა click საჭირო ცხრილზე და ვირჩევთ ბრძანებას "Edit Top 200 Rows":



2. მონაცემები უშუალოდ შეგვაქვს ცხრილში.



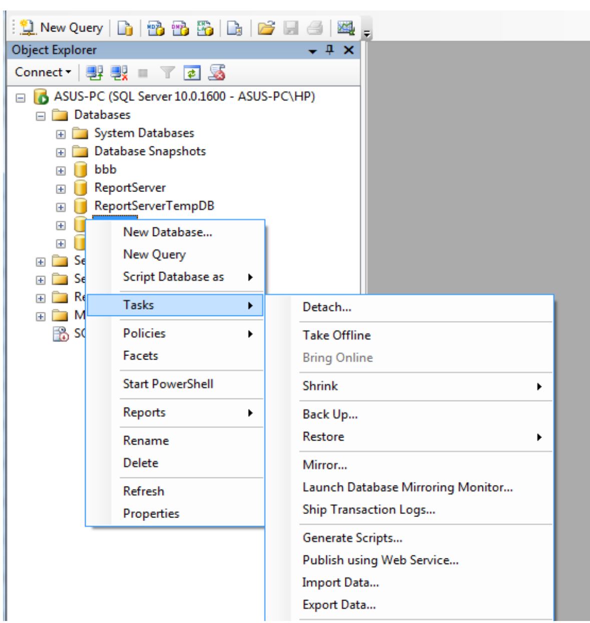
	IndividualId	FirstName	LastName	DateCreated
	1	Homer	Simpson	2009-04-27 11:3...
	NULL	Barney	Rubble	NULL
	NULL	Ozzy	Osbourne	NULL
	NULL	Fred	Flinsto	NULL
*	NULL	NULL	NULL	NULL



1. Open the design table tab
2. Highlight your two INT fields
3. Right click -> Set primary key

მონაცემთა ბაზის გამორთვა

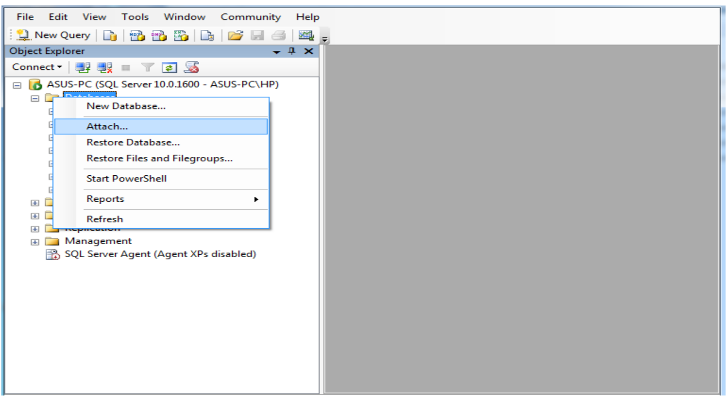
1. გავხსნათ **Databases** და ვირჩევთ მონაცემთა ბაზას, რომლის გამორთვაც გვინდა. თუ მონაცემთა ბაზა ამ დროისათვის გახსნილია, მაშინ მომხმარებელს ეკრძალება მასთან წვდომა.
2. Right-click მონაცემთა ბაზის სახელზე, პუნქტზე **Tasks** და click **Detach.**



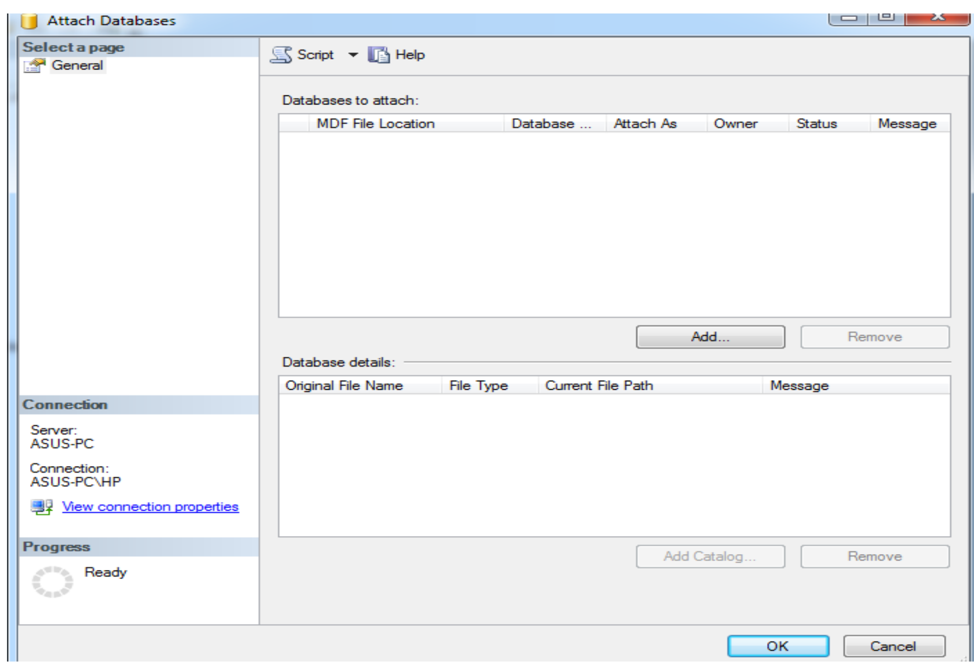
3. ჩნდება **Detach Database** დიალოგური ფანჯარა, სადაც **Databases to detach** ბადეში, **Database Name** სვეტში ვირჩევთ მონაცემთა ბაზას და click **OK**.

- **მონაცემთა ბაზის მიერთება**

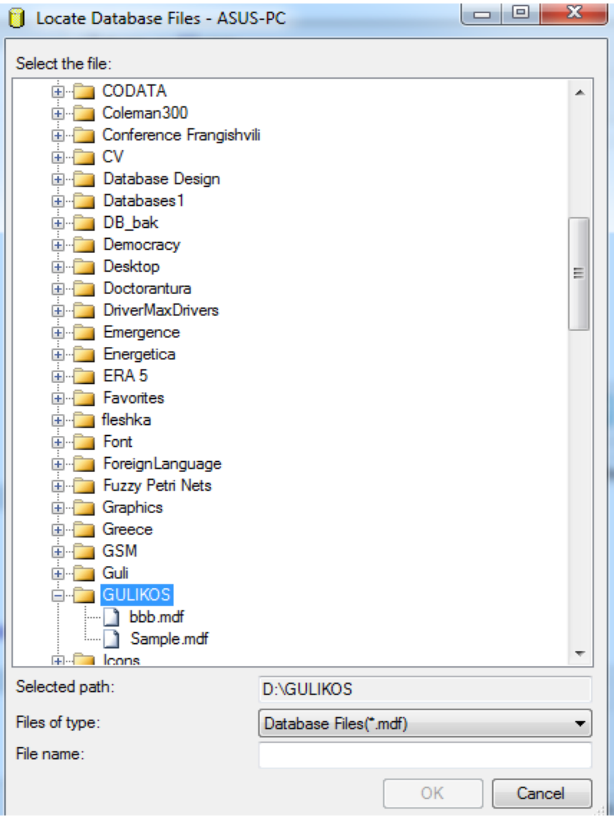
1. Right-click **Databases** და click **Attach**.



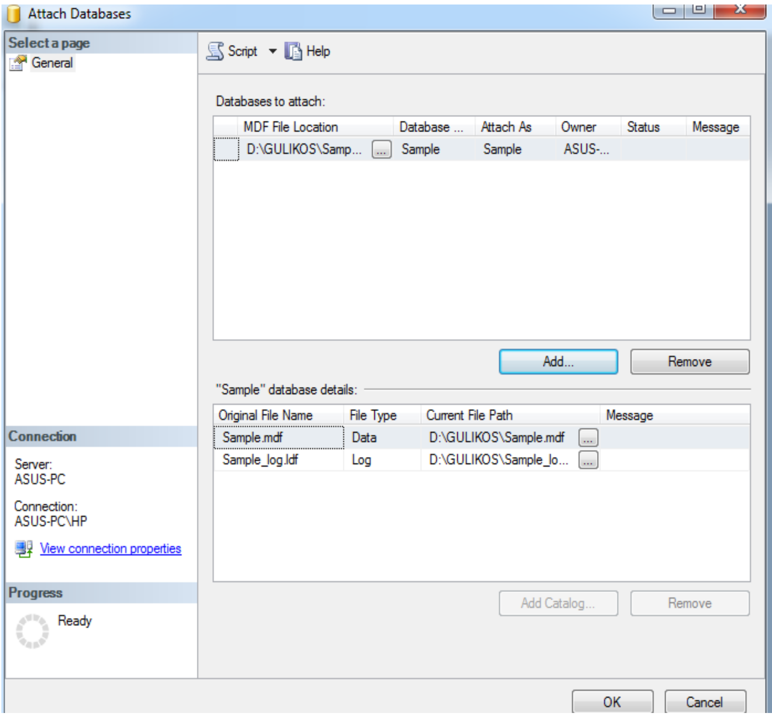
2. **Attach Databases** დიალოგურ ფანჯარაში click **Add**



3. **Locate Database Files** დიალოგურ ფანჯარაში ვირჩევთ დისკს, დირექტორიას, სადაც მოთავსებულია სასურველი მონაცემთა ბაზა. შემდეგ ვირჩევთ თვითონ მონაცემთა ბაზას.



4. **Attach Databases** ფანჯარაში ჩნდება სასურველი მონაცემთა ბაზა და click **OK**.



Sql ბრძანებები

DDL - Data Definition Language:

ბრძანება	აღწერა
CREATE	ახალი ცხრილის, წარმოდგენის ან მონაცემთა ბაზის სხვა ობიექტის შექმნა
ALTER	არსებული მონაცემთა ბაზის ობიექტის მოდიფიცირება, მაგალითად, როგორიცაა ცხრილი
DROP	ცხრილის, წარმოდგენის ან სხვა ობიექტის წაშლა

DML - Data Manipulation Language

ბრძანება	აღწერა
INSERT	ცხრილში სტრიქონის ჩასმა
UPDATE	სტრიქონის მოდიფიცირება
DELETE	სტრიქონის წაშლა

DCL - Data Control Language:

ბრძანება	აღწერა
GRANT	იძლევა გამოყენების პრივილეგიას
REVOKE	აბრუნებს მინიჭებულ პრივილეგიას

DQL - Data Query Language:

ბრძანება	აღწერა
SELECT	ამოარჩევს ჩანაწერებს ერთი ან რამდენიმე ცხრილიდან

რა არის RDBMS?

RDBMS არის რელაციური მონაცემთა ბაზების მართვის სისტემა. RDBMS -ში მონაცემები ინახება ობიექტებში, რომელსაც ეწოდება ცხრილები. ცხრილი არის ერთმანეთთან დაკავშირებული მონაცემების ჩანაწერების ნაკრები, რომელიც შედგება სვეტებისა და სტრიქონებისგან. ცხრილი არის მონაცემების შენახვის ყველაზე გავრცელებული და მარტივი ფორმა რელაციურ მონაცემთა ბაზაში. მაგალითად, განვიხილოთ კლიენტების ცხრილი:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

რა არის ველი

ყველა ცხრილი დაყოფილია უფრო მცირე ერთეულებად, რომელსაც ველები ეწოდება. CUSTOMERS ცხრილის ველები შედგება ID-ისგან,
სახელი, ასაკი, მისამართი და ხელფასი.
ველი არის ცხრილის სვეტი, რომელიც შექმნილია ცხრილის ყველა ჩანაწერის შესახებ სპეციფიკური ინფორმაციის შესანახად.

რა არის ჩანაწერი ან სტრიქონი

ჩანაწერი, რომელსაც ასევე უწოდებენ სტრიქონს, არის თითოეული ინდივიდუალური ჩანაწერი, რომელიც არსებობს ცხრილში. მაგალითად CUSTOMERS ცხრილში არის 7 ჩანაწერი. ერთი ჩანაწერი არის:

```
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
+----+----------+-----+-----------+----------+
```

რა არის სვეტი

სვეტი არის ვერტიკალური ობიექტი ცხრილში, რომელიც შეიცავს ცხრილის კონკრეტულ ველთან დაკავშირებულ ყველა ინფორმაციას. მაგალითად, CUSTOMERS ცხრილის სვეტი არის ADDRESS, რომელიც წარმოადგენს მდებარეობის აღწერას და შედგება შემდეგისგან:

```
+-----------+
| ADDRESS   |
+-----------+
| Ahmedabad |
| Delhi     |
| Kota      |
| Mumbai    |
| Bhopal    |
| MP        |
| Indore    |
+----+------+
```

რა არის NULL მნიშვნელობა?

NULL მნიშვნელობა ცხრილში არის მნიშვნელობა ველში, რომელიც ცარიელია, რაც ნიშნავს, რომ ველი NULL მნიშვნელობით არის ველი მნიშვნელობის გარეშე.
ძალიან მნიშვნელოვანია გვესმოდეს, რომ NULL მნიშვნელობა განსხვავდება ნულოვანი მნიშვნელობისგან ან ველისგან, რომელიც შეიცავს „ხარვეზებს“ (Space). NULL მნიშვნელობის მქონე ველი არის ისეთი ველი, რომელიც ცარიელი დარჩა ჩანაწერის შექმნისას.

SQL შეზღუდვები:

შეზღუდვები არის წესები, რომლებიც გამოიყენება მონაცემთა სვეტებზე ცხრილში. ისინი გამოიყენება მონაცემთა ტიპების შესაზღუდად, რომლებიც შეიძლება გადაეცეს ცხრილს. ეს უზრუნველყოფს მონაცემთა ბაზაში არსებული მონაცემების სიზუსტეს და სანდოობას.
შეზღუდვები შეიძლება იყოს სვეტის დონე ან ცხრილის დონე. სვეტის დონის შეზღუდვები გამოიყენება მხოლოდ ერთ სვეტზე, მაშინ როცა
ცხრილის დონის შეზღუდვები გამოიყენება მთელ ცხრილზე.
ქვემოთ მოცემულია გამოყენებული შეზღუდვები, რომლებიც ხელმისაწვდომია SQL-ში:

– NOT NULL შეზღუდვა: უზრუნველყოფს, რომ სვეტს არ ჰქონდეს NULL მნიშვნელობა.

- DEFAULT შეზღუდვა: უზრუნველყოფს სვეტის ნაგულისხმევ მნიშვნელობას, როდესაც არცერთი არ არის მითითებული.
- UNIQUE შეზღუდვა: უზრუნველყოფს, რომ სვეტის ყველა მნიშვნელობა განსხვავებულია.
- PRIMARY გასაღები: ცალსახად იდენტიფიცირებას უკეთებს თითოეულ სტრიქონს/ჩანაწერს მონაცემთა ბაზის ცხრილში.
- FOREIGN Key: ცალსახად აიდენტიფიცირებს სტრიქონებს/ჩანაწერებს მონაცემთა ბაზის ნებისმიერ სხვა ცხრილში.
- CHECK შეზღუდვა: CHECK შეზღუდვა უზრუნველყოფს, რომ სვეტის ყველა მნიშვნელობა აკმაყოფილებს გარკვეულ პირობებს.
- INDEX: გამოიყენება მონაცემთა ბაზიდან მონაცემთა ძალიან სწრაფად შესაქმნელად და ამოსაღებად.

NOT NULL შეზღუდვა

ნაგულისხმევად, სვეტი შეიძლება შეიცავდეს NULL მნიშვნელობას. თუ არ გსურთ, რომ სვეტს ჰქონდეს NULL მნიშვნელობა, მაშინ საჭიროა მიუთითოთ, რომ NULL ახლა არ არის დაშვებული ამ სვეტისთვის. მაგალითად, შემდეგი SQL ქმნის ახალ ცხრილს სახელწოდებით CUSTOMERS და ამატებს ხუთ სვეტს, რომელთაგან სამს: ID, NAME და AGE აკრძალული აქვს NULL მნიშვნელობები.

```
CREATE TABLE CUSTOMERS(
       ID   INT              NOT NULL,
       NAME VARCHAR (20)     NOT NULL,
       AGE  INT              NOT NULL,
       ADDRESS  CHAR (25) ,
       SALARY   DECIMAL (18, 2),
       PRIMARY KEY (ID)
);
```

ნაგულისხმევი შეზღუდვა:

DEFAULT შეზღუდვა უზრუნველყოფს სვეტის ნაგულისხმევ მნიშვნელობას,
როდესაც INSERT INTO განცხადება არ წარადგენს
კონკრეტული მნიშვნელობას.

მაგალითი:

მაგალითად, შემდეგი SQL ქმნის ახალ ცხრილს სახელწოდებით CUSTOMERS და ამატებს ხუთ სვეტს. ხელფასი სვეტი ნაგულისხმევად დაყენებულია 5000.00-ზე, ასე რომ, იმ შემთხვევაში, თუ განცხადება INSERT INTO არ იძლევა მნიშვნელობას ამ სვეტისთვის მაშინ ნაგულისხმევად ეს სვეტი იქნება 5000.00.

```
CREATE TABLE CUSTOMERS(
       ID   INT              NOT NULL,
       NAME VARCHAR (20)     NOT NULL,
       AGE  INT              NOT NULL,
       ADDRESS  CHAR (25) ,
       SALARY   DECIMAL (18, 2) DEFAULT 5000.00,
       PRIMARY KEY (ID)
);
```

თუ CUSTOMERS ცხრილი უკვე შექმნილია, მაშინ უნდა დაამატოთ DEFAULT შეზღუდვა SALARY სვეტში.

```
ALTER TABLE CUSTOMERS
   MODIFY SALARY  DECIMAL (18, 2) DEFAULT 5000.00;
```

ნაგულისხმევი შეზღუდვის მოსახსნელად გამოიყენება შემდეგი SQL კოდი:

```
ALTER TABLE CUSTOMERS
   ALTER COLUMN SALARY DROP DEFAULT;
```

უნიკალური შეზღუდვა:

UNIQUE შეზღუდვა გამორიცხავს ორ ჩანაწერს ჰქონდეს იდენტური მნიშვნელობები კონკრეტულ სვეტში.

მაგალითად, შემდეგი SQL ქმნის ახალ ცხრილს სახელწოდებით CUSTOMERS და ამატებს ხუთ სვეტს. აქ, AGE
სვეტზე დაყენებულია UNIQUE შეზღუდვა, ასე რომ არ შეგიძლიათ გქონდეთ ერთი და იგივე ასაკის ორი ჩანაწერი:

```
CREATE TABLE CUSTOMERS(
        ID   INT              NOT NULL,
       NAME VARCHAR (20)     NOT NULL,
       AGE  INT              NOT NULL UNIQUE,
       ADDRESS  CHAR (25) ,
       SALARY   DECIMAL (18, 2),
       PRIMARY KEY (ID)
);
```

თუ CUSTOMERS ცხრილი უკვე შექმნილია, მაშინ AGE სვეტში UNIQUE შეზღუდვის დასამატებლად საჭიროა დაიწეროს შემდეგი კოდი:

```
ALTER TABLE CUSTOMERS
   MODIFY AGE INT NOT NULL UNIQUE;
```

თქვენ ასევე შეგიძლიათ გამოიყენოთ შემდეგი სინტაქსი, რომელიც მხარს უჭერს შეზღუდვის დასახელებას მრავალ სვეტში:

```
ALTER TABLE CUSTOMERS
   ADD CONSTRAINT myUniqueConstraint UNIQUE(AGE, SALARY);
```

უნიკალური შეზღუდვის გაუქმება:

UNIQUE შეზღუდვის გასაუქმებლად გამოიყენეთ შემდეგი SQL კოდი:

```
ALTER TABLE CUSTOMERS
   DROP CONSTRAINT myUniqueConstraint;
```

თუ იყენებთ MySQL, მაშინ შეგიძლიათ გამოიყენოთ შემდეგი სინტაქსი:

```
ALTER TABLE CUSTOMERS
   DROP INDEX myUniqueConstraint;
```

პირველადი გასაღები:

პირველადი გასაღები არის ცხრილის ველი, რომელიც უნიკალურად განსაზღვრავს თითოეულ სტრიქონს/ჩანაწერს მონაცემთა ბაზის ცხრილში. ძირითადი გასაღებები უნდა შეიცავდეს უნიკალურ მნიშვნელობებს. პირველადი გასაღების სვეტს არ შეიძლება ჰქონდეს NULL მნიშვნელობები.

ცხრილს შეიძლება ჰქონდეს მხოლოდ ერთი პირველადი გასაღები, რომელიც შეიძლება შედგებოდეს ერთი ან რამდენიმე ველისგან. როდესაც გამოიყენება მრავალი მრავალთან კავშირი, მათ უწოდებენ კომპოზიტურ გასაღებს. თუ ცხრილს აქვს პირველადი გასაღები განსაზღვრული რომელიმე ველ(ებ)ზე, მაშინ არ შეიძლება გქონდეთ ერთი და იგივე მნიშვნელობის მქონე ორი ჩანაწერი. ეს ცნებები გამოიყენება მონაცემთა ბაზის ცხრილების შექმნისას.

განვსაზღვროთ ID ატრიბუტი, როგორც პირველადი გასაღები CUSTOMERS ცხრილში:

```
CREATE TABLE CUSTOMERS(
       ID   INT              NOT NULL,
       NAME VARCHAR (20)     NOT NULL,
       AGE  INT              NOT NULL,
       ADDRESS  CHAR (25) ,
       SALARY   DECIMAL (18, 2),
       PRIMARY KEY (ID)
);
```

"ID" სვეტზე PRIMARY KEY შეზღუდვის შესაქმნელად, როდესაც CUSTOMERS ცხრილი უკვე არსებობს, გამოიყენება შემდეგი SQL სინტაქსი:

```
ALTER TABLE CUSTOMER ADD PRIMARY KEY (ID);
```

მრავალ სვეტზე PRIMARY KEY შეზღუდვის დასადებად გამოიყენება შემდეგი SQL სინტაქსი:

```
CREATE TABLE CUSTOMERS(
       ID   INT              NOT NULL,
       NAME VARCHAR (20)     NOT NULL,
       AGE  INT              NOT NULL,
       ADDRESS  CHAR (25) ,
       SALARY   DECIMAL (18, 2),
       PRIMARY KEY (ID, NAME)
);
```

"ID" და "NAMES" სვეტებზე PRIMARY KEY შეზღუდვის შესაქმნელად, როდესაც CUSTOMERS ცხრილი უკვე არსებობს,
გამოიყენეთ შემდეგი SQL სინტაქსი:

```
ALTER TABLE CUSTOMERS
   ADD CONSTRAINT PK_CUSTID PRIMARY KEY (ID, NAME);
```

პირველადი გასაღების წაშლა:
თქვენ შეგიძლიათ წაშალოთ პირველადი გასაღების შეზღუდვა ცხრილიდან შემდეგი კოდით:

```
ALTER TABLE CUSTOMERS DROP PRIMARY KEY ;
```

გარე გასაღები:

გარე გასაღები არის გასაღები, რომელიც გამოიყენება ორი ცხრილის ერთმანეთთან დასაკავშირებლად. ამას ზოგჯერ უწოდებენ ბმულის გასაღებს. გარე გასაღები არის სვეტი ან სვეტების კომბინაცია, რომელთა მნიშვნელობები ემთხვევა სხვა ცხრილის პირველად გასაღებს.

მაგალითი:

CUSTOMERS ცხრილი:

```
CREATE TABLE CUSTOMERS(
        ID   INT              NOT NULL,
        NAME VARCHAR (20)     NOT NULL,
        AGE  INT              NOT NULL,
        ADDRESS  CHAR (25) ,
        SALARY   DECIMAL (18, 2),
        PRIMARY KEY (ID)
);
```

ORDERS ცხრილი:

```
CREATE TABLE ORDERS (
        ID          INT        NOT NULL,
        DATE        DATETIME,
        CUSTOMER_ID INT references CUSTOMERS(ID),
        AMOUNT     double,
        PRIMARY KEY (ID)
);
```

თუ ORDERS ცხრილი უკვე შექმნილია და გარე გასაღები ჯერ არ არის დაყენებული, გამოიყენეთ სინტაქსი გარე გასაღების მისათითებლად ცხრილში ცვლილების გზით:

```
ALTER TABLE ORDERS
    ADD FOREIGN KEY (Customer_ID) REFERENCES CUSTOMERS (ID);
```

გარე გასაღების შეზღუდვის წაშლა:

```
ALTER TABLE ORDERS
    DROP FOREIGN KEY;
```

CHECK შეზღუდვა:

CHECK შეზღუდვა მნიშვნელობის შემოწმების საშუალებას იძლევა. თუ პირობა მცდარია ჩანაწერი არღვევს პირობას და არ შედის ცხრილში. მაგალითად: შემდეგი კოდი ქმნის CUSTOMERS ცხრილს და ამატებს ხუთ სვეტს. აქ ვამატებთ ასაკი სვეტის შემოწმებას, რომ არ იყოს 18 წელზე ნაკლები ასაკის კლიენტი.

```
CREATE TABLE CUSTOMERS(
       ID   INT              NOT NULL,
       NAME VARCHAR (20)     NOT NULL,
       AGE  INT              NOT NULL CHECK (AGE >= 18),
       ADDRESS  CHAR (25) ,
       SALARY   DECIMAL (18, 2),
       PRIMARY KEY (ID)
);
```

თუ CUSTOMERS ცხრილი უკვე შექმნილია, მაშინ CHECK შეზღუდვის დასამატებლად AGE სვეტში, ვწერთ:

```
ALTER TABLE CUSTOMERS
   MODIFY AGE INT NOT NULL CHECK (AGE >= 18 );
```

ასევე შეგვიძლია გამოვიყენოთ შემდეგი სინტაქსი, რომელიც მხარს უჭერს შეზღუდვის დასახელებას მრავალ სვეტში:

```
ALTER TABLE CUSTOMERS
   ADD CONSTRAINT myCheckConstraint CHECK(AGE >= 18);
```

1.3. მონაცემთა განსაზღვრის ენა (Data Definition Language – DDL)

Transact-SQL – ის მონაცემთა განსაზღვრის ენა იყენებს შემდეგ ბრძანებებს:

- CREATE *object_name*
- ALTER *object_name*
- DROP *object_name*

მონაცემთა ბაზის და ცხრილების შექმნა
მონაცემთა ბაზის შექმნა

მონაცემთა ბაზის შექმნა შეგვიძლია SQL Server Management Studio–ში Object Explorer გრაფიკული ინტერფეისის და ასევე Transact-SQL-ის ბრძანებების საშუალებით. Transact-SQL ბრძანებებით მონაცემთა ბაზის შექმნისთვის გამოიყენება CREATE DATABASE ბრძანება. ზოგადად, მოცემული ბრძანების სინტაქსია:

```
CREATE DATABASE მონაცემთა_ბაზის_სახელი
[ ON [ PRIMARY ] [ <ფაილის_განსაზღვრა> [ ,...n] ] [ ,
<ფაილების_ჯგუფი> [ ,...n ] ] ]
[ LOG ON { <ფაილის_განსაზღვრა> [,...n] } ]
[ FOR ATTACH ]
```

სადაც, **ON** დარეზერვებული სიტყვის შემდეგ განისაზღვრება მონაცემთა ბაზის ფაილები.

PRIMARY ნიშნავს, რომ შემდგომში აღიწერება მონაცემთა ბაზის ფაილი.

LOG ON განსაზღვრავს ტრანზაქციების ჟურნალის ფაილებს.

FOR ATTACH მაშინ გამოიყენება როდესაც საჭიროა მონაცემთა ბაზის სერვერთან მიერთება.

მაგალითი 1

```
USE master
GO
CREATE DATABASE Sales
ON
( NAME = Sales_dat,
  FILENAME = 'c:\program files\microsoft sql server\mssql\data\saledat.mdf',
  SIZE = 10,
  MAXSIZE = 50,
  FILEGROWTH = 5 )
LOG ON
( NAME = 'Sales_log',
  FILENAME = 'c:\program files\microsoft sql server\mssql\data\salelog.ldf',
  SIZE = 5MB,
  MAXSIZE = 25MB,
  FILEGROWTH = 5MB )
GO
```

ცხრილის შექმნის სინტაქსი

მონაცემთა ბაზის ცხრილის შესაქმნელად CREATE TABLE ბრძანება გამოიყენება, მისი სინტაქსია:

```
CREATE TABLE [ მონაცემთა_ბაზის_სახელი.[ სქემის_სახელი. ]
ცხრილის_სახელი
(
{ <ველის_განსაზღვრა> | გამოთვლადი_ველის_სახელი AS
გამოსახულება
| <ცხრილის_შეზღუდვა> } [,...n]
)
```

ცხრილის შეზღუდვების სინტაქსია:

```
[ CONSTRAINT შეზღუდვის_სახელი ]
{
[ NULL | NOT NULL ] | [ { PRIMARY KEY | UNIQUE }
```

```
[ CLUSTERED | NONCLUSTERED ]
| [ [ FOREIGN KEY ] REFERENCES ref_ცხრილი [ ( ref_ველი ) ]
[ ON DELETE { CASCADE | NO ACTION } ]
[ ON UPDATE { CASCADE | NO ACTION } ]
| CHECK ( ლოგიკური_გამოსახულება )
}
```

მოცემული შეზღუდვები უკვე განვიხილეთ.

განვიხილოთ მონაცემთა ბაზა Sample შემდეგი ცხრილებით:

ცხრილი employee

emp_no INTEGER NOT NULL
emp_fname CHAR(20) NOT NULL
emp_lname CHAR(20) NOT NULL
dept_no CHAR(4) NULL)

ცხრილი department

dept_no CHAR(4) NOT NULL
dept_name CHAR(25) NOT NULL
location CHAR(30) NULL)

ცხრილი project

project_no CHAR(4) NOT NULL
project_name CHAR(15) NOT NULL
budget FLOAT NULL)

ცხრილი works_on

emp_no INTEGER NOT NULL
project_no CHAR(4) NOT NULL
job CHAR (15) NULL
enter_date DATE NULL)

The department table:

dept_no	dept_name	location
d1	Research	Dallas
d2	Accounting	Seattle
d3	Marketing	Dallas

The **employee** table:

emp_no	emp_fname	emp_lname	dept_no
25348	Matthew	Smith	d3
10102	Ann	Jones	d3
18316	John	Barrimore	d1
29346	James	James	d2
9031	Elsa	Bertoni	d2
2581	Elke	Hansel	d2
28559	Sybill	Moser	d1

The **project** table:

project_no	Project_name	budget
p1	Apollo	120000
p2	Gemini	95000
p3	Mercury	185600

The **works_on** table:

emp_no	project_no	job	enter_date
10102	p1	Analyst	2006.10.1
10102	p3	Manager	2008.1.1
25348	p2	Clerk	2007.2.15
18316	p2	NULL	2007.6.1
29346	p2	NULL	2006.12.15
2581	p3	Analyst	2007.10.15
9031	p1	Manager	2007.4.15
28559	p1	NULL	2007.8.1
28559	p2	Clerk	2008.2.1
9031	p3	Clerk	2006.11.15
29346	p1	Clerk	2007.1.4

ცხრილების შექმნის მაგალითები:

```
CREATE TABLE table_name
(col_name1 type1 [NOT NULL| NULL]
[{, col_name2 type2 [NOT NULL| NULL]} ...])
```

მაგალითი 2

```
USE sample;
CREATE TABLE employee (emp_no INTEGER NOT NULL,
emp_fname CHAR(20) NOT NULL,
emp_lname CHAR(20) NOT NULL,
dept_no CHAR(4) NULL);
CREATE TABLE department(dept_no CHAR(4) NOT NULL,
dept_name CHAR(25) NOT NULL,
location CHAR(30) NULL);
CREATE TABLE project (project_no CHAR(4) NOT NULL,
project_name CHAR(15) NOT NULL,
budget FLOAT NULL);
CREATE TABLE works_on (emp_no INTEGER NOT NULL,
project_no CHAR(4) NOT NULL,
job CHAR (15) NULL,
enter_date DATE NULL);
```

მაგალითი 3

```
USE sample;
CREATE TABLE employee (emp_no INTEGER NOT NULL,
emp_fname CHAR(20) NOT NULL,
emp_lname CHAR(20) NOT NULL,
dept_no CHAR(4) NULL,
CONSTRAINT prim_empl PRIMARY KEY (emp_no));
```

მაგალითი 4

```
USE sample;
CREATE TABLE employee
(emp_no INTEGER NOT NULL CONSTRAINT prim_empl PRIMARY KEY,
emp_fname CHAR(20) NOT NULL,
emp_lname CHAR(20) NOT NULL,
dept_no CHAR(4) NULL);
```

გარე გასაღები

```
[CONSTRAINT c_name]
[[FOREIGN KEY] ({col_name1} ,...)]
REFERENCES table_name ({col_name2},...)
[ON DELETE {NO ACTION| CASCADE | SET NULL | SET DEFAULT}]
[ON UPDATE {NO ACTION | CASCADE | SET NULL | SET DEFAULT}]
```

მაგალითი 5

```
USE sample;
CREATE TABLE works_on (emp_no INTEGER NOT NULL,
project_no CHAR(4) NOT NULL,
job CHAR (15) NULL,
enter_date DATE NULL,
CONSTRAINT prim_works PRIMARY KEY(emp_no, project_no),
CONSTRAINT foreign_works FOREIGN KEY(emp_no)
REFERENCES employee (emp_no));

USE sample;
INSERT INTO works_on (emp_no, ...)
VALUES (11111, ...);

USE sample;
UPDATE works_on
SET emp_no = 11111 WHERE emp_no = 10102;

USE sample;
UPDATE employee
SET emp_no = 22222 WHERE emp_no = 10102;
```

მაგალითი 6

```
USE sample;
CREATE TABLE department(dept_no CHAR(4) NOT NULL,
dept_name CHAR(25) NOT NULL,
location CHAR(30) NULL,
CONSTRAINT prim_dept PRIMARY KEY (dept_no));

CREATE TABLE employee (emp_no INTEGER NOT NULL,
emp_fname CHAR(20) NOT NULL,
emp_lname CHAR(20) NOT NULL,
dept_no CHAR(4) NULL,
CONSTRAINT prim_emp PRIMARY KEY (emp_no),
CONSTRAINT foreign_emp FOREIGN KEY(dept_no) REFERENCES
department(dept_no));

CREATE TABLE project (project_no CHAR(4) NOT NULL,
project_name CHAR(15) NOT NULL,
budget FLOAT NULL,
CONSTRAINT prim_proj PRIMARY KEY (project_no));

CREATE TABLE works_on (emp_no INTEGER NOT NULL,
project_no CHAR(4) NOT NULL,
job CHAR (15) NULL,
```

```
enter_date DATE NULL,
CONSTRAINT prim_works PRIMARY KEY(emp_no, project_no),
CONSTRAINT foreign1_works FOREIGN KEY(emp_no) REFERENCES
employee(emp_no),
CONSTRAINT foreign2_works FOREIGN KEY(project_no) REFERENCES
project(project_no));
```

მაგალითი 7

```
USE sample;
CREATE TABLE works_on1
(emp_no INTEGER NOT NULL,
project_no CHAR(4) NOT NULL,
job CHAR (15) NULL,

enter_date DATE NULL,
CONSTRAINT prim_works1 PRIMARY KEY(emp_no, project_no),
CONSTRAINT foreign1_works1 FOREIGN KEY(emp_no)
REFERENCES employee(emp_no) ON DELETE CASCADE,
CONSTRAINT foreign2_works1 FOREIGN KEY(project_no)
REFERENCES project(project_no) ON UPDATE CASCADE);
```

მონაცემთა ბაზის ობიექტების შეცვლა

მონაცემთა ბაზის შეცვლა

მონაცემთა ბაზის კონფიგურირების ან პარამეტრების შესაცვლელად გამოიყენება ALTER DATABASE ბრძანება, რომლის სინტაქსია:

```
ALTER DATABASE მონაცემთა_ბაზის_სახელი
{|
<ფაილების_დამატება_შეცვლა> |
<ფაილების_ჯგუფის_დამატება_შეცვლა>
| <მონაცემთა_ბაზის_რეჟიმები> | MODIFY NAME =
მონაცემთა_ბაზის_ახალი_სახელი
} [;]
```

<ფაილების_დამატება_შეცვლა> კონსტრუქციის სინტაქსია:

```
<ფაილების_დამატება_შეცვლა> ::=
{
ADD FILE <ფაილის_განსაზღვრა> [ ,...n ]
[ TO FILEGROUP { ფაილების_ჯგუფის_სახელი | DEFAULT } ]
| ADD LOG FILE <ფაილის_განსაზღვრა> [ ,...n ]
| REMOVE FILE ფაილის_ლოგიკური_სახელი
| MODIFY FILE <ფაილის_განსაზღვრა>
}
```

მაგალითი 8

```
USE master;
GO
ALTER DATABASE projects
ADD FILE (NAME=projects_dat1,
FILENAME = 'C:\projects1.mdf', SIZE = 10,
MAXSIZE = 100, FILEGROWTH = 5);
```

ცხრილის სტრუქტურის ცვლილება

მაგალითი 9

```
USE sample;
ALTER TABLE employee
ADD telephone_no CHAR(12) NULL;
```

მაგალითი 10

```
USE sample;
ALTER TABLE employee
DROP COLUMN telephone_no;
```

მაგალითი 11

```
USE sample;
ALTER TABLE department
ALTER COLUMN location CHAR(25) NOT NULL;
```

მაგალითი12

მაგალითი 13

```
USE sample;
ALTER TABLE sales
ADD CONSTRAINT primaryk_sales PRIMARY KEY(order_no);
```

მაგალითი 14

```
USE sample;
ALTER TABLE sales
DROP CONSTRAINT order_check;
```

მაგალითი 15

```
USE sample;
ALTER TABLE sales
NOCHECK CONSTRAINT ALL;
```

მაგალითი 16

```
USE sample;
EXEC sp_rename @objname = department, @newname = subdivision
```

Example 5.25 renames the **department** table to **subdivision**.

ცვლილებები მონაცემთა ბაზაში

მაგალითი 17

```
USE master;
GO
ALTER DATABASE projects
ADD FILE (NAME=projects_dat1,
FILENAME = 'C:\projects1.mdf', SIZE = 10,
MAXSIZE = 100, FILEGROWTH = 5);
```

ცვლილებები ცხრილში

მაგალითი 18

```
USE sample;
ALTER TABLE employee
ADD telephone_no CHAR(12) NULL;
```

მაგალითი 19

```
USE sample;
ALTER TABLE employee
DROP COLUMN telephone_no;
```

სვეტის თვისებების მოდიფიცირება

მაგალითი 20

```
USE sample;
ALTER TABLE department
ALTER COLUMN location CHAR(25) NOT NULL;
```

შეზღუდვების დამატება ან ამოგდება

მაგალითი 21

```
USE sample;
CREATE TABLE sales
(order_no INTEGER NOT NULL,
order_date DATE NOT NULL,
ship_date DATE NOT NULL);
ALTER TABLE sales
ADD CONSTRAINT order_check CHECK(order_date <= ship_date);
```

მაგალითი 22

```
USE sample;
ALTER TABLE sales
ADD CONSTRAINT primaryk_sales PRIMARY KEY(order_no);
```

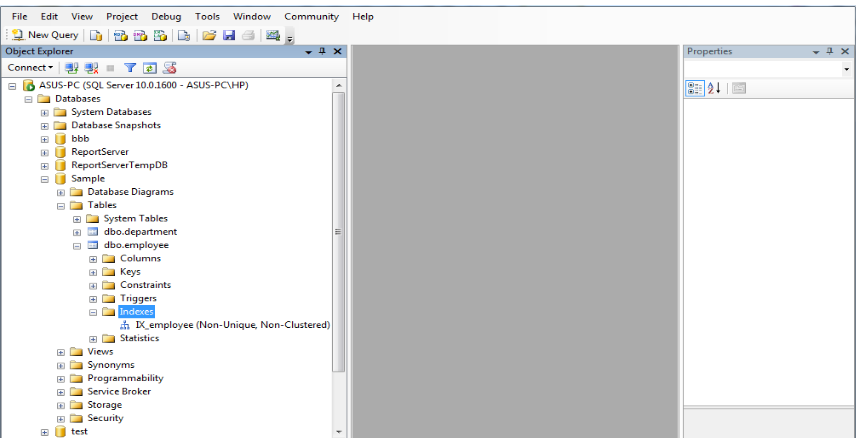
მაგალითი 23

```
USE sample;
ALTER TABLE sales
DROP CONSTRAINT order_check;
```

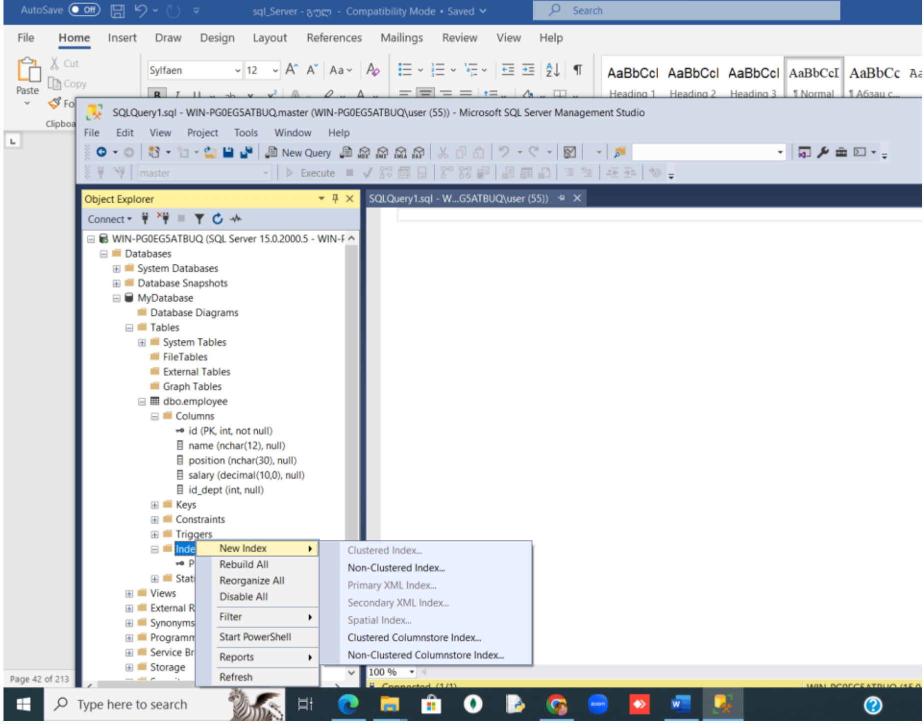
ინდექსები

ინდექსის შექმნა SQL Server Management Studio –ში

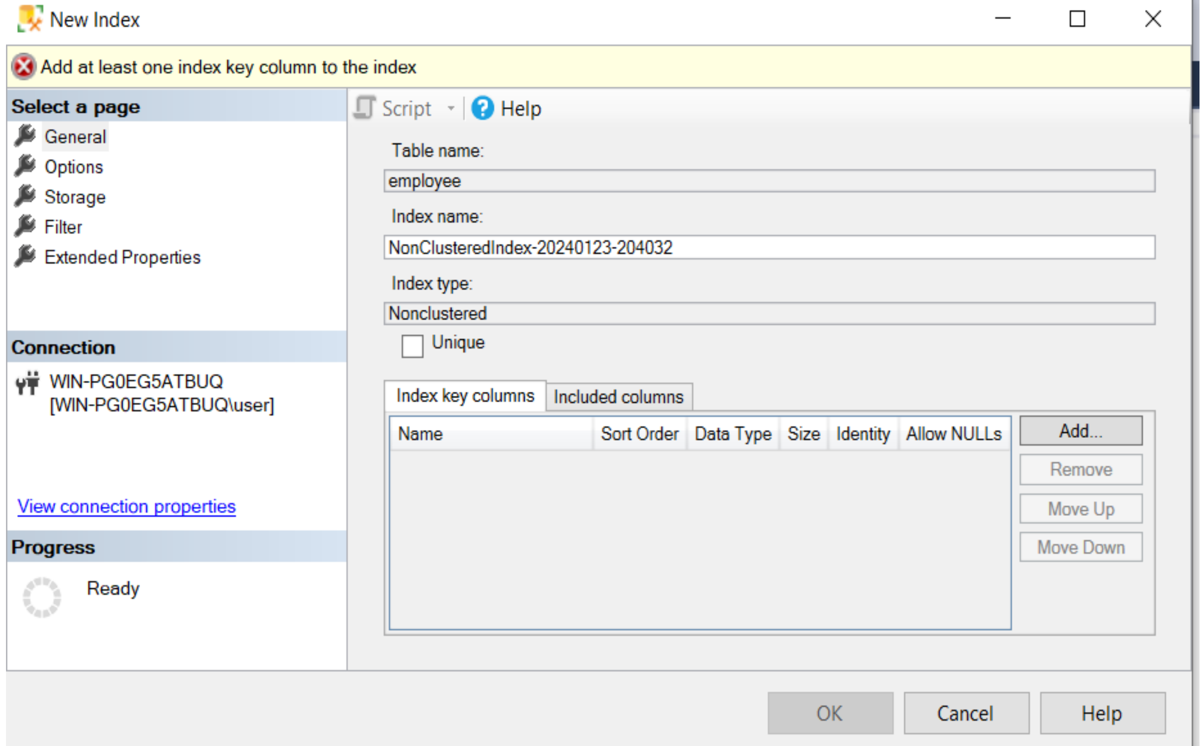
1. Object Explorer-ში ვხსნით მონაცემთა ბაზას, შემდეგ ჩვენი ცხრილისათვის ვააქტიურებთ Indexes.



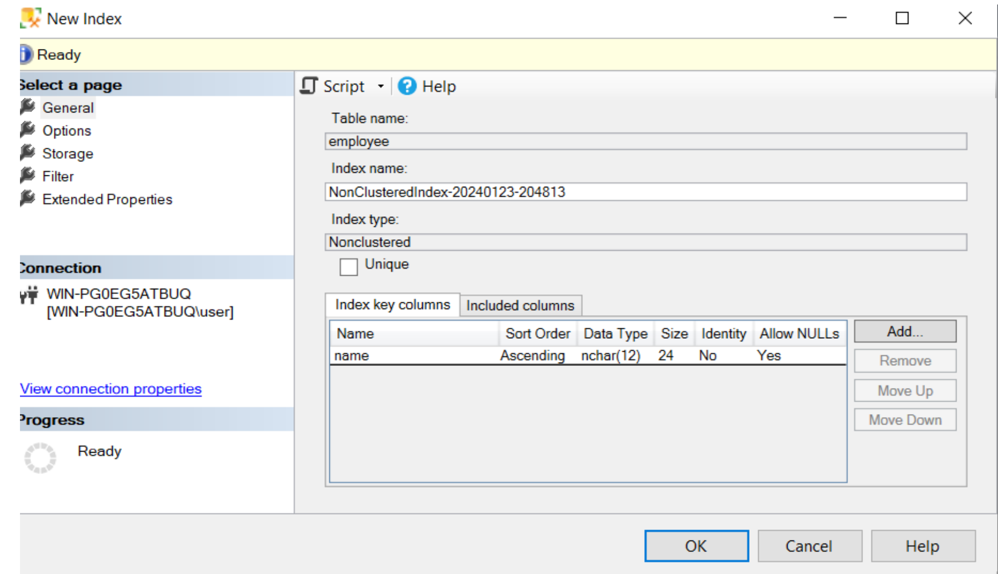
2. კონტექსტურ მენიუში ვირჩევთ New Index.



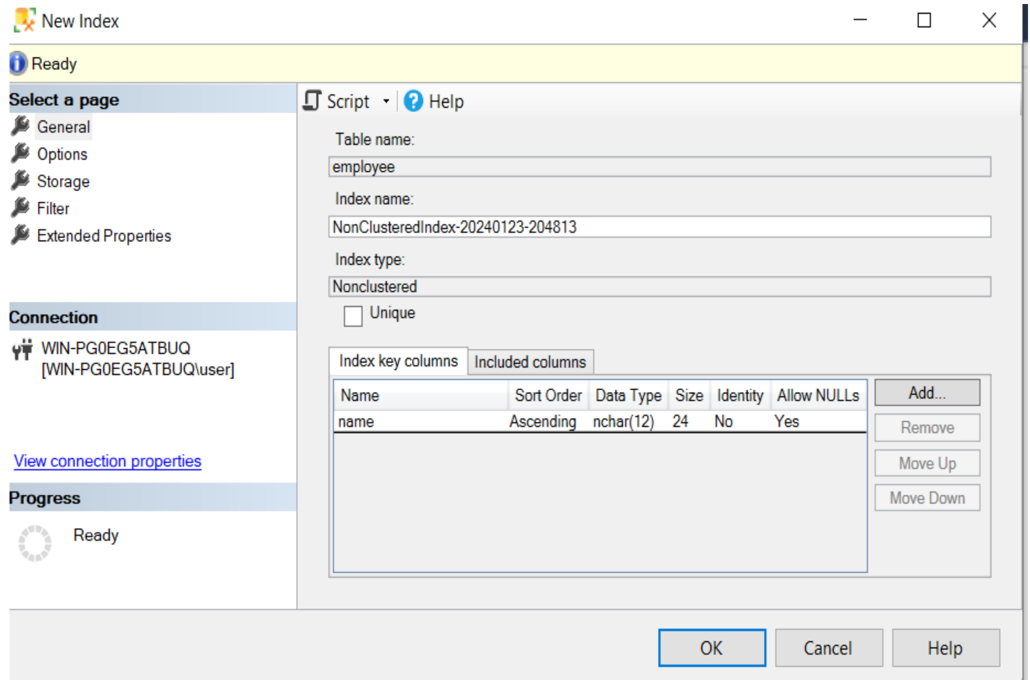
3. იხსნება ფანჯარა New Index .

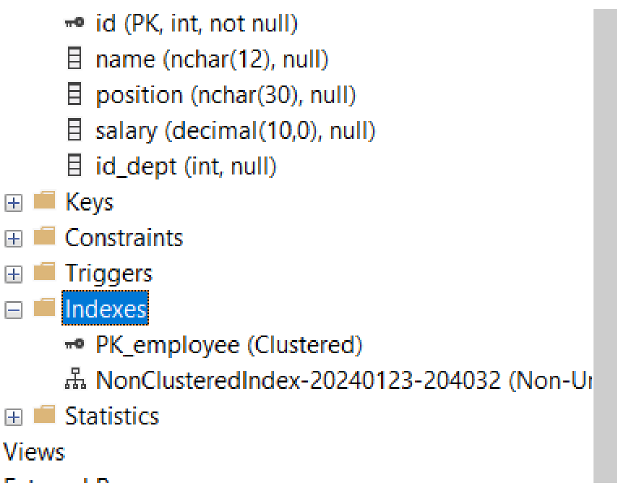


1. ველში Index name ინდექსს ვარქმევთ სახელს. ველში Index type ვირჩევთ ინდექსის ტიპს. შემდეგ Add ღილაკზე დაჭერით იხსნება ფანჯარა Select columns ვირჩევთ სვეტის სახელს, რომლის ინდექსირებაც გვსურს და OK.



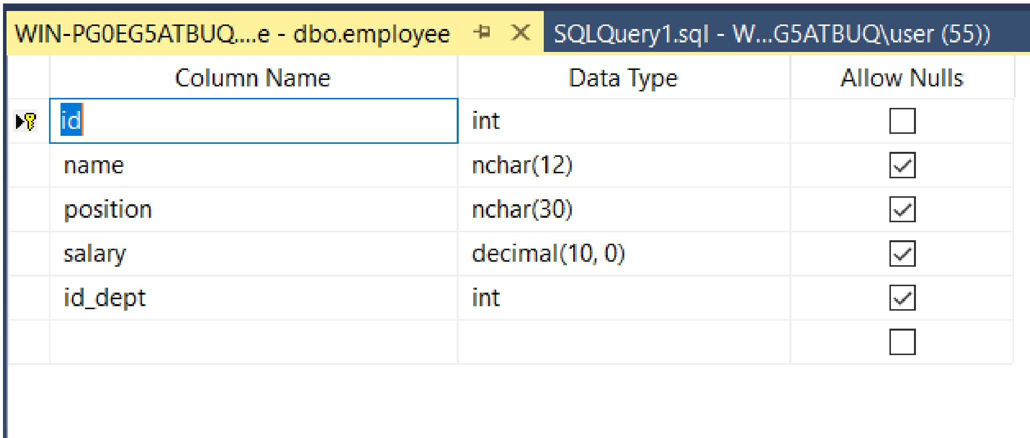
2. შედეგი აისახება New Index ფანჯარაში



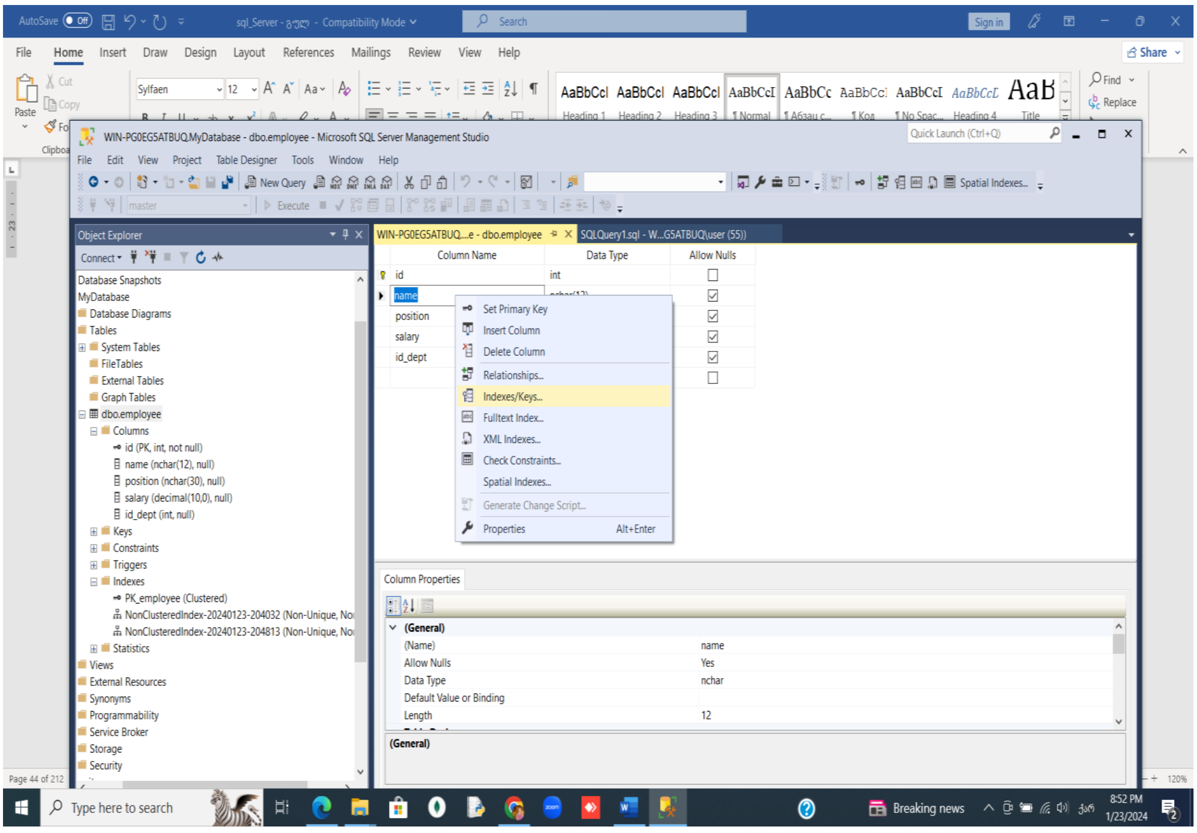


ინდექსის რედაქტირება SSMS –ში

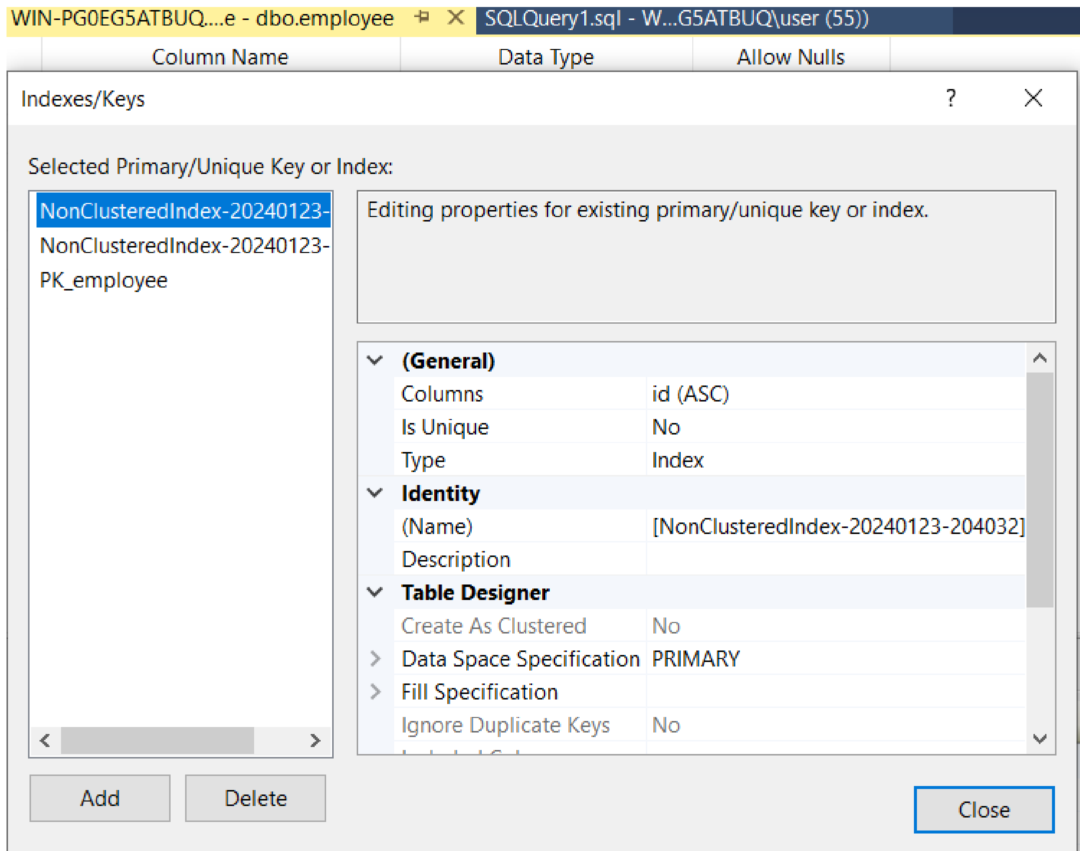
1. Object Explorer-ში ვხსნით ჩვენს ცხრილს.



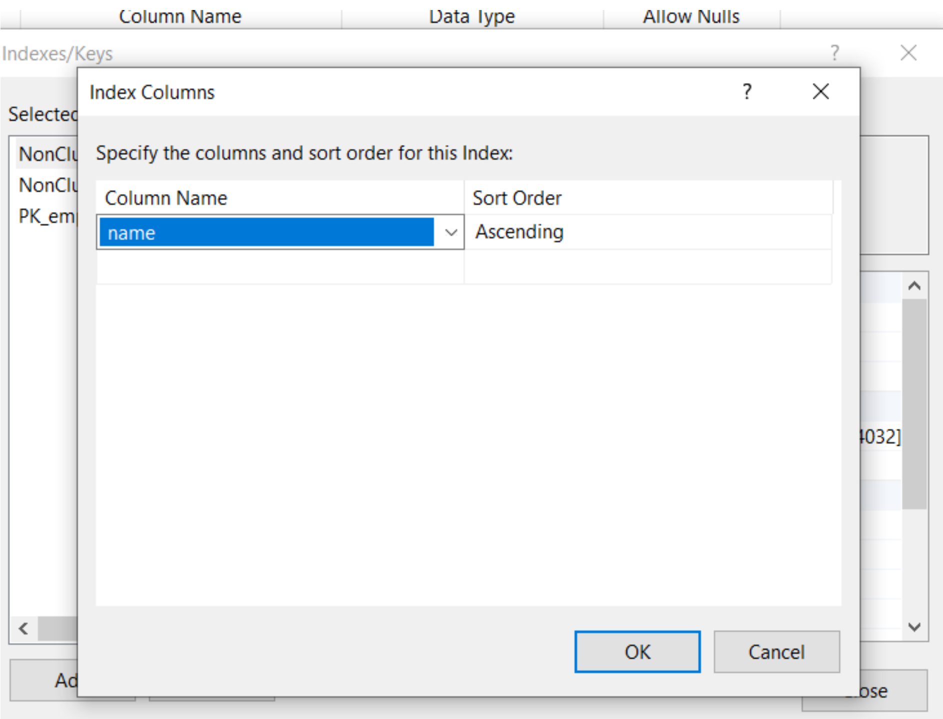
2. მარჯვენა click-ით სვეტზე, რომლის ინდექსის შექმნაც გვინდა, ვხსნით კონტექსტურ მენიუს, სადაც ვირჩევთ პუნქტს Indexes/Keys.



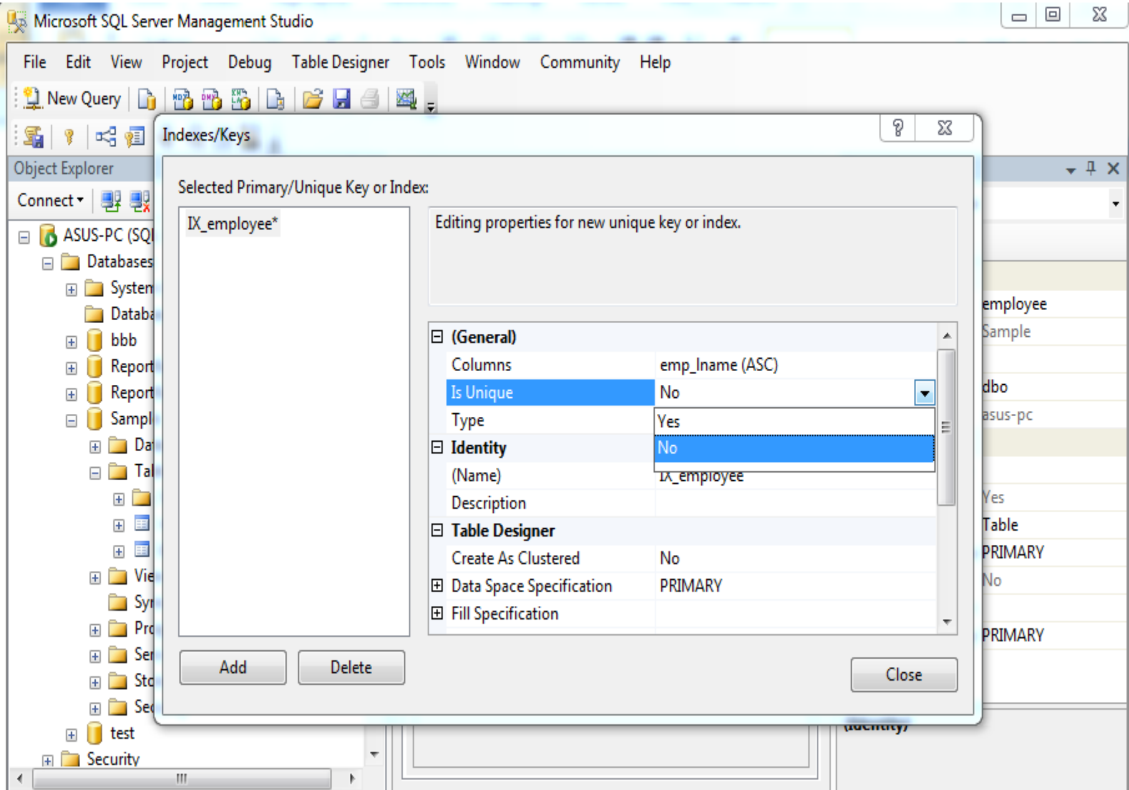
3. იხსნება შესაბამისი ფანჯარა. ვაგრძელებთ მუშაობას თვისებების რედაქტორის ველში.



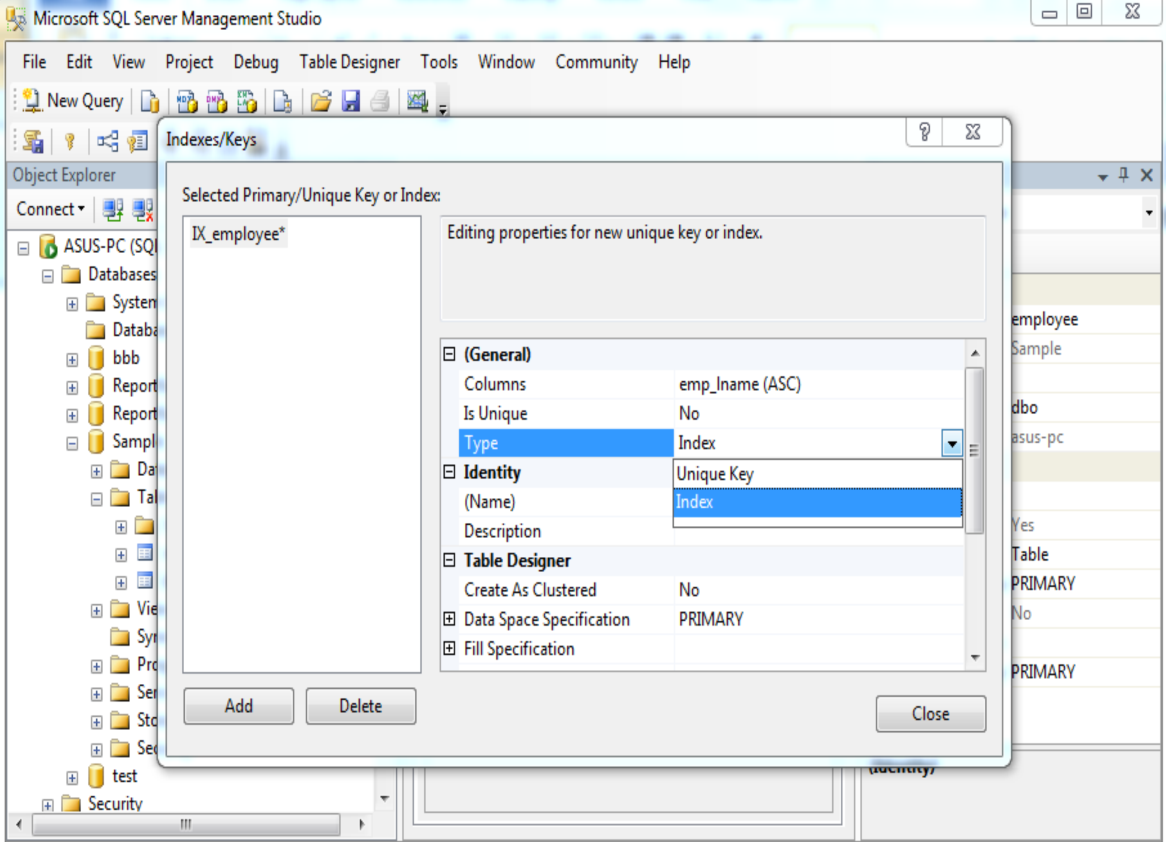
4. პუნქტში General ვირჩევთ სვეტის სახელს და სურვილის შემთხვევაში სორტირებას.



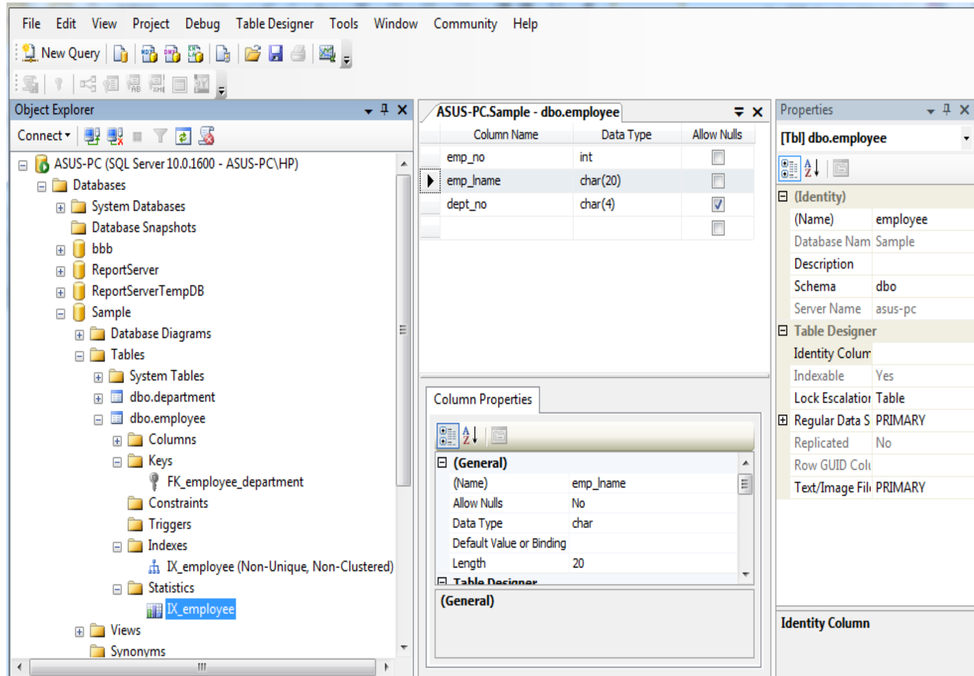
5. შემდეგ ვირჩევთ უნიკალურობის თვისების არსებობის პირობას.



6. ბოლოს ვირჩევთ ინდექსის ტიპს.



7. ინდექსის რედაქტირების შედეგს ვამოწმებთ Object Explorer ფანჯარაში.



INDEX გამოიყენება მონაცემთა ბაზიდან მონაცემების ძალიან სწრაფად შესაქმნელად და ამოსაღებად. ინდექსი შეიძლება შეიქმნას ერთი ან რამდენიმე სვეტის გამოყენებით ცხრილში. ინდექსის შექმნისას სორტირებამდე მას ენიჭება ROWID თითოეული მწკრივისთვის. ინდექსები კარგია დიდ მონაცემთა ბაზებში მუშაობისთვის მონაცემების სწრაფად მოსაძებნად და ამოსაღებად. მაგრამ ინდექსის შექმნისას ფრთხილად უნდა ვიყოთ. ველების შერჩევა დამოკიდებულია იმაზე, თუ რას ვიყენებთ SQL მოთხოვნებში. მაგალითი:

მაგალითად, შემდეგი SQL ქმნის ახალ ცხრილს სახელწოდებით CUSTOMERS და ამატებს ხუთ სვეტს:

```
CREATE TABLE CUSTOMERS(
 ID INT NOT NULL,
 NAME VARCHAR (20) NOT NULL,
 AGE INT NOT NULL,
 ADDRESS CHAR (25) ,
 SALARY DECIMAL (18, 2),
 PRIMARY KEY (ID)
);
```

შეგვიძლია შევქმნათ ინდექსი ერთ ან მრავალ სვეტზე შემდეგი სინტაქსის გამოყენებით:

```
CREATE INDEX index_name
 ON table_name ( column1, column2.....);
```

AGE სვეტზე INDEX-ის შესაქმნელად, კონკრეტული ასაკის მომხმარებელთა ძიების ოპტიმიზაციისთვის, ქვემოთ მოცემულია SQL

სინტაქსი:

```
CREATE INDEX idx_age
   ON CUSTOMERS ( AGE );
```

ინდექსის შეზღუდვის გაუქმება:

```
ALTER TABLE CUSTOMERS
   DROP INDEX idx_age;
```

მონაცემთა მთლიანობა:

მონაცემთა მთლიანობის შემდეგი კატეგორიები არსებობს თითოეულ RDBMS-ში:

- ობიექტის მთლიანობა, როდესაც ცხრილში არ არის განმეორებადი სტრიქონი;
- დომენის მთლიანობა: უზრუნველყოფს ჩანაწერების წვდომადობას მოცემული სტრიქონისთვის ტიპის, ფორმატის ან მნიშვნელობის დიაპაზონის შეზღუდვით;
- ბმულის მთლიანობა: შეუძლებელია სტრიქონის წაშლა, რომელიც გამოიყენება სხვა ჩანაწერების მიერ;
- მომხმარებლის მიერ განსაზღვრული მთლიანობა: უზრუნველყოფს ზოგიერთი კონკრეტული ბიზნეს-წესების დაცვას, რომლებიც არ მიეკუთვნება დანარჩენ მთლიანობას.

მაგალითი 1

```
USE sample;
CREATE INDEX i_empno ON employee (emp_no);
```

ეს მაგალითი გვიჩვენებს უნიკალური შედგენილი ინდექსის შექმნას.

მაგალითი 2

```
USE sample;
CREATE UNIQUE INDEX i_empno_prno
ON works_on (emp_no, project_no)
WITH FILLFACTOR= 80;
```

ინდექსების მართვა TRANSACT SQL–ში

ინდექსის ცვლილება

ინდექსის შეცვლისთვის გამოიყენება ALTER INDEX ბრძანება, რომლის სინტაქსია:

```
ALTER INDEX { ინდექსის_სახელი | ALL } ON <ობიექტი>
REBUILD
[ WITH ( <ხელახალი_აგების_რეჟიმი> [ ,...n ] ) ]
<ხელახალი_აგების_რეჟიმი> კონსტრუქციის სინტაქსია:
<ხელახალი_აგების_რეჟიმი> ::=
{
PAD_INDEX = { ON | OFF }
| FILLFACTOR = შევსების_ფაქტორი
| SORT_IN_TEMPDB = { ON | OFF }
| IGNORE_DUP_KEY = { ON | OFF }
| STATISTICS_NORECOMPUTE = { ON | OFF }
}
```

ინდექსის გაუქმება

ინდექსის გასაუქმებლად გამოიყენება DROP INDEX ბრძანება, რომლის ინდექსია:

```
DROP INDEX ინდექსის_სახელი, [,...n]
```

DROP INDEX ერთი ბრძანებით შესაძლებელია არა ერთი, არამედ რამდენიმე ინდექსის წაშლა.

მაგალითი 3

```
USE sample;
DROP INDEX i_empno ON employee;
```

ინდექსები WHERE განყოფილებაში

თუ WHERE განყოფილება SELECT ბრძანებაში შეიცავს ერთ სვეტში ძებნის პირობებს, მაშინ შესაძლებელია ამ სვეტში ინდექსის შექმნა.

მაგალითი 4

```
USE sample;
CREATE INDEX i_works ON works_on(emp_no, enter_date);
SELECT emp_no, project_no, enter_date
FROM works_on
WHERE emp_no = 29346 AND enter_date='1.4.2006';
```

ინდექსები და Join ოპერატორი.

შეერთების ოპერატორის შემთხვევაში რეკომენდებულია შეერთების თითოეული სვეტის ინდექსირება.

მაგალითი 5

```
USE sample;
SELECT emp_lname, emp_fname
FROM employee, works_on
WHERE employee.emp_no = works_on.emp_no
AND enter_date = '10.15.2007';
```

ვიდრე ინდექსს შევქმნით უმჯობესია გამოვიტანოთ ინფორმაცია იმის შესახებ თუ რომელი ინდექსები არის უკვე ჩვენს ცხრილში. ამისათვის გამოვიყენოთ შემდეგი ბრძანება:

```
sp_helpindex [ @objname = ] 'ცხრილის_სახელი'
```

მონაცემთა შეტანა ცხრილებში

არსებობს ცხრილში მონაცემების ჩამატების რამდენიმე ვარიანტი:

– INSERT ბრძანების გამოყენება, როდესაც ცხრილში შეიძლება ერთი ან მეტი სტრიქონის ჩამატება.

– SELECT INTO ბრძანების დროს ცხრილში ჩაიწერება SELECT მოთხოვნის მიერ გაცემული სტრიქონები.

– bcp.exe და BULK INSERT ბრძანებების გამოყენების შემთხვევაში საწყისი მონაცემები ცხრილს ტექსტური ფაილიდან ჩაემატება.

– მონაცემთა ბაზის პროგრამული ინტერფეისის გამოყენება (Database API) ითვალისწინებს ADO, OLE DB, ODBC და DB ბიბლიოთეკების გამოყენებას.

– Data Transformation Services ტექნოლოგია, რომელიც საშუალებას გვაძლევს შევასრულოთ მონაცემების გადატანისა და გარდაქმნის ოპერაციები მონაცემების ისეთი წყაროებიდან, როგორიცაა Oracle, Excel, Paradox, Access და ა.შ. განვიხილოთ ზოგიერთი მათგანი.

INSERT INTO ბრძანება

ბრძანების სინტაქსია:

INSERT [INTO] { ცხრილის_სახელი | წარმოდგენის_სახელი} {[(სვეტების_სია)]
{ VALUES ({ DEFAULT | NULL | გამოსახულება})|მიღებული_ცხრილი}}|DEFAULT VALUES

განვიხილოთ არგუმენტების დანიშნულება.

– [INTO] არააუცილებელი არგუმენტია, რომელსაც მოსდევს იმ ცხრილის სახელი, რომელშიც მონაცემების ჩამატება ხდება;

– ცხრილის_სახელი იმ ცხრილის სახელია, რომელშიც სტრიქონების ჩამატება ხდება;

– წარმოდგენის_სახელი იმ წარმოდგენის სახელია, რომელშიც ხდება სტრიქონების ჩამატება. უნდა გვახსოვდეს, რომ სტრიქონების ჩამატება შესაძლებელია წარმოდგენის მხოლოდ ერთ ცხრილში;

– (სვეტების_სია) იმ სვეტების სახელების სიაა, რომლებშიც უნდა მოხდეს მონაცემების ჩამატება. სვეტების მნიშვნელობები მიეთითება VALUES არგუმენტში. თუ გვინდა რამდენიმე სვეტის მნიშვნელობის შეტანა ჩვენთვის სასურველი მიმდევრობით, მაშინ მათი სახელები ასეთივე მიმდევრობით უნდა მივუთითოთ სვეტების_სია არგუმენტში. შესაბამისად, VALUE არგუმენტში ვუთითებთ მონაცემებს შესაბამისი მიმდევრობით.

– VALUES ({DEFAULT|NULL|გამოსახულება}). VALUES საკვანძო სიტყვა განსაზღვრავს ცხრილში ჩასასმელ მონაცემებს. მისი არგუმენტების რაოდენობა უნდა ემთხვეოდეს ცხრილში ან სვეტების_სია არგუმენტში მითითებული სვეტების რაოდენობას. DEFAULT არგუმენტი მიუთითებს, რომ მოხდება ავტომატურად განსაზღვრული მნიშვნელობების ჩასმა.

– თუ სვეტისთვის ავტომატური მნიშვნელობა განსაზღვრული არ არის და ნებადართულია NULL მნიშვნელობის შენახვა, მაშინ სვეტში NULL მნიშვნელობა მოთავსდება. DEFAULT არგუმენტის გამოყენება არ შეიძლება იმ სვეტებისათვის, რომლებისთვისაც განსაზღვრულია IDENTITY თვისება. გამოსახულება არგუმენტი შეიძლება იყოს მუდმივა, ცვლადი ან გამოსახულება,რომელიც განსაზღვრავს სვეტში ჩასასმელ მნიშვნელობასა და ტიპს.

– მიღებული_ცხრილი არგუმენტი შეიძლება შეიცავდეს SELECT ბრძანებას, რომლის საშუალებითაც მოცემულ ცხრილში ჩასასმელ სტრიქონებს სხვა ცხრილიდან მივიღებთ. ორივე ცხრილს ერთნაირი სვეტები უნდა ჰქონდეს.

– DEFAULT VALUES არგუმენტის მითითების შემთხვევაში თითოეულ სვეტში მოთავსდება ავტომატურად განსაზღვრული მნიშვნელობა.

მაგალითი 1

```
USE sample;
INSERT INTO employee VALUES (25348, 'Matthew', 'Smith','d3');
INSERT INTO employee VALUES (10102, 'Ann', 'Jones','d3');
INSERT INTO employee VALUES (18316, 'John', 'Barrimore', 'd1');
INSERT INTO employee VALUES (29346, 'James', 'James', 'd2');
INSERT INTO employee VALUES (9031, 'Elsa', 'Bertoni', 'd2');
INSERT INTO employee VALUES (2581, 'Elke', 'Hansel', 'd2');
INSERT INTO employee VALUES (28559, 'Sybill', 'Moser', 'd1');
```

მაგალითი 2

```
USE sample;
INSERT INTO department VALUES ('d1', 'Research', 'Dallas');
INSERT INTO department VALUES ('d2', 'Accounting', 'Seattle');
INSERT INTO department VALUES ('d3', 'Marketing', 'Dallas');
```

მაგალითი 3

```
USE sample;
INSERT INTO project VALUES ('p1', 'Apollo', 120000.00);
INSERT INTO project VALUES ('p2', 'Gemini', 95000.00);
INSERT INTO project VALUES ('p3', 'Mercury', 186500.00);
```

მაგალითი 4

```
USE sample;
INSERT INTO works_on VALUES (10102,'p1', 'Analyst', '2006.10.1');
INSERT INTO works_on VALUES (10102, 'p3', 'Manager', '2008.1.1');
INSERT INTO works_on VALUES (25348, 'p2', 'Clerk', '2007.2.15');
INSERT INTO works_on VALUES (18316, 'p2', NULL, '2007.6.1');
INSERT INTO works_on VALUES (29346, 'p2', NULL, '2006.12.15');
INSERT INTO works_on VALUES (2581, 'p3', 'Analyst', '2007.10.15');
INSERT INTO works_on VALUES (9031, 'p1', 'Manager', '2007.4.15');
INSERT INTO works_on VALUES (28559, 'p1', 'NULL', '2007.8.1');
INSERT INTO works_on VALUES (28559, 'p2', 'Clerk', '2008.2.1');
INSERT INTO works_on VALUES (9031, 'p3', 'Clerk', '2006.11.15');
INSERT INTO works_on VALUES (29346, 'p1','Clerk', '2007.1.4');
```

არსებობს ახალ ატრიქონში მონაცემების შეტანის კიდევ რამდენიმე განსხვავებული გზა. შეიძლება ველების მიმდევრობა არ დავიცვათ, მაგრამ გავითვალისწინოთ, რომ რა მიმდევრობითაც დავასახელებთ ცხრილის ველებს იმავე მიმდევრობით უნდა დავასახელოთ მონაცემები:

მაგალითი 5 და მაგალითი 7 გვიჩვენებს ზოგიერთ ასეთ შესაძლებლობას.

მაგალითი 5

```
USE sample;
INSERT INTO employee VALUES (15201, 'Dave', 'Davis', NULL);
```

მაგალითი 6

```
USE sample;
INSERT INTO employee (emp_no, emp_fname, emp_lname)
VALUES (15201, 'Dave', 'Davis');
```

მაგალითი 7

```
USE sample;
INSERT INTO employee (emp_lname, emp_fname, dept_no, emp_no)
VALUES ('Davis', 'Dave', 'd1', 15201);
```

მაგალითი 8

```
USE sample;
CREATE TABLE dallas_dept
```

```
(dept_no CHAR(4) NOT NULL,
dept_name CHAR(20) NOT NULL);
INSERT INTO dallas_dept (dept_no, dept_name)
SELECT dept_no, dept_name
FROM department
WHERE location = 'Dallas';
```

მაგალითი 9

```
USE sample;
CREATE TABLE clerk_t
(emp_no INT NOT NULL,
project_no CHAR(4),
enter_date DATE);
INSERT INTO clerk_t (emp_no, project_no, enter_date)
SELECT emp_no, project_no, enter_date
FROM works_on
WHERE job = 'Clerk'
AND project_no = 'p2';
```

შედეგად მივიღებთ:

emp_no	project_no	enter_date
25348	p2	2007-02-15
28559	p2	2008-02-01

მაგალითი 10

```
USE sample;
INSERT INTO department VALUES ('d4', 'Human Resources', 'Chicago'),
('d5', 'Distribution', 'New Orleans'),
('d6', 'Sales', 'Chicago');
```

SELECT ... INTO ბრძანება

SELECT მოთხოვნის მიერ გაცემული სტრიქონების ცხრილში ჩაწერისათვის გამოიყენება SELECT ... INTO ბრძანება, რომლის შესრულების შედეგად შეიქმნება ახალი ცხრილი, რომელშიც SELECT მოთხოვნის მიერ ამორჩეული სტრიქონები მოთავსდება. მისი სინტაქსია:

SELECT { სვეტის_სახელი [[AS] სვეტის_ფსევდონიმი], ... n }
INTO ახალი_ცხრილის_სახელი FROM { საწყისი_ცხრილის_სახელი, ... n }

განვიხილოთ არგუმენტების დანიშნულება.

– სვეტის_სახელი [[AS] სვეტის_ფსევდონიმი] განსაზღვრავს იმ სვეტის სახელს, რომელიც ჩართული იქნება შედეგში. თუ სხვადასხვა ცხრილის სვეტებს ერთნაირი სახელები აქვს, მაშინ მათი სახელების წინ უნდა მივუთითოთ ფსევდონიმი (ცხრილის სახელი). ფსევდონიმის გამოყენება სასარგებლოა იმ

შემთხვევაშიც, როცა გვინდა, რომ შესაქმნელი ცხრილის სვეტებს საწყისი ცხრილისაგან განსხვავებული სახელები ჰქონდეს;

– INTO ახალი_ცხრილის_სახელი მიუთითებს შესაქმნელი ცხრილის სახელს. თუ დროებით ცხრილს ვქმნით, მაშინ ცხრილის სახელის წინ უნდა მივუთითოთ # ან ##;

– FROM { საწყისი_ცხრილის_სახელი, ... n } არგუმენტი შეიცავს საწყისი ცხრილების სახელებს.

სინტაქსი, ყველა სვეტის გადაწერა ახალ ცხრილში:

SELECT *
INTO *newtable* [IN *externaldb*]
FROM *oldtable*
WHERE *condition*;

ზოგიერთი სვეტის გადაწერა ახალ ცხრილში:

SELECT *column1*, *column2*, *column3*,
INTO *newtable* [IN *externaldb*]
FROM *oldtable*
WHERE *condition;*

მაგალითი: ქმნის Customers სარეზერვო ასლს:

```
SELECT * INTO CustomersBackup2017
FROM Customers;
```

მაგალითი: იყენებს IN ბრძანებას ცხრილის ახალ ცხრილში გადასაკოპირებლად, სხვა მონაცემთა ბაზაში:

```
SELECT * INTO CustomersBackup2017 IN 'Backup.mdb'
FROM Customers;
```

მაგალითი: SQL კოდი აკოპირებს მხოლოდ რამდენიმე სვეტს ახალ ცხრილში:

```
SELECT CustomerName, ContactName INTO CustomersBackup2017
FROM Customers;
```

მაგალითი: SQL კოდი აკოპირებს მხოლოდ გერმანელ მომხმარებლებს ახალ ცხრილში:

```
SELECT * INTO CustomersGermany
FROM Customers
WHERE Country = 'Germany';
```

SELECT INTO ასევე შეიძლება გამოყენებულ იქნას ახალი, ცარიელი ცხრილის შესაქმნელად სხვისი სქემის გამოყენებით. უბრალოდ დაამატეთ WHERE პუნქტი, რომელიც იწვევს მოთხოვნას მონაცემების გარეშე:

SELECT * INTO *newtable*
FROM *oldtable*
WHERE 1 = 0;

UPDATE ბრძანება

ცხრილებში მონაცემების შესაცვლელად გამოიყენება UPDATE ბრძანება. მისი სინტაქსია:

UPDATE { ცხრილის_სახელი | წარმოდგენის_სახელი }
SET { სვეტის_სახელი = { გამოსახულება | DEFAULT | NULL }
| @ცვლადის_სახელი = გამოსახულება
| @ცვლადის_სახელი = სვეტის_სახელი = გამოსახულება } [,...n]
{ [FROM { <საწყისი_ცხრილი> } [,...]] [WHERE <ძებნის_პირობა>] }

განვიხილოთ არგუმენტების დანიშნულება.

- ცხრილის_სახელი იმ ცხრილის სახელია, რომელშიც უნდა შესრულდეს მონაცემების ცვლილება.
- SET საკვანძო სიტყვა იწყებს ბლოკს, რომელშიც მითითებულია შესაცვლელი სვეტების ან ცვლადების სია.
- სვეტის_სახელი = { გამოსახულება | DEFAULT | NULL }. თითოეული სვეტისათვის უნდა მიეთითოს მისთვის მისანიჭებელი მნიშვნელობა. DEFAULT საკვანძო სიტყვა მიუთითებს, რომ სვეტს უსიტყვოდ უნდა მიენიჭოს განსაზღვრული მნიშვნელობა. NULL საკვანძო სიტყვა მიუთითებს, რომ სვეტს უნდა მიენიჭოს NULL მნიშვნელობა.
- @ცვლადის_სახელი = გამოსახულება არგუმენტი ცვლადს ანიჭებს გამოსახულების მნიშვნელობას.
- @ცვლადის_სახელი = სვეტის_სახელი = გამოსახულება კონსტრუქცია საშუალებას გვაძლევს შევათავსოთ ცვლადებისა და სვეტების სახელების გამოყენება.

მაგალითი 11

```
USE sample;
UPDATE works_on
SET job = 'Manager'
WHERE emp_no = 18316
AND project_no = 'p2';
```

მაგალითი 12

```
USE sample;
UPDATE project
```

```
SET budget = budget*0.51;
```

WHERE პირობის ამოღების გამო **project** ცხრილის ყველა სტრიქონი შეიცვლება. შეცვლილი სტრიქონები შეიძლება გამოვიტანოთ ეკრანზე შემდეგი Transact-SQL ინსტრუქციით: SELECT * FROM project;

შედეგად მივიღებთ:

project_no	**project_name**	**budget**
p1	Apollo	61200
p2	Gemini	48450
p3	Mercury	95115

მაგალითი 13

```
USE sample;
UPDATE works_on
SET job = NULL
WHERE emp_no IN
(SELECT emp_no
FROM employee
WHERE emp_lname = 'Jones');
```

მაგალითი 14

```
USE sample;
UPDATE works_on
SET job = NULL
FROM works_on, employee
WHERE emp_lname = 'Jones'
AND works_on.emp_no = employee.emp_no;
```

მაგალითი **15** ცხადყოფს CASE გამოსახულების გამოყენებას UPDATE ინსტრუქციაში.

მაგალითი 15

```
USE sample;
UPDATE project
SET budget = CASE
WHEN budget >0 and budget < 100000 THEN budget*1.2
WHEN budget >= 100000 and budget < 200000 THEN budget*1.1
ELSE budget*1.05
END
```

სინტაქსი:

```
UPDATE table_name
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];
```

მაგალითები:

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune'
WHERE ID = 6;
```

მიიღება შედეგი:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | Pune      |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

UPDATE CUSTOMERS SET ADDRESS = 'Pune', SALARY = 1000.00

```
+----+----------+-----+---------+---------+
| ID | NAME     | AGE | ADDRESS | SALARY  |
+----+----------+-----+---------+---------+
|  1 | Ramesh   |  32 | Pune    | 1000.00 |
|  2 | Khilan   |  25 | Pune    | 1000.00 |
|  3 | kaushik  |  23 | Pune    | 1000.00 |
|  4 | Chaitali |  25 | Pune    | 1000.00 |
|  5 | Hardik   |  27 | Pune    | 1000.00 |
|  6 | Komal    |  22 | Pune    | 1000.00 |
|  7 | Muffy    |  24 | Pune    | 1000.00 |
+----+----------+-----+---------+---------+
```

DELETE ბრძანება

ცხრილიდან სტრიქონების წასაშლელად გამოიყენება DELETE ბრძანება. მისი სინტაქსია:

DELETE [FROM] { ცხრილის_სახელი | წარმოდგენის_სახელი }

[FROM { <ცხრილის_სახელი>} [,…]]

[WHERE <ძებნის_პირობა>]

თუ WHERE განყოფილება არ არის მითითებული, მაშინ ცხრილიდან ყველა სტრიქონი წაიშლება.

მაგალითი 16

USE sample;
DELETE FROM works_on
WHERE job = 'Manager';

მაგალითი 17

USE sample;

```
DELETE FROM works_on
WHERE emp_no IN
(SELECT emp_no
FROM employee
WHERE emp_lname = 'Moser');
DELETE FROM employee
WHERE emp_lname = 'Moser';
```

მაგალითი 18

```
USE sample;
DELETE works_on
FROM works_on, employee
WHERE works_on.emp_no = employee.emp_no
AND emp_lname = 'Moser';
DELETE  FROM employee
WHERE emp_lname = 'Moser';
```

1.4. მონაცემთა მანიპულირების ენა (DML). მონაცემთა ამორჩევა. SELECT ბრძანება

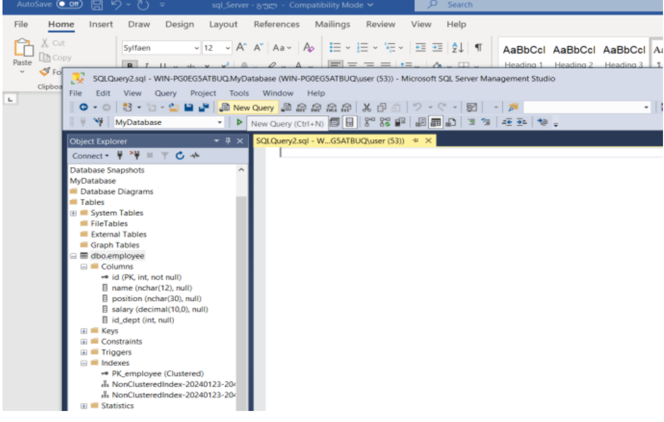
1. მონაცემთა მანიპულირების ენა. SELECT ბრძანები

Transact-SQL-ის მონაცემთა მანიპულირების ენა (DML) იყენებს შემდეგ ბრძა-ნებებს:

- SELECT
- INSERT
- UPDATE
- DELETE

მოთხოვნის შექმნა

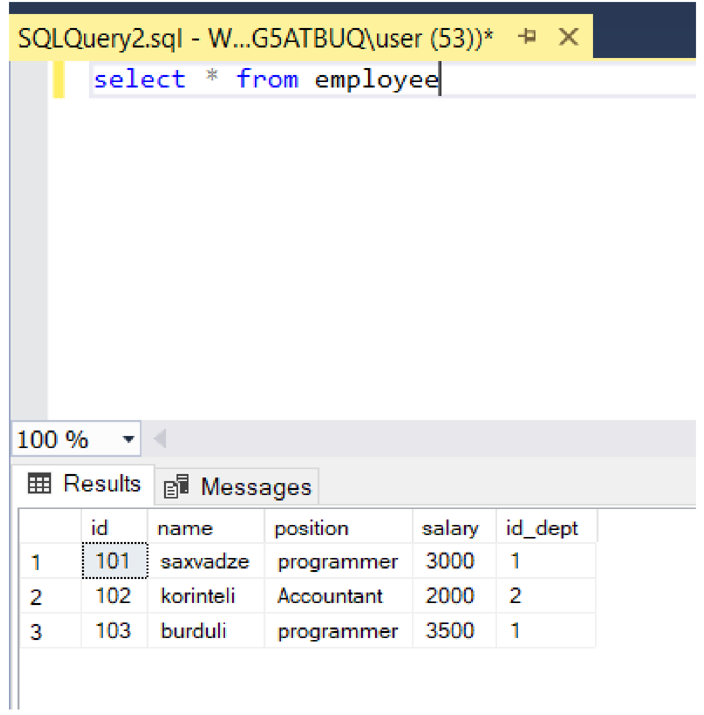
ახალი მოთხოვნის შექმნისათვის click ღილაკზე "New Query":



ინტერფეისი, რომელიც ჩნდება ეკრანზე, შესაძლებელია გამოვიყენოთ მონაცემთა ბაზის ობიექტების (როგორიცაა ცხრილები, წარმოდგენები და ა.შ.) შესაქმნელად, მონაცემების შესატანად, ამორჩევისათვის, ცვლილებებისა და წაშლისათვის. SQL მოთხოვნისათვის:

1. ვკრეფთ მოთხოვნის ტექსტს სამუშაო არეში მარჯვენა პანელზე.
2. Click ღილაკზე "Execute" (ან ვაჭერთ F5 –ზე).

შედეგები მიიღება დისპლეის ორივე პანელზე.



SQL SELECT ბრძანებები

ყველა SQL განცხადება იწყება ნებისმიერი საკვანძო სიტყვით, როგორიცაა SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, CREATE, USE, SHOW და ყველა განცხადება მთავრდება მძიმით (;). მნიშვნელოვანი პუნქტი, რომელიც უნდა აღინიშნოს, არის ის, რომ SQL არ არის რეგისტრდამოკიდებული, რაც ნიშნავს, რომ SELECT და Select იგივე მნიშვნელობა აქვთ. განვიხილოთ **SELECT** ბრძანებების სინტაქსები:

SQL SELECT ბრძანების ზოგადი სტრუქტურა:

```
SELECT column1, column2....columnN
  FROM table_name;
```

SQL DISTINCT ბრძანება:

```
SELECT DISTINCT column1, column2....columnN
   FROM table_name;
```

SQL WHERE ბრძანება:

```
SELECT column1, column2....columnN
  FROM table_name WHERE CONDITION;
```

SQL AND/OR ბრძანება:

```
SELECT column1, column2....columnN FROM table_name WHERE CONDITION-1
{AND|OR} CONDITION-2
```

SQL IN ბრძანება:

```
SELECT column1, column2....columnN
FROM table_name WHERE column_name IN (val-1, val-2,...val-N);
```

SQL BETWEEN ბრძანება:

```
 SELECT column1, column2....columnN FROM table_name
WHERE column_name BETWEEN val-1 AND val-2;
```

SQL LIKE ბრძანება:

```
 SELECT column1, column2....columnN
FROM table_name
 WHERE column_name LIKE { PATTERN };
```

SQL ORDER BY ბრძანება:

```
SELECT column1, column2....columnN
FROM table_name
WHERE CONDITION
ORDER BY column_name {ASC|DESC};
```

SQL GROUP BY ბრძანება:

```
SELECT SUM(column_name)
FROM table_name
WHERE CONDITION
GROUP BY column_name;
```

SQL COUNT ბრძანება:

```
SELECT COUNT(column_name)
FROM table_name WHERE CONDITION;
```

SQL HAVING ბრძანება:

```
SELECT SUM(column_name)
FROM table_name
WHERE CONDITION
GROUP BY column_name
```

HAVING (arithematic function condition);

მაგალითები:

```
SQL> SELECT * FROM CUSTOMERS;
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
7 rows in set (0.00 sec)
```

```
SQL> SELECT * FROM CUSTOMERS WHERE SALARY > 5000;
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  5 | Hardik   |  27 | Bhopal  |  8500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
3 rows in set (0.00 sec)

SQL>  SELECT * FROM CUSTOMERS WHERE SALARY = 2000;
+----+---------+-----+-----------+---------+
| ID | NAME    | AGE | ADDRESS   | SALARY  |
+----+---------+-----+-----------+---------+
|  1 | Ramesh  |  32 | Ahmedabad | 2000.00 |
|  3 | kaushik |  23 | Kota      | 2000.00 |
+----+---------+-----+-----------+---------+
2 rows in set (0.00 sec)

SQL>  SELECT * FROM CUSTOMERS WHERE SALARY != 2000;
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  2 | Khilan   |  25 | Delhi   |  1500.00 |
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  5 | Hardik   |  27 | Bhopal  |  8500.00 |
|  6 | Komal    |  22 | MP      |  4500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
```

SELECT * FROM CUSTOMERS WHERE SALARY <> 2000;

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Delhi	1500.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

5 rows in set (0.00 sec)

SQL> SELECT * FROM CUSTOMERS WHERE SALARY >= 6500;

ID	NAME	AGE	ADDRESS	SALARY
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
7	Muffy	24	Indore	10000.00

3 rows in set (0.00 sec)

SELECT * FROM CUSTOMERS WHERE AGE >= 25 AND SALARY >= 6500;

ID	NAME	AGE	ADDRESS	SALARY
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00

SELECT * FROM CUSTOMERS WHERE AGE >= 25 OR SALARY >= 6500;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
7	Muffy	24	Indore	10000.00

SELECT * FROM CUSTOMERS WHERE AGE IS NOT NULL;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

SELECT * FROM CUSTOMERS WHERE NAME LIKE 'Ko%';

ID	NAME	AGE	ADDRESS	SALARY
6	Komal	22	MP	4500.00

SELECT * FROM CUSTOMERS WHERE AGE IN (25, 27);

2	Khilan	25	Delhi	1500.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00

SELECT * FROM CUSTOMERS WHERE AGE BETWEEN 25 AND 27;

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Delhi	1500.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00

SELECT AGE FROM CUSTOMERS WHERE EXISTS (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);

AGE
32
25
23
25
27
22
24

SELECT * FROM CUSTOMERS WHERE AGE > ALL (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00

SELECT * FROM CUSTOMERS WHERE AGE > ANY (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00

SQL გამოსახულება

სინტაქსი:

SELECT column1, column2, columnN
FROM table_name
WHERE [CONDITION|EXPRESSION];

SQL ლოგიკური გამოსახულება:

SELECT column1, column2, columnN
FROM table_name
WHERE SINGLE VALUE MATCHTING EXPRESSION;

მაგალითები:

SELECT * FROM CUSTOMERS;

```
SQL> SELECT * FROM CUSTOMERS;
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

SELECT * FROM CUSTOMERS WHERE SALARY = 10000;

```
+----+-------+-----+---------+----------+
| ID | NAME  | AGE | ADDRESS | SALARY   |
+----+-------+-----+---------+----------+
|  7 | Muffy |  24 | Indore  | 10000.00 |
+----+-------+-----+---------+----------+
1 row in set (0.00 sec)
```

SQL რიცხვითი გამოსახულება:

SELECT numerical_expression as OPERATION_NAME

[FROM table_name

WHERE CONDITION] ;

```
SQL> SELECT (15 + 6) AS ADDITION
+----------+
| ADDITION |
+----------+
|       21 |
+----------+
1 row in set (0.00 sec)
```

SELECT (15 + 6) AS ADDITION

```
+----------+
| ADDITION |
+----------+
|       21 |
+----------+
1 row in set (0.00 sec)
```

SELECT COUNT(*) AS "RECORDS" FROM CUSTOMERS;

```
+---------+
| RECORDS |
+---------+
|       7 |
+---------+
1 row in set (0.00 sec)
```

SQL – Date გამოსახულება:

SELECT CURRENT_TIMESTAMP;

```
+---------------------+
| Current_Timestamp   |
+---------------------+
| 2009-11-12 06:40:23 |
+---------------------+
1 row in set (0.00 sec)
```

SELECT GETDATE();

```
+-------------------------+
| GETDATE                 |
+-------------------------+
| 2009-10-22 12:07:18.140 |
+-------------------------+
1 row in set (0.00 sec)
```

მაგალითები:

```
USE sample;
SELECT dept_no, dept_name, location
FROM department;
```

```
USE sample;
SELECT dept_name, dept_no
FROM department
WHERE location = 'Dallas';
```

```
USE sample;
SELECT emp_lname, emp_fname
FROM employee
WHERE emp_no >= 15000;
```

```
USE sample;
SELECT project_name
FROM project
WHERE budget*0.51 > 60000;
```

```
USE sample;
SELECT project_no, emp_no
FROM works_on
WHERE project_no = 'p1'
OR project_no = 'p2';
USE sample;
SELECT DISTINCT emp_no
FROM works_on
WHERE project_no = 'p1'
OR project_no = 'p2';
```

```
USE sample;
SELECT emp_fname, DISTINCT emp_no
FROM employee
WHERE emp_lname = 'Moser';
```

```
USE sample;
SELECT emp_no, emp_fname, emp_lname
FROM employee
WHERE emp_no = 25348 AND emp_lname = 'Smith'
OR emp_fname = 'Matthew' AND dept_no = 'd1';
SELECT emp_no, emp_fname, emp_lname
FROM employee
WHERE ((emp_no = 25348 AND emp_lname = 'Smith')
OR emp_fname ='Matthew') AND dept_no = 'd1';
```

```
USE sample;
SELECT emp_no, emp_fname, emp_lname
FROM employee
WHERE (emp_no = 25348 AND emp_lname = 'Smith')
OR (emp_fname = 'Matthew' AND dept_no = 'd1');
```

```
USE sample
SELECT emp_no, emp_lname
FROM employee
WHERE NOT dept_no = 'd2';
```

```
USE sample;
SELECT emp_no, emp_fname, emp_lname
FROM employee
WHERE emp_no IN (29346, 28559, 25348);
```

```
USE sample;
SELECT emp_no, emp_fname, emp_lname, dept_no
FROM employee
WHERE emp_no NOT IN (10102, 9031);
```

```
USE sample;
SELECT project_name, budget
FROM project
WHERE budget BETWEEN 95000 AND 120000;
```

```
USE sample;
SELECT project_name, budget
FROM project
WHERE budget >= 95000 AND budget <= 120000;
```

```
USE sample;
SELECT project_name
FROM project
WHERE budget NOT BETWEEN 100000 AND 150000;
USE sample;
SELECT project_name
FROM project
WHERE budget < 100000 OR budget > 150000;
```

```
USE sample;
SELECT emp_no, project_no
FROM works_on
WHERE project_no = 'p2'
AND job IS NULL;
```

```
USE sample;
SELECT project_no, job
FROM works_on
WHERE job <> NULL;
```

```
USE sample;
SELECT emp_no, ISNULL(job, 'Job unknown') AS task
FROM works_on
WHERE project_no = 'p1';
```

SQL LIKE ბრძანება

SQL LIKE პუნქტი გამოიყენება მნიშვნელობების შესადარებლად მსგავს მნიშვნელობებთან ოპერატორების გამოყენებით. გამოიყენება პროცენტის ნიშანი (%) და ქვედა ხაზი (_). პროცენტის ნიშანი წარმოადგენს ნულს, ერთ ან მრავალ სიმბოლოს, ქვედა ხაზგასმა წარმოადგენს ერთ რიცხვს ან თვისებას. შეიძლება მათი გამოყენება კომბინაციაში.

სინტაქსი:

```
SELECT column1, column2, ...
FROM table_name
WHERE columnN LIKE pattern;
```

ქვემოთ მოცემული კოდი დააბრუნებს ყველა მომხმარებელს ქალაქიდან, რომელიც იწყება "L"-ით, რასაც მოჰყვება ერთი სიმბოლო, შემდეგ "nd" და შემდეგ ორი სიმბოლო:

```
SELECT * FROM Customers
WHERE city LIKE 'L_nd__';
```

მაგალითები:

```
SELECT FROM table_name
WHERE column LIKE 'XXXX%'

or

SELECT FROM table_name
WHERE column LIKE '%XXXX%'

or

SELECT FROM table_name
WHERE column LIKE 'XXXX_'

or

SELECT FROM table_name
WHERE column LIKE '_XXXX'

or

SELECT FROM table name
WHERE column LIKE '_XXXX_'
```

კლიენტი ცხრილისთვის შევასრულოთ LIKE ბრძანებები:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

ქვემოთ მოცემულია მაგალითი, რომელიც აჩვენებს ყველა ჩანაწერს CUSTOMERS ცხრილიდან, სადაც ხელფასი იწყება 200-ით:

SQL> SELECT * FROM CUSTOMERS

WHERE SALARY LIKE '200%';

მიიღება შედეგი:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
+----+----------+-----+-----------+----------+
```

მაგალითები:

```
USE sample;
SELECT emp  fname, emp  lname, emp  no
FROM employee
WHERE emp  fname LIKE '  a%';
```

```
USE sample;
SELECT *
FROM department
WHERE location LIKE '[C-F]%';
```

```
USE sample;
SELECT emp  no, emp  fname, emp  lname
FROM employee
WHERE emp  lname LIKE '[^J-O]%'
AND emp  fname LIKE '[^EZ]%';
```

```
USE sample;
SELECT emp  no, emp  fname, emp  lname
FROM employee
WHERE emp  fname NOT LIKE '%n';
```

```
USE sample;
SELECT project  no, project  name
FROM project
WHERE project  name LIKE '%[  ]%';
SELECT project  no, project  name
FROM project
WHERE project  name LIKE '%!  %' ESCAPE '!';
```

SQL TOP ბრძანება

SQL TOP პუნქტი გამოიყენება ცხრილიდან TOP N ნომრის ან X პროცენტის ჩანაწერების მისაღებად. ყველა მონაცემთა ბაზა არ უჭერს მხარს TOP პუნქტს.

TOP პუნქტის ძირითადი სინტაქსია:

SELECT TOP number|percent column_name(s)

FROM table_name WHERE [condition]

ქვემოთ მოცემულია მაგალითი SQL სერვერზე, რომელიც მიიღებს დასაწყისიდან 3 ჩანაწერს CUSTOMERS ცხრილიდან:

SQL> SELECT TOP 3 * FROM CUSTOMERS;

მიიღება შედეგი:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00

იგივე ბრძანება MySQL-ში დაიწერება შემდეგი სახით:

SQL> SELECT * FROM CUSTOMERS LIMIT 3;

იგივე ბრძანება Oracle -ში დაიწერება შემდეგი სახით:

SQL> SELECT * FROM CUSTOMERS WHERE ROWNUM <= 3;

სამივე ბრძანება მოგვცემს ერთნაირ შედეგს.

SQL ORDER BY ბრძანება

SQL ORDER BY პუნქტი გამოიყენება მონაცემების ზრდის ან კლების მიხედვით დასალაგებლად, ერთი ან მეტი სვეტისთვის.

სინტაქსი:

SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];

ქვემოთ მოცემულია მაგალითი, რომელიც დაალაგებს შედეგს ზრდადი თანმიმდევრობით სახელისა და ხელფასის მიხედვით:

SQL> SELECT * FROM CUSTOMERS ORDER BY NAME, SALARY;

შედეგი:

ID	NAME	AGE	ADDRESS	SALARY
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
3	kaushik	23	Kota	2000.00
2	Khilan	25	Delhi	1500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00
1	Ramesh	32	Ahmedabad	2000.00

ქვემოთ მოცემულია მაგალითი, რომელიც დაალაგებს შედეგს კლებადობით NAME-ის მიხედვით:

SQL> SELECT * FROM CUSTOMERS
ORDER BY NAME DESC;

შედეგი:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
7	Muffy	24	Indore	10000.00
6	Komal	22	MP	4500.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
5	Hardik	27	Bhopal	8500.00
4	Chaitali	25	Mumbai	6500.00

მაგალითები:

```
SELECT emp  fname, emp  lname, dept  no
FROM employee
WHERE emp  no < 20000
ORDER BY emp  lname, emp  fname;
```

```
SELECT project  no, COUNT(*) emp  quantity
FROM works  on
GROUP BY project  no
ORDER BY 2 DESC
```

SQL Group By ბრძანება

SQL GROUP BY პუნქტი გამოიყენება SELECT განცხადებასთან ერთად იდენტური მონაცემების დასაჯგუფებლად. GROUP BY პუნქტი მიჰყვება WHERE პუნქტს SELECT განცხადებაში და წინ უსწრებს ORDER BY პუნქტს.

სინტაქსი:

```
SELECT column1, column2
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2
ORDER BY column1, column2
```

შევასრულოთ ბრძანება CUSTOMERS ცხრილისთვის:

გამოვიტანოთ თითოეული მომხმარებლის ხელფასის მთლიანი ოდენობა, GROUP BY მოთხოვნა იქნება შემდეგი:

```
SQL> SELECT NAME, SUM(SALARY)
FROM CUSTOMERS GROUP BY NAME;
```

შედეგი:

NAME	SUM(SALARY)
Chaitali	6500.00
Hardik	8500.00
kaushik	2000.00
Khilan	1500.00
Komal	4500.00
Muffy	10000.00
Ramesh	2000.00

ვთქვათ გვაქვს ცხრილი, რომელშიც მეორდება სახელები:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Ramesh	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	kaushik	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

გავიგოთ ხელფასის ჯამური რაოდენობა, მოცემული ცხრილისთვის:

```
SQL> SELECT NAME, SUM(SALARY)
FROM CUSTOMERS GROUP BY NAME;
```

შედეგი:

NAME	SUM(SALARY)
Hardik	8500.00
kaushik	8500.00
Komal	4500.00
Muffy	10000.00
Ramesh	3500.00

SQL Distinct ბრძანება

SQL DISTINCT საკვანძო სიტყვა გამოიყენება SELECT განცხადებასთან ერთად ყველა დუბლიკატის აღმოსაფხვრელად, მიიღება მხოლოდ უნიკალური ჩანაწერები. შეიძლება იყოს სიტუაცია, როდესაც თქვენ გაქვთ რამდენიმე დუბლიკატი ჩანაწერი ცხრილში. ასეთი ჩანაწერების გამოტანისას უფრო ლოგიკურია მხოლოდ უნიკალური ჩანაწერების მიღება, დუბლიკატი ჩანაწერების მიღების ნაცვლად.

სინტაქსი:

SELECT DISTINCT column1, column2,.....columnN
FROM table_name
WHERE [condition]

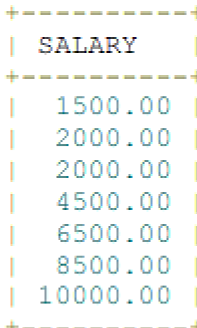
მაგალითად გვაქვს CUSTOMERS ცხრილი, რომელშიც არის დუბლიკატი ჩანაწერები.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

ჯერ ვნახოთ, როგორ აბრუნებს შემდეგი SELECT მოთხოვნა ხელფასის დუბლიკატებს:

SQL> SELECT SALARY
FROM CUSTOMERS ORDER BY SALARY;

შედეგი:



SALARY
1500.00
2000.00
2000.00
4500.00
6500.00
8500.00
10000.00

ახლა გამოვიყენოთ DISTINCT გასაღებური სიტყვა მოთხოვნაში:

```
SQL> SELECT DISTINCT SALARY
 FROM CUSTOMERS ORDER BY SALARY;
```

მიღებული შედეგიდან ჩანს, რომ დუბლიკატები არარ გამოიტანა:

```
+-----------+
| SALARY    |
+-----------+
|  1500.00 |
|  2000.00 |
|  4500.00 |
|  6500.00 |
|  8500.00 |
| 10000.00 |
+-----------+
```

Sub მოთხოვნები

განვიხილოთ ჩაშენებული მოთხოვნების გამოყენების მაგალითები:

```
USE sample;
SELECT emp_fname, emp_lname
FROM employee
WHERE dept_no =
(SELECT dept_no
FROM department
WHERE dept_name = 'Research');
```

ჩაშენებულ მოთხოვნაში IN ოპერატორის გამოყენების მაგალითები:

```
SELECT *
FROM employee
WHERE dept_no IN
(SELECT dept_no
FROM department
WHERE location = 'Dallas');
```

```
USE sample;
SELECT emp_lname
FROM employee
WHERE emp_no IN
(SELECT emp_no
FROM works_on
WHERE project_no IN
(SELECT project_no
FROM project
WHERE project_name = 'Apollo'));
```

ჩაშენებულ მოთხოვნაში ANY და ALL ოპერატორების გამოყენება:

```
USE sample
SELECT DISTINCT emp_no, project_no, job
FROM works_on
WHERE enter_date > ANY
(SELECT enter_date
FROM works_on)
```

აგრეგატული ფუნქციები

განვიხილოთ აგრეგატული ფუნქციების გამოყენების მაგალითები:

```
USE sample;
SELECT emp_lname, MIN(emp_no)
FROM employee;
```

```
SELECT MIN(emp_no) AS min_employee_no
FROM employee;
```

```
USE sample;
SELECT emp_no, emp_lname
FROM employee
WHERE emp_no =
(SELECT MIN(emp_no)
FROM employee);
```

```
SELECT emp_no
FROM works_on
WHERE enter_date =
(SELECT MAX(enter_date)
FROM works_on
WHERE job = 'Manager');
```

```
USE sample;
SELECT SUM (budget) sum_of_budgets
FROM project;
```

```
SELECT SUM (budget) sum_of_budgets
FROM project
GROUP BY();
```

```
SELECT AVG(budget) avg_budget
FROM project
WHERE budget > 100000;
```

```
SELECT project_no, COUNT(DISTINCT job) job_count
FROM works_on
GROUP BY project_no;
```

```
SELECT job, COUNT(*) job_count
FROM works_on
GROUP BY job;
```

```
SELECT project_no
FROM works_on
GROUP BY project_no
HAVING COUNT(*) < 4;
```

```
SELECT job
FROM works_on
GROUP BY job
HAVING job LIKE 'M%';
```

SELECT ოპერატორში IDENTITY თვისების გამოყენება

```
CREATE TABLE product
(product  no INTEGER IDENTITY(10000,1) NOT NULL,
product  name CHAR(30) NOT NULL,
price MONEY);
SELECT IDENTITYCOL
FROM product
WHERE product_name = 'Soap';
```

CASE გამოსახულების გამოყენება

CASE გამოსახულების სინტაქსი:

```
CASE გამოსახულება_1
{WHEN გამოსახულება_2 THEN შედეგი_1} ...
[ELSE შედეგი_n]
END
```

ძებნის CASE გამოსახულების სინტაქსია:

```
 CASE
{WHEN პირობა_1 THEN შედეგი_1} ...
[ELSE შედეგი_n]
END
```

მაგალითები:

```
SELECT project  name,
CASE
WHEN budget > 0 AND budget < 100000 THEN 1
WHEN budget >= 100000 AND budget < 200000 THEN 2
WHEN budget >= 200000 AND budget < 300000 THEN 3
ELSE 4
END budget  weight
FROM project;
```

```
SELECT project  name,
CASE
WHEN p1.budget < (SELECT AVG(p2.budget) FROM project p2)
THEN 'below average'
WHEN p1.budget = (SELECT AVG(p2.budget) FROM project p2)
THEN 'on average'
WHEN p1.budget > (SELECT AVG(p2.budget) FROM project p2)
THEN 'above average'
END budget  category
FROM project p1;
```

ლოგიკური ოპერატორების გამოყენება

```
SELECT project  no, emp  no
FROM works  on
WHERE project  no = 'p1'
OR project  no = 'p2';
```

```
SELECT DISTINCT emp  no
FROM works  on
WHERE project  no = 'p1'
OR project  no = 'p2';

SELECT emp  fname, DISTINCT emp  no
FROM employee
WHERE emp  lname = 'Moser';

SELECT emp  no, emp  fname, emp  lname
FROM employee
WHERE emp  no = 25348 AND emp  lname = 'Smith'
OR emp  fname = 'Matthew' AND dept  no = 'd1';

SELECT emp  no, emp  fname, emp  lname
FROM employee
WHERE ((emp  no = 25348 AND emp  lname = 'Smith')
OR emp  fname ='Matthew') AND dept  no = 'd1';

SELECT emp  no, emp  fname, emp  lname
FROM employee
WHERE (emp  no = 25348 AND emp  lname = 'Smith')
OR (emp  fname = 'Matthew' AND dept  no = 'd1');

SELECT emp  no, emp  lname
FROM employee
WHERE NOT dept  no = 'd2';

SELECT emp  no, emp  fname, emp  lname
FROM employee
WHERE emp  no IN (29346, 28559, 25348);

SELECT emp  no, emp  fname, emp  lname, dept  no
FROM employee
WHERE emp  no NOT IN (10102, 9031);

SELECT project  name, budget
FROM project
WHERE budget BETWEEN 95000 AND 120000;

SELECT project  name, budget
FROM project
WHERE budget >= 95000 AND budget <= 120000;

USE sample;
SELECT project  name
FROM project
WHERE budget NOT BETWEEN 100000 AND 150000;

USE sample;
SELECT project  name
FROM project
WHERE budget < 100000 OR budget > 150000;
```

```
USE sample;
SELECT emp_no, project_no
FROM works_on
WHERE project_no = 'p2'
AND job IS NULL;

USE sample;
SELECT project_no, job
FROM works_on
WHERE job <> NULL;

USE sample;
SELECT emp_no, ISNULL(job, 'Job unknown') AS task
FROM works_on
WHERE project_no = 'p1';
```

SOME და ANY ოპერატორები. ეს ოპერატორები სკალარულ სიდიდეს ადარებენ ქვემოთხოვნის მიერ გაცემულ თითოეულ მნიშვნელობას. მათი სინტაქსია:
სკალარული_გამოსახულება { = | <> | != | > | >= | !> | < | <= | !< } ANY (ქვემოთხოვნა)
სკალარული_გამოსახულება { = | <> | != | > | >= | !> | < | <= | !< } SOME (ქვემოთხოვნა)
ეს ოპერატორები ხშირად გამოიყენება იმის გასარკვევად, არის თუ არა სვეტში საჭირო მნიშვნელობა.

BETWEEN ოპერატორი. იგი საშუალებას გვაძლევს მივუთითოთ მნიშვნელობების დიაპაზონი. მისი სინტაქსია:

გამოსახულება [NOT] BETWEEN საწყისი_გამოსახულება AND საბოლოო_გამოსახულება

საწყისი_გამოსახულება განსაზღვრავს დიაპაზონის დასაწყისს, საბოლოო_გამოსახულება კი - დასასრულს.

EXISTS ოპერატორი. ოპერატორი გასცემს true მნიშვნელობას, თუ ქვემოთხოვნა გასცემს თუნდაც ერთ სტრიქონს. წინააღმდეგ შემთხვევაში, გაიცემა false მნიშვნელობა. მისი სინტაქსია:

EXISTS (ქვემოთხოვნა)

IN ოპერატორი. ამ ოპერატორის საშუალებით შეიძლება შემოწმება ემთხვევა თუ არა გამოსახულების მნიშვნელობა ქვემოთხოვნის მიერ დაბრუნებული მნიშვნელობებიდან ან ჩამოთვლილი სიდიდეებიდან ერთ-ერთს. მისი სინტაქსია:

გამოსახულება [NOT] IN (ქვემოთხოვნა | გამოსახულება [,...n])

Set ოპერატორები

განირჩევა სამი Set ოპერატორი, რომელიც გამოიყენება Transact-SQL ენაში:

- UNION
- INTERSECT

– EXCEPT

UNION Set ოპერატორი

ორი სიმრავლის გაერთიანებისათვის გამოიყენება UNION ოპერატორი:

select_1 UNION [ALL] select_2 {[UNION [ALL] select_3]}...

INTERSECT და EXCEPT Set ოპერატორები

ოპერატორი INTERSECT აღწერს ორ ცხრილს შორის თანაკვეთას, ოპერატორი EXCEPT განსაზღვრავს მათ შორის სხვაობას.

1.5. JOIN, INSERT, UPDATE, DELETE ბრძანებები

მონაცემთა მანიპულირების ენა (Data Manipulation Language – DML), გარდა ამორჩევის SELECT ბრძანებისა, ცხრილში მონაცემების მოსათავსებლად იყენებს INSERT ბრძანებას, ხოლო ცხრილში მოთავსებული მონაცემების მოდიფიცირებისათვის გამოიყენება შემდეგი ბრძანებები: შეცვლა (UPDATE) და წაშლა (DELETE).

JOIN ოპერატორის ვარიანტები

ხშირად მოთხოვნათა შედგენის ანუ მონაცემების ამორჩევის დროს რამდენიმე ცხრილს შორის კავშირის უზრუნველსაყოფად გამოიყენება ოპერატორი JOIN. განირჩევა ამ ოპერატორის რამდენიმე სახესხვაობა:

– CROSS JOIN
– [INNER] JOIN
– LEFT [OUTER] JOIN
– RIGHT [OUTER] JOIN
– FULL [OUTER] JOIN

ცხრილებს შორის კავშირის მეშვეობით მიიღება ე.წ. ბმული ცხრილი, რომლის სინტაქსია:

<ბმული_ცხრილი> ::=

<მარცხენა_ცხრილი> <ბმის_ტიპი> <მარჯვენა_ცხრილი> ON <ბმის_პირობა>

| <მარცხენა_ცხრილი> CROSS JOIN <მარჯვენა_ცხრილი> | <ბმული_ცხრილი>

აქ <მარცხენა_ცხრილი> კონსტრუქცია შეიცავს მთავარი ცხრილის სახელს, რომელსაც დაუკავშირდება სხვა ცხრილები. <ბმის_ტიპი> კონსტრუქცია განსაზღვრავს ორ ცხრილს შორის კავშირის ტიპს. მთავარი ცხრილის სახელი მიეთითება <ბმის_ტიპი> კონსტრუქციის მარცხნივ (მას მარცხენა ცხრილი ეწოდება - left table), მარჯვნივ კი მიეთითება დამოკიდებული ცხრილის სახელი (მას მარჯვენა ცხრილი ეწოდება - right table). <ბმის_ტიპი> კონსტრუქციის სინტაქსია:

<ბმის_ტიპი> ::= [INNER | { { LEFT | RIGHT | FULL } [OUTER] }] JOIN

განვიხილოთ არგუმენტების დანიშნულება.

- INNER კავშირის ტიპი ავტომატურად გამოიყენება. შედეგში ჩართული იქნება მარცხენა ცხრილის მხოლოდ ის სტრიქონები, რომლებისთვისაც არსებობენ სტრიქონები ბმულ (მარჯვენა) ცხრილში. ჩართული იქნება, აგრეთვე, მარჯვენა ცხრილის მხოლოდ ის სტრიქონები, რომლებისთვისაც არსებობენ შესაბამისი სტრიქონები მარცხენა ცხრილში.

- LEFT [OUTER] არგუმენტის გამოყენების შედეგად შედეგში ჩართული იქნება მარცხენა ცხრილის ყველა სტრიქონი, მიუხედავად იმისა, არსებობს თუ არა მათთვის შესაბამისი სტრიქონი მარჯვენა ცხრილში. მარჯვენა ცხრილის შესაბამის ველებს NULL მნიშვნელობა ექნებათ.

- RIGHT [OUTER] არგუმენტის გამოყენების შედეგად შედეგში ჩართული იქნება მარჯვენა ცხრილის ყველა სტრიქონი მიუხედავად იმისა, აქვთ თუ არა მათ შესაბამისი სტრიქონები მარცხენა ცხრილში. მარცხენა ცხრილის შესაბამის ველებს NULL მნიშვნელობა ექნებათ.

- FULL [OUTER] არგუმენტის გამოყენების შედეგად შედეგში ჩართული იქნება მარცხენა და მარჯვენა ცხრილების ყველა სტრიქონი.

- JOIN არგუმენტის შემდეგ მიეთითება მარჯვენა ცხრილი.

- ON <ბმის_პირობა> არის ორივე ცხრილის დაკავშირების ლოგიკური პირობა.

- CROSS JOIN საკვანძო სიტყვის გამოყენებისას სრულდება მარცხენა ცხრილის თითოეული სტრიქონის დაკავშირება მარჯვენა ცხრილის თითოეულ სტრიქონთან. თუ ცხრილებს შორის კავშირი არ არის მითითებული, მაშინ ავტომატურად სრულდება მარცხენა ცხრილის თითოეული სტრიქონის დაკავშირება მარჯვენა ცხრილის თითოეულ სტრიქონთან.

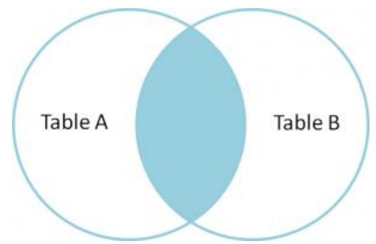
JOIN ოპერატორის ვარიაციების საილუსტრაციოდ განვიხილოთ შემდეგი მაგალითები.

ცხრილი A მარცხნივ და **ცხრილი B** მარჯვნივ შემდეგი მონაცემებით:

```
id  name        id   name
--  ----        --   ----
1  Pirate       1   Rutabaga
2  Monkey       2   Pirate
3  Ninja        3   Darth Vader
4  Spaghetti    4   Ninja
```

INNER JOIN

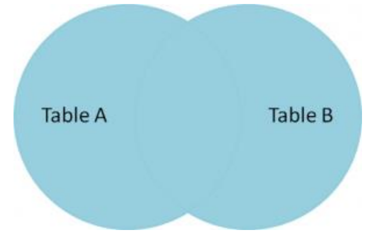
```
 SELECT * FROM TableA
INNER JOIN TableB
ON TableA.name = TableB.name
```




```
id   name        id    name
--   ----        --    ----
1    Pirate      2     Pirate
3    Ninja       4     Ninja
```

FULL OUTER JOIN

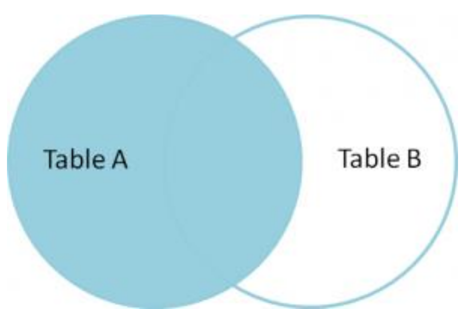
```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.name = TableB.name
```



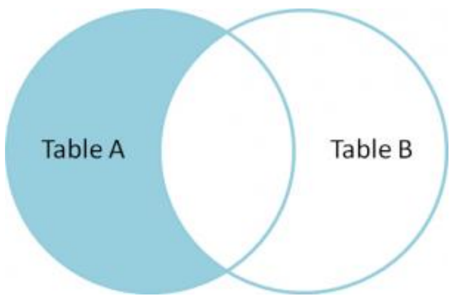

```
id     name        id     name
--     ----        --     ----
1      Pirate      2      Pirate
2      Monkey      null   null
3      Ninja       4      Ninja
4      Spaghetti   null   null
null   null        1      Rutabaga
null   null        3      Darth Vader
```

LEFT OUTER JOIN

```
SELECT * FROM TableA
LEFT OUTER JOIN TableB
ON TableA.name = TableB.name
```

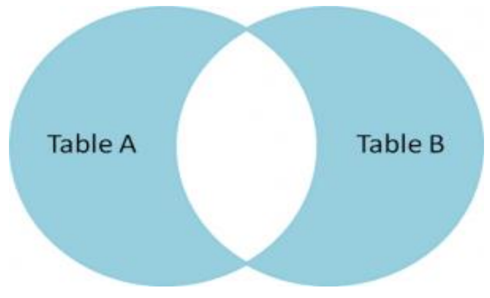



```
id  name       id    name
--  ----       --    ----
1   Pirate     2     Pirate
2   Monkey     null  null
3   Ninja      4     Ninja
4   Spaghetti  null  null
```




```
SELECT * FROM TableA
LEFT OUTER JOIN TableB
ON TableA.name = TableB.name
WHERE TableB.id IS null
```

```
id  name       id    name
--  ----       --    ----
2   Monkey     null  null
4   Spaghetti  null  null
```




```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.name = TableB.name
WHERE TableA.id IS null
```

```
OR TableB.id IS null

id    name       id    name
--    ----       --    ----
2     Monkey     null  null
4     Spaghetti  null  null
null  null       1     Rutabaga
null  null       3     Darth Vader
```

CROSS JOIN – დეკარტული ნამრავლი.

ნატურალური Join

მაგალითი 1

USE sample;
SELECT employee.*, department.*
FROM employee INNER JOIN department
ON employee.dept_no = department.dept_no;

“Old-style” join:

მაგალითი 2

USE sample;
SELECT employee.*, department.*
FROM employee, department
WHERE employee.dept_no = department.dept_no;

მაგალითი 3

USE sample;
SELECT emp_no, project.project_no, job, enter_date, project_name, budget
FROM works_on JOIN project
ON project.project_no = works_on.project_no
WHERE project_name = 'Gemini';

მაგალითი 4

USE sample;
SELECT emp_no, project.project_no, job, enter_date, project_name, budget
FROM works_on, project
WHERE project.project_no = works_on.project_no
AND project_name = 'Gemini';

emp_no	project_no	job	enter_date	project_name	budget
25348	p2	Clerk	2007-02-15	Gemini	95000.0
18316	p2	NULL	2007-06-01	Gemini	95000.0
29346	p2	NULL	2006-12-15	Gemini	95000.0
28559	p2	Clerk	2008-02-01	Gemini	95000.0

მაგალითი 5

USE sample;
SELECT dept_no

```
FROM employee JOIN works_on
ON employee.emp_no = works_on.emp_no
WHERE enter_date = '10.15.2007';
```

მაგალითი 6

```
USE sample;
SELECT emp_fname, emp_lname
FROM works_on JOIN employee ON works_on.emp_no=employee.emp_no
JOIN department ON employee.dept_no=department.dept_no
AND location = 'Seattle'
AND job = 'analyst';
```

მაგალითი 7

```
USE sample;
SELECT DISTINCT project_name
FROM project JOIN works_on
ON project.project_no = works_on.project_no
JOIN employee ON works_on.emp_no = employee.emp_no
JOIN department ON employee.dept_no = department.dept_no
WHERE dept_name = 'Accounting';
```

მაგალითი 8

```
USE sample;
SELECT employee.*, department.*
FROM employee CROSS JOIN department;
```

მაგალითი 9

```
USE sample;
SELECT employee_enh.*, department.location
FROM employee_enh JOIN department
ON domicile = location;
```

მაგალითი 10

```
USE sample;
SELECT employee_enh.*, department.location
FROM employee_enh LEFT OUTER JOIN department
ON domicile = location;
```

emp_no	emp_fname	emp_lname	dept_no	domicile	location
25348	Matthew	Smith	d3	San Antonio	NULL
10102	Ann	Jones	d3	Houston	NULL
18316	John	Barrimore	d1	San Antonio	NULL
29346	James	James	d2	Seattle	Seattle
9031	Elsa	Bertoni	d2	Portland	NULL
2581	Elke	Hansel	d2	Tacoma	NULL
28559	Sybill	Moser	d1	Houston	NULL

მაგალითი 11

```
USE sample;
SELECT employee_enh.domicile, department.*
FROM employee_enh RIGHT OUTER JOIN department
ON domicile =location;
```

domicile	dept_no	dept_name	location
Seattle	d2	Accounting	Seattle
NULL	d1	Research	Dallas
NULL	d3	Marketing	Dallas

მაგალითი 12

```
USE sample;
SELECT employee_enh.*, department.location
FROM employee_enh JOIN department
ON domicile = location
UNION
SELECT employee_enh.*, 'NULL'
FROM employee_enh
WHERE NOT EXISTS
(SELECT *
FROM department
WHERE location = domicile);
```

მაგალითი 13

```
USE sample;
SELECT emp_fname, emp_lname, domicile, location
FROM employee_enh JOIN department
ON domicile < location;
```

მაგალითი 14

```
USE sample;
SELECT t1.dept_no, t1.dept_name, t1.location
FROM department t1 JOIN department t2
ON t1.location = t2.location
WHERE t1.dept_no <> t2.dept_no;
```

მაგალითი 15

```
USE sample;
SELECT emp_no, emp_lname, e.dept_no
FROM employee e JOIN department d
ON e.dept_no = d.dept_no
WHERE location = 'Dallas';
```

მაგალითები:

“orders” ცხრილი

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

“Customer” ცხრილი

CustomerID	CustomerName	ContactName	Country
1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mexico

SQL კოდი, (რომელიც შეიცავს INNER JOIN-ს), ირჩევს ჩანაწერებს, რომლებსაც აქვთ შესაბამისი მნიშვნელობები ორივე ცხრილში:

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

INNER JOIN საკვანძო სიტყვა ირჩევს ყველა სტრიქონს ორივე ცხრილიდან, თუ არის შესაბამისობა სვეტებს შორის. თუ "შეკვეთების" ცხრილში არის ჩანაწერები, რომლებსაც არ აქვთ შესატყვისი "მომხმარებლებში", ეს შეკვეთები არ გამოჩნდება!

LEFT JOIN დააბრუნებს ყველა ჩანაწერს მარცხენა ცხრილიდან (Customers), მაშინაც კი თუ არა აქვს შესატყვისი მნიშვნელობები მარჯვენა ცხრილში (Orders).

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID
ORDER BY Customers.CustomerName;
```

RIGHT JOIN დააბრუნებს ყველა ჩანაწერს მარჯვენა ცხრილიდან (Employees), მაშინაც კი თუ არა აქვს შესატყვისი მნიშვნელობები მარცხენა ცხრილში (Orders)

```
SELECT Orders.OrderID, Employees.LastName, Employees.FirstName
FROM Orders
RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
ORDER BY Orders.OrderID;
```

FULL OUTER JOIN

კოდი ამოარჩევს ყველა Customers და ყველა Orders:

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
FULL OUTER JOIN Orders ON Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerName;
```

UNION ოპერატორი

UNION ოპერატორი, რომელიც აერთიანებს რამდენიმე მოთხოვნის შესრულების შედეგს. მისი სინტაქსია:

select_1 UNION [ALL] select_2 {[UNION [ALL] select_3]}...

შემდეგი SQL განცხადება აბრუნებს ქალაქებს (მხოლოდ განსხვავებულ მნიშვნელობებს) "მომხმარებელთა" და "მომწოდებლების" ცხრილიდან:

```
SELECT City FROM Customers
UNION
SELECT City FROM Suppliers
ORDER BY City;
```

შემდეგი SQL განცხადება აბრუნებს გერმანიის ქალაქებს (მხოლოდ განსხვავებულ მნიშვნელობებს) "მომხმარებელთა" და "მომწოდებლების" ცხრილიდან:

```
SELECT City, Country FROM Customers
WHERE Country='Germany'
UNION
SELECT City, Country FROM Suppliers
WHERE Country='Germany'
ORDER BY City;
```

კორელირებული ქვემოთხოვნები

ქვემოთხოვნა არის კორელირებული, როდესაც შიდა მოთხოვნა დამოკიდებულია გარე მოთხოვნაზე.

მაგალითი **16** გვიჩვენებს კორელირებულ ქვემოთხოვნას.

მაგალითი 16

```
USE sample;
SELECT emp_lname
FROM employee
WHERE 'p3' IN
(SELECT project_no
FROM works_on
WHERE works_on.emp_no = employee.emp_no);
```

შედეგად მივიღებთ:

emp_lname
Jones
Bertoni
Hansel

Set ოპერატორების გამოყენება

განვიხილოთ მაგალითები Set ოპერატორების გამოყენებით:

UNION ოპერატორი

UNION ოპერატორი, რომელიც აერთიანებს რამდენიმე მოთხოვნის შესრულების შედეგს. მისი სინტაქსია:

select_1 UNION [ALL] select_2 {[UNION [ALL] select_3]}...

sample მონაცემთა ბაზაში ამ ოპერატორს შემოჰყავს ახალი ცხრილი **employee_enh**, რომელიც არსებული **employee** ცხრილის იდენტურია, მხოლოდ დამატებულია სვეტი **domicile** :

emp_no	emp_fname emp_lname	dept_no	domicile
25348	Matthew Smith	d3	San Antonio
10102	Ann Jones	d3	Houston
18316	John Barrimore	d1	San Antonio
29346	James James	d2	Seattle
9031	Elke Bertoli	d2	Portland
2581	Elisa Kim	d2	Tacoma
28559	Sybill Moser	d1	Houston

მაგალითში **17** ნაჩვენებია **employee_enh ცხრილის** შექმნა.

მაგალითი 17

```
USE sample;
SELECT emp_no, emp_fname, emp_lname, dept_no
INTO employee_enh
FROM employee;
ALTER TABLE employee_enh
```

```
ADD domicile CHAR(25) NULL;
```

ამ მაგალითში SELECT INTO ახდენს **employee_enh** ცხრილის გენერაციას და შეჰყავს მასში ყველა სტრიქონი საწყისი ცხრილიდან (**employee**) ახალ ცხრილში.

მაგალითი 18

```
USE sample;
SELECT domicile
FROM employee_enh
UNION
SELECT location
FROM department;
```

შედეგად მივიღებთ:

domicile

San Antonio
Houston
Portland
Tacoma
Seattle
Dallas

მაგალითი 19

```
USE sample;
SELECT emp_no
FROM employee
WHERE dept_no = 'd1'
UNION
SELECT emp_no
FROM works_on
WHERE enter_date < '01.01.2007'
ORDER BY 1;
```

შედეგად მივიღებთ:

emp_no
9031
10102
18316
28559
29346

INTERSECT და EXCEPT ოპერატორები

ოპერატორი INTERSECT აღწერს ორ ცხრილს შორის თანაკვეთას, ოპერატორი EXCEPT განსაზღვრავს მათ შორის სხვაობას.

მაგალითი 20

```
USE sample;
SELECT emp_no
FROM employee
WHERE dept_no = 'd1'
INTERSECT
SELECT emp_no
FROM works_on
WHERE enter_date < '01.01.2008';
```

შედეგად მივიღებთ:

emp_no
18316
28559

მაგალითი **21** გვიჩვენებს EXCEPT ოპერატორის გამოყენების შემთხვევას.

მაგალითი 21

```
USE sample;
SELECT emp_no
FROM employee
WHERE dept_no = 'd3'
EXCEPT
SELECT emp_no
FROM works_on
WHERE enter_date > '01.01.2008';
```

შედეგად მივიღებთ:

emp_no
10102
25348

CASE გამოსახულებების გამოყენება

მრავალგანშტოებიანი კონსტრუქცია იყენებს ძირითადად ორი ფორმის (მარტივი და ძებნის) CASE გამოსახულებას. მარტივი გამოსახულების სინტაქსია:

```
CASE გამოსახულება_1
{WHEN გამოსახულება_2 THEN შედეგი_1} ...
[ELSE შედეგი_n]
```

END

ძებნის CASE გამოსახულების სინტაქსია:

```
 CASE
{WHEN პირობა_1 THEN შედეგი_1} ...
[ELSE შედეგი_n]
END
```

მაგალითი **22** გვიჩვენებს CASE გამოსახულების გამოყენების შემთხვევას. expression.

მაგალითი 22

```
USE sample;
SELECT project_name,
CASE
WHEN budget > 0 AND budget < 100000 THEN 1
WHEN budget >= 100000 AND budget < 200000 THEN 2
WHEN budget >= 200000 AND budget < 300000 THEN 3
ELSE 4
END budget_weight
FROM project;
```

შედეგად მივიღებთ:

project_name	budget_weight
Apollo	2
Gemini	1
Mercury	2

მაგალითი 23

```
SELECT project_name,
CASE
WHEN p1.budget < (SELECT AVG(p2.budget) FROM project p2)
THEN 'below average'
WHEN p1.budget = (SELECT AVG(p2.budget) FROM project p2)
THEN 'on average'
WHEN p1.budget > (SELECT AVG(p2.budget) FROM project p2)
THEN 'above average'
END budget_category
FROM project p1;
```

შედეგად მივიღებთ:

project_name budget_category

Apollo below average

Gemini below average

Mercury above average

მაგალითი 24

```
USE sample;
CREATE TABLE #project_temp
(project_no CHAR(4) NOT NULL,
project_name CHAR(25) NOT NULL);
```

მაგალითი 25

```
USE sample;
SELECT project_no, project_name
INTO #project_temp
FROM project;
```

1.6. წარმოდგენები

წარმოდგენა (view) არის ე.წ. ვირტუალური ცხრილი, რომელიც ფიზიკურად არ არსებობს, არამედ დინამიურად იწარმოება ერთი ან მეტი ცხრილიდან და/ან სხვა წარმოდგენიდან.

წარმოდგენის შესაქმნელად გამოიყენება CREATE VIEW ბრძანება. მისი სინტაქსია:

CREATE VIEW [სქემის_სახელი.] წარმოდგენის_სახელი [(სვეტის_სახელი [,...n])] [WITH <წარმოდგენის_ატრიბუტები> [,...n]] AS select_ბრძანება [WITH CHECK OPTION]

<წარმოდგენის_ატრიბუტები> კონსტრუქციის სინტაქსია:

<წარმოდგენის_ატრიბუტები> ::= { ENCRYPTION | SCHEMABINDING | VIEW_METADATA }

წარმოდგენის_სახელი არის სახელი, რომელიც განსაზღვრავს წარმოდგენას.

სვეტის_სახელი არის წარმოდგენაში ჩართული სვეტის სახელი.

ENCRYPTION არგუმენტი მიუთითებს, რომ უნდა მოხდეს უსაფრთხოების გაზრდის მიზნით წარმოდგენის კოდის დაშიფვრა, რათა წარმოდგენის განსაზღვრების ნახვა შეუძლებელი გახდეს.

SCHEMABINDING არგუმენტის მითითების შემთხვევაში სერვერი წარმოდგენის სტრუქტურას დააკავშირებს იმ ობიექტების სტრუქტურასთან, რომლებსაც SELECT მოთხოვნა მიმართავს.

VIEW_METADATA არგუმენტის მითითების შემთხვევაში სერვერი გასცემს წარმოდგენის მეტამონაცემებს DB-LIBRARY და OLE DB ტექნოლოგიებისათვის.

WITH CHECK OPTION არგუმენტის მითითების დროს ხდება წარმოდგენის საშუალებით შესრულებული ცვლილებების შემოწმება.

წარმოდგენის შექმნისას მისი სახელი შეინახება მიმდინარე მონაცემთა ბაზის sys.objects სისტემურ წარმოდგენაში (SELECT name FROM sys.objects ORDER BY name), ხოლო კოდი კი - syscomments სისტემურ წარმოდგენაში (SELECT text FROM syscomments). წარმოდგენაში განსაზღვრული სვეტების სია ინახება sys.columns წარმოდგენაში (SELECT name FROM sys.columns ORDER BY name), ხოლო ინფორმაცია წარმოდგენის საწყის ცხრილებზე დამოკიდებულების შესახებ კი - sysdepends სისტემურ წარმოდგენაში (SELECT * FROM sysdepends).

წარმოდგენების მართვა

წარმოდგენის შეცვლა

შექმნილი წარმოდგენის შესაცვლელად გამოიყენება ALTER VIEW ბრძანება. მისი საშუალებით შესაძლებელია ინდექსირებული წამოდგენის შეცვლაც. მისი სინტაქსია:

ALTER VIEW [სქემის_სახელი .] წარმოდგენის_სახელი [(სვეტის_სახელი [,...n])]
[WITH <წარმოდგენის_ატრიბუტები> [,...n]]
AS
select_ბრძანება [;]
[WITH CHECK OPTION]

სვეტის_სახელი წარმოდგენის ერთი ან მეტი სვეტის სახელია. სვეტის ნებართვები მოქმედებს მხოლოდ მაშინ, როცა სვეტებს ერთნაირი სახელები აქვთ ALTER VIEW ბრძანების შესრულებამდე და შესრულების შემდეგ.
თუ წარმოდგენა შეიქმნა **WITH ENCRYPTION** ან **CHECK** OPTION არგუმენტების გამოყენებით, მაშინ ეს რეჟიმები ძალაში იქნება იმ შემთხვევაში, თუ ეს არგუმენტები მითითებული იქნება ALTER VIEW ბრძანებაში.

წარმოდგენის სახელის შეცვლა

წარმოდგენის სახელის შესაცვლელად გამოიყენება sp_rename ბრძანება.
მაგალითი. View_1 წარმოდგენას დავარქვათ View_2 სახელი.
sp_rename View_1, View_2

წარმოდგენის წაშლა

წარმოდგენის წასაშლელად გამოიყენება DROP VIEW ბრძანება. მისი სინტაქსია:

DROP VIEW [სქემის_სახელი .] წარმოდგენის_სახელი [,...n]

წარმოდგენის წაშლის უფლება აქვს მხოლოდ მის მფლობელს და ეს უფლება არ შეიძლება სხვას გადაეცეს. თუ საჭიროა, რომ სხვა მომხმარებელმა წაშალოს წარმოდგენა, მაშინ მას უნდა გადავცეთ წარმოდგენის ფლობის უფლება. db_owner და sysadmin როლის წევრებს შეუძლიათ წარმოდგენების წაშლა.
წარმოდგენის წაშლის შედეგად წაიშლება ყველა მისგან წარმოებული წარმოდგენები.

წარმოდგენის შესახებ ინფორმაციის მიღება

წარმოდგენის შესახებ ინფორმაციის მისაღებად გამოიყენება sp_help შენახვადი პროცედურა.
მაგალითი.

sp_help 'View_1'

ამ ბრძანების შესრულების შედეგად გამოვა ცხრილი, რომელიც შეიცავს წარმოდგენის სვეტებს მათი თვისებების აღწერით. კოდის მისაღებად, რომლის საშუალებითაც შეიქმნა წარმოდგენა, შეგვიძლია გამოვიყენოთ sp_helptext შენახვადი პროცედურა. მისი სინტაქსია:
sp_helptext [@objname =] 'წარმოდგენის_სახელი'

წარმოდგენის დამოკიდებულებების ნახვა

წარმოდგენის დამოკიდებულებების ასახვისათვის გამოიყენება შენახვადი პროცედურა sp_depends, რომელსაც არ გამოაქვს ინფორმაცია იმ ობიექტების შესახებ, რომლებიც სხვა მონაცემთა ბაზაშია განთავსებული.
მაგალითი. მივიღოთ იმ ობიექტების სია, რომლებზეც View_1 წარმოდგენაა დამოკიდებული:
sp_depends 'View_1'

ინდექსირებული წარმოდგენები

ინდექსირებული წარმოდგენების შექმნა. წარმოდგენების ინდექსირების პროცესი ორბიჯიანია:
1. წარმოდგენის შექმნა CREATE VIEW ბრძანებით WITH SCHEMABINDING–თან ერთად.
2. შესაბამისი კლასტერული ინდექსის შექმნა.

მაგალითი 1

```
USE sample;
GO
CREATE VIEW v_clerk
AS SELECT emp_no, project_no, enter_date
FROM works_on
WHERE job = 'Clerk';
```

მაგალითი 2

```
USE sample;
GO
CREATE VIEW v_without_budget
AS SELECT project_no, project_name
```

```
FROM project;
```

მაგალითი 3

```
USE sample;
GO
CREATE VIEW v_count(project_no, count_project)
AS SELECT project_no, COUNT(*)
FROM works_on
GROUP BY project_no;
```

მაგალითი 4

```
USE sample;
GO
CREATE VIEW v_count1
AS SELECT project_no, COUNT(*) count_project
FROM works_on
GROUP BY project_no;
```

მაგალითი 5

```
USE sample;
GO
CREATE VIEW v_project_p2
AS SELECT emp_no
FROM v_clerk
WHERE project_no ='p2';
```

მაგალითი 6

```
USE sample;
GO
ALTER VIEW v_without_budget
AS SELECT project_no, project_name
FROM project
WHERE project_no >= 'p3';
```

მაგალითი 7

```
USE sample;
GO
DROP VIEW v_count;
```

მაგალითი 8

```
USE sample;
GO
DROP VIEW v_clerk;
```

მაგალითი 9

```
sp_helptext 'View_1'
```

მაგალითი 10

```
USE sample;
GO
CREATE VIEW v_dept
AS SELECT dept_no, dept_name
FROM department;
GO
INSERT INTO v_dept
VALUES('d4', 'Development');
```

მაგალითი 11

```
USE sample;
GO
CREATE VIEW v_2006_check
AS SELECT emp_no, project_no, enter_date
FROM works_on
WHERE enter_date BETWEEN '01.01.2006' AND '12.31.2006'
WITH CHECK OPTION;
GO

INSERT INTO v_2006_check
VALUES (22334, 'p2', '1.15.2007');
```

მაგალითი 12

```
USE sample;
GO
CREATE VIEW v_2006_nocheck
AS SELECT emp_no, project_no, enter_date
FROM works_on
WHERE enter_date BETWEEN '01.01.2006' AND '12.31.2006';
GO
INSERT INTO v_2006_nocheck
VALUES (22334, 'p2', '1.15.2007');
SELECT *
FROM v_2006_nocheck;
```

მაგალითი 13

```
USE sample;
```

```
GO
CREATE VIEW v_sum(sum_of_budget)
AS SELECT SUM(budget)
FROM project;
GO
SELECT *
FROM v_sum;
```

მაგალითი 14

```
USE sample;
GO
CREATE VIEW v_p1
AS SELECT emp_no, job
FROM works_on
WHERE project_no = 'p1';
GO
UPDATE v_p1
SET job = NULL
WHERE job = 'Manager';
```

მაგალითი 15

```
USE sample;
GO
CREATE VIEW v_100000
AS SELECT project_no, budget
FROM project
WHERE budget > 100000
WITH CHECK OPTION;
GO
UPDATE v_100000
SET budget = 93000
WHERE project_no = 'p3';
```

მაგალითი 16

```
USE sample;
GO
CREATE VIEW v_uk_pound (project_number, budget_in_pounds)
AS SELECT project_no, budget*0.65
FROM project
WHERE budget > 100000;
```

```
GO
SELECT *
FROM v_uk_pound;
```

მაგალითი 17

```
USE sample;
GO
CREATE VIEW v_project_p1
AS SELECT emp_no, job
FROM works_on
WHERE project_no = 'p1';
GO
DELETE FROM v_project_p1
WHERE job = 'Clerk';
```

მაგალითი 18

```
USE sample;
GO
CREATE VIEW v_budget (budget_reduction)
AS SELECT budget*0.9
FROM project;
GO
DELETE FROM v_budget;
```

მაგალითი 19

```
USE sample;
GO
CREATE VIEW v_enter_month
WITH SCHEMABINDING
AS SELECT emp_no, DATEPART(MONTH, enter_date) AS enter_month
FROM dbo.works_on;
```

1.7. შენახვადი პროცედურები. მომხმარებლის მიერ შემუშავებული ფუნქციები

შენახვადი პროცედურები

შენახვადი პროცედურები (stored procedure) წარმოადგენს მონაცემთა ბაზის დამოუკიდებელ ობიექტს, რომელთა დახმარებით იქმნება ქვეპროგრამები, ინახება და მუშაობენ უშუალოდ სერვერზე. SQL ენა არაპროცედურულია, მაგრამ SQL Server-ში გამოიყენება გასაღებური სიტყვები, რომლებიც მონაწილეობენ პროცედურების შექმნის დროს და ისინი შეიძლება შენახულ იქნას შემდეგ შესრულებამდე. პროგრამირების სტანდარტული ენების (C, Visual Basic და სხვ.) გამოყენების ნაცვლად, მონაცემთა ბაზებში ოპერაციათა შესასრულებლად შეიძლება შენახვადი პროცედურების გამოყენება. მათი გამოძახება შესაძლებელია კლიენტის პროგრამის, სხვა შენახვადი პროცედურის ან ტრიგერის მიერ.

შენახვადი პროცედურების შესრულების მართვისათვის გამოიყენება Transact - SQL-ის შემდეგი კონსტრუქციები:

კონსტრუქცია IF . . . ELSE:

კონსტრუქცია BEGIN . . . END:

ინსტრუქცია WHILE:

გასაღებური სიტყვა BREAK:

გასაღებური სიტყვა CONTINUE:

Transact - SQL-ში ლოკალური ცვლადები განისაზღვრება ინსტრუქციით:

DECLARE @ ცვლადის-სახელი მონაცემთა-ტიპი [, . . . n]

გლობალური ცვლადები პროგრამაში არ განისაზღვრება, არამედ სისტემაში სერვერის დონეზე.

ინსტრუქცია PRINT ცვლადებთან:

გასაღებური სიტყვა GOTO

გასაღებური სიტყვა RETURN

გასაღებური სიტყვა RAISERROR

ინსტრუქცია WAITFOR

გამოსახულება CASE

შენახვადი პროცედურის კოდის შეცვლა შეუძლია მხოლოდ მის მფლობელს ან მონაცემთა ბაზის db_owner ფიქსირებული როლის წევრს. საჭიროების შემთხვევაში შენახვადი პროცედურის ფლობის უფლება შეგვიძლია ერთი მომხმარებლიდან მეორეს გადავცეთ. შენახვადი პროცედურის გამოყენებას გააჩნია შემდეგი უპირატესობები:

– შენახვადი პროცედურის შესრულების წინ ხდება მათი დაგეგმვა ანუ სერვერი მისთვის ადგენს შესრულების გეგმას (execution plan), ასრულებს მის ოპტიმიზებას და

კომპილირებას. ეს თავის მხრივ, იძლევა ოპერაციების შესრულების სიჩქარის გაზრდის შესაძლებლობას, რადგან მათი განმეორებით გამოყენების შემთხვევაში, ისინი უკვე ჩატვირთულია ოპერატიულ მეხსიერებაში, სადაც მათი მოძებნა გაცილებით სწრაფად ხდება.

– შენახვადი პროცედურის შესასრულებლად საკმარისია მისი სახელის მითითება, რაც ამცირებს მოთხოვნის ზომას, რომელიც ქსელში იგზავნება კლიენტიდან სერვერისკენ. შედეგად, შენახვადი პროცედურების გამოყენება ამცირებს ქსელზე დატვირთვებს.

– შენახვადი პროცედურების გამოყენება აადვილებს მოდულური პროექტირების პრინციპის რეალიზებას, რადგან პროცედურები იძლევიან დიდი ამოცანების უფრო პატარა, დამოუკიდებელ და ადვილად სამართავ ამოცანებად დაყოფის საშუალებას.

შენახვადი პროცედურების ტიპები

განირჩევა შენახვადი პროცედურების რამდენიმე ტიპი:

– ***სისტემური შენახვადი პროცედურები*** (System stored procedures) ინახება master სისტემურ მონაცემთა ბაზაში და შეგვიძლია ნებისმიერი მონაცემთა ბაზიდან გამოვიძახოთ. ისინი განკუთვნილია ისეთი ადმი-ნისტრაციული მოქმედებების შესასრულებლად, როგორიცაა სააღრიცხვო ჩანაწერების შექმნა, მონაცემთა ბაზების ობიექტების შესახებ ინფორმაციის მიღება, სერვერისა და მონაცემთა ბაზის თვისებების მართვა, ავტომატიზებისა და რეპლიკაციის ქვესისტემის მართვა და ა.შ. სისტემურ შენახულ პროცედურებს აქვთ **sp_** პრეფიქსი (system procedure).

– ***მომხმარებლის მიერ განსაზღვრული შენახვადი პროცედურები*** (User_defined stored procedures) ინახება შესაბამის მონაცემთა ბაზაში და წარმოადგენს ამ მონაცემთა ბაზის ობიექტს.

– ***დროებითი შენახვადი პროცედურები*** (Temporary stored procedures). განირჩევა ლოკალური და გლობალური დროებითი პროცედურები:

➢ *ლოკალური დროებითი შენახვადი პროცედურების* (Local temporary stored procedures) სახელები # სიმბოლოთი იწყება. ისინი ინახება tempdb მონაცემთა ბაზაში და ავტომატურად იშლება მომხმარებლის გამორთვისას, სერვერის გაჩერების ან ხელახალი გაშვების დროს.

➢ *გლობალური დროებითი შენახვადი პროცედურები* (Global temporary stored procedures) შეიძლება გამოძახებული იყოს ნებისმიერი შეერთებიდან. ასეთი პროცედურის განსაზღვრისას მისი სახელის წინ უნდა მოვათავსოთ ## სიმბოლოები. ეს პროცედურები ინახება tempdb მონაცემთა ბაზაში და იშლება სერვერის გაჩერების ან ხელახალი გაშვების დროს, აგრეთვე, იმ შეერთების დახურვისას, რომლშიც ისინი შეიქმნა.

შენახვადი პროცედურის შექმნა და შესრულება

შენახვადი პროცედურის შესაქმნელად გამოიყენება CREATE PROCEDURE ბრძანება. მისი სინტაქსია:

CREATE PROC [EDURE] [სქემის_სახელი.] პროცედურის_სახელი
[{ @პარამეტრის_სახელი მონაცემთა_ტიპი } [VARYING] [= DEFAULT] [OUTPUT]] [,...n]
[WITH { RECOMPILE | ENCRYPTION | RECOMPILE, ENCRYPTION }]
AS sql_ბრძანება [,...n]

განვიხილოთ არგუმენტების დანიშნულება.

– პროცედურის_სახელი არის შესაქმნელი პროცედურის სახელი. თუ გამოვიყენებთ sp_, # და ## პრეფიქსებს, მაშინ შესაბამისად შეგვიძლია შევქმნათ სისტემური, ლოკალური და გლობალური დროებითი პროცედურები. შენახვადი პროცედურა ყოველთვის იქმნება მიმდინარე მონაცემთა ბაზაში.

– @პარამეტრის_სახელი ცვლადი შეიცავს იმ პარამეტრის სახელს, რომელსაც შენახვადი პროცედურა გამოიყენებს შესასვლელი და გამოსასვლელი მონაცემების გადაცემის მიზნით. შენახვადი პროცედურის პარამეტრების სახელები @ სიმბოლოთი უნდა იწყებოდეს. პარამეტრები ერთმანეთისაგან მძიმეებით გამოიყოფა. შენახვადი პროცედურის პარამეტრები წარმოადგენენ ლოკალურ პარამეტრებს, ამიტომ სხვადასხვა შენახულ პროცედურებს შეიძლება ერთნაირი პარამეტრები ჰქონდეს. შენახულ პროცედურაში პარამეტრის სახელი ცვლადის სახელს არ უნდა ემთხვეოდეს.

– მონაცემთა_ტიპი არის პარამეტრის ტიპი. პარამეტრს შეიძლება ნებისმიერი ტიპი ჰქონდეს, მათ შორის მომხმარებლის მიერ განსაზღვრული ტიპები. რაც შეეხება cursor ტიპს, ის შეგვიძლია გამოვიყენოთ მხოლოდ გამოსასვლელი პარამეტრებისათვის.

– OUTPUT არგუმენტი მიუთითებს, რომ პარამეტრი არის გამოსასვლელი, ანუ შეგვიძლია გამოვიყენოთ მონაცემების დასაბრუნებლად შენახვადი პროცედურიდან. ასეთი პარამეტრი შეგვიძლია გამოვიყენოთ როგორც შესასვლელიც. შენახვადი პროცედურიდან გამოსვლისას გამოსასვლელი პარამეტრის მიმდინარე მნიშვნელობა ენიჭება იმ ლოკალურ ცვლადს, რომელიც მითითებული იყო შენახვადი პროცედურის გამოძახების დროს.

– VARYING არგუმენტს აქვს cursor ტიპი და გამოიყენება OUTPUT არგუმენტთან ერთად. ის მიუთითებს, რომ გამოსასვლელი პარამეტრი იქნება სიმრავლე.

– DEFAULT არგუმენტი მიუთითებს იმ მნიშვნელობას, რომელსაც პარამეტრი ავტომატურად მიიღებს თუ მისი მნიშვნელობა არ იქნება მითითებული.

– RECOMPILE არგუმენტი მიუთითებს, რომ შენახვადი პროცედურის ყოველი გამოძახებისას უნდა შედგეს მისი შესრულების გეგმა და შესრულდეს შენახვადი პროცედურის კოდის ხელახალი კომპილირება.

– ENCRYPTION არგუმენტი მიუთითებს, რომ უნდა შესრულდეს შენახვადი პროცედურის კოდის დაშიფვრა.

– AS არგუმენტი იწყებს შენახვადი პროცედურის ტანს (კოდს). ტანში დასაშვებია Transact_SQL-ის პრაქტიკულად ყველა ბრძანების გამოყენება, ტრანზაქციების გამოცხადება და სხვა შენახვადი პროცედურების გამოძახება. შენახვადი პროცედურიდან გამოსასვლელად შეგვიძლია გამოვიყენოთ RETURN ბრძანება.

ერთი შენახვადი პროცედურის ტანიდან მეორე შენახვადი პროცედურის გამოძახებისათვის იქმნება ჩადგმული პროცედურები (nested procedure). ჩადგმულობის მიმდინარე დონის მისაღებად შეგვიძლია @@NESTLEVEL ცვლადის გამოყენება (SELECT @@NESTLEVEL AS [ჩადგმულობის დონე]).

არსებობს შენახვადი პროცედურის აქტივიზაციის ორი გზა:

- მხოლოდ მისი სახელის მითითება;
- EXECUTE ბრძანების გამოყენება.

მხოლოდ სახელის მითითებით შენახვადი პროცედურის გამოძახების დროს უნდა იყოს ერთადერთი ბრძანება შესასრულებლად გადასაცემ პაკეტში. წინააღმდეგ შემთხვევაში გამოიყენება EXECUTE ბრძანება. იგი შეიძლება გამოვიყენოთ, აგრეთვე, შენახვადი პროცედურის გამოძახებისას სხვა შენახვადი პროცედურიდან ან ტრიგერიდან.

EXECUTE ბრძანების სინტაქსია:

[EXEC [UTE]] პროცედურის_სახელი
[[@პარამეტრის_სახელი =] { სიდიდე | @ცვლადის_სახელი } [OUTPUT] | [DEFAULT]] [,...n]

OUTPUT ან DEFAULT არგუმენტის გამოყენება ნებადართულია იმ შემთხვევაში, როცა შესაბამისი პარამეტრი გამოცხადებული იყო OUTPUT სიტყვის გამოყენებით შენახვადი პროცედურის შექმნის დროს.

შენახვადი პროცედურის შეცვლა

შენახვადი პროცედურის შესაცვლელად გამოიყენება ALTER PROCEDURE ბრძანება. მისი სინტაქსია:

ALTER { PROC | PROCEDURE } [სქემის_სახელი.] პროცედურის_სახელი
[{ @პარამეტრის_სახელი მონაცემთა_ტიპი }
[VARYING] [= DEFAULT] [[OUT [PUT]] [,...n]
[WITH <პროცედურის_რეჟიმი> [,...n]]
[FOR REPLICATION]
AS
{ <sql_ბრძანება> [...n] }

<პროცედურის_რეჟიმი> კონსტრუქციის სინტაქსია:

<პროცედურის_რეჟიმი> ::=

[ENCRYPTION] [RECOMPILE]
<sql_ბრძანება> კონსტრუქციის სინტაქსია:
<sql_ბრძანება> ::=
{ [BEGIN] ბრძანებები [END] }

შენახვადი პროცედურის შესახებ ინფორმაციის მიღება

ნებისმიერი ობიექტის შესახებ, მათ შორის შენახვადი პროცედურის შესახებ ინფორმაციის მისაღებად, უნდა გამოვიყენოთ sp_help პროცედურა. მისი სინტაქსია:

sp_help 'პროცედურის_სახელი'

შენახვადი პროცედურის სახელის შეცვლა

შენახვადი პროცედურის სახელის შესაცვლელად გამოიყენება sp_rename ბრძანება. შენახვადი პროცედურისათვის სახელის შეცვლის დროს, იცვლება მხოლოდ სახელი, რომელიც sysobjects სისტემურ წარმოდგენაში ინახება. ამ პროცედურაზე მიმართვები უცვლელი რჩება. ამიტომ, ჩვენ თვითონ მოგვიწევს ამ მიმართვების შეცვლა.

შენახვადი პროცედურის წაშლა

შენახვადი პროცედურის წასაშლელად გამოიყენება DROP PROCEDURE ბრძანება. მისი სინტაქსია:
DROP PROC [EDURE] [სქემის_სახელი.] პროცედურის_სახელი [,...n]

შენახვადი პროცედურის წაშლის შედეგად მისი სახელი წაიშლება ამავე მონაცემთა ბაზის sysobjects სისტემური წარმოდგენიდან. იმისათვის, რომ წავშალოთ და ისევ შევქმნათ ერთი და იგივე სახელის მქონე ობიექტი ერთსა და იმავე პაკეტში, ბრძანებებს შორის უნდა მივუთითოთ GO ბრძანება.

შენახვადი პროცედურის ავტომატურად შესრულების მართვა

შენახვადი პროცედურის ავტომატურად შესრულებისათვის საჭიროა მისი კონფიგურირება, რაც სრულდება მისი თვისებების ცვლილების გზით. ამისათვის, გამოიყენება sp_procoption პროცედურა. მისი სინტაქსია:
sp_procoption [@ProcName =] 'პროცედურის_სახელი',
[@OptionName =] 'რეჟიმი', [@OptionValue =] 'მნიშვნელობა'
განვიხილოთ არგუმენტების დანიშნულება.

– რეჟიმი არის შენახვადი პროცედურის იმ თვისების სახელი, რომელსაც ვცვლით. ამ შემთხვევაში, ეს არის startup თვისება, რომელიც განსაზღვრავს შენახვადი პროცედურის ავტომატურად გაშვების შესაძლებლობას.
– მნიშვნელობა არგუმენტი იღებს true (ON) ან false (OFF) მნიშვნელობას.

sp_procoption სისტემური შენახვადი პროცედურის შესრულების უფლება აქვს მხოლოდ სერვერის ფიქსირებული როლის წევრებს. ნებადართულია მხოლოდ master

მონაცემთა ბაზის შენახვადი პროცედურების ავტომატურად გაშვება, როდესაც შენახვადი პროცედურა ეკუთვნის მონაცემთა ბაზის მფლობელს - dbo მომხმარებელს.

მაგალითი 1

```
USE sample;
GO
CREATE PROCEDURE increase_budget (@percent INT=5)
AS UPDATE project
SET budget = budget + budget*@percent/100;
```

მაგალითი 2

```
USE sample;
EXECUTE increase_budget 10;
```

მაგალითი 3

```
USE sample;
GO
CREATE PROCEDURE modify_empno (@old_no INTEGER, @new_no INTEGER)
AS UPDATE employee
SET emp_no = @new_no
WHERE emp_no = @old_no
UPDATE works_on
SET emp_no = @new_no
WHERE emp_no = @old_no
```

მაგალითი 4

```
USE sample;
GO
CREATE PROCEDURE delete_emp @employee_no INT, @counter INT OUTPUT
AS SELECT @counter = COUNT(*)
FROM works_on
WHERE emp_no = @employee_no
DELETE FROM employee
WHERE emp_no = @employee_no
DELETE FROM works_on
WHERE emp_no = @employee_no
```

მოცემული შენახვადი პროცედურა შეიძლება შესრულდეს შემდეგი ბრძანებებით:

```
DECLARE @quantity INT
EXECUTE delete_emp @employee_no=28559, @counter=@quantity OUTPUT
```

მაგალითი 5

```
USE sample;
DECLARE @ret INT
EXECUTE @ret=GetEmployeeCount
PRINT @ret
```

The PRINT statement returns the current number of the rows in the **employee** table.

მომხმარებლის მიერ შემუშავებული ფუნქციები

მომხმარებლის მიერ შემუშავებული ფუნქციები (User Defined Functions – UDF) წარმოადგენენ მონაცემთა ბაზის დამოუკიდებელ ობიექტებს და განთავსებულნი არიან შესაბამის მონაცემთა ბაზაში. იქმნება CREATE FUNCTION ბრძანებით, რომლის სინტაქსია:

```
CREATE FUNCTION [სქემის_სახელი.]ფუნქციის_ სახელი
[({@პარამეტრი }ტიპი [= default]) {,...}
RETURNS {სკალარი_ტიპი | [@ცვლადი]ცხრილი}
[WITH {ENCRYPTION | SCHEMABINDING}
[AS] {ბლოკი | RETURN (შერჩეული ინსტრუქცია)}
```

სქემის_სახელი არგუმენტი მიუთითებს იმ სქემის სახელს, რომელსაც ფუნქცია ეკუთვნის.

@პარამეტრის_სახელი მონაცემთა_ტიპი [= DEFAULT], რომელიც ფუნქციის პარამეტრებს განსაზღვრავს, უნდა იყოს უნიკალური და იწყებოდეს @ სიმბოლოთი.

WITH კონსტრუქციას აქვს შემდეგი სინტაქსი:

```
<ფუნქციის_რეჟიმი> ::= { ENCRYPTION | SCHEMABINDING }
```

ENCRYPTION საკვანძო სიტყვის გამოყენება იწვევს CREATE FUNCTION ბრძანების კოდის დაშიფვრას. SCHEMABINDING ფუნქცია შეიძლება მიმართავდეს მონაცემთა ბაზის სხვადასხვა ობიექტებს. AS საკვანძო სიტყვას მოსდევს ფუნქციის ტანი. BEGIN...END საკვანძო სიტყვებს შორის მოთავსებულია ფუნქციის ტანი (კოდი). RETURN ფუნქციის მუშაობას ამთავრებს და შედეგს გასცემს. UDF შეიძლება გაუმჯობესდეს Transact-SQL ისეთი ბრძანებებით, როგორიცაა SELECT, INSERT, UPDATE ან DELETE.

მომხმარებლის მიერ შემუშავებული ფუნქციის შეცვლა

ფუნქციის შესაცვლელად გამოიყენება ALTER FUNCTION ბრძანება, რომლის სინტაქსია:

```
ALTER FUNCTION ფუნქციის_სახელი
(
[ { @პარამეტრის_სახელი [ AS ] მონაცემთა_ტიპი [ = DEFAULT ] } [ ,...n ] ]
)
RETURNS დასაბრუნებელი_მნიშვნელობის_ტიპი
```

```
[ WITH <ფუნქციის_რეჟიმი> [ ,...n ] ]
[ AS ]
BEGIN
ფუნქციის_კოდი
RETURN სკალარული_გამოსახულება
END
[ ; ]
```

მომხმარებლის მიერ შემუშავებული ფუნქციის წაშლა

ფუნქციის წასაშლელად გამოიყენება DROP FUNCTION ბრძანება. მისი სინტაქსია:

```
DROP FUNCTION { [ სქემის_სახელი.] ფუნქციის_სახელი } [,...n]
```

მხოლოდ ფუნქციის მფლობელს (ან **db_owner** და **sysadmin–ის** წევრებს) შეუძლიათ ფუნქციის წაშლა. ამიტომ, თუ ვშლით ფუნქციას, რომელიც სხვა მომხმარებელს ან მონაცემთა ბაზის სხვა მფლობელს ეკუთვნის, მაშინ უნდა მივუთითოთ ფუნქციის მფლობელის სახელიც. ერთი ბრძანებით შეიძლება რამდენიმე ფუნქციის წაშლა.

განვიხილოთ რამდენიმე მაგალითი. მაგალითი 1 **compute_costs** ფუნქციის შექმნას გვიჩვენებს:

მაგალითი 1

```
USE sample;
GO
CREATE FUNCTION compute_costs (@percent INT =10)
RETURNS DECIMAL(16,2)
BEGIN
DECLARE @additional_costs DEC (14,2), @sum_budget dec(16,2)
SELECT @sum_budget = SUM (budget) FROM project
SET @additional_costs = @sum_budget * @percent/100
RETURN @additional_costs
END
```

მაგალითი 2-ის შემთხვევაში **compute_costs** ფუნქცია გამოიყენება SELECT ინსტრუქციაში.

მაგალითი 2

```
USE sample;
SELECT project_no, project_name
FROM project
WHERE budget < dbo.compute_costs(25)
```

შედეგად მივიღებთ:

project_no	project_name
p2	Gemini

მაგალითი 3-ში ფუნქცია აბრუნებს TABLE მონაცემთა ტიპის ცვლადს.

მაგალითი 3

```
USE sample;
GO
CREATE FUNCTION employees_in_project (@pr_number CHAR(4))
RETURNS TABLE
AS RETURN (SELECT emp_fname, emp_lname
FROM works_on, employee
WHERE employee.emp_no = works_on.emp_no
AND project_no = @pr_number)
```

მაგალითი 4 მიგვითითებს **employees_in_project** ფუნქციის გამოყენებაზე.

მაგალითი 4

```
USE sample;
SELECT *
FROM employees_in_project('p3')
```

შედეგად მივიღებთ:

emp_fname	emp_lname
Ann	Jones
Elsa	Bertoni
Elke	Hansel

1.8. პარალელიზმის მართვა

ტრანზაქციები. ბლოკირებები.

ტრანზაქციების გამოყენების შემთხვევაში მკაფიოდ მიეთითება ტრანზაქციის დაწყება და დამთავრება, რისთვისაც შესაბამისად გამოიყენება ბრძანებები: BEGIN TRANSACTION, COMMIT და ROLLBACK.

BEGIN TRANSACTION ბრძანება განსაზღვრავს ტრანზაქციის დასაწყისს. ამ დროს ტრანზაქციების ჟურნალში ფიქსირდება შესაცვლელი მონაცემების საწყისი მნიშვნელობები და ტრანზაქციის დაწყების მომენტი. მისი სინტაქსია:

```
BEGIN TRAN[SACTION]
[ ტრანზაქციის_სახელი | @tran_name_variable [ WITH MARK [ 'აღწერა' ]
] ]
```

განვიხილოთ არგუმენტების დანიშნულება.

– ტრანზაქციის_სახელი ჩვეულებრივ, გამოიყენება მხოლოდ ჩადგმულ ტრანზაქციებთან სამუშაოდ ყველაზე დაბალი დონის ტრანზაქციის სახელდებისთვის. სახელის სიგრძე არ უნდა აღემატებოდეს 32 სიმბოლოს და უნდა აკმაყოფილებდეს ობიექტის სახელდების წესებს.

– @tran_name_variable ლოკალური ცვლადია, რომელიც ტრანზაქციის სახელს შეიცავს. მისი ტიპი უნდა იყოს char, varchar, nchar ან nvarchar. ჩადგმული ტრანზაქციების სახელდებისას უნდა გავითვალისწინოთ ის, რომ სერვერზე რეგისტრირდება მხოლოდ პირველი (ზედა დონის) ტრანზაქციის სახელი. დანარჩენი ტრანზაქციების სახელები იგნორირდება. ჩვეულებრივ შემთხვევაში ტრანზაქციის სახელდება საჭირო არ არის.

– WITH MARK ['აღწერა'] არგუმენტი საშუალებას გვაძლევს სპეციალური გზით მოვახდინოთ ტრანზაქციების მარკირება ტრანზაქციების ჟურნალში. ასეთი მარკირება საჭიროა სარეზერვო ასლიდან ტრანზაქციების ჟურნალის აღდგენისათვის, მონაცემთა ბაზის დასაბრუნებლად იმ მდგომარეობაში, რომელშიც ის იყო ტრანზაქციის დაწყებამდე ან მისი დამთავრების შემდეგ. ამისათვის, გამოიყენება RESTORE LOG ბრძანება შესაბამისად STOPBEFOREMARK და STOPATMARK საკვანძო სიტყვებით.

COMMIT TRANSACTION ან COMMIT WORK ბრძანებები განსაზღვრავენ ტრანზაქციის დასასრულს. თუ ტრანზაქციის შესრულების დროს ადგილი არ ჰქონდა შეცდომებს, მაშინ შესრულებული ცვლილებები დაფიქსირდება (roll forward). ამის შემდეგ, ტრანზაქციების ჟურნალში მოინიშნება, რომ ცვლილებები დაფიქსირებულია და ტრანზაქცია დამთავრებული.

COMMIT ბრძანებების სინტაქსია:

COMMIT [WORK] |

COMMIT [TRAN [SACTION] [ტრანზაქციის_სახელი | @tran_name_variable]]

ROLLBACK TRANSACTION ან ROLLBACK WORK ბრძანებების შესრულება იწვევს ტრანზაქციის შეწყვეტას და უკუქცევას (roll back). უკუქცევის დროს, შესრულებული ცვლილებები უქმდება და აღდგება სისტემის პირვანდელი მდგომარეობა. ტრანზაქციების ჟურნალში მოინიშნება, რომ ტრანზაქცია იყო გაუქმებული. ROLLBACK ბრძანებების სინტაქსია:

ROLLBACK [WORK] |

ROLLBACK [TRAN [SACTION]

[ტრანზაქციის_სახელი | @tran_name_variable

| შენახვის_წერტილის_სახელი | @savepoint_variable]]

განვიხილოთ არგუმენტების დანიშნულება.

– შენახვის_წერტილის_სახელი არგუმენტი არის ტრანზაქციის საკონტროლო წერტილის სახელი, რომელიც შეიქმნა SAVE TRANSACTION ბრძანებით. გარკვეულ წერტილამდე ტრანზაქციის უკუქცევა საშუალებას გვაძლევს გავაუქმოთ მხოლოდ ტრანზაქციის ნაწილი, ანუ საკონტროლო წერტილის შექმნის შემდეგ გაკეთებული ცვლილებები.

– @savepoint_variable არგუმენტი არის ცვლადი, რომელიც შეიცავს ტრანზაქციის საკონტროლო წერტილის სახელს. მისი ტიპი უნდა იყოს char, varchar, nchar და nvarchar.

განაწილებული ტრანზაქციები

განაწილებული ტრანზაქცია (distributed transaction) წარმოადგენს რამდენიმე ცალკეულ ტრანზაქციას, რომელიც ლოკალურად სრულდება ცალკეულ მონაცემთა ბაზაში. განაწილებული ტრანზაქციების ფიქსირება და უკუქცევა იმართება ტრანზაქციების მენეჯერის (transaction manager) მიერ. ის ახდენს ლოკალური რესურსების მენეჯერების მუშაობის კოორდინირებას და იძლევა იმის გარანტიას, რომ ყველა ლოკალური ტრანზაქცია დაფიქსირებული ან უკუქცეული იქნება. არ შეიძლება ტრანზაქციის ერთი ნაწილის ფიქსირება და მეორე ნაწილის უკუქცევა. განაწილებული ტრანზაქციების მუშაობის ყველაზე რთული ეტაპი დაკავშირებულია მათ დამთავრებასთან.

არსებობს განაწილებული ტრანზაქციის დაწყების რამდენიმე გზა:

– თუ პროგრამა-დანართი ლოკალურ ტრანზაქციაში იყენებს განაწილებულ მოთხოვნას მონაცემებთან მიმართვისათვის, მაშინ სერვერი ავტომატურად იწყებს განაწილებული ტრანზაქციის შესრულებას.

– თუ პროგრამა-დანართი იწყებს ლოკალურ ტრანზაქციას და მისგან იძახებს დაშორებულ შენახულ პროცედურას დაყენებული REMOTE_PROC_TRANSACTION პარამეტრის შემთხვევაში, მაშინ ეს ტრანზაქცია ავტომატურად ფართოვდება განაწილებულ ტრანზაქციამდე.

– პროგრამა-დანართს განაწილებული ტრანზაქციის დაწყება შეუძლია OLE DB-ის მეთოდების ან ODBC-ის ფუნქციების გამოყენებით.

– სერვერი იწყებს განაწილებული ტრანზაქციის შესრულებას BEGIN DISTRIBUTED TRANSACTION ბრძანების საშუალებით. მისი სინტაქსია:

```
BEGIN DISTRIBUTED TRAN [ SACTION ]
[ ტრანზაქციის_სახელი | @tran_name_variable ]
```

განაწილებული ტრანზაქციის მუშაობის დროს არ შეიძლება WITH MARK პარამეტრის გამოყენება ტრანზაქციის მარკირებისათვის.

ჩვენ შეგვიძლია ნება დავრთოთ ან ავკრძალოთ ლოკალური ტრანზაქციის ავტომატურად გაფართოება განაწილებულ ტრანზაქციამდე, ამისათვის ტრანზაქციის ტანში, დაშორებული შენახვადი პროცედურის გამოძახების დროს, უნდა შევასრულოთ სერვერის კონფიგურირება:

```
EXEC sp_configure 'remote proc trans' , { 0 | 1 }
```

კონფიგურირების ამ პარამეტრის მოქმედება ძალაში შედის დაუყოვნებლივ და არ ითხოვს სერვერის ხელახალ გაშვებას. ეს შენახვადი პროცედურა მოქმედებს სერვერის დონეზე და აყენებს ავტომატურ მნიშვნელობას ყველა გახსნილი

შეერთებისათვის. ტრანზაქციის გაფართოების მართვა შეიძლება, აგრეთვე ცალკეული შეერთების დონეზე. ამისათვის, გამოიყენება შემდეგი ბრძანება:

SET REMOTE_PROC_TRANS { ON | OFF }

ამ ბრძანების მოქმედება კონკრეტული შეერთების ფარგლებში ფარავს sp_configure შენახვადი პროცედურის მიერ დაყენებულ მნიშვნელობას.

მაგალითი 1

```
USE sample;
BEGIN TRANSACTION /* The beginning of the transaction */
UPDATE employee
SET emp_no = 39831
WHERE emp_no = 10102
IF (@@error <> 0)
ROLLBACK /* Rollback of the transaction */
UPDATE works_on
SET emp_no = 39831
WHERE emp_no = 10102
IF (@@error <> 0)
ROLLBACK
COMMIT /*The end of the transaction */
```

მაგალითი 2

```
BEGIN TRANSACTION;
INSERT INTO department (dept_no, dept_name)
VALUES ('d4', 'Sales');
SAVE TRANSACTION a;
INSERT INTO department (dept_no, dept_name)
VALUES ('d5', 'Research');
SAVE TRANSACTION b;
INSERT INTO department (dept_no, dept_name)
VALUES ('d6', 'Management');
ROLLBACK TRANSACTION b;
INSERT INTO department (dept_no, dept_name)
VALUES ('d7', 'Support');
ROLLBACK TRANSACTION a;
COMMIT TRANSACTION;
```

ბლოკირებები

ბლოკირების მონიტორინგისათვის გამოიყენება Transact-SQL-ის sp_lock შენახვადი პროცედურა, რომლის საშუალებით შეგვიძლია მივიღოთ ინფორმაცია

კონკრეტული პროცესის მიერ დაყენებული ბლოკირების შესახებ. ამ პროცედურის სინტაქსია:

sp_lock [[@spid1 =] 'პროცესის_იდენტიფიკატორი_1'
[, [@spid2 =] 'პროცესის_იდენტიფიკატორი_2']

'პროცესის_იდენტიფიკატორი_1' არგუმენტი შეიცავს პროცესის საიდენტიფიკაციო ნომერს. 'პროცესის_იდენტიფიკატორი_2' არგუმენტი დამატებითია და გამოიყენება ერთდროულად ორი პროცესის შესახებ ინფორმაციის მისაღებად. თუ არგუმენტები არ არის მითითებული, მაშინ გაიცემა ინფორმაცია ყველა პროცესის მიერ დაყენებული ბლოკირების შესახებ.

პროცესის შესაწყვეტად გამოიყენება KILL ბრძანება. მისი სინტაქსია:

KILL პროცესის_იდენტიფიკატორი

კონკრეტულ მონაცემთა ბაზაში გახსნილი ტრანზაქციების სანახავად გამოიყენება DBCC OPENTRAN ბრძანება. მისი სინტაქსია:

DBCC OPENTRAN ({ 'მონაცემთა_ბაზის_სახელი' | მონაცემთა_ბაზის_იდენტიფიკატორი })
[WITH TABLERESULTS [, NO_INFOMSGS]]

მაგალითი 3

```
SELECT resource_type, DB_NAME(resource_database_id) as db_name,
request_session_id, request_mode, request_status
FROM sys.dm_tran_locks
WHERE request_status = 'WAIT;'
```

„მკვდარი“ ბლოკირება

მაგალითი 4

```
BEGIN TRANSACTION
UPDATE employee
SET dept_no = 'd2'
WHERE emp_no = 9031
WAITFOR DELAY '00:00:10'
DELETE FROM works_on
WHERE emp_no = 18316
AND project_no = 'p2'
COMMIT

BEGIN TRANSACTION
UPDATE works_on
SET job = 'Manager'
WHERE emp_no = 18316
AND project_no = 'p2'
WAITFOR DELAY '00:00:10'
```

```
UPDATE employee
SET emp_lname = 'Green'
WHERE emp_no = 9031
COMMIT
```

1.9.ტრიგერებთან მუშაობა TRANSACT SQL-ში

ტრიგერი (trigger) არის სპეციალური ტიპის შენახვადი პროცედურა, რომელიც სრულდება კონკრეტულ ცხრილზე კონკრეტული მოქმედებების განხორციელებისათვის. ტრიგერი სამი ძირითადი ნაწილისაგან შედგება: სახელი, ქმედება და შესრულება.

ტრიგერის სახელის მაქსიმალური ზომა 128 სიმბოლოთი განისაზღვრება. ტრიგერის ქმედება შეიძლება იყოს DML (INSERT, UPDATE ან DELETE) ან DDL ნებისმიერი ბრძანება. შესაბამისად არსებობს ტრიგერების ორი ფორმა: DML ტრიგერები და DDL ტრიგერები.

DML-ტრიგერები განსხვავდებიან იმ ბრძანებების ტიპის მიხედვით, რომლებზეც ისინი რეაგირებენ. არსებობს ტრიგერების სამი ტიპი:

- INSERT TRIGGER. ამ ტიპის ტრიგერები გაიშვება INSERT ბრძანებით მონაცემების ჩასმის მცდელობის დროს.

მონაცემების შეცვლის მცდელობის დროს.

- DELETE TRIGGER. ამ ტიპის ტრიგერები გაიშვება DELETE ბრძანებით მონაცემების წაშლის მცდელობის დროს.

DML ტრიგერის შექმნა

ტრიგერის შექმნისათვის CREATE TRIGGER ბრძანება გამოიყენება:

```
CREATE TRIGGER [სქემის_სახელი.]ტრიგერის_სახელი
ON {ცხრილის_სახელი|წარმოდგენის_სახელი}
[WITH dml_ტრიგერის_ოპცია[,…]]
{FOR | AFTER | INSTEAD OF} { [INSERT] [,] [UPDATE] [,] [DELETE]}
[WITH APPEND]
{AS sql_ბრძანება EXTERNAL NAME მეთოდის_სახელი}
```

ტრიგერის_სახელი უნდა იყოს უნიკალური მონაცემთა ბაზის ფარგლებში.

ცხრილის_სახელი | წარმოდგენის_სახელი. იმ ცხრილის ან წარმოდგენის სახელია, რომელსაც ტრიგერი უკავშირდება.

AFTER. ამ არგუმენტის მითითებით ტრიგერი მხოლოდ მისი გამომძახებელი ბრძანებების წარმატებით შესრულების შემთხვევაში გაიშვება.

INSTEAD OF. ამ არგუმენტის მითითებით ტრიგერი შესრულდება მისი გამომძახებელი მოთხოვნის ნაცვლად.

[DELETE] [,] [INSERT] [,] [UPDATE]. ეს კონსტრუქცია განსაზღვრავს ბრძანებას რომელზეც ტრიგერი მოახდენს რეაგირებას. შეიძლება ორი ან სამი ბრძანების მითითებაც.

FOR [INSERT] [,] [UPDATE]. განსაზღვრავს ბრძანებას, რომლის შესრულებითაც ტრიგერი გაიშვება. დასაშვებია ტრიგერის დაკავშირება რამდენიმე ბრძანებასთან.

IF UPDATE (სვეტის_სახელი). მისი მითითებით ტრიგერი სრულდება ცხრილის კონკრეტული სვეტის მოდიფიცირების დროს.

{ AND | OR } UPDATE (სვეტის_სახელი). ამ კონსტრუქციის გამოყენებით ხდება ტრიგერის გაშვება რამდენიმე სვეტის მოდიფიცირების შემთხვევაში.

IF (COLUMNS_UPDATED()). ეს კონსტრუქცია გვამცნობს თუ რომელი სვეტები შეიცვალა ან დაემატა.

AS sql_ბრძანება [,...n]. ეს არგუმენტი შეიცავს Transact_SQL-ის ბრძანებებს, რომლებიც შესრულდება ტრიგერის გაშვებისას.

ტრიგერის სტრუქტურის შეცვლა

ტრიგერის სტრუქტურის შესაცვლელად გამოიყენება ALTER TRIGGER ბრძანება, რომლის შესრულების დროს ჯერ იშლება არსებული ტრიგერი, შემდეგ კი - იქმნება ახალი. მისი სინტაქსია:

ALTER TRIGGER [სქემის_სახელი .] ტრიგერის_სახელი
ON (ცხრილის_სახელი | წარმოდგენის_სახელი)
[WITH ENCRYPTION]
(FOR | AFTER | INSTEAD OF)
{ [DELETE] [,] [INSERT] [,] [UPDATE] }
AS { sql_ბრძანება [;] [...n] }

ტრიგერის წაშლა

ტრიგერის წასაშლელად გამოიყენება DROP TRIGGER ბრძანება. მისი სინტაქსია:

DROP TRIGGER [სქემის_სახელი .] ტრიგერის_სახელი [,...n]

ანუ შესაძლებელია რამდენიმე ტრიგერის ერთდროულად წაშლა.

ტრიგერების მართვა

ტრიგერის სახელის შეცვლა

ტრიგერის სახელის შესაცვლელად გამოიყენება sp_rename სისტემური შენახვადი პროცედურა.

ტრიგერის შესახებ ინფორმაციის მიღება

ტრიგერის გამოძახების დროს შესრულებული კოდის მისაღებად უნდა შევასრულოთ sp_helptext სისტემური შენახვადი პროცედურა. მისი სინტაქსია:

sp_helptext [@objname =] 'ტრიგერის_სახელი'

'ტრიგერის_სახელი' არის იმ ტრიგერის სახელი, რომლის შესახებ გვინდა ინფორმაციის მიღება.

კონკრეტული ცხრილისთვის განსაზღვრული ტრიგერების სიის მისაღებად გამოიყენება sp_helptrigger სისტემური შენახვადი პროცედურა. მისი სინტაქსია:

sp_helptrigger [@tabname =] 'ცხრილის_სახელი'
[, [@triggertype =] 'ტრიგერის_ტიპი']

'ცხრილის_სახელი' არის იმ ცხრილის სახელი, რომლის ტრიგერების სიაც გვაინტერესებს. 'ტრიგერის_ტიპი' არის იმ ტრიგერების ტიპი, რომელთა შესახებ გვაინტერესებს ინფორმაცია. ის იღებს მნიშვნელობებს: 'INSERT', 'UPDATE' და 'DELETE'.

მაგალითი 5

```
USE sample;
GO
CREATE TABLE audit_budget
(project_no CHAR(4) NULL,
user_name CHAR(16) NULL,
date DATETIME NULL,
budget_old FLOAT NULL,
budget_new FLOAT NULL);
GO
CREATE TRIGGER modify_budget
ON project AFTER UPDATE
AS IF UPDATE(budget)
BEGIN
DECLARE @budget_old FLOAT
DECLARE @budget_new FLOAT
DECLARE @project_number CHAR(4)
SELECT @budget_old = (SELECT budget FROM deleted)
SELECT @budget_new = (SELECT budget FROM inserted)
SELECT @project_number = (SELECT project_no FROM deleted)
INSERT INTO audit_budget VALUES
(@project_number,USER_NAME(),GETDATE(),@budget_old, @budget_new)
END
```

მაგალითი 6

```
USE sample;
```

```
GO
CREATE TRIGGER total_budget
ON project AFTER UPDATE
AS IF UPDATE (budget)
BEGIN
DECLARE @sum_old1 FLOAT
DECLARE @sum_old2 FLOAT
DECLARE @sum_new FLOAT
SELECT @sum_new = (SELECT SUM(budget) FROM inserted)
SELECT @sum_old1 = (SELECT SUM(p.budget)
FROM project p WHERE p.project_no
NOT IN (SELECT d.project_no FROM deleted d))
SELECT @sum_old2 = (SELECT SUM(budget) FROM deleted)
IF @sum_new > (@sum_old1 + @sum_old2) *1.5
BEGIN
PRINT 'No modification of budgets'
ROLLBACK TRANSACTION
END
ELSE
PRINT 'The modification of budgets executed'
END
```

მაგალითი 7

```
USE sample;
GO
CREATE TRIGGER workson_integrity
ON works_on AFTER INSERT, UPDATE
AS IF UPDATE(emp_no)
BEGIN
IF (SELECT employee.emp_no
FROM employee, inserted
WHERE employee.emp_no = inserted.emp_no) IS NULL
BEGIN
ROLLBACK TRANSACTION
PRINT 'No insertion/modification of the row'
END
ELSE PRINT 'The row inserted/modified'
END
```

მაგალითი 8

```
USE sample;
GO
CREATE TRIGGER refint_workson2
ON employee AFTER DELETE, UPDATE
AS IF UPDATE (emp_no)
BEGIN
IF (SELECT COUNT(*)
FROM WORKS_ON, deleted
WHERE works_on.emp_no = deleted.emp_no) > 0
BEGIN
ROLLBACK TRANSACTION
PRINT 'No modification/deletion of the row'
END
ELSE PRINT 'The row is deleted/modified'
END
```

მაგალითი 9

```
CREATE VIEW all_orders
AS SELECT orderid, price, quantity, orderdate, total, shippeddate
FROM orders;
GO
CREATE TRIGGER tr_orders
ON all_orders INSTEAD OF INSERT
AS BEGIN
INSERT INTO orders
SELECT orderid, price, quantity, orderdate
FROM inserted
END
```

მაგალითი 10

```
USE sample;
GO
CREATE TRIGGER prevent_drop_triggers
ON DATABASE FOR DROP_TRIGGER
AS PRINT 'You must disable "prevent_drop_triggers" to drop any trigger'
ROLLBACK
```

მაგალითი 11

```
USE master;
GO
```

```
CREATE LOGIN login_test WITH PASSWORD = 'login_test§$',
CHECK_EXPIRATION = ON;
GO
GRANT VIEW SERVER STATE TO login_test;
GO
CREATE TRIGGER connection_limit_trigger
ON ALL SERVER WITH EXECUTE AS 'login_test'
FOR LOGON AS
BEGIN
IF ORIGINAL_LOGIN()= 'login_test' AND
(SELECT COUNT(*) FROM sys.dm_exec_sessions
WHERE is_user_process = 1 AND
original_login_name = 'login_test') > 1
ROLLBACK;
END;
```

1.20. კურსორები

კურსორების მართვა

– კურსორთან მუშაობის დროს შეგვიძლია ხუთი ძირითადი ოპერაცია შევასრულოთ:

– კურსორის შექმნა. კურსორი უნდა შევქმნათ მის გამოყენებამდე.

– კურსორის გახსნა. შექმნილი კურსორი მონაცემებს არ შეიცავს. გახსნის ოპერაცია კურსორს მონაცემებით ავსებს.

– სტრიქონების ამორჩევა კურსორიდან და მათი შეცვლა კურსორის საშუალებით. სტრიქონებით კურსორის შევსების შემდეგ შეგვიძლია მათთან მუშაობა. კურსორის ტიპზე დამოკიდებულებით შეგვეძლება სტრიქონების ან მხოლოდ წაკითხვა ან წაკითხვა და შეცვლა.

– კურსორის დახურვა. კურსორთან მუშაობის დამთავრების შემდეგ ის უნდა დავხუროთ. ამ დროს სერვერი ათავისუფლებს სივრცეს tempdb მონაცემთა ბაზაში, რომელიც კურსორს გამოეყო შექმნის დროს.

– კურსორის გათავისუფლება. ამ დროს კურსორი იშლება.

კურსორის შექმნა

კურსორის შესაქმნელად გამოიყენება DECLARE CURSOR ბრძანება. მისი სინტაქსია:

DECLARE კურსორის_სახელი [INSENSITIVE] [SCROLL] CURSOR FOR select_ბრძანება [FOR { READ ONLY | UPDATE [OF სვეტის_სახელი [,...n]] }]

განვიხილოთ არგუმენტების დანიშნულება.

– INSENSITIVE მიუთითებს, რომ უნდა შეიქმნას სტატიკური კურსორი. თუ ის მითითებული არ არის, მაშინ შეიქმნება დინამიური კურსორი.
– SCROLL მიუთითებს, რომ შეიქმნება გადახვევადი კურსორი. თუ ის მითითებული არ არის, მაშინ შეიქმნება მიმდევრობითი კურსორი.
– FOR select_ბრძანება. ეს არგუმენტი შეიცავს SELECT მოთხოვნას, რომელიც გასცემს კურსორის სტრიქონების შედეგობრივ ნაკრებს. SELECT მოთხოვნა არ უნდა შეიცავდეს INTO, FOR BROWSE და COMPUTE BY განყოფილებებს. თუ მოთხოვნა კონფლიქტშია კურსორის ტიპთან, მაშინ სერვერი შეასრულებს კურსორის არაცხად გარდაქმნას თავსებად ტიპში.
– FOR READ ONLY. თუ ის მითითებულია, მაშინ შეიქმნება „მხოლოდ წაკითხვადი" კურსორი. „მხოლოდ წაკითხვადი" კურსორი განსხვავდება სტატიკური კურსორისაგან. „მხოლოდ წაკითხვადი" შეიძლება იყოს დინამიური კურსორიც. ამ შემთხვევაში შესაძლებელი იქნება სხვა მომხმარებლების მიერ შესრულებული ცვლილებების ასახვა.
– FOR UPDATE [OF სვეტის_სახელი [,...n]]. თუ ის მითითებულია, მაშინ კურსორში შესაძლებელი იქნება სტრიქონების ცვლილება. თუ მითითებული არ არის OF სვეტის_სახელი არგუმენტი, მაშინ შესაძლებელია კურსორის ყველა სვეტის შეცვლა. წინააღმდეგ შემთხვევაში, დასაშვები იქნება მხოლოდ სვეტის_სახელი სიაში მითითებული სვეტების შეცვლა.

DECLARE CURSOR ბრძანების Transact-SQL-ით განსაზღვრული სინტაქსია:

DECLARE კურსორის_სახელი CURSOR [LOCAL | GLOBAL] [FORWARD_ONLY | SCROLL] [STATIC | KEYSET | DYNAMIC | FAST_FORWARD] [READ_ONLY | SCROLL_LOCKS | OPTIMISTIC] [TYPE_WARNING] FOR select_ბრძანება [FOR UPDATE [OF სვეტის_სახელი [,...n]]]

განვიხილოთ არგუმენტების დანიშნულება.

LOCAL მიუთითებს, რომ შეიქმნება ლოკალური კურსორი, რომელიც ხილული იქნება მხოლოდ ამ კურსორის შემქმნელი პაკეტის, ტრიგერის, შენახვადი პროცედურის ან მომხმარებლის მიერ შექმნილი ფუნქციის შიგნით. პაკეტის, ტრიგერის ან შენახვადი პროცედურის დამთავრებისთანავე კურსორი წაიშლება. კურსორის შემცველობის გადასაცემად მისი შემქმნელი კონსტრუქციის გარეთ, საჭიროა ის მივანიჭოთ OUTPUT პარამეტრს.

GLOBAL მიუთითებს, რომ შეიქმნება გლობალური კურსორი, რომელიც იარსებებს მიმდინარე შეერთების დახურვამდე.

FORWARD_ONLY მიუთითებს, რომ შეიქმნება მიმდევრობითი კურსორი.

SCROLL მიუთითებს, რომ შეიქმნება გადახვევადი კურსორი.

STATIC მიუთითებს, რომ შეიქმნება სტატიკური კურსორი.

KEYSET მიუთითებს, რომ შეიქმნება საგასაღებო კურსორი.

DYNAMIC მიუთითებს, რომ შეიქმნება დინამიური კურსორი.

FAST_FORWARD. თუ ის მითითებულია READ_ONLY არგუმენტთან ერთად, მაშინ მოხდება შექმნილი კურსორის ოპტიმიზება მონაცემებთან სწრაფი მიმართვის მიზნით. ეს არგუმენტი არ შეიძლება გამოყენებული იყოს FORWARD_ONLY და OPTIMISTIC არგუმენტებთან ერთად.

OPTIMISTIC მიუთითებს, რომ კურსორში იკრძალება იმ სტრიქონების შეცვლა ან წაშლა, რომლებიც ცხრილში შეიცვალა კურსორის გახსნის შემდეგ.

TYPE_WARNING. თუ ის მითითებულია, მაშინ სერვერი მომხმარებელს შეატყობინებს კურსორის ტიპის შეცვლის შესახებ, თუ ის არათავსებადია SELECT მოთხოვნის ტიპთან.

კურსორის გახსნა

კურსორის გასახსნელად და მონაცემებით შესავსებად გამოიყენება OPEN ბრძანება. მისი სინტაქსია:

OPEN { { [GLOBAL] კურსორის_სახელი } | @cursor_variable_name }

@cursor_variable_name ცვლადი უნდა შეიცავდეს კურსორის სახელს. თუ საჭიროა გლობალური კურსორის გახსნა, მაშინ უნდა მივუთითოთ GLOBAL არგუმენტი.

მონაცემების წაკითხვა

კურსორიდან მონაცემების წასაკითხად გამოიყენება FETCH ბრძანება. მისი სინტაქსია:

```
FETCH [ [ NEXT | PRIOR | FIRST | LAST
| ABSOLUTE { n | @nvar } | RELATIVE { n | @nvar } ]
FROM
]
{ { [ GLOBAL ] კურსორის_სახელი } | @cursor_variable_name }
[ INTO @variable_name [,...n] ]
```

განვიხილოთ არგუმენტების დანიშნულება.

– FIRST. მისი მითითების შემთხვევაში გაიცემა კურსორის შედეგობრივი ნაკრების პირველი სტრიქონი, რომელიც მიმდინარე გახდება.

– LAST. მისი მითითების შემთხვევაში გაიცემა კურსორის შედეგობრივი ნაკრების უკანასკნელი სტრიქონი, რომელიც მიმდინარე გახდება.

– NEXT. მისი მითითების შემთხვევაში გაიცემა კურსორის შედეგობრივი ნაკრების მიმდინარე სტრიქონის შემდეგ მოთავსებული სტრიქონი, რომელიც მიმდინარე გახდება.

– PRIOR. მისი მითითების შემთხვევაში გაიცემა კურსორის შედეგობრივი ნაკრების მიმდინარე სტრიქონის წინ მოთავსებული სტრიქონი, რომელიც მიმდინარე გახდება.

– ABSOLUTE { n | @nvar }. გასცემს სტრიქონს კურსორის მთლიან შედეგობრივ ნაკრებში მისი აბსოლუტური რიგითი ნომრის მიხედვით. სტრიქონის ნომერი

შეიძლება მივუთითოთ მუდმივას ან ცვლადის სახით. ცვლადს მთელი ტიპი უნდა ჰქონდეს.

– RELATIVE { n | @nvar }. გასცემს სტრიქონს, რომელიც მდებარეობს მიმდინარე სტრიქონიდან n სტრიქონის შემდეგ, თუ n დადებითია და n სტრიქონით წინ, თუ n უარყოფითია. გაცემული სტრიქონი ხდება მიმდინარე. თუ მითითებულია ნულოვანი მნიშვნელობა, მაშინ მიმდინარე სტრიქონი გაიცემა.

– INTO @variable_name [,...n]. ეს არგუმენტი მიუთითებსს იმ ცვლადების სიას, რომლებიც შეიცავენ დასაბრუნებელი სტრიქონის სვეტების შესაბამის მნიშვნელობებს. ცვლადების მითითების მიმდევრობა უნდა შეესაბამებოდეს კურსორში სვეტების მიმდევრობას. ამასთან, ცვლადის ტიპი უნდა ემთხვეოდეს სვეტის ტიპს. თუ ეს არგუმენტი მითითებული არ არის, მაშინ მონაცემები ეკრანზე გამოიცემა.

მონაცემების შეცვლა

კურსორის საშუალებით მონაცემების შესაცვლელად UPDATE ბრძანება გამოიყენება. მისი სინტაქსია:

UPDATE ცხრილის_სახელი SET { სვეტის_სახელი = { DEFAULT | NULL |
გამოსახულება } } [,..n]
WHERE CURRENT OF კურსორის_სახელი

განვიხილოთ არგუმეტების დანიშნულება.

– სვეტის_სახელი იმ სვეტის სახელია, რომელიც უნდა შეიცვალოს. ერთი ოპერაციის შედეგად შეიძლება რამდენიმე სვეტის შეცვლა, მაგრამ ყველა შესაცვლელი სვეტი უნდა ეკუთვნოდეს ერთ ცხრილს. სვეტს მიენიჭება ავტომატური მნიშვნელობა, თუ მითითებულია DEFAULT საკვანძო სიტყვა. სვეტს მიენიჭება NULL მნიშვნელობა, თუ მითითებულია NULL საკვანძო სიტყვა. გამოსახულება შეიცავს სვეტისთვის მისანიჭებელ მნიშვნელობას.

– კურსორის_სახელი იმ კურსორის სახელია, რომელშიც უნდა შესრულდეს ცვლილებები.

კურსორის დახურვა

კურსორის დახურვა ათავისუფლებს მისთვის გამოყოფილ რესურსებს და შლის კურსორში მოთავსებულ სტრიქონებს. დახურვის დროს მოიხსნება კურსორის მუშაობის დროს დაყენებული ბლოკირებები. კურსორი, რომელიც დაიხურა, მაგრამ არ გათავისუფლდა, შეიძლება განმეორებით გაიხსნას. კურსორის დასახურად გამოიყენება CLOSE ბრძანება, რომლის სინტაქსია:

CLOSE { { [GLOBAL] კურსორის_სახელი } | @cursor_variable_name }

დამატებითი საშუალებები

კურსორების უფრო მოქნილი მართვისთვის უნდა გამოვიყენოთ სისტემური შენახვადი პროცედურები და Transzct-SQL-ის ფუნქციები.

sp_cursor_list შენახვადი პროცედურა გასცემს მიმდინარე შეერთებაში გახსნილი კურსორებისა და მათი ატრიბუტების სიას. მისი სინტაქსია:

```
sp_cursor_list [@cursor_return = ] კურსორის_სახელი OUTPUT,
[@cursor_scope = ] კურსორის_დიაპაზონი
```

განვიხილოთ არგუმენტების დანიშნულება.

– კურსორის_სახელი არის ცვლადის სახელი, რომელსაც CURSOR ტიპი აქვს. მასში ინახება sp_cursor_list პროცედურის მიერ გაცემული შედეგი. შედეგი არის ცხრილი, რომლის თითოეული სტრიქონი ერთ კურსორს შეესაბამება.
– კურსორის_დიაპაზონი არის კურსორის ტიპების დიაპაზონი, რომლებისთვისაც გამოტანილი იქნება ინფორმაცია. ის იღებს მნიშვნელობებს 1 (ყველა ლოკალური კურსორი), 2 (ყველა გლობალური კურსორი) და 3 (ყველა ლოკალური და გლობალური კურსორი).

კურსორის შესახებ ინფორმაციის მისაღებად შეგვიძლია გამოვიყენოთ sp_describe_cursor შენახვადი პროცედურა. მისი სინტაქსია:

```
sp_describe_cursor [ @cursor_return = ] კურსორის_სახელი OUTPUT
{
[ , [ @cursor_source = ] N'local' , [ @cursor_identity = ]
N'ლოკალური_კურსორის_სახელი' ]
| [ , [ @cursor_source = ] N'global' , [ @cursor_identity = ]
N'გლობალური_კურსორის_სახელი' ]
| [ , [ @cursor_source = ] N'variable' , [ @cursor_identity = ]
N'კურსორის_ცვლადის_სახელი' ]
}
```

მოცემული შენახვადი პროცედურა გასცემს იმავე მონაცემებს, რასაც sp_cursor_list პროცედურა. განსხვავება იმაშია, რომ sp_describe_cursor შენახვადი პროცედურის საშუალებით ინფორმაცია შეგვიძლია მივიღოთ მხოლოდ ერთი კურსორის შესახებ.

კურსორის კონკრეტული სვეტის შესახებ ინფორმაციის მისაღებად უნდა გამოვიყენოთ sp_describe_cursor_columns შენახვადი პროცედურა. მისი სინტაქსია:

```
sp_describe_cursor_columns [ @cursor_return = ] კურსორის_სახელი OUTPUT
{
[ , [ @cursor_source = ] N'local' , [ @cursor_identity = ]
N'ლოკალური_კურსორის_სახელი' ] | [ , [ @cursor_source = ] N'global' , [
```

@cursor_identity =] N'გლობალური_კურსორის_სახელი'] | [, [@cursor_source =] N'variable' , [@cursor_identity =] N'კურსორის_ცვლადის_სახელი'] }

განვიხილოთ ამ შენახვადი პროცედურის არგუმენტები.

- [@cursor_return =] კურსორის_სახელი OUTPUT. განსაზღვრავს ცვლადის სახელს, რომლის ტიპია CURSOR. ეს ცვლადი შეიცავს ინფორმაციას მითითებული კურსორის სვეტების შესახებ. რადგან ეს ცვლადი სინამდვილეში კურსორია, ამიტომ ინფორმაციის მისაღებად შეგვიძლია გამოვიყენოთ კურსორთან მუშაობის სტანდარტული მეთოდები.

ცხრილების სიის მისაღებად, რომელთა ბაზაზეც აგებულია კურსორი, გამოიყენება sp_describe_cursor_tables სისტემური შენახვადი პროცედურა. მისი სინტაქსია:

sp_describe_cursor_tables [@cursor_return =] კურსორის_სახელი OUTPUT { [, [@cursor_source =] N'local' , [@cursor_identity =] N'ლოკალური_კურსორის_სახელი'] | [, [@cursor_source =] N'global' , [@cursor_identity =] N'გლობალური_კურსორის_სახელი'] | [, [@cursor_source =] N'variable' , [@cursor_identity =] N'კურსორის_ცვლადის_სახელი'] }

1.21. სარეზერვო ასლების შექმნა. სარეზერვო ასლიდან აღდგენა

მონაცემთა ბაზების სარეზერვო ასლი ყოველთვის იქმნება ერთი მონაცემთა ბაზისთვის და წარმოადგენს ერთ ფაილს, რომელიც შეგვიძლია სხვა დისკზე ან კატალოგში გადავწეროთ, აგრეთვე, გადავიტანოთ სხვა სერვერზე და იქ აღვადგინოთ.

განირჩევა სარეზერვო ასლის ოთხი ტიპი:

1. მონაცემთა ბაზის სრული სარეზერვო ასლი (Database Backup);
2. ტრანზაქციების ჟურნალის სარეზერვო ასლი (Transaction log Backup);
3. დიფერენცირებული სარეზერვო ასლი (Differential Database Backup);
4. ფაილებისა და ფაილების ჯგუფის სარეზერვო ასლი (File and Filegroup Backup).

სრული და დიფერენცირებული სარეზერვო ასლების შექმნა

მონაცემთა ბაზის სრული სარეზერვო ასლის შექმნის დროს ხდება მთელი მონაცემთა ბაზის ასლის შექმნა, ხოლო დიფერენცირებული სარეზერვო ასლი შეიცავს მონაცემთა ბაზის ცვლილებების იმ ნაწილს, რომლებიც შესრულდა მონაცემთა ბაზაში მისი ბოლო სრული სარეზერვო ასლის შექმნის შემდეგ.

სრული და დიფერენცირებული სარეზერვო ასლების შესაქმნელად გამოიყენება BACKUP DATABASE ბრძანება, რომლის სინტაქსია:

```
BACKUP DATABASE {db_name | @variable}
TO device_list
[MIRROR TO device_list2]
```

```
[WITH | option_list]
```

მაგალითად:

```
USE master;
BACKUP DATABASE sample
TO DISK = 'C:\sample.bak'
WITH INIT, COMPRESSION;
```

ასევე ტრანზაქციების ჟურნალის სარეზერვო ასლებისათვის:

```
BACKUP LOG {db_name | @variable}
TO device_list
[MIRROR TO device_list2]
    [WITH option_list]
```

BACKUP DATABASE { მონაცემთა_ბაზის_სახელი | @database_name_var } TO <ინფორმაციის_მატარებელი> [,...n] [WITH [BLOCKSIZE = { ბლოკის_ზომა | @blocksize_variable }] [[,] DESCRIPTION = { 'ტექსტი' | @text_variable }] [[,] DIFFERENTIAL] [[,] EXPIREDATE = { თარიღი | @date_var } |

სრული და დიფერენცირებული სარეზერვო ასლიდან აღდგენა

სრული და დიფერენცირებული სარეზერვო ასლიდან აღსადგენად გამოიყენება RESTORE DATABASE ბრძანება, რომლის სინტაქსია:

```
RESTORE DATABASE {db_name | @variable}
[FROM device_list]
    [WITH option_list]
```

ტრანზაქციების ჟურნალიდან აღდგენა

ტრანზაქციების ჟურნალიდან აღსადგენად გამოიყენება ბრძანება:

RESTORE LOG { მონაცემთა_ბაზის_სახელი | @database_name_var } [FROM <ინფორმაციის_მატარებელი> [,...n]] [WITH [RESTRICTED_USER] [[,] FILE = { ფაილის_ნომერი | @file_number }] [[,] MOVE 'ფაილის_ლოგიკური_სახელი' TO 'ფაილის_ფიზიკური_სახელი'] [,...n] [[,] MEDIANAME =

სისტემური მონაცემთა ბაზების სარეზერვო ასლის შექმნა

სარეზერვო ასლების რეგულარულ შექმნას უნდა ექვემდებარებოდეს სისტემური მონაცემთა ბაზებიც. master სისტემური მონაცემთა ბაზისთვის შესაძლებელია მხოლოდ სრული სარეზერვო ასლის შექმნა, როცა ხდება ახალი სააღრიცხვო ჩანაწერის შექმნა, მონაცემთა ბაზების შექმნა ან წაშლა, სისტემური გაწყობების შეცვლა და ა.შ. master მონაცემთა ბაზის აღდგენის შემდეგ ჯერ უნდა აღვადგინოთ სისტემური, შემდეგ კი - მომხმარებლების მონაცემთა ბაზები. Msdb მონაცემთა ბაზისთვის შესაძლებელია როგორც სრული, ისე დიფერენცირებული სარეზერვო ასლის შექმნა. Model ბაზა თუ არ იცვლება, მაშინ მისი სარეზერვო ასლის

შექმნა დიდ აუცილებლობას არ წარმოადგენს. Tempdb მონაცემთა ბაზის სარეზერვო ასლის შექმნას აზრი არ აქვს, რადგან სერვერის გაჩერების დროს ხდება მისი წაშლა და ხელახალი შექმნა სერვერის გაშვების დროს.

1.22. მონაცემთა იმპორტი და ექსპორტი

სერვერს აქვს პროგრამები, რომლებიც იძლევიან მონაცემების გაცვლის საშუალებას SQL სერვერსა და მონაცემების დამუშავების სხვა სისტემებს შორის, როგორიცაა Access, FoxPRo, Excel და ა.შ.

არსებობს მონაცემების გაცვლის ორი სახე - იმპორტი და ექსპორტი:
– მონაცემების იმპორტი - ესაა პროცესი, როცა ხდება მონაცემების ამოღება გარე წყაროებიდან (მაგალითად, Access), დამუშავება და SQL სერვერის მონაცემთა ბაზების ცხრილებში ჩაწერა.
– მონაცემების ექსპორტი - ესაა პროცესი, როცა ხდება მონაცემების ამოღება SQL სერვერის მონაცემთა ბაზების ცხრილებიდან, დამუშავება და მიმღების (მაგალითად, Access) ცხრილებში ჩაწერა.

მონაცემების დამუშავების ბევრ სისტემას, მაგალითად Access ან Excel, შეუძლია პირდაპირ მიმართოს მონაცემებს SQL სერვერზე, მათი წინასწარი გარდაქმნის გარეშე.

TRANSACT SQL–ში მონაცემების გადაცემის ძირითადი მეთოდებია:
– Data Transformation Services (DTS) Import and Export Wizard პროგრამა-ოსტატის გამოყენება TDS პაკეტის შექმნისათვის. ეს მეთოდი უზრუნველყოფს მონაცემების ექსპორტსა და იმპორტს მონაცემების ჰეტეროგენულ წყაროებს შორის.
– საბრძანებო სტრიქონის bcp (Bulk Copy Program) უტილიტის გამოყენება. ის საშუალებას გვაძლევს მონაცემები გავცვალოთ ტექსტურ ფაილსა და სერვერის მონაცემთა ბაზას შორის.
– BULK INSERT ბრძანების გამოყენება ტექსტური ფაილიდან ცხრილში მონაცემების ჩასასმელად.
– მონაცემთა იმპორტ-ექსპორტი SQL Server Management Studio-ის გამოყენებით.

მოცემული ლაბორატორიული სამუშაო სწორედ ამ უკანასკნელის ათვისებას ისახავს მიზნად.

მონაცემების გადატანა ტექსტურ ფაილში გამოიყენება იმ შემთხვევაში, როცა შეუძლებელია პირდაპირი კავშირის დამყარება SQL სერვერსა და კლიენტის პროგრამას შორის. ამ მიზნით შეგვიძლია bcp უტილიტა გამოვიყენოთ. ამ დროს, სერვერიდან მონაცემები შეგვიძლია ჩავწეროთ ტექსტურ ფაილში, საიდანაც შემდეგ წაკითხული იქნება კლიენტის პროგრამის მიერ, და პირიქით. მონაცემების გადატანა სამაგიდო პროგრამა-დანართში.

1.23. მონაცემთა ბაზების უსაფრთხოების სისტემა

მონაცემთა ბაზების უსაფრთხოების სისტემა ემყარება შემდეგ ძირითად ცნებებს:
– აუტენტიფიკაციის (authentication) გამოყენების ერთ-ერთ მთავარ არსს წარმოადგენს არასანქცირებული მომხმარებლის სისტემაში შეღწევის დროული გამოვლინება და თავიდან აცილება. ეს პროცესი შეიძლება მნიშვნელოვნად გაძლიერდეს დაშიფვრის გამოყენებით.
– ავტორიზაცია (authorization) გამოიყენება მომხმარებლის (აუტენტიფიცკაციის შემოწმების შემდეგ) იდენტიფიკაციის მიზნით, როდესაც სისტემა (სტრუქტურული და სისტემური კატალოგი) განსაზღვრავს რომელ რესურსებთან წვდომის ნებართვები ენიჭება კონკრეტულ მომხმარებელს.
– დაშიფვრა (encryption) გამოიყენება მონაცემების არაწაკითხვად ფორმამდე შესაცვლელად, რათა არ მოხდეს კონფიდენციალური ინფორმაციის წაკითხვა.
– Tracking changes ნიშნავს, რომ არასანქცირებული მომხმარებლის ქმედებები იქნება სათანადოდ რეაგირებული და შესაბამისად დოკუმენტულად აღრიცხული. ეს პროცესი განსაკუთრებით სასარგებლოა სისტემის დასაცავად მაღალი პრივილეგიების მქონე მომხმარებლებისგანაც.

აუტენტიფიციკაცია

მომხმარებლის აუტენტიფიცირება/იდენტიფიცირება ხდება სააღრიცხვო ჩანაწერის სახელისა და პაროლის მიხედვით. მომხმარებლების აუტენტიფიცირების ორი რეჟიმი არსებობს:

– აუტენტიფიცირების რეჟიმი Windows NT-ის საშუალებებით (Windows NT Authentication);
– აუტენტიფიცირების შერეული რეჟიმი (Windows NT Authentication and SQL Server Authentication).

შერეული რეჟიმი საშუალებას იძლევა დავრეგისტრირდეთ როგორც სერვერის, ისე Windows NT-ის საშუალებებით.

უსაფრთხოების სისტემის აწყობა DDL–ის გამოყენებით

განირჩევა სამი ძირითადი ბრძანება CREATE LOGIN, ALTER LOGIN და DROP LOGIN, რომელთაც შეესაბამება სისტემური პროცედურები **sp_addlogin** და **sp_droplogin**. ბრძანება CREATE LOGIN ქმნის ახალ SQL სერვერის სააღრიცხვო ჩანაწერს (login). მისი სინტაქსია:

```
CREATE LOGIN login_name
{ WITH option_list1 |
FROM {WINDOWS [ WITH option_list2 [,...] ]
| CERTIFICATE certname | ASYMMETRIC KEY key_name }}
```

მაგალითი 1

```
USE sample;
CREATE LOGIN mary WITH PASSWORD = 'you1know4it9';
```

მაგალითი 2

```
USE sample;
CREATE LOGIN [NTB11901\pete] FROM WINDOWS;
```

ბრძანება ALTER LOGIN ცვლის კერძო სააღრიცხვო ჩანაწერის თვისებებს, ხოლო ბრძანება DROP LOGIN შლის არსებულ სააღრიცხვო ჩანაწერს.

სქემები

სქემა არის მონაცემთა ბაზაში შემავალი ობიექტების კოლექცია, რომელიც ერთ პიროვნების საკუთრებას წარმოადგენს და სახელების ერთიან სივრცეს ქმნის. სქემა შეიძლება შეიცავდეს ცხრილებს, წარმოდგენებს, ფუნქციებს, პროცედურებს, ტრიგერებს, ინდექსებსა და მომხმარებლებს.

მონაცემთა ბაზის შექმნის დროს იქმნება სტანდარტული სქემები. ერთ-ერთი მათგანია dbo. ნაგულისხმევი სქემა შეიძლება განისაზღვროს და შეიცვალოს CREATE USER და ALTER USER ბრძანებების DEFAULT_SCHEMA რეჟიმის გამოყენებით. თუ DEFAULT_SCHEMA განუსაზღვრელია, მაშინ ნაგულისხმევი სქემა იქნება dbo.

სქემებთან დაკავშირებული DDL ბრძანებები. სქემებთან მიმართებით განირჩევა სამი ბრძანება:

- CREATE SCHEMA
- ALTER SCHEMA
- DROP SCHEMA

CREATE SCHEMA ქმნის სქემებს და ახორციელებს მონაცემთა ბაზის უსაფრთხოების მართვას.

მაგალითი 3

```
USE sample;
GO
CREATE SCHEMA my_schema AUTHORIZATION peter
GO
CREATE TABLE product
(product_no CHAR(10) NOT NULL UNIQUE, product_name CHAR(20) NULL, price
MONEY NULL);
GO
CREATE VIEW product_info
AS SELECT product_no, product_name
FROM product;
```

```
GO
GRANT SELECT TO mary;
DENY UPDATE TO mary;
```

ALTER SCHEMA ბრძანებით ობიექტის გარდაქმნა ხდება იმავე მონაცემთა ბაზის სხვადასხვა სქემებს შორის. მისი სინტაქსია:

```
ALTER SCHEMA schema_name TRANSFER object_name
```

DROP SCHEMA ბრძანება შლის სქემას მონაცემთა ბაზიდან მხოლოდ იმ შემთხვევაში, თუ სქემა არ შეიცავს არც ერთ ობიექტს. წინააღმდეგ შემთხვევაში სისტემა სქემის საკუთრების შეცვლის საშუალებას იძლევა ALTER AUTHORIZATION ბრძანების მეშვეობით.

მაგალითი 4

```
USE sample;
ALTER AUTHORIZATION ON SCHEMA ::my_schema TO mary;
```

მონაცემთა ბაზის უსაფრთხოება, როლები, ავტორიზაცია

მონაცემთა ბაზის ნებართვების აწყობა შესაძლებელია Transact-SQL ბრძანებების ან Management Studio-ს ან კიდევ პროცედურების სისტემის მეშვეობით.

მომხმარებლის დამატება Transact-SQL-ის მეშვეობით

მომხმარებლის დამატებისათვის მიმდინარე ბაზაში გამოიყენება ბრძანება CREATE USER, რომლის სინტაქსია:

```
CREATE USER user_name
[FOR {LOGIN login |CERTIFICATE cert_name |ASYMMETRIC KEY key_name}]
[ WITH DEFAULT_SCHEMA = schema_name ]
```

მაგალითი 5

```
USE sample;
CREATE USER peter FOR LOGIN [NTB11901\pete];
CREATE USER mary FOR LOGIN mary WITH DEFAULT_SCHEMA =my_schema;
```

ბრძანება ALTER USER ცვლის მონაცემთა ბაზის მომხმარებლის სახელს, მის default schema ან გადააწყობს მომხმარებელს სხვა სააღრიცხვო ჩანაწერზე.

ბრძანება DROP USER შლის მომხმარებელს მიმდინარე მონაცემთა ბაზიდან, თუ მისი ობიექტები არ არიან საკუთრივ დაცული.

სერვერის როლები

მომხმარებლები ერთნაირი ფუნქციებით, შეიძლება პირობითად სერვერის როლების (server role) შესაბამისად დაჯგუფდნენ. განირჩევა ორი სტანდარტული როლი: სერვერის დონეზე და მონაცემთა ბაზის დონეზე. მომხმარებელს ადმინისტრატორმა რომ მიანიჭოს ის უფლებები, რომელიც აქვს სერვერის რომელიმე

ფიქსირებულ როლს, ამისათვის საჭიროა მომხმარებლის ამ როლში ჩართვა. სერვერის ნებისმიერ როლში შეგვიძლია ჩავრთოთ სერვერის ან Windows NT-ის სააღრიცხვო ჩანაწერი. სერვერის სტანდარტული როლები (fixed server role) და მათი უფლებები მოყვანილია ცხრილში.

სერვერის ფიქსირებული როლები

სერვერის ფიქსირებული როლები	დანიშნულება
Sysadmin	შეუძლია ნებისმიერი მოქმედების შესრულება სერვერზე
Serveradmin	ასრულებს სერვერის კონფიგურირებასა და გამორთვას
Setupadmin	მართავს ბმულ სერვერებსა და პროცედურებს, რომლებიც ავტომატურად გაიშვება სერვერის გაშვებისას
Securityadmin	მართავს სააღრიცხვო ჩანაწერებსა და მონაცემთა ბაზის შექმნის წესებს, ასევე, შეუძლია შეცდომების ჟურნალის წაკითხვა
Processadmin	მართავს სერვერის მიერ გაშვებულ პროცესებს
Dbcreator	შეუძლია მონაცემთა ბაზების შექმნა და მოდიფიცირება
Diskadmin	მართავს სერვერის ფაილებს
Bulkadmin (Bulk Insert administrators)	შეუძლია ჩასვას მონაცემები მასობრივი გადაწერის საშუალებების გამოყენებით

მონაცემთა ბაზების როლები

მონაცემთა ბაზების როლები (database role) საშუალებას გვაძლევს მომხმარებლები გავაერთიანოთ ერთ ადმინისტრაციულ ერთეულში. კონკრეტული როლისთვის მონაცემთა ბაზის ობიექტებთან მიმართვის უფლებების განსაზღვრით ავტომატურად ვაძლევთ იგივე უფლებებს ამ როლის ყველა წევრს. მომხმარებლისათვის როლის შესაბამისი უფლებების მინიჭებისათვის, ეს მომხმარებელი ამ როლში უნდა ჩავრთოთ, რისთვისაც მიმდინარე მონაცემთა ბაზაში როლის დასამატებლად sp_addrole შენახვადი პროცედურა გამოიყენება. მისი სინტაქსია:

sp_addrole [@rolename =] 'როლის_სახელი' [, [@ownername =] 'მფლობელის_სახელი']

მფლობელი უნდა იყოს მიმდინარე მონაცემთა ბაზის მომხმარებელი ან მიმდინარე მონაცემთა ბაზის როლი. მიმდინარე მონაცემთა ბაზის როლში მონაცემთა ბაზის მომხმარებელის, მონაცემთა ბაზის როლის, Windows-ის სააღრიცხვო ჩანაწერისა და Windows-ის ჯგუფის ჩასართავად გამოიყენება sp_addrolemember შენახვადი პროცედურა, რომლის სინტაქსია:

sp_addrolemember [@rolename =] 'როლის_სახელი',
[@membername =] 'უსაფრთხოების_ობიექტის_სახელი'

სადაც:

–'როლის_სახელი'. მონაცემთა ბაზის როლის სახელია მიმდინარე მონაცემთა ბაზაში;

–'უსაფრთხოების_ობიექტის_სახელი' არის უსაფრთხოების სისტემის ობიექტის სახელი, რომელიც როლში უნდა ჩავრთოთ.

მონაცემთა ბაზის შექმნისას განისაზღვრება მონაცემთა ბაზის სტანდარტული როლები, რომლებიც არ შეიძლება იყოს წაშლილი ან შეცვლილი. მონაცემთა ბაზის სტანდარტული როლები (fixed database role) და მათი უფლებები მოყვანილია ცხრილში.

მონაცემთა ბაზის ფიქსირებული როლები	დანიშნულება
db_owner	აქვს ყველა უფლება მონაცემთა ბაზაში
db_accessadmin	შეუძლია მომხმარებლების დამატება და წაშლა
db_securityadmin	მართავს ყველა ნებართვას, ობიექტს, როლს და როლების წევრებს
db_ddladmin	შეუძლია DDL ბრძანებების შესრულება, გარდა GRANT, DENY და REVOKE ბრძანებებისა
db_backupoperator	შეუძლია DBCC, CHECKPOINT და BACKUP ბრძანებების შესრულება
db_datareader	შეუძლია ნახოს ნებისმიერი მონაცემები მონაცემთა ბაზის ნებისმიერ ცხრილში
db_datawriter	შეუძლია შეცვალოს ნებისმიერი მონაცემები მონაცემთა ბაზის ნებისმიერ ცხრილში
db_denydatareader	ეკრძალება ნახოს ნებისმიერი

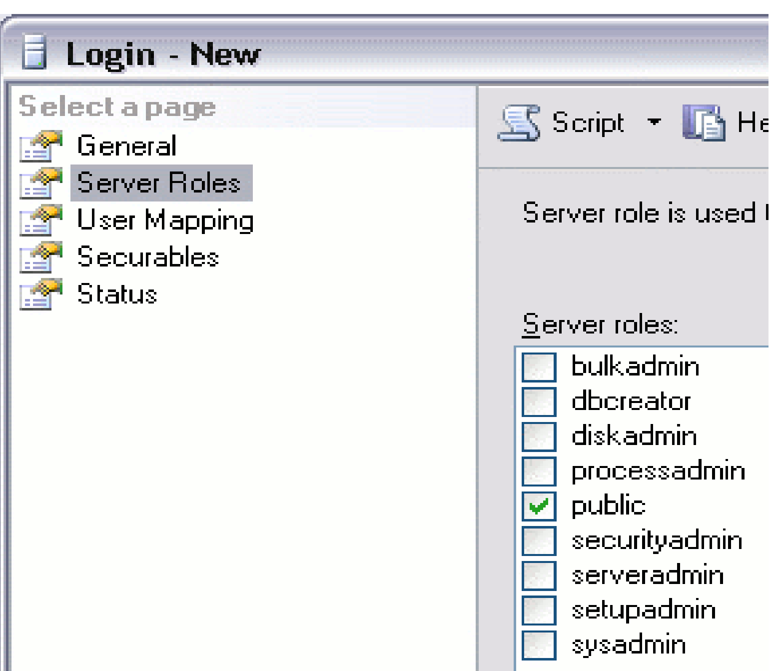
	მონაცემები მონაცემთა ბაზის ნებისმიერ ცხრილში
db_denydatawriter	ეკრძალება შეცვალოს ნებისმიერი მონაცემები მონაცემთა ბაზის ნებისმიერ ცხრილში

sa საალრიცხვო ჩანაწერი

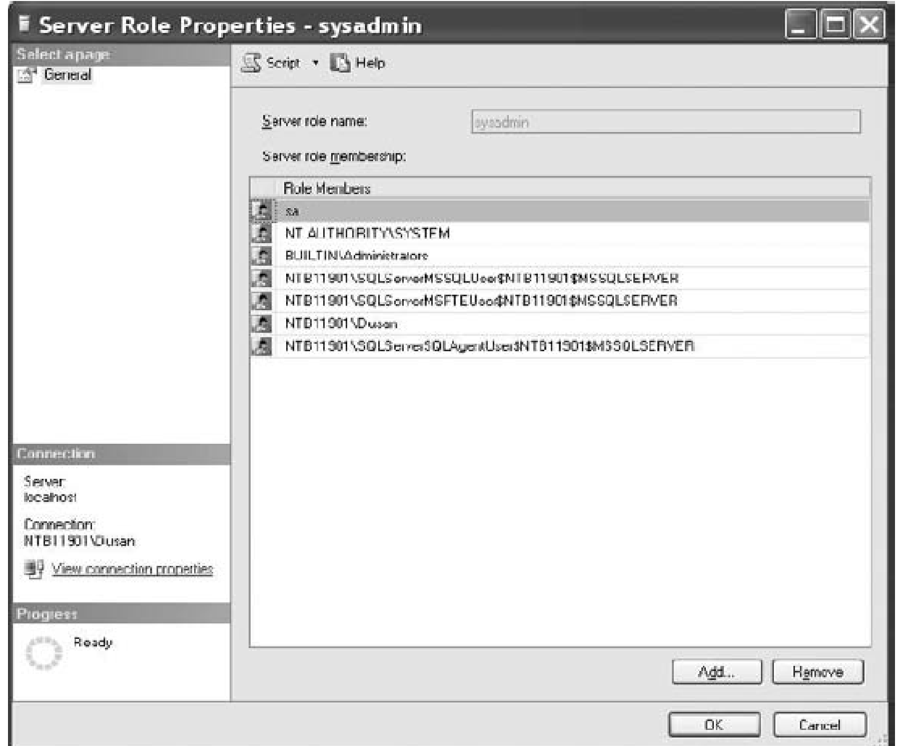
sa საალრიცხვო ჩანაწერი ეკუთვნის სისტემურ ადმინისტრატორს, რომელიც ყოველთვის არის **sysadmin** სერვერის სტანდარტული როლის წევრი და მისი ამ როლიდან ამოშლა არ შეიძლება.

საალრიცხვო ჩანაწერის დაყენება სერვერის სტანდარტულ როლში:

1. ვხსნით სერვერს;
2. შემდეგ ვხსნით **Security**–ს;
3. ვხსნით **Server Roles**–ს. მარჯვენა-click როლზე, რომელშიც გვინდა საალრიცხვო ჩანაწერის დამატება და შემდეგ click **Properties**–ზე.



4. **Server Role Properties** დიალოგური ფანჯრის მთავარ გვერდზე **General**, click **Add**–ზე. ვეძებთ საალრიცხვო ჩანაწერს, რომლის დამატება გვინდა მოცემული როლისათვის.



გამოყენებითი როლების მართვა

როლების შექმნის, ცვლილებისა და წაშლისათვის გამოიყენება Transact-SQL ბრძანებები ან სისტემური პროცედურები (**sp_addapprole**, **sp_setapprole** და **sp_dropapprole**). ბრძანება CREATE APPLICATION ROLE ქმნის როლს მიმდინარე მონაცემთა ბაზაში.

მაგალითი 6

```
USE sample;
CREATE APPLICATION ROLE weekly_reports
WITH PASSWORD ='x1y2z3w4',
DEFAULT_SCHEMA =my_schema;
```

ბრძანება ALTER APPLICATION ROLE ცვლის არსებულ გამოყენებით როლს (სახელს, პაროლს ან default schema). ბრძანება DROP APPLICATION ROLE შლის გამოყენებით როლს მიმდინარე მონაცემთა ბაზიდან.

გამოყენებითი როლების გააქტიურება

შეერთების შემდეგ სრულდება სისტემური პროცედურა **sp_setapprole** - გამოყენებითი როლის შესაბამისი ნებართვების გააქტიურება. პროცედურის სინტაქსია:

```
sp_setapprole [@rolename =] 'role' ,
[@password =] 'password'
[,[@encrypt =] 'encrypt_style']
```

გამოყენებითი როლების მართვა Management Studio–ს მეშვეობი:

1. გავხსნათ სერვერი, გავხსნათ **Databases;**
2. შემდეგ მონაცემთა ბაზა და მისი **Security** საქაღალდე;
3. მარჯვენა-click **Roles–**ზე, click **New**–ზე და შემდეგ click **New Application Role**–ზე. დიალოგურ ფანჯარაში **Application Role** შეგვაქვს ახალი როლის სახელი და პაროლი. დამატებით შეგვიძლია შევიტანოთ აგრეთვე default schema.

მომხმარებლის მიერ განსაზღვრული როლები

ამ როლების შექმნისა და წაშლისათვის გამოიყენება Transact-SQL ბრძანებები ან სისტემური პროცედურები (**sp_addrole** და **sp_droprole**). Transact-SQL ბრძანებები CREATE ROLE, ALTER ROLE და DROP ROLE შესაბამისად ქმნის, ცვლის და შლის მომხმარებლის მიერ განსაზღვრულ როლებს.

მიმდინარე მონაცემთა ბაზაში როლის დამატების შემდეგ ამ როლისათვის წევრების დამატება ხდება სისტემური პროცედურის **sp_addrolemember** მეშვეობით. მხოლოდ მონაცემთა ბაზის **db_owner** როლს შეუძლია ამ სისტემური პროცედურის შესრულება. როლიდან წევრის ამოშლა სრულდება სისტემური პროცედურის **sp_droprolemember** მიერ.

ავტორიზაცია

მხოლოდ ავტორიზებულ მომხმარებლებს ენიჭებათ მონაცემთა ბაზებთან მიმართვის ნებართვები.

GRANT ბრძანება გამოიყენება მონაცემთა ბაზის ობიექტებთან მომხმარებლის მიმართვის უფლებების მართვისთვის. მისი სინტაქსია:

```
GRANT {ALL [PRIVILEGES]} | permission_list
[ON [class::] securable] TO principal_list [WITH GRANT OPTION]
[AS principal ]
```

მაგალითი 7

```
USE sample;
GRANT CREATE TABLE, CREATE PROCEDURE
TO peter, mary;
```

მონაცემთა ბაზის ობიექტებთან მიმართვის უფლებები

მონაცემთა ბაზების ობიექტებთან (ცხრილები, სვეტები, წარმოდგენები, შენახვადი პროცედურები) მიმართვის უფლებები მართავენ მომხმარებლების მიერ ბრძანებების (SELECT, INSERT, UPDATE და DELETE) შესრულების შესაძლებლობას ცხრილებისა და წარმოდგენებისათვის. სხვადასხვა ობიექტებისთვის გამოიყენება მათთან მიმართვის უფლებების სხვადასხვა ნაკრებები:

- SELECT და UPDATE უფლებები გამოიყენება ცხრილის ან წარმოდგენის სვეტის მიმართ;
- SELECT, INSERT, UPDATE, DELETE და REFERENCES უფლებები გამოიყენება ცხრილებისა და წარმოდგენებისათვის;
- EXECUTE უფლება გამოიყენება მხოლოდ შენახვადი პროცედურებისა და ფუნქციების მიმართ.

INSERT უფლება, რომელიც შეიძლება გაიცეს მხოლოდ ცხრილის ან წარმოდგენის და არა სვეტის დონეზე, საშუალებას გვაძლევს ცხრილში ან წარმოდგენაში ჩავსვათ ახალი სტრიქონები.

UPDATE უფლება გაიცემა ცხრილის დონეზე, რაც საშუალებას გვაძლევს მთელ ცხრილში შევცვალოთ მონაცემები, ან სვეტის დონეზე, კონკრეტულ სვეტში შევცვალოთ მონაცემები.

DELETE უფლება გვაძლევს ცხრილიდან ან წარმოდგენიდან სტრიქონების წაშლის საშუალებას. იგი შეიძლება გაიცეს მხოლოდ ცხრილის ან წარმოდგენის დონეზე და არა სვეტის დონეზე.

SELECT უფლება იძლევა მონაცემების ამორჩევის შესაძლებლობას. SELECT გაიცემა როგორც ცხრილის, ისე სვეტის დონეზე.

REFERENCES უფლება იძლევა მითითებულ ობიექტთან მიმართვის შესაძლებლობას. ცხრილთან მიმართებით მომხმარებელს უფლებას აძლევს შექმნას ამ ცხრილის გარე გასაღები ან ამ ცხრილის უნიკალური სვეტი. წარმოდგენასთან მიმართებით მომხმარებელს უფლებას აძლევს წარმოდგენა დააკავშიროს ცხრილების სქემებთან, რომელთა საფუძველზეც იგება ეს წარმოდგენა.

მაგალითი 8

```
USE sample;
GRANT CREATE FUNCTION TO mary;
```

მაგალითი 9

```
USE sample;
GRANT SELECT ON employee
TO peter, mary;
```

მაგალითი 10

```
USE sample;
GRANT UPDATE ON works_on (emp_no, enter_date) TO peter;
```

მაგალითი 11

```
USE sample;
GRANT VIEW DEFINITION ON OBJECT::employee TO peter;
GRANT VIEW DEFINITION ON SCHEMA::dbo TO peter;
```

მაგალითი 12

```
USE sample;
GRANT CONTROL ON DATABASE::sample TO peter;
```

მაგალითი 13

```
USE sample;
GRANT SELECT ON works_on TO mary
WITH GRANT OPTION;
```

DENY ბრძანება გამოიყენება მონაცემთა ბაზის ობიექტთან მომხმარებლის მიმართვის აკრძალვისათვის. მისი სინტაქსია:

DENY { ALL [PRIVILEGES] | ნებართვა [,...n] } { [(სვეტის_სახელი [,...n])] ON { ცხრილის_სახელი | წარმოდგენის_სახელი } | ON { ცხრილის_სახელი | წარმოდგენის_სახელი } [(სვეტის_სახელი [,...n])] | ON შენახვადი_პროცედურის_სახელი | ON მომხმარებლის_ფუნქციის_სახელი } TO უსაფრთხოების_სისტემის_ობიექტის_სახელი [,...n] [CASCADE]

DENY ბრძანების არგუმენტები GRANT ბრძანების არგუმენტების ანალოგიურია.

CASCADE არგუმენტი საშუალებას გვაძლევს უფლებები ჩამოვართვათ არა მხოლოდ მოცემულ მომხმარებელს, არამედ იმ მომხმარებლებსაც, რომლებსაც ამ მომხმარებელმა გადასცა უფლებები.

მაგალითი 14

```
USE sample;
DENY CREATE TABLE, CREATE PROCEDURE
TO peter;
```

მაგალითი 15

```
USE sample;
GRANT SELECT ON project
TO PUBLIC;
DENY SELECT ON project
TO peter, mary;
```

REVOKE ბრძანება აუქმებს GRANT და DENY ბრძანებებში ადრე მინიჭებულ ერთ ან რამდენიმე ნებართვას. მისი სინტაქსია:

REVOKE [GRANT OPTION FOR] { ALL [PRIVILEGES] | ნებართვა [,...n] } { [(სვეტის_სახელი [,...n])] ON { ცხრილის_სახელი | წარმოდგენის_სახელი } | ON { ცხრილის_სახელი | წარმოდგენის_სახელი } [(სვეტის_სახელი [,...n])] | ON შენახვადი_პროცედურის_სახელი | ON მომხმარებლის_ფუნქციის_სახელი } { TO | FROM } უსაფრთხოების_სისტემის_ობიექტის_სახელი [,...n] [CASCADE] [AS { ჯგუფის_სახელი | როლის_სახელი }]

Transact_SQL-ის ბრძანების შესრულების უფლების არაცხადი უარყოფისათვის გამოიყენება ბრძანება:
REVOKE { ALL | ბრძანება [,...n] } FROM
უსაფრთხოების_სისტემის_ობიექტის_სახელი [,...n]
არგუმენტების დანიშნულება ისეთივეა, როგორიც GRANT და DENY ბრძანებებში.

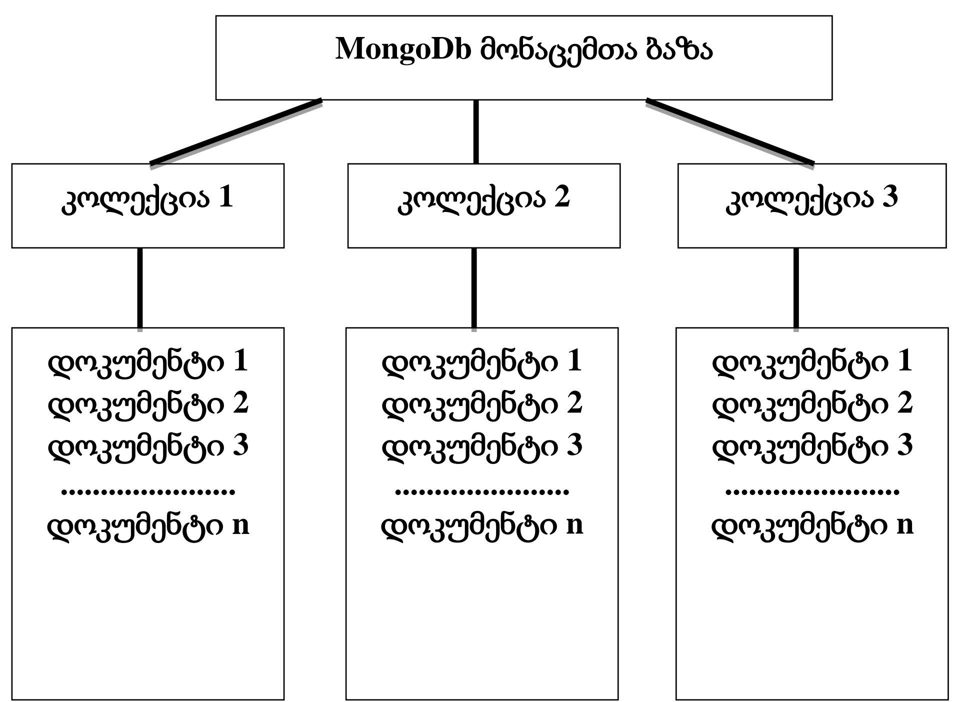
მაგალითი 16

```
USE sample;
REVOKE SELECT ON project
FROM PUBLIC;
```

II თავი. NoSql მონაცემთა ბაზებთან მუშაობა (Mongo DB)

2.1. NoSql მონაცემთა ბაზის სტრუქტურა

მთელი მონაცემთა ბაზის მოწყობილობის მოდელი MongoDB-ში შეიძლება წარმოდგენილი იყოს შემდეგი სტრუქტურის სახით:



თუ რელაციური მონაცემთა ბაზები შედგება ცხრილებისგან, მაშინ mongodb-ში მონაცემთა ბაზა შედგება კოლექციებისგან. თითოეულ კოლექციას აქვს თავისი უნიკალური სახელი - ნებისმიერი იდენტიფიკატორი, რომელიც შედგება არაუმეტეს 128 განსხვავებული ალფაციფრული სიმბოლოსა და ქვედა ხაზისგან.

რელაციური მონაცემთა ბაზებისგან განსხვავებით, MongoDB არ იყენებს ცხრილის მოწყობილობას სვეტების და მონაცემთა ტიპების ფიქსირებული რაოდენობით. MongoDB არის დოკუმენტზე ორიენტირებული სისტემა, რომელშიც ცენტრალური კონცეფცია არის დოკუმენტი.

დოკუმენტი შეიძლება ჩაითვალოს როგორც ობიექტი, რომელიც ინახავს გარკვეულ ინფორმაციას. გარკვეული გაგებით, ის რელაციური ბაზების სტრიქონების მსგავსია, სადაც სტრიქონები ინახავს ინფორმაციას ერთი ელემენტის შესახებ. მაგალითად, ტიპიური დოკუმენტი:

```
{
  "name": "Tom",
  "surname": "Smith",
  "age": "37",
  "company": {
```

```
        "name" : "Microsoft",
        "salary" : "100"
    }
}
```

გასაღები წარმოადგენს სტრიქონებს. მნიშვნელობები შეიძლება განსხვავდებოდეს მონაცემთა ტიპის მიხედვით. ამ შემთხვევაში, თითქმის ყველა მნიშვნელობა ასევე წარმოადგენს სტრიქონის ტიპს და მხოლოდ ერთი გასაღები (კომპანია) იგზავნება ერთ ობიექტზე. საერთო ჯამში, არსებობს შემდეგი ტიპის მნიშვნელობები:

String - სტრიქონული მონაცემთა ტიპი, (სტრიქონები იყენებენ UTF-8 დაშიფვრას);

Array - მონაცემთა ტიპი მასივის ელემენტების შესანახად;

Binary data - მონაცემთა ტიპი ორობით ფორმატში შესანახად;

Boolean - ლოგიკური მონაცემთა ტიპი, რომელიც ინახავს ლოგიკურ მნიშვნელობებს TRUE ან FALSE, მაგალითად, {"married": FALSE};

Date - თარიღის ტიპის მონაცემების შესანახად;

Double - ციფრული მონაცემთა ტიპი მცურავი წერტილის ნომრების შესანახად;

Integer - გამოიყენება 32 ბიტიანი მნიშვნელობების შესანახად, მაგ. {"ასაკი": 29};

Long - გამოიყენება 64 ბიტიანი მთელი რიცხვების შესანახად;

JavaScript - მონაცემთა ტიპი Javascript კოდის შესანახად;

Min key/Max key - გამოიყენება მნიშვნელობების შესადარებლად უმცირეს/უდიდეს BSON ელემენტებთან;

Null - მონაცემთა ტიპი ნულოვანი მნიშვნელობის შესანახად;

Object - ობიექტი, რომელიც შეიცავს თვისებების ნაკრებს;

ObjectId – მონაცემთა ტიპი დოკუმენტის ID-ის შესანახად;

Regular expression - გამოიყენება რეგულარული გამოსახულებების შესანახად;

Decimal128 - მონაცემთა ტიპი 128 ბიტის ზომით ათობითი წილადი რიცხვების შესანახად, რომელიც წყვეტს გამოთვლის სიზუსტის პრობლემას წილადი რიცხვების გამოყენებისას, რომლებიც წარმოადგენს Double ტიპის;

Timestamp: - გამოიყენება დროის შესანახად;

სტრიქონებისგან განსხვავებით, დოკუმენტები შეიძლება შეიცავდეს სხვადასხვა ინფორმაციას. ასე რომ, ზემოთ აღწერილი დოკუმენტის გვერდით, ერთ კოლექციაში შეიძლება იყოს სხვა ობიექტი, მაგალითად:

```
{
    "name": "Bob",
    "birthday": "1985.06.28",
    "place" : "Berlin",
    "languages" :[
        "english",
        "german",
```

```
    "spanish"
  ]
}
```

თითქოს სხვადასხვა ობიექტებია, ცალკეული თვისებების გარდა, მაგრამ ისინი ყველა შეიძლება იყოს ერთსა და იმავე კოლექციაში.

შენიშვნა: MongoDB - ში მოთხოვნები რეგისტრდამოკიდებულია და მკაცრად ტიპიზებული, მაგალითად ეს ორი დოკუმენტი ვერ იქნება იდენტური:

```
{"age" : "28"}
{"age" : 28}
```

თუ პირველ შემთხვევაში „ასაკი" გასაღებისთვის მნიშვნელობა განსაზღვრულია როგორც სტრიქონი, მაშინ მეორე შემთხვევაში მნიშვნელობა არის რიცხვი.

MongoDB-ში ყველა დოკუმენტს აქვს უნიკალური იდენტიფიკატორი, სახელწოდებით _id. როდესაც დოკუმენტი ემატება კოლექციას, ეს იდენტიფიკატორი ავტომატურად გენერირებულია. თუმცა, დეველოპერს შეუძლია თავად დააყენოს იდენტიფიკატორი და არ დაეყრდნოს ავტომატურად გენერირებულებს, შესაბამისი გასაღებისა და მისი მნიშვნელობის დოკუმენტში მითითებით.

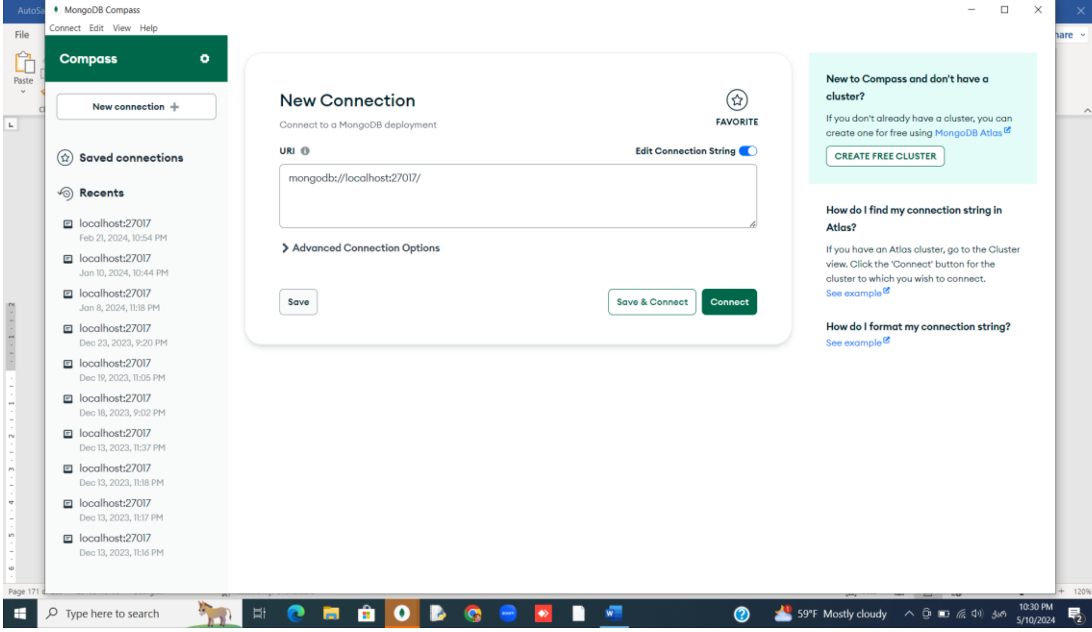
ამ ველს უნდა ჰქონდეს უნიკალური მნიშვნელობა კოლექციაში. ხოლო თუ შევეცდებით კოლექციას დავამატოთ ერთი და იგივე ID-ის ორი დოკუმენტი, მაშინ მათგან მხოლოდ ერთი დაემატება, ხოლო მეორის დამატებისას მივიღებთ შეცდომას.

თუ იდენტიფიკატორი ცხადად არ არის მითითებული, მაშინ MongoDB ქმნის სპეციალურ ორობით მნიშვნელობას 12 ბაიტით. ეს მნიშვნელობა შედგება რამდენიმე სეგმენტისგან: 4-ბაიტიანი დროის ანაბეჭდის მნიშვნელობა (რომელიც წარმოადგენს წამების რაოდენობას Unix-ის ეპოქიდან), 5-ბაიტიანი შემთხვევითი რიცხვი და 3-ბაიტიანი მრიცხველი, რომელიც ინიციალიზებულია შემთხვევითი რიცხვით. იდენტიფიკატორის აგების ასეთი მოდელი გარანტიას იძლევა მაღალი ხარისხის ალბათობით, რომ მას ექნება უნიკალური მნიშვნელობა.

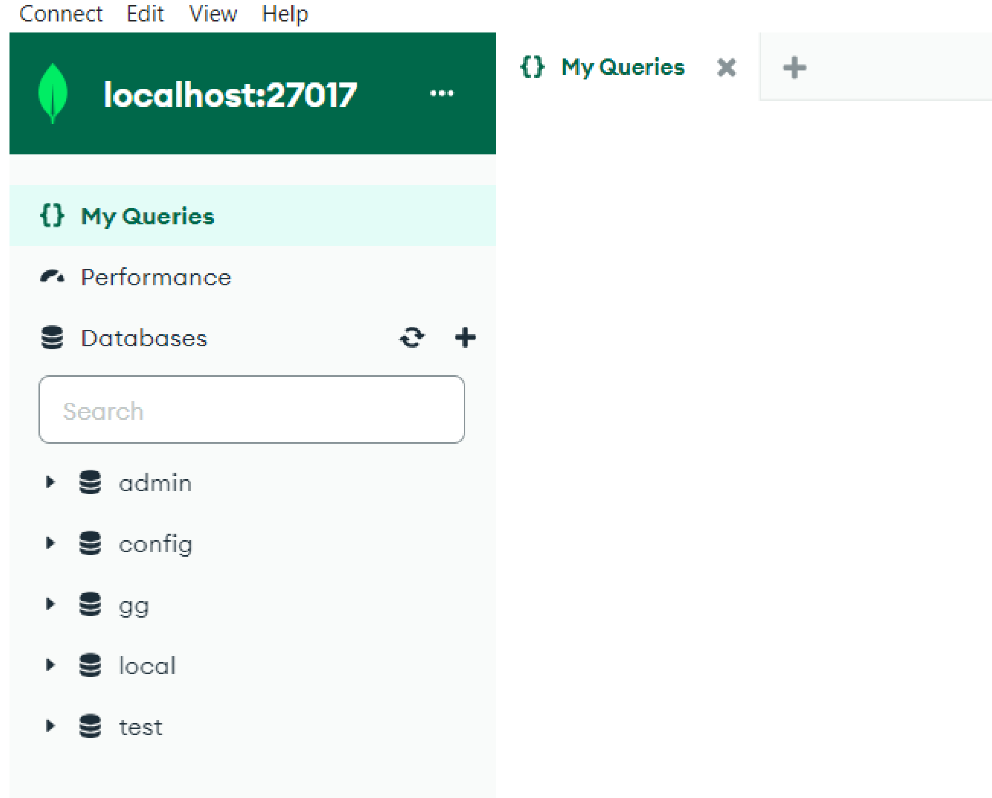
ჩვენ გავაშუქებთ მონაცემთა ძირითად ოპერაციებს MongoDB-ში, როგორც mongosh კონსოლის გარსის, ასევე MongoDB Compass გრაფიკული კლიენტის გამოყენებით. თუმცა, ნებისმიერ შემთხვევაში, როდესაც იწყებთ სერვერთან მუშაობას, არ უნდა დაგვავიწყდეს თავად სერვერის გაშვება - ეს არის MongoDB აპლიკაცია.

MongoDB-თან მუშაობის დაწყებისას mongosh shell-ში, პირველი ნაბიჯი არის ჩვენთვის საჭირო მონაცემთა ბაზის დაყენება მიმდინარე მონაცემთა ბაზად, რათა მოგვიანებით გამოვიყენოთ იგი. ამისათვის გამოიყენეთ გამოყენების ბრძანება, რომელსაც მოჰყვება მონაცემთა ბაზის სახელი. არ აქვს მნიშვნელობა ასეთი მონაცემთა ბაზა არსებობს თუ არა. თუ ის არ არსებობს, MongoDB ავტომატურად შექმნის მას, როდესაც მას მონაცემები დაემატება.

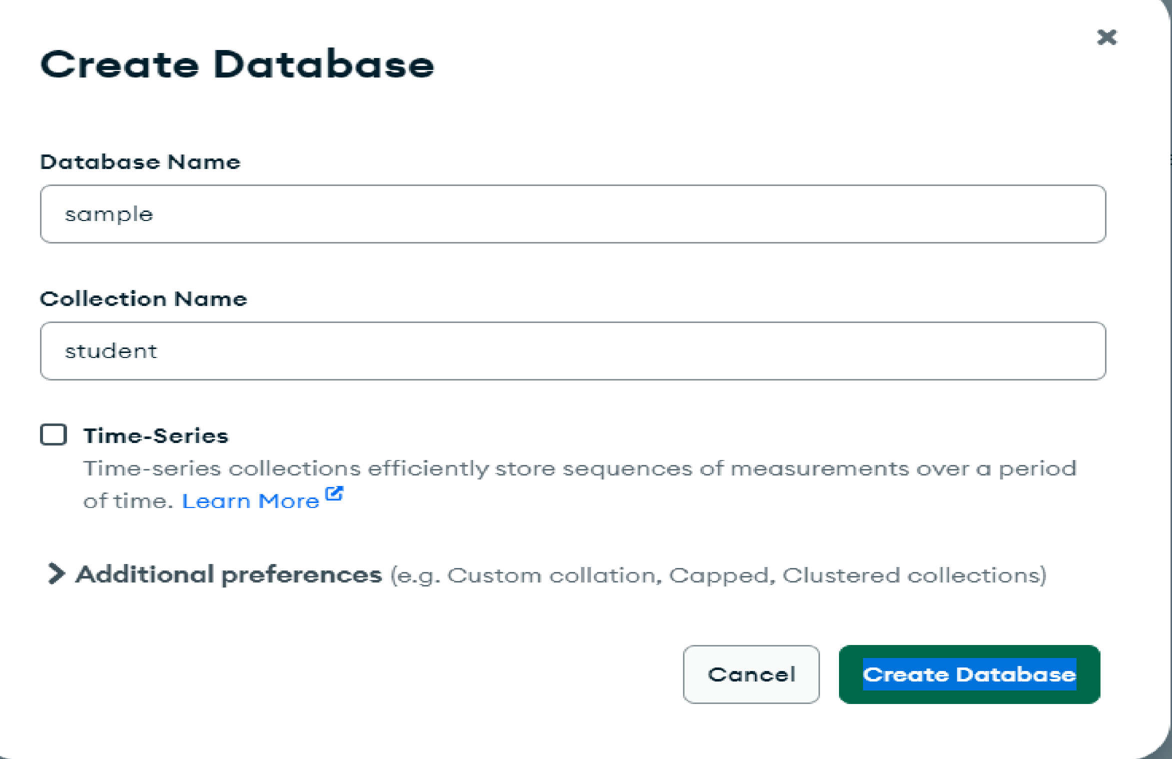
MongoDB გარემო მოცემულია სურათზე:



ვაჭერთ connect ღილაკზე, მიიღება შემდეგი ფანჯარა:



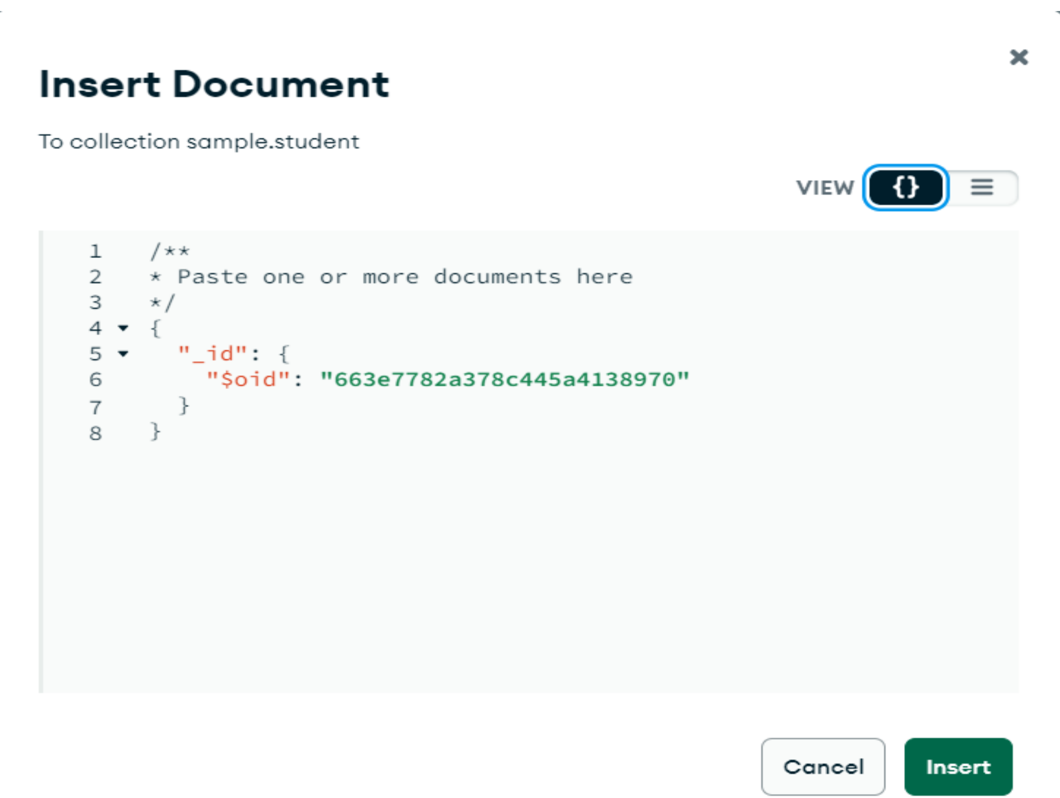
ვქმნით ახალ მონაცემთა ბაზას, შესაბამისად Database Name ველში ავკრიფოთ ბაზის სახელი, ხოლო Collection Name ველში - კოლექციის სახელი, დავაწკაპოთ Create Database ღილაკზე, როგორც სურათზეა მოცემული:



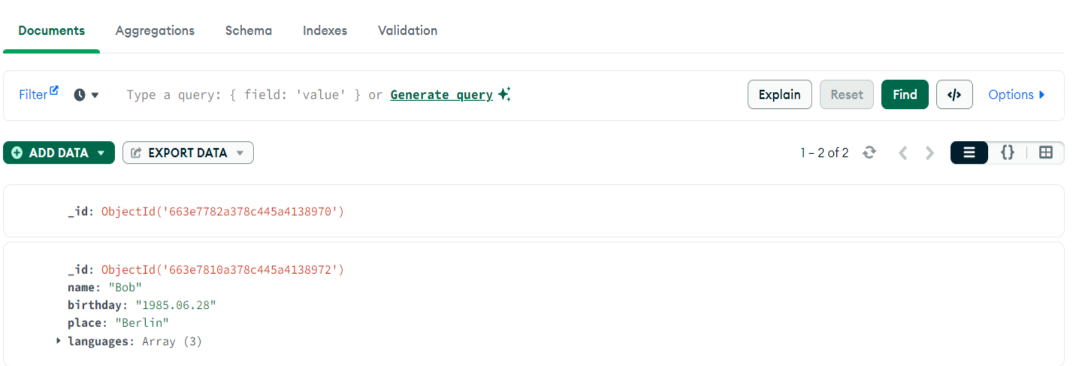
ჩვენ მიერ შექმნილი მონაცემთა ბაზა გამოჩნდება მონაცემთა ბაზების სიაში, სადაც ჩანს კოლექციაც:

MongoDB Compass - localhost:27017/sample.student
Connect Edit View Collection Help
localhost:27017
My Queries
Performance
Databases
Search
admin
config
gg
local
sample
student
My Queries | student
sample.student
Documents Aggregations Schema Indexes Validation
Filter Type a query: { field: 'value' } or Generate query
ADD DATA EXPORT DATA

დოკუმენტის დასამატებლად ავირჩიოთ Insert Document:

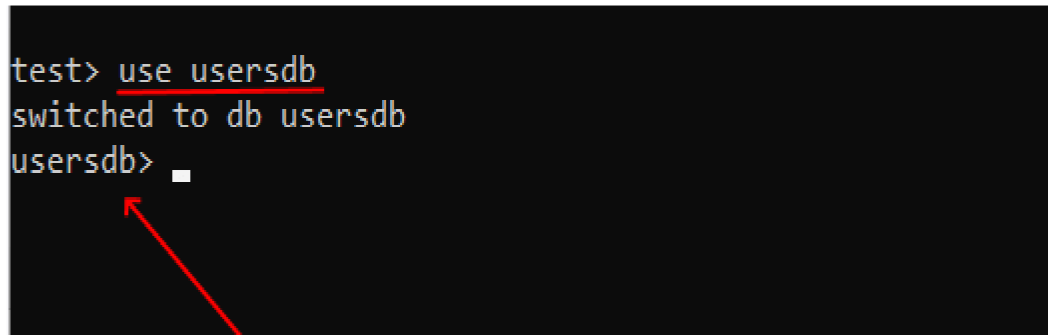


გამოსული კოდი შეგვიძლია გამოვიყენოთ ან სრულად წავშალოთ და ისე შევიტანოთ ჩვენ მიერ შექმნილი კოდი. შექმნილი დოკუმენტი გამოჩნდება ფანჯარაში:



კონსოლოის რეჟიმში მუშაობისთვის MongoDB ფანჯრის ქვედა მარჯვენა ნაწილში დავაწკაპოთ ^ ღილაკზე, გამოვა ფანჯარა, სადაც შეგვიძლია შევიტანოთ ბრძანებები. გავუშვათ mongosh კონსოლის გარემო და შეიყვანოთ შემდეგი ბრძანება:

use usersdb



ახლა usersdb მონაცემთა ბაზა დაყენდება როგორც მიმდინარე. არ აქვს მნიშვნელობა, რომ თავდაპირველად ასეთი მონაცემთა ბაზა შეიძლება არ არსებობდეს: თუ ის არ არსებობს, მაშინ პირველი ოპერაციის დროს ის იქმნება.

თუ არ ხართ დარწმუნებული, რომ ამ სახელის მონაცემთა ბაზა უკვე არსებობს, მაშინ show dbs ბრძანების გამოყენებით შეგიძლიათ აჩვენოთ ყველა არსებული მონაცემთა ბაზის სახელები კონსოლზე:

```
usersdb> show dbs
admin   40.00 KiB
config  72.00 KiB
local   72.00 KiB
test    40.00 KiB
usersdb>
```

გაითვალისწინეთ, რომ usersdb მონაცემთა ბაზა ჯერ არ არის მონაცემთა ბაზების სიაში, რადგან ჯერ არ გაგვიკეთებია მასთან რაიმე ოპერაცია.

მონაცემთა ბაზას შეიძლება მიენიჭოს ნებისმიერი სახელი, მაგრამ არსებობს გარკვეული შეზღუდვები. მაგალითად, სახელი არ შეიძლება შეიცავდეს /, \, ., ", *, <, >, :, |, ?, $. ასევე მონაცემთა ბაზის სახელები შეზღუდულია 64 ბაიტით.

ასევე არის დარეზერვებული სახელები, რომელთა გამოყენება შეუძლებელია: local, admin, config. ეს სახელები წარმოადგენს მონაცემთა ბაზებს, რომლებიც უკვე სერვერზეა ნაგულისხმევად და განკუთვნილია მომსახურე მიზნებისთვის.

უფრო მეტიც, როგორც ხედავთ, სატესტო მონაცემთა ბაზა არ არის ამ სიაში, რადგან მასში მონაცემები ჯერ არ დაგვიმატებია.

მონაცემთა ბაზების გარდა, ჩვენ შეგვიძლია ჩამოვთვალოთ ყველა კოლექცია მიმდინარე მონაცემთა ბაზაში ბრძანებით:

show collections

სტატისტიკის მიღება:

db.stats() ბრძანების გამოყენებით შეგიძლიათ მიიღოთ სტატისტიკა მიმდინარე მონაცემთა ბაზაზე. მაგალითად, ჩვენ გვაქვს სატესტო მონაცემთა ბაზა, როგორც მიმდინარე:

mongosh mongodb://127.0.0.1:27017/?directConnection=true&se...

```
usersdb> use test
switched to db test
test> show collections
users
test> db.stats()
{
  db: 'test',
  collections: 1,
  views: 0,
  objects: 1,
  avgObjSize: 36,
  dataSize: 36,
  storageSize: 20480,
  indexes: 1,
  indexSize: 20480,
  totalSize: 40960,
  scaleFactor: 1,
  fsUsedSize: 179737477120,
test> _
  ok: 1
```

ანალოგიურად, ჩვენ შეგვიძლია გავარკვიოთ კონკრეტული კოლექციის ყველა სტატისტიკა. მაგალითად, მოდით გავარკვიოთ სტატისტიკა “users” (თუ ასეთი გვაქვს) კოლექციის შესახებ:

db.users.stats()

განმარტებები:

db: 'test', // მონაცემთა ბაზის დასახელება

collections: 1, // კოლექციის რაოდენობა ბაზაში

views: 0, // წარმოდგენების რაოდენობა ბაზაში

objects: 2, // დოკუმენტების რაოდენობა ბაზაში (ბაიტებში)

dataSize: 114, // დოკუმენტების ზომა ბაზაში (ბაიტებში)

storageSize: 36864, // მონცემთა ბაზის მიერ დისკზე დაკავებული მეხსიერება

indexes: 1, // ინდექსების რაოდენობა ბაზაში

indexSize: 36864, // ინდექსების ზომა ბაზაში (ბაიტებში)

totalSize: 73728, // მონაცემთა ბაზის სრული ზომა, მონაემების და ინდექსების ჩათვლით

scaleFactor: 1, // მონაცემთა ბაზის მასშტაბური კოეფიციენტი

fsUsedSize: 32249266176, //საცავის გამოყენებული მოცულობა დისკზე მონაცემთა ბაზისთვის (ბაიტებში)

fsTotalSize: 110445457408, // მისაწვდომი საცავის საერთო მოცულობა დისკზე სისტემისთვის (ბაიტებში)

ok: 1 // ალამი, ოპერაციის წარმატებით შესრულების შეტყობინებაზე

2.2. კოლექციის შექმნა და წაშლა. დოკუმენტის შექმნა და წაშლა

როდესაც იყენებთ insert() მეთოდს დოკუმენტის ჩასართავად, თქვენ მიუთითებთ კოლექციას, რომელშიც ჩასმული იქნება დოკუმენტი. თუ კოლექცია ჯერ არ არსებობს, ის შეიქმნება.

db.artists.insert({ artistname: "The Tea Party" })

ამ შემთხვევაში მხატვრის კოლექცია მანამდე არ არსებობდა, ამიტომ ის ჩვენთვის შეიქმნა.

კოლექციის შექმნა createCollection() მეთოდის გამოყენებით:

db.createCollection("producers")

შეიძლება კოლექციის შექმნა პარამეტრების მითითებით:

db.createCollection(name, options).

მაგალითი:

db.createCollection("log", { capped : true, size : 4500500, max : 4000 })

ხელმისაწვდომი პარამეტრები:

capped ოპცია არის boolean ტიპია. true მნიშვნელობის დროს ქმნის შეზღუდულ კოლექციას, რომელიც არის ფიქსირებული ზომის კოლექცია, რომელიც ავტომატურად გადაიწერს თავის ყველაზე ძველ ჩანაწერებს, როდესაც მიაღწევს მაქსიმალურ ზომას. თუ მიუთითებთ true-ს ასევე უნდა დააყენოთ მაქსიმალური ზომა size ველში.

autoIndexId boolean მიუთითეთ false, რათა გაითიშოს ინდექსის ავტომატური შექმნა _id ველში. ეს ველი დაძველდა და ამოღებულია 3.4 ვერსიაში.

size number – მაქსიმალური სიგრძე ბაიტებში, შეზღუდული კოლექციისთვის. გამოიყენება მხოლოს დახურულ კოლექციებში.

max number - დახურულ კოლექციაში დასაშვები დოკუმენტების მაქსიმალური რაოდენობა. size ველს აქვს უპირატესობა max ველზე. თუ კოლექცია მიაღწევს ზომის ზღვარს მანამ, სანამ მიიღწევა დოკუმენტის ლიმიტი, MongoDB მაინც წაშლის დოკუმენტებს.

კოლექციის წაშლა

ვთქვათ გვინდა ვნახოთ კოლექციები ჩვენს მუსიკალურ მონაცემთა ბაზაში;

show collections

შედეგი;

artists

musicians

producers

წავშალოთ არტისტების კოლექცია:

db.artists.drop()

შედეგი:
true

გადავამოწმოთ თუ წაიშალა:
show collections
შედეგი:
musicians
producers

ჩვენს ბაზაში აღარ არის artists კოლექცია. მოვსინჯოთ მისი წაშლა:
db.artists.drop()
შედეგი:
false

დოკუმენტის წაშლა

დოკუმენტის წაშლისთვის არის სამი მეთოდი:
db.collection.deleteOne()
db.collection.deleteMany()
db.collection.remove()

db.collection.deleteOne() მეთოდი წაშლის ერთ დოკუმენტს, თუნდაც პირობას შეესაბამებოდეს ერთზე მეტი დოკუმენტი:
ჯერ გავუშვათ მოთხოვნა, რომელიც დააბრუნებს რამდენიმე შედეგს:
(
db.artists.find({ artistname: { $in: ["The Kooks", "Gang of Four", "Bastille"] } })
შედეგი:
{"_id" : ObjectId("5781d7f245re8c6b3ffb014d"), "artistname" : "The Kooks"}
{"_id" : ObjectId("5781d7f245re8c6b3ffb014e"), "artistname" : "Bastille"}
{"_id" : ObjectId("5781d7f245re8c6b3ffb014f"), "artistname" : "Gang of Four"}
ვნახეთ, რომ გვაქვს სამი დოკუმენტი, რომელიც ამ პირობას შეესაბამება.
გამოვიყენოთ db.collection.deleteOne() მეთოდი:
db.artists.deleteOne({ artistname: { $in: ["The Kooks", "Gang of Four", "Bastille"] } })
შედეგი:
{ "acknowledged" : true, "deletedCount" : 1 }
წაიშალა მხოლოდ ერთი, თუმცა კრიტერიუმს სამივე შეესაბამებოდა.
გავუშვათ find() მოთხოვნა და გადავამოწმოთ შედეგი:
db.artists.find({ artistname: { $in: ["The Kooks", "Gang of Four", "Bastille"] } })
შედეგი:
{ "_id" : ObjectId("5781d7f568ef8c6b3ffb014e"), "artistname" : "Bastille" }
{ "_id" : ObjectId("5781d7f568ef8c6b3ffb014f"), "artistname" : "Gang of Four" }

db.collection.deleteMany() მეთოდი

db.artists.deleteMany({ artistname: { $in: ["The Kooks", "Gang of Four", "Bastille"] } })

შედეგი:

{ "acknowledged" : true, "deletedCount" : 2 }

წაიშალა დარჩენილი ორივე დოკუმენტი.

db.collection.remove() მეთოდი

წაშლის ერთ ან ყველა დოკუმენტს, რომელიც შეესაბამება მითითებულ კრიტერიუმებს:

db.artists.remove({ artistname: "AC/DC" })

წაშლის ყველა დოკუმენტს AC/DC სახელით.

შედეგი:

WriteResult({ "nRemoved" : 1 })

კოლექციიდან ყველა დოკუმენტის წაშლა:

db.artists.remove({})

შედეგი:

WriteResult({ "nRemoved" : 8 })

მონაცემთა ბაზის წაშლა. db.dropDatabase() მეთოდი

შევამოწმოთ მონაცემთა ბაზის ჩამონათვალი:

> show databases

local 0.000GB

music 0.000GB

test 0.005GB

ავირჩიოთ მონაცემთა ბაზა, რომელსაც ვშლით, მაგალითად:

use music

შევასრულოთ წაშლის ბრძანება:

db.dropDatabase()

ისევ შევამოწმოთ ბაზის ჩამონათვალი:

show databases

local 0.000GB

test 0.005GB

dropDatabase ბრძანება

მონაცემთა ბაზის წაშლა შეიძლება შემდეგი ბრძანებითაც:

use test

db.runCommand({ dropDatabase: 1 })

{ "dropped" : "test", "ok" : 1 }

ბაზების სია ისევ შემცირდა:

> show databases

local 0.000GB

2.3. მონაცემთა დამატება

მონაცემთა ბაზის დაყენების შემდეგ, ჩვენ შეგვიძლია დავამატოთ მასში მონაცემები. ყველა მონაცემი ინახება მონაცემთა ბაზაში BSON ფორმატში, რომელიც ახლოსაა JSON-თან, ამიტომ ჩვენც უნდა შევიტანოთ მონაცემები ამ ფორმატში. და მიუხედავად იმისა, რომ შეიძლება არ გვქონდეს ერთი კოლექცია ამ მომენტში, როდესაც მას ვამატებთ მონაცემებს, ის ავტომატურად იქმნება.

მონაცემთა ბაზის დაინსტალირების შემდეგ, ჩვენ შეგვიძლია დავამატოთ მასში მონაცემები. ყველა მონაცემი ინახება მონაცემთა ბაზაში BSON ფორმატში, რომელიც ახლოსაა JSON-თან, ამიტომ ჩვენც უნდა შევიტანოთ მონაცემები ამ ფორმატში. და მიუხედავად იმისა, რომ შეიძლება არ გვქონდეს არცერთი კოლექცია ამ მომენტში, როდესაც მას ვამატებთ მონაცემებს, ის ავტომატურად იქმნება.

როგორც უკვე აღვნიშნეთ, კოლექციის სახელი არის ნებისმიერი იდენტიფიკატორი, რომელიც შედგება 128-მდე სხვადასხვა ალფავიტ-ციფრული სიმბოლოსგან და ქვედა ხაზისგან. ამავდროულად, კოლექციის სახელი არ უნდა დაიწყოს სისტემის პრეფიქსით, რადგან ის დაცულია შიდა კოლექციებისთვის (მაგალითად, system.users კოლექცია შეიცავს მონაცემთა ბაზის ყველა მომხმარებელს). და ასევე სახელი არ უნდა შეიცავდეს დოლარის ნიშანს - $.

კოლექციაში დასამატებლად გამოიყენება მისი 3 მეთოდი:

insertOne(): ერთი დოკუმენტის დამატება;

insertMany(): რამდენიმე დოკუმენტის დამატება;

ვთქვათ, ვიყენებთ მონაცემთა ბაზას “test”. მოდით დავამატოთ მას ერთი დოკუმენტი:

test> db.users.insertOne({"name": "Tom", "age": 28, languages: ["english", "spanish"]})

დოკუმენტი წარმოადგენს გასაღები-მნიშვნელობის წყვილების ერთობლიობას. ამ შემთხვევაში, დამატებულ დოკუმენტს აქვს სამი გასაღები: სახელი, ასაკი, ენები და თითოეულ მათგანს ენიჭება კონკრეტული მნიშვნელობა. მაგალითად, languages გასაღებს მნიშვნელობის სახით შეესაბამება მასივი.

აღსანიშნავია, რომ გასაღებების სახელები შეიძლება ჩასმული იყოს ბრჭყალებში ან მის გარეშე.

ზოგიერთი შეზღუდვა გასაღების სახელების გამოყენებისას:

- ✓ $ სიმბოლო არ შეიძლება იყოს საკვანძო სახელის პირველი სიმბოლო
- ✓ გასაღების სახელი არ შეიძლება შეიცავდეს სიმბოლო „წერტილს“.

მონაცემთა დამატებისას, თუ ჩვენ ცალსახად არ მივუთითეთ მნიშვნელობა "_id" ველისთვის (ანუ უნიკალური დოკუმენტის იდენტიფიკატორი), მაშინ ის ავტომატურად გენერირებულია. ამდენად, დამატების ოპერაციის შესრულების შემდეგ, კონსოლი აჩვენებს დამატებული დოკუმენტისთვის გენერირებულ იდენტიფიკატორს:

test> db.users.insertOne({"name": "Tom", "age": 28, languages: ["english", "spanish"]})

გამოიტანს:

```
{
  acknowledged: true,
  insertedId: ObjectId("62e27b1b06adfcddf4619fc1")
```

}
test>

სერვერის პასუხში ჩვენ მივიღებთ ობიექტს, რომლის insertedId პარამეტრი შეიცავს იდენტიფიკატორს.
მაგრამ პრინციპში, ჩვენ შეგვიძლია დავაყენოთ ეს იდენტიფიკატორი მონაცემთა დამატებისას:
test> db.users.insertOne({"_id": 123457, "name": "Tom", "age": 28, languages: ["english", "spanish"]})
ან გამოვიყენოთ იდენტიფიკატორისთვის ObjectId ტიპი.
test> db.users.insertOne({"_id": ObjectId("62e27b1b06adfcddf4619fc6"), "name": "Tom", "age": 28, languages: ["english", "spanish"]})
აღსანიშნავია, რომ თუ იდენტიფიკატორის განსაზღვრისთვის გამოიყენება ObjectId ტიპი, მაშინ იგი უნდა შეიცავდეს 12 ბაიტის მქონე სტრიქონს ან 24 სიმბოლოიან სტრიქონს.
წარმატებული დამატების შემთხვევაში კონსოლზე გამოვა დამატებული დოკუმენტის იდენტიფიკატორი.
იმაში დასარწმუნებლად, რომ დოკუმენტი მონაცემთა ბაზაშია, გამოვიტანოთ იგი find ფუნქციით:
test> db.users.find()
დუმილით, ის გამოსცემს კოლექციაში არსებულ ყველა დოკუმენტს:

```
mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
test> db.users.insertOne({"name": "Tom", "age": 28, languages: ["english", "spanish"]})
{
  acknowledged: true,
  insertedId: ObjectId("62e2d183e75ce6a476c170aa")
}
test> db.users.find()
[
  {
    _id: ObjectId("62e2d183e75ce6a476c170aa"),
    name: 'Tom',
    age: 28,
    languages: [ 'english', 'spanish' ]
  }
]
test>
```

თუ ჩვენ გვჭირდება რამდენიმე დოკუმენტის დამატება, მაშინ შეგვიძლია გამოვიყენოთ insertMany() მეთოდი, რომელიც იღებს ობიექტების მასივს:
db.users.insertMany([{"name": "Bob", "age": 26, languages: ["english", "french"]},
{"name": "Alice", "age": 31, languages:["german", "english"]}])
დამატების შემდეგ, კონსოლი გამოსცემს დამატებული დოკუმენტების ID-ებს:

```
mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
]
test> db.users.insertMany([{"name": "Bob", "age": 26, languages: ["english", "french"]},
... {"name": "Alice", "age": 31, languages:["german", "english"]}])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("62e2d255e75ce6a476c170ab"),
    '1': ObjectId("62e2d255e75ce6a476c170ac")
  }
}
test> 
```

არსებობს დოკუმენტის მონაცემთა ბაზაში დამატების კიდევ ერთი გზა, რომელიც მოიცავს ორ ნაბიჯს: დოკუმენტის განსაზღვრა (document = ({ ... })) და უშუალოდ მისი დამატება:

document=({"name": "Bill", "age": 32, languages: ["english", "french"]})

db.users.insertOne(document)

შეიძლება ყველასთვის მოსახერხებელი არ იყოს ყველა წყვილი გასაღებისა და თვისების ერთ ხაზზე შეყვანა. მაგრამ MongoDB-ის javascript-ზე დაფუძნებული ჭკვიანი თარჯიმანი საშუალებას გაძლევთ შეიყვანოთ მრავალ ხაზოვანი ბრძანებებიც. თუ გამოსახულება არ არის დასრულებული (JavaScript ენის თვალსაზრისით) და დააჭირეთ Enter-ს, მაშინ მისი შემდეგი ნაწილის შეყვანა ავტომატურად გადაიტანება შემდეგ სტრიქონზე:

```
mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
}
test> doc=({
... "name": "Sam",
... "age": 26,
... "languages":[
... "english",
... "spanish"]
... })
{ name: 'Sam', age: 26, languages: [ 'english', 'spanish' ] }
test> db.users.insertOne(doc)
{
  acknowledged: true,
  insertedId: ObjectId("62e2d348e75ce6a476c170ae")
}
test>
```

ფაილიდან მონაცემების ჩატვირთვა

mongodb მონაცემთა ბაზის მონაცემები შეიძლება განისაზღვროს უბრალო ტექსტურ ფაილში, რაც საკმაოდ მოსახერხებელია, რადგან ჩვენ შეგვიძლია ამ ფაილის

გადატანა ან გადაგზავნა mongodb მონაცემთა ბაზისგან დამოუკიდებლად. მაგალითად, მოდით განვსაზღვროთ users.js ფაილი სადმე მყარ დისკზე, შემდეგი შინაარსით:

```
db.users.insertMany([
{"name": "Alice", "age": 31, languages: ["english", "french"]},
{"name": "Lene", "age": 29, languages: ["english", "spanish"]},
{"name": "Kate", "age": 30, languages: ["german", "russian"]}
])
```

ანუ, აქ, insertMany მეთოდის გამოყენებით, „users" კოლექციას ემატება სამი დოკუმენტი:

ფაილის მიმდინარე მონაცემთა ბაზაში ჩასატვირთად გამოიყენება load() ფუნქცია, რომელშიც პარამეტრად გადაეცემა გზა ფაილისაკენ:

```
load("D:/users.js")
```

ამ შემთხვევაში, ივარაუდება, რომ ფაილი მდებარეობს მისამართზე: "D:/users.js".

2.4. ამორჩევა და ფილტრაცია

კოლექციიდან დოკუმენტების ამოღების უმარტივესი გზაა find() ფუნქციის გამოყენება. ეს ფუნქცია მუშაობს ისევე, როგორც ჩვეულებრივი SQL SELECT * FROM Table მოთხოვნა, რომელიც ამოიღებს ყველა სტრიქონს. მაგალითად, წინა თემაში შექმნილი users კოლექციიდან ყველა დოკუმენტის დასაბრუნებლად, შეგვიძლია გამოვიყენოთ ბრძანება:

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

```
Выбрать mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutM...

test> db.users.find()
[
  {
    _id: ObjectId("62e2d183e75ce6a476c170aa"),
    name: 'Tom',
    age: 28,
    languages: [ 'english', 'spanish' ]
  },
  {
    _id: ObjectId("62e2d255e75ce6a476c170ab"),
    name: 'Bob',
    age: 26,
    languages: [ 'english', 'french' ]
  },
  {
    _id: ObjectId("62e2d255e75ce6a476c170ac"),
    name: 'Alice',
    age: 31,
```

მონაცემთა ფილტრაცია

თუმცა, რა მოხდება, თუ ჩვენ გვჭირდება არა ყველა დოკუმენტის მიღება, არამედ მხოლოდ ის, ვინც აკმაყოფილებს გარკვეულ მოთხოვნას. მაგალითად, ჩვენ ადრე დავამატეთ შემდეგი დოკუმენტები ბაზაში:

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

გამოვიტანოთ ყველა დოკუმენტი, რომელშიც name=Tom:

```
db.users.find({name: "Tom"})
```

შედეგი გამოვა:

```
test> db.users.find({name: "Tom"})
[
  {
   _id: ObjectId("62e2d6a5e75ce6a476c170b3"),
   name: 'Tom',
   age: 28,
   languages: [ 'english', 'spanish' ]
  },
  { _id: ObjectId("62e2d348e75ce6a476c170ae"),
   _id: ObjectId("62e2d6a5e75ce6a476c170b5"),
   name: 'Tom',
   age: 32,
   languages: [ 'english', 'german' ]
  }
]  _id: ObjectId("62e2d3d5e75ce6a476c170af"),
test>
```

ახლა უფრო რთული მოთხოვნა: ჩვენ უნდა გამოვიტანოთ ის ობიექტები, რომლებთათვისაც ერთდროულად name=Tom და age=32. ეს მოთხოვნა SQL-ში შეიძლება ასე გამოიყურებოდეს: SELECT * FROM Table WHERE Name='Tom' AND Age=32. ამ კრიტერიუმს შეესაბამება ბოლო დამატებული ობიექტი. დავწეროთ ეს მოთხოვნა:

```
db.users.find({name: "Tom", age: 32})
```

გაფილტვრა გამოტოვებული თვისებების მიხედვით

ზოგიერთ დოკუმენტს შეიძლება ჰქონდეს გარკვეული თვისება, ზოგს კი არა. რა მოხდება, თუ ჩვენ გვინდა მოვიპოვოთ დოკუმენტები, რომლებსაც არა აქვთ გარკვეული თვისება? ამ შემთხვევაში თვისებას გადაეცემა null მნიშვნელობა. საკუთრებაზე ითვლება ნული. მაგალითად, ვიპოვოთ ყველა დოკუმენტი, სადაც აკლია თვისება languages:

db.users.find({languages: null})
ან იპოვეთ ყველა დოკუმენტი, სადაც name = "Tom", მაგრამ თვისება languages არ არის განსაზღვრული:
db.users.find({name: "Tom", languages: null})

ფილტრაცია მასივის ელემენტების მიხედვით

ასევე ადვილია მასივის ელემენტის მიხედვით პოვნა. მაგალითად, შემდეგი მოთხოვნა აბრუნებს ყველა დოკუმენტს, რომელსაც ენების მასივში აქვთ ინგლისური ენა:
db.users.find({languages: "english"})
გავართულოთ შეკითხვა და მივიღოთ ის დოკუმენტები, რომლებსაც აქვთ ენების მასივში ერთდროულად ორი ენა: "ინგლისური" და "გერმანული":
db.users.find({languages: ["english", "german"]})
და ამ თანმიმდევრობით, "ინგლისური" განსაზღვრული პირველი და "გერმანული" მეორე.
ახლა გამოვიტანოთ ყველა დოკუმენტი, რომელშიც "ინგლისური" პირველ ადგილზეა ენების მასივში:
db.users.find({"languages.0": "english"})
გაითვალისწინეთ, რომ "languages.0" უზრუნველყოფს რთულ თვისებას და ამიტომ არის ბრჭყალებში. შესაბამისად, თუ ჩვენ გვჭირდება დოკუმენტების ჩვენება, სადაც ინგლისური მეორე ადგილზეა (მაგალითად, ["german", "english"]), მაშინ ნულის ნაცვლად ვაყენებთ ერთს: "languages.1".
განვიხილოთ უფრო რთული მაგალითი, სადაც მასივის ელემენტი წარმოადგენს რთულ ობიექტს. ვთქვათ, ჩვენ გვაქვს შემდეგი დოკუმენტები ჩვენს ბაზაში:
db.users.insertOne({"name": "Bob", "age": 28, friends: [{"name": "Tim"}, {"name": "Tom"}]})
db.users.insertOne({"name": "Tim", "age": 29, friends: [{"name": "Bob"}, {"name": "Tom"}]})
db.users.insertOne({"name": "Sam", "age": 31, friends: [{"name": "Tom"}]})
db.users.insertOne({"name": "Tom", "age": 32, friends: [{"name": "Bob"}, {"name": "Tim"}, {"name": "Sam"}]})

გამოვიტანოთ ყველა დოკუმენტი, სადაც friends მასივში პირველი ელემენტის name თვისება "Bob" ტოლია:
test> db.users.find({"friends.0.name": "Bob"})
გამოტანილი შედეგი:

```
test> db.users.find({"friends.0.name": "Bob"})
[
  {
    _id: ObjectId("62e39da1c881653067e87901"),
    name: 'Tim',
```

```
    age: 29,
    friends: [ { name: 'Bob' }, { name: 'Tom' } ]
  },
  {
    _id: ObjectId("62e39da1c881653067e87903"),
    name: 'Tom',
    age: 32,
    friends: [ { name: 'Bob' }, { name: 'Tim' }, { name: 'Sam' } ]
  }
]
test>
```

2.5. პროექცია

დოკუმენტს შეიძლება ჰქონდეს მრავალი ველი, მაგრამ ყველა ეს ველი არ არის აუცილებელი და მნიშვნელოვანი ჩვენთვის მოთხოვნისას. და ამ შემთხვევაში, ჩვენ შეგვიძლია შევიტანოთ მხოლოდ საჭირო ველები შერჩევაში პროექციის გამოყენებით. მაგალითად, ვაჩვენოთ მხოლოდ ინფორმაციის ნაწილი, მაგალითად, გამოვიტანოთ „age“ ველების მნიშვნელობები ყველა დოკუმენტისთვის, რომელშიც name=Tom:

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

ერთის გამოყენება {age: 1} პარამეტრად მიუთითებს, რომ მოთხოვნამ უნდა დააბრუნოს მხოლოდ ასაკის თვისების შიგთავსი.

```
test> db.users.find({name: "Tom"}, {age: 1})
[
  { _id: ObjectId("62e2d6a5e75ce6a476c170b3"), age: 28 },
  { _id: ObjectId("62e2d6a5e75ce6a476c170b5"), age: 32 },
  { _id: ObjectId("62e2d799e75ce6a476c170b7"), age: 28 },
  { _id: ObjectId("62e39da1c881653067e87903"), age: 32 }
]
test>
```

და საპირისპირო სიტუაცია: ჩვენ გვინდა ვიპოვოთ დოკუმენტის ყველა ველი, გარდა ასაკობრივი თვისებისა. ამ შემთხვევაში, პარამეტრად მიუთითეთ 0:

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

გავითვალისწინოთ, რომ თუ აღვნიშნავთ, რომ ჩვენ გვინდა მივიღოთ მხოლოდ name ველი, ველი _id ასევე ჩაერთვება ამორჩევაში. ამიტომ თუ არ გვინდა ამორჩევაში გვქონდეს ეს ველი, მაშინ ცხადად უნდა მივუთითოთ: {"_id":0}

ალტერნატიულად, true და false შეიძლება გამოყენებულ იქნას 1 და 0-ის ნაცვლად:

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

თუ არ გვსურს დავაკონკრეტოთ ამორჩევა, მაგრამ გვინდა ყველა დოკუმენტის გამოტანა, მაშინ შეგვიძლია პირველი ფიგურული ფრჩხილები ცარიელი დავტოვოთ:

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

2.6. მოთხოვნები ჩაშენებულ ობიექტებთან

წინა მოთხოვნები გამოიყენებოდა მარტივი ობიექტებისთვის. მაგრამ დოკუმენტები შეიძლება იყოს ძალიან რთული სტრუქტურის. მაგალითად, მოდით დავამატოთ შემდეგი დოკუმენტი users კოლექციას:

db.users.insertOne({"name": "Alex", "age": 28, company: {"name":"Microsoft", "country":"USA"}})

აქ განსაზღვრულია ჩაშენებული ობიექტი კომპანია გასაღებით. ყველა დოკუმენტის საპოვნელად, რომელსაც აქვს გასაღები „კომპანია“, საჭიროა ოპერატორ „წერტილის“ გამოყენება:

db.users.find({"company.name": "Microsoft"})

JavaScript გამოყენება

მონაცემთა ბაზის მოთხოვნების შესრულების გარდა, ჩვენ შეგვიძლია JavaScript გამოსახულების შესრულება. მაგალითად, ჩვენ შეგვიძლია შევქმნათ გარკვეული ფუნქცია და გამოვიყენოთ იგი:

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

კონსოლის შედეგი:

```
test> function sqrt(n) { return n*n; }
[Function: sqrt]
test> sqrt(5)
25
test>
```

ჩვენ შეგვიძლია გამოვიყენოთ მსგავსი ფუნქციები და JavaScript გამოსახულებები მონაცემთა ბაზის მოთხოვნებში. მაგალითად, მოვიძიოთ ყველა დოკუმენტი, სადაც ასაკის ველი უდრის sqrt(5)+3:

```
test> db.users.find({age: sqrt(5)+3})
[
  {
    _id: ObjectId("62e2d6a5e75ce6a476c170b3"),
```

```
    name: 'Tom',
    age: 28,
    languages: [ 'english', 'spanish' ]
  },
  { _id: ObjectId("62e2d76ae75ce6a476c170b6"), name: 'Tomas', age: 28 },
  { _id: ObjectId("62e2d799e75ce6a476c170b7"), name: 'Tom', age: 28 },
  {
    _id: ObjectId("62e39da1c881653067e87900"),
    name: 'Bob',
    age: 28
  }
]
test>
```

რეგულარული გამოსახულებების გამოყენება

მოთხოვნების შექმნის კიდევ ერთი შესანიშნავი შესაძლებლობაა რეგულარული გამოსახულებების გამოყენება. მაგალითად, ვიპოვოთ ყველა დოკუმენტი, რომელშიც name გასაღების მნიშვნელობა იწყება B ასოთი:

```
db.users.find({name:/^B\w+/i})
```

კონსოლის შედეგის ნიმუში:

```
test> db.users.find({name:/^B\w+/i})
[
  {
    _id: ObjectId("62e2d6a5e75ce6a476c170b4"),
    name: 'Bill',
    age: 32,
    languages: [ 'english', 'french' ]
  },
  {
    _id: ObjectId("62e39da1c881653067e87900"),
    name: 'Bob',
    age: 28
  }
]
test>
```

ერთი დოკუმენტის ძებნა

თუ ყველა დოკუმენტი ამოიღება find-ის მიერ, მაშინ ერთი დოკუმენტი ამოიღება findOne-ის მიერ

მაგალითად, მოდით ამოვარჩიოთ ერთი ელემენტი, რომელშიც name="Tom":

```
test> db.users.findOne({name: "Tom"})
{
  _id: ObjectId("62e2d6a5e75ce6a476c170b3"),
  name: 'Tom',
  age: 28,
  languages: [ 'english', 'spanish' ]
}
test>
```

2.7. კურსორი

ამორჩევის შედეგს, რომელიც მიღებულია find ფუნქციის მიერ ეწოდება კურსორი. საჭიროების შემთხვევაში ჩვენ შეგვიძლია გადავცეთ კურსორი ცალკეულ ცვლადში:

```
var cursor = db.users.find()
```

კურსორი ინკაპსულაციას უკეთებს მონაცემთა ბაზის ობიექტებიდან მიღებულ ნაკრებს. javascript ენის სინტაქსის და კურსორის მეთოდების გამოყენებით შეგვიძლია გამოვიტანოთ მიღებული დოკუმენტები ეკრანზე და დავამუშავოთ ისინი. მაგალითად:

```
var cursor = db.users.find()
 while(cursor.hasNext()){
  obj = cursor.next();
  print(obj["name"]);
}
```

კურსორს აქვს hasNext მეთოდი, რომელიც გადარჩევისას უჩვენებს ნაკრებს არის თუ არა კიდევ დოკუმენტი. ხოლო next მეთოდი ამოიღებს მიმდინარე დოკუმენტს და გადააადგილებს კურსორს შემდეგ დოკუმენტზე. შედეგად obj ცვლადში აღმოჩნდება დოკუმენტი, რომლის ველებზე ჩვენ შეგვიძლია წვდომის მიღება.

```
 test> var cursor = db.users.find()
test> while(cursor.hasNext()){
...   obj = cursor.next();
...   print(obj["name"]);
... }
Tom
Bob
Sam
test>
```

ასევე კურსორსი დოკუმენტების გადარჩევისთვის ალტერნატივის სახით ჩვენ შეგვიძლია გამოვიყენოთ **javascript** იტერატორის კონსტრუქცია - **forEach.**

```
var cursor = db.users.find()
```

```
cursor.forEach(function(obj){
  print(obj.name);
})
```

ფუნქცია limit

ფუნქცია განსაზღვრავს მისაღები დოკუმენტების რაოდენობას. რაოდენობა გადაეცემა რიცხვითი პარამეტრის სახით. მაგალითად შევზღუდოთ ამორჩევა 3 დოკუმენტით:

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

ამ შემთხვევაში მივიღებთ პირველ სამ დოკუმენტს (თუ კრებულში 3 ან მეტი დოკუმენტია). მაგრამ რა მოხდება, თუ ჩვენ გვსურს ამორჩევა გავაკეთოთ არა თავიდან, არამედ რამდენიმე დოკუმენტის გამოტოვებით? ამაში skip ფუნქცია დაგვეხმარება. მაგალითად, გამოვტოვოთ პირველი სამი ჩანაწერი:

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

ორივე ფუნქციის კომბინაციით, ჩვენ შეგვიძლია მივიღოთ დოკუმენტების გარკვეული რაოდენობა, დაწყებული გარკვეული დოკუმენტიდან. მაგალითად, მოდით ავირჩიოთ დოკუმენტები 4-დან 6-მდე:

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

MongoDB უზრუნველყოფს მონაცემთა ბაზიდან მიღებული მონაცემთა ნაკრების დახარისხების შესაძლებლობას დახარისხების ფუნქციის გამოყენებით. ამ ფუნქციისთვის 1 ან -1 მნიშვნელობების გადაცემით, შეგვიძლია განვსაზღვროთ რა თანმიმდევრობით დახარისხდება: აღმავალი (1) ან კლებადი (-1). მრავალი თვალსაზრისით, ეს ფუნქცია მსგავსია ORDER BY გამოსახულების SQL-ში. მაგალითად, დახარისხება ზრდადი თანმიმდევრობით სახელის ველის მიხედვით:

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

მაგალითად, ვაჩვენოთ მხოლოდ "სახელის" ველის მნიშვნელობები მონაცემთა ბაზიდან, დავალაგოთ ისინი ზრდადი თანმიმდევრობით:

```
test> db.users.find({}, {name:1, _id: 0}).sort({name: 1})
[
  { name: 'Bill' },
  { name: 'Bob' },
  { name: 'Sam' },
  { name: 'Tim' },
  { name: 'Tom' },
  { name: 'Tom' },
  { name: 'Tom' },
  { name: 'Tom' },
  { name: 'Tomas' }
]
test>
```

$slice ოპერატორი

ეს ოპერატორი არის limit და skip ფუნქციების კომბინაცია. მაგრამ მათგან განსხვავებით **$slice** შეუძლია მასივებთან მუშაობა. მას აქვს ორი ფორმა:

$slice: limit

$slice: skip, limit

limit პარამეტრი მიუთითებს დასაბრუნებელი დოკუმენტების საერთო რაოდენობაზე, ხოლო skip მიუთითებს საიდან დაიწყოს ამორჩევა:

მაგალითად, ცალკეულ დოკუმენტში განსაზღვრულია languages მასივი ენების შესანახად, რომელზეც საუბრობს ადამიანი. იგი შეიძლება იყოს 1,2,3 ან მეტი. დავუშვათ, ადრე ჩვენ დავამატეთ შემდეგი ობიექტი:

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

და ჩვენ გვინდა დოკუმენტების გამოტანისას სიაში მოხვდეს ერთი ენა languages სიიდან და არა მთელი მასივი:

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

მოცემული მოთხოვნა დოკუმენტის ამოღებისას დატოვებს მხოლოდ პირველ ენას languages მასივიდან, მოცემულ შემთხვევაში english-ს:

```
test> db.users.find ({name: "Tom"}, {languages: {$slice : 1}})
[
  {
    _id: ObjectId("62e3c70079a0a7792a9de20c"),
    name: 'Tom',
    age: 32,
    languages: [ 'english' ]
  }
]
test>
```

საპირისპირო სიტუაცია: ჩვენ ასევე უნდა დავტოვოთ ერთი ელემენტი მასივში, მაგრამ არა თავიდან, არამედ ბოლოდან. ამ შემთხვევაში, პარამეტრს უნდა გადაეცეს უარყოფითი მნიშვნელობა:

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

ახლა მასივი შეიცავს "ესპანურს", რადგან ის ბოლოდან პირველია დამატებულ ელემენტში.

გამოვიყენოთ ორი პარამეტრი ერთდროულად:

```
db.users.find ({name: "Tom"}, {languages: {$slice : [-2, 1]}});
```

პირველი პარამეტრი ამბობს, რომ ელემენტების შერჩევის დაწყება უნდა მოხდეს ბოლოდან (როგორც უარყოფითი მნიშვნელობა), ანუ შერჩევა იწყება ბოლოდან მეორე

ელემენტიდან, ხოლო მეორე პარამეტრი მიუთითებს დაბრუნებული მასივის ელემენტების რაოდენობაზე. შედეგად, ენის მასივი შეიცავს "გერმანულს".

2.8. ინდექსები

მცირე კოლექციებში დოკუმენტების ძიებისას რაიმე განსაკუთრებული პრობლემა არ შეგვექმნება. თუმცა, როდესაც კოლექციები შეიცავს მილიონობით დოკუმენტს და ჩვენ გვჭირდება არჩევანის გაკეთება გარკვეული ველის მიხედვით, მაშინ საჭირო მონაცემების მოძიებას შეიძლება გარკვეული დრო დასჭირდეს, რაც შეიძლება კრიტიკული აღმოჩნდეს ჩვენი ამოცანისთვის. ამ შემთხვევაში ინდექსები დაგვეხმარება.

ინდექსები საშუალებას გაძლევთ მოაწესრიგოთ მონაცემები კონკრეტულ ველზე, რაც შემდგომ დააჩქარებს ძიებას. მაგალითად, თუ ჩვენ ვართ ჩვენს აპლიკაციაში ან ამოცანაში, როგორც წესი, ჩვენ ვეძებთ ველს NAME, მაშინ შეიძლება კოლექციის ინდექსირება ამ ველის მიხედვით:

ინდექსის შექმნა

ინდექსის შესაქმნელად გამოიყენება createIndex() ფუნქცია, რომელსაც გადაეცემა ობიექტი იმ ველების მითითებით, რომლებისთვისაც იქმნება ინდექსი. მაგალითად, “name” ველზე ინდექსის შექმნა:

```
db.users.createIndex({"name" : 1})
```

ინდექსის შექმნისას კონსოლი გვაბრუნებს ინდექსის სახელს:

```
test> db.users.createIndex({"name" : 1})
name_1
test>
```

ანუ ზემოთ მოცემულ მაგალითში შეიქმნა ინდექსი სახელწოდებით "name_1" სახელის ველზე. MongoDB საშუალებას გაძლევთ დააყენოთ 64-მდე ინდექსი თითო კოლექციაზე.

მრავალი ინდექსის შესაქმნელად გამოიყენება createIndexes() ფუნქცია - მას გადაეცემა ობიექტების მასივი, რომელიც ადგენს ველებს ინდექსებისთვის:

```
db.users.createIndexes([{"name" : 1}, {"age": 1}])
```

მოცემულ შემთხვევაში იქმნება 2 ინდექსი - ერთი name ველისთვის, ხოლო მეორე - age ველისთვის.

ინდექსის წაშლა

ინდექსების წასაშლელად გამოიყენება dropIndex() ფუნქცია, რომელსაც გადაეცემა ინდექსის სახელი. მაგალითად, წავშალოთ ზემოთ განსაზღვრული ინდექსი „name_1“:

```
db.users.dropIndex("name_1")
```

ინდექსების აწყობა

თუ ჩვენ უბრალოდ განვსაზღვრავთ ინდექსს კოლექციისთვის, მაგალითად, db.users.createIndex({"name" : 1}), მაშინ შევძლებთ კოლექციაზე დოკუმენტის დამატებას name გასაღების ერთნაირი მნიშვნელობით. მაგრამ ჩვენ თუ გვინდა კოლექციაში არ განმეორდეს დოკუმენტი, რომლის ველებს აქვა ერთიდაიგივე მნიშვნელობა, მაშინ უნდა აღვმართოთ ალამი unique:

```
db.users.createIndex({"name" : 1}, {"unique" : true})
```

ამ დროს კოლექციაში არ იქნება დოკუმენტი, რომელსაც განსაზღვრული ველისთვის აქვს ერთნაირი მნიშვნელობა. თუ ამას შევეცდებით, მივიღებთ შეცდომას.

აქ არის თავისი სპეციფიკა. დოკუმენტს შეიძლება არ ჰქონდეს name გასაღები, ამ შემთხვევაში დასამატებელი დოკუმენტისთვის ავტომატურად შეიქმნება name გასაღები null მნიშვნელობით. ამიტომ მეორე დოკუმენტის დამატებისას, სადაც არ არის განსაზღვრული name გასაღები, ამუშავდება გამონაკლისი, რადგანაც name გასაღები null მნიშვნელობით უკვე არის კოლექციაში.

ასევე შეიძლება შიქმნას ერთი ინდექსი ერთდროულად ორი ველისთვის:

```
db.users.createIndex({"name" : 1, "age" : 1})
```

მხოლოდ, ამ შემთხვევაში ყველა დასამატებელ დოკუმენტს უნდა ჰქონდეს უნიკალური მნიშვნელობა ორივე ველისთვის.

ამავდროულად, ველის მნიშვნელობა, რომელზედაც ხდება ინდექსაცია, არ უნდა იყოს 1024 ბაიტზე მეტი.

ინდექსების მართვა

ყველა ინდექსი ინახება სისტემურ კოლექციაში - indexes. მასზე მიმართვისათვის შეიძლება გამოვიყენოთ getIndexes ფუნქცია, მაგალითად, გამოვიტანოთ მთელი ინფორმაცია ინდექსებზე კონკრეტული კოლექციისთვის:

```
db.users.getIndexes()
```

ეს ბრძანება გამოიტანს შემდეგის მსგავსს:

```
test> db.users.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { name: 1 }, name: 'name_1' }
]
test>
```

როგორც ვხედავთ, users კოლექციისთვის (სატესტო ბაზიდან) არის განსაზღვრული 2 ინდექსი: id და სახელი. key ველი გამოიყენება მაქსიმალური და მინიმალური მნიშვნელობების მოსაძებნად, სხვადასხვა ოპერაციებისთვის, სადაც უნდა იქნას გამოყენებული ეს ინდექსი. name ველი გამოიყენება იდენტიფიკატორის სახით ადმინისტრირების ოპერაციებისთვის, მაგალითად, როგორიცაა ინდექსის წაშლა.

2.9. აგრეგატული ფუნქციები

ელემენტების რაოდენობა კოლექციაში

countDocuments() ფუნქციის გამოყენებით, შეგიძლიათ მიიღოთ დოკუმენტების მთლიანი რაოდენობა კოლექციაში:

db.users.countDocuments()

თუ ჩვენ უნდა გავიგოთ არა კოლექციაში არსებული დოკუმენტების მთლიანი რაოდენობა, არამედ მხოლოდ დოკუმენტების რაოდენობა კონკრეტულ შერჩევაში, მაშინ შეგვიძლია გამოვიყენოთ count () ფუნქცია. მაგალითად, დავთვალოთ დოკუმენტების რაოდენობა, რომლებსაც აქვთ name=Tom:

db.users.find({name: "Tom"}).count()

გარდა ამისა, ჩვენ შეგვიძლია შევქმნათ ფუნქციების ჯაჭვები დათვლის პირობების დასაკონკრეტებლად:

db.users.find({name: "Tom"}).skip(2).count(true)

უნდა აღინიშნოს, რომ დუმილით count ფუნქცია არ გამოიყენება limit და skip ფუნქციებით. რათა ისინი გამოვიყენოთ, როგორც ზემოთ მაგალითში, count ფუნქციას უნდა გადავცეთ ბულის მნიშვნელობა - true.

distinct ფუნქცია

კოლექცია შეიძლება შეიცავდეს დოკუმენტებს, რომლებიც შეიცავენ იგივე მნიშვნელობებს ერთი ან მეტი ველისთვის. მაგალითად, რამდენიმე დოკუმენტი განსაზღვრავს სახელს: " Tom" და ჩვენ უნდა ვიპოვოთ მხოლოდ უნიკალური განსხვავებული მნიშვნელობები დოკუმენტის ერთ-ერთი ველისთვის. ამისათვის ჩვენ შეგვიძლია გამოვიყენოთ distinct ფუნქცია. მაგალითად, დავუშვათ, რომ მონაცემთა ბაზას დაემატა შემდეგი დოკუმენტები:

```
db.users.insertOne({"name": "Tom", "age": 38, languages: ["english", "spanish"]})
db.users.insertOne({"name": "Bob", "age": 41, languages: ["english", "french"]})
db.users.insertOne({"name": "Sam", "age": 28, languages: ["english"]})
db.users.insertOne({"name": "Tom", "age": 22, languages: ["english", "german"]})
```

გამოვიტანოთ ყველა უნიკალური მნიშვნელობა name ველისთვის:

```
test> db.users.distinct("name")
[ 'Bob', 'Sam', 'Tom' ]
test>
```

min და max ფუნქციები

min ფუნქცია განსაზღვრავს მინიმალურ მნიშვნელობას კონკრეტული ველისთვის, რომელიც უნდა იყოს შერჩეული. ამ შემთხვევაში, ამ ფუნქციებს შეუძლიათ გამოიყენონ მხოლოდ ის ველები, რომლებისთვისაც დაყენებულია

ინდექსები. მაგალითად, ავიღოთ ზემოთ განსაზღვრული db.users კოლექცია და განვსაზღვროთ მასში ინდექსი age ველისთვის:

ფუნქციის შესრულებისას ასევე უნდა გამოიყენოთ hint() ფუნქცია, რომელსაც გადაეცემა ინდექსი. მაგალითად, ავირჩიოთ ყველა დოკუმენტი, რომლებშიც age ველი 30 წელზე მეტია:

```
test> db.users.find().min({age:30}).hint({age:1})
[
  {
    _id: ObjectId("62e3d63a79a0a7792a9de210"),
    name: 'Tom',
    age: 38,
    languages: [ 'english', 'spanish' ]
  },
  {
    _id: ObjectId("62e3d63a79a0a7792a9de211"),
    name: 'Bob',
    age: 41,
    languages: [ 'english', 'french' ]
  }
]
test>
```

max() ფუნქცია მუშაობს ანალოგიურად, რომელიც ადგენს მაქსიმალურ მნიშვნელობას. მაგალითად, ავირჩიოთ დოკუმენტები, სადაც age 30 წელზე ნაკლებია:

```
test> db.users.find().max({age:30}).hint({age:1})
[
  {
    _id: ObjectId("62e3d64079a0a7792a9de213"),
    name: 'Tom',
    age: 22,
    languages: [ 'english', 'german' ]
  },
  {
    _id: ObjectId("62e3d63a79a0a7792a9de212"),
    name: 'Sam',
    age: 28,
    languages: [ 'english' ]
  }
]
test>
```

2.10. ამორჩევის ოპერატორები

პირობის ოპერატორები

- **$eq** (ტოლია)
- **$ne** (არ უდრის)
- **$gt** (მეტი ვიდრე)
- **$lt** (ნაკლები ვიდრე)
- **$gte** (მეტი ან ტოლი)
- **$lte** (ნაკლები ან ტოლი)
- **$in** განსაზღვრავს მნიშვნელობების მასივს, რომელთაგან ერთს უნდა ჰქონდეს დოკუმენტის ველი
- **$nin** განსაზღვრავს მნიშვნელობების მასივს, რომელიც არ უნდა ჰქონდეს დოკუმენტის ველს

მაგალითად, მოდი ვიპოვოთ ყველა დოკუმენტი, რომელთაგან age გასაღების მნიშვნელობა 30 წელზე ნაკლებია:

```
db.users.find ({age: {$lt : 30}})
```

სხვა შედარების ოპერატორების გამოყენება მსგავსი იქნება. მაგალითად, იგივე გასაღები, მხოლოდ 30-ზე მეტი:

```
db.users.find ({age: {$gt : 30}})
```

გაითვალისწინეთ, რომ შედარება აქ არის მთელ რიცხვებზე და არა სტრიქონებზე. თუ ასაკის გასაღები წარმოადგენს სტრიქონების მნიშვნელობებს, მაშინ შედარება უნდა მოხდეს სტრიქონებზე შესაბამისად:

db.users.find ({age: {$gt : "30"}}), შედეგი იგივე იქნება

მაგრამ თუ გვაქვს სიტუაცია, როდესაც უნდა ვიპოვოთ ყველა ობიექტი, რომლის ასაკობრივი ველის მნიშვნელობა 30-ზე მეტია, მაგრამ 50-ზე ნაკლები. ამ შემთხვევაში, ჩვენ შეგვიძლია გავაერთიანოთ ორი ოპერატორი:

```
db.users.find ({age: {$gt : 30, $lt: 50}})
```

ვიპოვოთ მომხმარებლები, რომელთა ასაკი ტოლის 22-ის:

```
db.users.find ({age: {$eq : 22}})
```

ეს ანალოგიურია შემდეგი მოთხოვნის:

```
db.users.find ({age: 22})
```

ვიპოვოთ მომხმარებლები, რომელთა ასაკი არ უდრის 22-ს:

```
db.users.find ({age: {$ne : 22}})
```

$in ოპერატორი განსაზღვრავს შესაძლო გამონათქვამების მასივს და ეძებს იმ გასაღებებს, რომელთა მნიშვნელობაც არის მასივში:

```
db.users.find ({age: {$in : [22, 32]}})
```

$nin ოპერატორი მუშაობს საპირისპიროდ - ის განსაზღვრავს შესაძლო გამონათქვამების მასივს და ეძებს იმ გასაღებებს, რომელთა მნიშვნელობა არ არის ამ მასივში:
db.users.find ({age: {$nin : [22, 32]}})

2.11. ლოგიკური ოპერატორები

$or: აკავშირებს ორ პირობას და დოკუმენტი უნდა შეესაბამებოდეს ერთ-ერთ ამ პირობას
$and: აკავშირებს ორ პირობას და დოკუმენტი უნდა შეესაბამებოდეს ორივე პირობას
$not: დოკუმენტი არ უნდა ემთხვეოდეს პირობას
$nor: აკავშირებს ორ პირობას და დოკუმენტი არ უნდა ემთხვეოდეს ორივე პირობას
მაგალითები:
$or ოპერატორი წარმოადგენს ლოგიკურ OR ოპერაციას და განსაზღვრავს გასაღები-მნიშვნელობის წყვილების ერთობლიობას, რომელიც უნდა ჰქონდეს დოკუმენტს. და თუ დოკუმენტს აქვს მინიმუმ ერთი ასეთი გასაღები-მნიშვნელობის წყვილი, მაშინ ის ემთხვევა მოცემულ მოთხოვნას და ამოღებულია მონაცემთა ბაზიდან:
db.users.find ({$or : [{name: "Tom"}, {age: 22}]})
ეს გამოთქმა დააბრუნებს ყველა დოკუმენტს, რომელსაც აქვს სახელი=ტომი ან ასაკი=22.
კიდევ ერთი მაგალითი დააბრუნებს ყველა დოკუმენტს, რომლებშიც სახელი=ტომი და ასაკი არის 22 წლის ან აქვს "გერმანული" ენებს შორის:
db.users.find ({name: "Tom", $or : [{age: 22}, {languages: "german"}]})
პირობითი ოპერატორები შეიძლება გამოყენებულ იქნას or ქვეგამოსახულებაში:
db.users.find ({$or : [{name: "Tom"}, {age: {$gte:30}}]})
ამ შემთხვევაში, ჩვენ ვირჩევთ ყველა დოკუმენტს, სადაც name="Tom" ან ასაკის ველს აქვს 30 ან მეტი მნიშვნელობა.

$and ოპერატორი

$and ოპერატორი წარმოადგენს ლოგიკურ AND ოპერაციას (ლოგიკური გამრავლება) და განსაზღვრავს კრიტერიუმების ერთობლიობას, რომელსაც დოკუმენტი უნდა აკმაყოფილებდეს. $or ოპერატორისგან განსხვავებით, დოკუმენტი უნდა შეესაბამებოდეს ყველა მითითებულ კრიტერიუმს. მაგალითად:საზღვრავს კრიტერიუმების ერთობლიობას, რომელსაც დოკუმენტი უნდა აკმაყოფილებდეს. $or ოპერატორისგან განსხვავებით, დოკუმენტი უნდა შეესაბამებოდეს ყველა მითითებულ კრიტერიუმს. მაგალითად:

```
db.users.find ({$and : [{name: "Tom"}, {age: 22}]})
```

აქ შერჩეულ დოკუმენტებს უნდა ჰქონდეს სახელი ტომი და 22 წლის ასაკი - ორივე ეს ნიშანი.

2.12. ძებნა მასივების მიხედვით

რიგი ოპერატორები შექმნილია მასივებთან მუშაობისთვის:
$all: განსაზღვრავს მნიშვნელობების ერთობლიობას, რომელიც უნდა იყოს წარმოდგენილი მასივში;
$size: განსაზღვრავს ელემენტების რაოდენობას, რომლებიც უნდა იყოს მასივში;
$elemMatch: განსაზღვრავს პირობას, რომელთაც უნდა შეესაბამებოდეს ელემენტები მასივში.

$all

$all ოპერატორი განსაზღვრავს შესაძლო გამოსახულებების მასივს და მოითხოვს რომ დოკუმენტებს ჰქონდეს გამოსახულებების მთელი განსაზღვრული ნაკრები. შესაბამისად, იგი გამოიყენება მასივში საძიებლად. მაგალითად, დოკუმენტებში დოკუმენტებში არის languages მასივი, რომელიც ინახავს უცხო ენებს, რომელზეც საუბრობს მომხმარბელი. რათა ვიპოვოთ ყველა ადამიანი, რომელიც საუბრობს ერთდროულად ინგლისურად და ფრანგულად, ჩვენ შეგვიძლია გამოვიყენოთ შემდეგი გამოსახულება:

```
db.users.find ({languages: {$all : ["english", "french"]}})
```

$elemMatch ოპერატორი

$elemMatch ოპერატორი საშუალებას იძლევა ავარჩიოთ დოკუმენტები, რომელშიც მასივი შეიცავს ელემენტებს რომლებიც შეესაბამება გარკვეულ პირობას. მაგალითად, მონაცემთა ბაზაში არის კოლექცია, რომელიც შეიცავს მომხმარებელთა შეფასებებს განსაზღვრულ კურსებზე. დავამატოთ რამდენიმე დოკუმენტი:
db.grades.insertMany([{student: "Tom", courses:[{name: "Java", grade: 5}, {name: "MongoDB", grade: 4}]},
{student: "Alice", courses:[{name: "C++", grade: 3}, {name: "MongoDB", grade: 5}]}])

თითოეულ დოკუმენტს აქვს courses მასივი, რომელიც, თავის მხრივ, შედგება ჩადგმული დოკუმენტებისგან.
ახლა მოდით ვიპოვოთ სტუდენტები, რომლებსაც აქვთ 4-ზე მაღალი შეფასება MongoDB კურსში:

```
db.grades.find({courses: {$elemMatch: {name: "MongoDB", grade: {$gt: 4}}}})
```

$size ოპერატორი

$size ოპერატორი გამოიყენება დოკუმენტების საპოვნელად, რომლებშიც მასივები შეიცავს **$size** მნიშვნელობის ტოლი ელემენტების რაოდენობას. მაგალითად, მოდით ამოვიღოთ ყველა დოკუმენტი, რომელსაც აქვს ორი ელემენტი **laguages** მასივში:
db.users.find ({languages: {$size:2}})

ასეთი მოთხოვნა შეესაბამება, მაგალითად, შემდეგ დოკუმენტს:
{"name": "Tom", "age": 32, languages: ["english", "german"]}

$exists ოპერატორი

$exists ოპერატორი საშუალებას გაძლევთ ამოიღოთ მხოლოდ ის დოკუმენტები, რომლებშიც კონკრეტული გასაღები არის ან არ არსებობს. მაგალითად, დავაბრუნოთ ყველა დოკუმენტი, რომელშიც არის company გასაღები:
db.users.find ({company: {$exists:true}})
თუ $exists ოპერატორს პარამეტრად მივუთითებთ false მნიშვნელობას, მაშინ მოთხოვნა დააბრუნებს მხოლოდ იმ დოკუმენტებს, რომელშიც არ არის განსაზღვრული company გასაღები:

$type ოპერატორი

$type ოპერატორი ამოიღებს მხოლოდ იმ დოკუმენტებს, რომელშიც განსაზღვრულ გასაღებს აქვს განსაზღვრული ტიპის მნიშვნელობა, მაგალითად, სტრიქონი ან რიცხვი:
db.users.find ({age: {$type:"string"}})
db.users.find ({age: {$type:"number"}})

$regex ოპერატორი

$regex ოპერატორი იძლევა რეგულარულ გამოსახულებას, რომელსაც უნდა შეესაბამებოდეს ველის მნიშვნელობა. მაგალითად, დავუშვათ name ველს აუცილებლად აქვს "b" სიმბოლო:
db.users.find ({name: {$regex:"b"}})
$regex მიიღებს არა უბრალოდ სტრიქონებს, არამედ სახელდობრ რეგულარულ გამოსახულებას, მაგალითად: name: {$regex:"om$"} - რაც ნიშნავს, რომ name მნიშვნელობა მთავრდება om-ით.

2.13. მონაცემთა განახლება

მონაცემთა ბაზის მართვის სხვა სისტემების მსგავსად, MongoDB უზრუნველყოფს მონაცემთა განახლების შესაძლებლობას. ამისათვის რამდენიმე ფუნქციაა ხელმისაწვდომი.

replaceOne

თუ ჩვენ გვჭირდება ერთი დოკუმენტის მთლიანად ჩანაცვლება მეორით, შეიძლება გამოყენებულ იქნას replaceOne ფუნქცია:
db.collection.replaceOne(filter, update, options)
filter: იღებს მოთხოვნას დოკუმენტის ამორჩევაზე, რომელიც უნდა განახლდეს;
update: წარმოადგენს ახალ დოკუმენტს, რომელიც განახლებისას ჩაანაცვლებს ძველს;

options: განსაზღვრავს დამატებით პარამეტრებს დოკუმენტების განახლებისას, რომელთაგან მთავარია upsert პარამეტრი.
თუ upsert პარამეტრს აქვს true მნიშვნელობ, mongodb განაახლებს დოკუმენტს თუ იპოვის, ხოლო შექმნის ახალს, თუ ასეთი დოკუმენტი არ არის. თუ მას აქვს false მნიშვნელობა, მაშინ mongodb არ შექმნის ახალ დოკუმენტს, მაშინაც კი თუ ამორჩევაზე მოთხოვნა ვერ იპოვის ვერცერთ დოკუმენტს.
მაგალითად:

```
db.users.replaceOne({name: "Bob"}, {name: "Bob", age: 25})
```

მოცემულ შემთხვევაში ვპოულობთ დოკუმენტ, რომელშიც name ="Bob" და შევცვლით მას დოკუმენტით: {name: "Bob", age: 25}
შესრულების შემდეგ კონსოლი დააბრუნებს განახლების შედეგებს:

```
test> db.users.replaceOne({name: "Bob"}, {name: "Bob", age: 25})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

მიღებულ შედეგში matchedCount პარამეტრი უჩვენებს დოკუმენტების რაოდენობას, რომელიც შეესაბამება მოთხოვნას. modifiedCount უჩვენებს შეცვლილი დოკუმენტების როდენობას. ანუ მოცემულ შემთხვევაში მოთხოვნას შეესაბამება 1 დოკუმენტი და მოხდა მისი შეცვლა.

updateOne и updateMany

ხშირად არ არის საჭირო მთელი დოკუმენტის განახლება, არამედ საჭიროა მხოლოდ მისი ერთი ან რამდენიმე თვისების განახლება. ამისათვის გამოიყენება updateOne() (მხოლოდ ერზტი დოკუმნტის განახლება) ფუნქცია და updateMany (მრავალი დოკუმენტის განახლება) ფუნქცია.

ცალკეული ველების განსაახლებლად ამ ფუნქციებში გამოიყენება $set ოპერატორი. თუ დოკუმენტი არ შიცავს განსაახლებელ ველს, მაშინ იგი შეიქმნება.

```
db.users.updateOne({name : "Tom", age: 22}, {$set: {age : 28}})
```

ვეძებთ დოკუმენტს name="Tom" და age=22 და შევცვლით ასაკს - 28-ით.
თუ საჭიროა მთელი დოკუმენტის განახლება, რაიმე კრიტერიუმის შესაბამისად, მაშინ გამოიყენება ფუნქცია:
updateMany():

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

თუ დოკუმენტში არ არის განსაახლებელი ველი, მაშინ იგი ემატება:

```
db.users.updateOne({name : "Tom", age: 28}, {$set: {salary : 300}})
```

თუ საჭიროა რამდენიმე ველის მნიშვნელობების განახლება, მაშინ ისინი გადაეცემა $set ოპერატორს, გამოყოფილი მძიმეებით:
db.users.updateOne({name : "Tom"}, {$set: {name: "Tomas", age : 25}})
იმისათვის, რომ უბრალოდ გავზარდოთ რიცხვითი ველის მნიშვნელობა გარკვეული რაოდენობის ერთეულით, გამოიყენეთ $inc ოპერატორი. თუ დოკუმენტი არ შეიცავს განსაახლებელ ველს, მაშინ იგი შეიქმნება. ეს ოპერატორი გამოიყენება მხოლოდ ციფრული მნიშვნელობებისათვის.
db.users.updateOne({name : "Tom"}, {$inc: {age:2}})

ველის წაშლა

ცალკეული გასაღების წასაშლელად გამოიყენება $unset ოპერატორი:
db.users.updateOne({name : "Tom"}, {$unset: {salary: 1}})
თუ ასეთი გასაღები არ არსებობს დოკუმენტში, მაშინ ოპერატორს არანაირი გავლენა არ აქვს. ასევე შეგვიძლია წავშალოთ რამდენიმე ველი ერთდროულად:
db.users.updateOne({name : "Tom"}, {$unset: {salary: 1, age: 1}})

მასივების განახლება
$push ოპერატორი

$push ოპერატორი საშუალებას იძლევა დავამატოთ კიდევ ერთი მნიშვნელობა არსებულს. მაგალითად, თუ გასაღების მნიშვნელობა არის მასივი:
db.users.updateOne({name : "Tom"}, {$push: {languages: "georgian"}})
ზემოთ გამოყენებული იყო updateOne ფუნქცია, ეს ოპერატორი ასევე გამოიყენება updateMany ფუნქციაშიც:
db.users.updateMany({name : "Tom"}, {$push: {languages: " georgian"}})
თუ გასაღები, რომელსაც გვინდა დავამატოთ მნიშვნელობა, არ წარმოადგენს მასივს, მაშინ მივიღებთ შეცდომას `Cannot apply` $push/$pushAll modifier to non-array.

$each ოპერატორის გამოყენებით ერთდროულად შეგვიძლია რამდენიმე მნიშვნელობის დამატება:

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

ჩასმას უზრუნველყოფს ოპერატორების წყვილი:

$position ოპერატორი მიუთითებს მასივში ადგილს, სადაც უნდა ჩაისვას ელემენტი, ხოლო $slice ოპერატორი მიუთითებს რამდენი ელემენტი დარჩეს მასივში ჩასმის შემდეგ:

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

მოცემულ მაგალითში ["german", "spanish", "italian"] დაემატება languages მასივს 1-ლი ინდექსიდან, მასივში დარცება 5 ელემენტი.

$addToSet ოპერატორი

$addToSet ოპერატორი $push ოპერატორის მსგავსად დაამატებს მასივში ობიექტებს. განსხვავება არის იმაში, რომ $addToSet დაამატებს მონაცემებს, თუ ისინი ჯერ კიდევ არ არიან მასივში ($push-ით დამატებისას მონაცემები დუბლირდებიან, თუ ემატება ელემენტები, რომლებიც უკვე არის მასივში).

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

მასივიდან ელემენტის წაშლა

$pop მასივიდან ელემენტის წაშლის საშუალებას იძლევა:

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

languages გასარებისთვის თუ მივუთითებთ მნიშვნელობა 1-ს, ჩვენ წავშლით ბოლოდან პირველ ელემენტს. დასაწყისიდან მასივის პირველი ელემენტის წასაშლელად უნდა გადავცეთ უარყოფითი მნიშვნელობა;

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

$pull წაშლის მასივში ელემენტის ცალკეულ შესვლას. მაგალითად, $push ოპერატორით ჩვენ შეგვიძლია დავამატოთ მასივში ერთიდაიგივე მნიშვნელობა რამდენჯერმე. ხოლო შემდეგ $pull გამოყენებით წავშალოთ იგი:

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

თუ გვინდა წავშალოთ ერთდროულად რამდენიმე მნიშვნელობა, მაშინ უნდ გამოვიყენოთ $pullAll ოპერატორი:

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

მონაცემების წაშლა

დოკუმენტების წასაშლელად, MongoDB-ში გათვალისწინებულია ფუნქციები deleteOne() - წაშლის ერთ დოკუმენტს და deleteMany() - გაძლევთ საშუალებას წაშალოთ მრავალი დოკუმენტი. პარამეტრის სახით ამ ფუნქციებში გადაეცემა წასაშლელი დოკუმენტების ფილტრი.

წავშალოთ დოკუმენტი, რომელშიც name="Tom":

```
db.users.deleteOne({name : "Tom"})
```

შედეგად, პირველი ნაპოვნი დოკუმენტი name=Tom წაიშლება. წაშლის შემდეგ, კონსოლი აჩვენებს ობიექტს, რომელშიც deletedCount პარამეტრი მიუთითებს წაშლილი დოკუმენტების რაოდენობას:

```
test> db.users.deleteOne({name : "Tom"})
{ acknowledged: true, deletedCount: 1 }
```

ყველა დოკუმენტის წასაშლელად, რომლებიც შეესაბამება ფილტრს, გამოიყენება deleteMany() ფუნქცია:

```
db.users.deleteMany({name : "Tom"})
```

ამავდროულად, როგორც find შემთხვევაში, ჩვენ შეგვიძლია დავსვათ ამორჩევის პირობა სხვადასხვა ხერხით წასაშლელად (რეგულარული გამოსახულების სახით, პირობითი კონსტრუქციის სახით და ა.შ.):

```
db.users.deleteOne({name : /^T\w+/i})
db.users.deleteOne({age: {$lt : 30}})
```

კოლექციიდან ყველა დოკუმენტის ერთდროულად წასაშლელად, უნდა დავტოვოთ მოთხოვნის პარამეტრი ცარიელი:

```
db.users.deleteMany({})
```

კოლექციის წაშლა მონაცემთა ბაზიდან

ჩვენ შეგვიძლია წავშალოთ არა მხოლოდ დოკუმენტები, არამედ კოლექციები და მონაცემთა ბაზები. drop ფუნქცია გამოიყენება კოლექციების წასაშლელად:

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

თუ კოლექციის წაშლა წარმატებით დასრულდა, მაშინ კონსოლზე გამოჩნდება:

```
true
```

მთელი მონაცემთა ბაზის წასაშლელად გამოიყენება dropDatabase() ფუნქცია:

```
db.dropDatabase()
```

2.14. კოლექციების მართვა

კოლექციის ცხადი სახით შექმნა

წინა თემებში კოლექცია არაცხადად ავტომატურად იქმნებოდა, როდესაც მას დაემატებოდა პირველი მონაცემები. მაგრამ ჩვენ ასევე შეგვიძლია იგი შევქმნათ ცხადად db.createCollection (სახელი, პარამეტრები) მეთოდის გამოყენებით, სადაც სახელი არის კოლექციის სახელი და options არასავალდებულო ობიექტია ინიციალიზაციის დამატებითი პარამეტრებით.

მაგალითად:

```
db.createCollection("accounts")
{"ok" : 1}
```

ასეთი სახით შეიქმნება accounts კოლექცია.

კოლექციაზე სახელის გადარქმევა

მუშაობისას შეიძლება საჭირო გახდეს კოლექციის სახელის შეცვლა. მაგალითად, თუ პირველად მონაცემთა დამატებისას მის სახელში იყო შეცდომა, და იმისათვის, რომ არ წავშალოთ და ხელახლა არ შევქმნათ კოლექცია, უნდა გამოვიყენოთ renameCollection ფუნქცია:

```
db.users.renameCollection("ახალი_სახელი")
```

თუ ყველაფერმა ჩაიარა წარმატებით, კონსოლზე გამოვა შეტყობინება:

```
{"ok" : 1}
```

კოლექციის შეზღუდვა

როდესაც ჩვენ ვაგზავნით მოთხოვნას მონაცემთა ბაზაში ამორჩევისთვის, MongoDB გვიბრუნებს დოკუმენტებს იმ თანმიმდევრობით, როგორითაც ისინი დაემატა. თუმცა ასეთი მიმდევრობა ყოველთვის არ არის გარანტირებული, რადგან მონაცემები შეიძლება წაიშალოს, გადაადგილდეს, შეიცვალოს. ამიტომ MongoDB-ში არის შეზღუდული კოლექციის ცნება (capped collection). ასეთი კოლექცია გარანტიას იძლევა, რომ დოკუმენტები იქნებიან განლაგებული იმავე მიმდევრობით, როგორითაც ისინი დაემატა კოლექციას. შეზღუდულ კოლექციას აქვს ფიქსირებული ზომა და როდესაც კოლექციაში უკვე აღარ არის ადგილი, შედარებით ძველი დოკუმენტები წაიშლება, ხოლო ბოლოში დაემატება ახალი მონაცემები.

ჩვეულებრივი კოლექციებისგან განსხვავებით, შეზღუდული კოლექციები შეიძლება ცხადი სახით გამოვაცხადოთ. მაგალითად, შევქმნათ შეზღუდული კოლექცია სახელწოდებით profiles და დავაყენოთ მისი ზომა 9500 ბაიტი:

```
db.createCollection("profiles", {capped:true, size:9500})
```

კოლექციის წარმატებით შექმნის სემდეგ კონსოლზე გამოვა შეტყობინება:

```
{"ok":1}
```

ასევე შეიძლება შეიზღუდოს კოლექციაში არსებული დოკუმენტების რაოდენობა, მისი max პარამეტრში მითითებით:

```
> db.createCollection("profiles", {capped:true, size:9500, max: 150})
```

თუმცა, კოლექციის შექმნის ამ მეთოდით, უნდა გავითვალისწინოთ, რომ თუ კოლექციისთვის მთელი სივრცე სავსეა (მაგალითად, ჩვენს მიერ გამოყოფილი 9500 ბაიტი) და დოკუმენტების რაოდენობას ჯერ არ მიუღწევია მაქსიმუმს, ამ შემთხვევაში 150, მაშინ ამ შემთხვევაში ახალი დოკუმენტის დამატებისას წაიშლება უძველესი დოკუმენტი და მის ადგილას ჩასმული იქნება ახალი დოკუმენტი.

ასეთ კოლექციებში დოკუმენტების განახლებისას უნდა გავითვალისწინოთ, რომ დოკუმენტები ზომაში არ უნდა გაიზარდოს, წინააღმდეგ შემთხვევაში განახლება ვერ მოხერხდება.

ასევე, არ შეიძლება წაიშალოს დოკუმენტები ასეთი კოლექციებიდან, მხოლოდ შეიძლება წაიშალოს მთელი კოლექცია.

ქვეკოლექციები

კოლექციებში მონაცემთა ორგანიზების გასამარტივებლად, ჩვენ შეგვიძლია გამოვიყენოთ ქვეკოლექციები. მაგალითად, users კოლექციის მონაცემები შეიძლება გავმიჯნოთ „პროფილებად“ და „სააღრიცხვო მონაცემებად“ და ჩვენ შეგვიძლია გამოვიყენოთ db.users.profiles და db.users.accounts კოლექციის შექმნა. ამ დროს ისინი არაფრით არ იქნება დაკავშირებული users კოლექციასთან. ანუ ჯამში იქნება სამი სხვადასხვა კოლექცია. მონაცემთა შენახვის ლოგიკური ორგანიზების კუთხით ასეთი მიდგომა ზოგიერთ შემთხვევაში შეიძლება იყოს უფრო მარტივი.

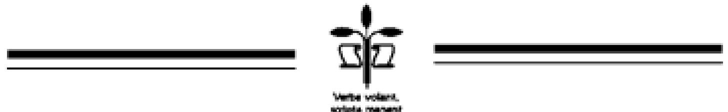
ლიტერატურა

1. ბადრი მეფარიშვილი, გულნარა ჯანელიძე „საინფორმაციო სისტემების აგება MS SQL Server-ის გამოყენებით", თბილისი, 2013წ.
2. Andreas Meier · Michael Kaufmann, SQL & NoSQL Databases Models, Languages, Consistency Options and Architectures https://doi.org/10.1007/978-3-658-24549-8, 2019.
3. Vatika Sharma1, Meenu Dave2, SQL and NoSQL Databases, International Journal of Advanced Research in Computer Science and Software Engineering.
4. Itzik Ben-Gan, T-SQL Fundamentals (Developer Reference) 3rd Edition, ISBN- 978-1509302000, August 3, 2016.
5. Walter Shields, SQL QuickStart Guide, November 18, 2019, 242 pages;
6. Pedro Lopes, Pam Lahoud, Learn T-SQL Querying, ISBN-978-1789348811, May 3, 2019;
7. Pramod Sadalage, Martin Fowler, NoSQL Distilled, ISBN-978-0321826626, August 2012;
8. Johannes Zollmann, NoSQLDatabases, 21 pages;
9. MongoDB. Sharding- mongodb. http: //www.mongodb.org/display/DOCS/Sharding, July 2012.
10. C.Strozzi. Nosql relational database management system. http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL/Home Page, July 2012.

კომპიუტერული უზრუნველყოფა: გ. სურგულაძე, გ. ჯანელიძე,
გ. დალაქიშვილი

(იბეჭდება ავტორის ხარჯით)

წარმოებას გადაეცა 10.06.2024. ხელმოწერილია დასაბეჭდად 10.07.2024.
ოფსეტური ქაღალდის ზომა 60X84 1/16. პირობითი ნაბეჭდი თაბახი 10,6.
ტირაჟი 50 ეგზ.



სტუ-ს „IT-კონსალტინგის სამეცნირო ცენტრი"
თბილისი, მ. კოსტავას 77