ივანე ჯავახიშვილის სახელობის თბილისის
სახელმწიფო უნივერსიტეტი

ავთანდილ აბრამიშვილი

ძებნის ინსტრუმენტების შერჩევა მონაცემთა ბაზებში

ინფორმაციული ტექნოლოგიები

ნაშრომი შესრულებულია ინფორმაციული ტექნოლოგიების
მაგისტრის აკადემიური ხარისხის მოსაპოვებლად

ხელმძღვანელი: მაია არჩუაძე

მაგდა ცინცაძე

თბილისი 2015

შინაარსი

ანოტაცია

ნაშრომში განხილულია ინფორმაციის ძებნის მეთოდები მონაცემთა ბაზებში და ის თანამედროვე ალგორითმები, რომლებიც გამოიყენება ძებნის სისტემებში. გაანალიზებულია ამ ალგორითმების სუსტი და ძლიერი მხარეები. წარმოდგენილია მონაცემთა ძებნის მეთოდები მონაცემთა ბაზებში და შესაბამისი მაგალითები თავისი ანალიზით. შემუშავებულია ძებნის „კომბინირებული“ მოდელი, რომელიც წარმოადგენს არსებული სამი მოდელის ეტაპობრივ გამოყენებას. მოდელში ძებნის პროცესი განიხილება, როგორც სამ დონიანი პროცესი, რომელსაც წინ უძღვის მოთხოვნის, იდენტიფიცირების და შედეგების რანჟირების ეტაპები.

Annotation

The paper deals with the information search methods in databases and the algorithms that are used in search engines. Analyzed the strengths and weaknesses of these algorithms. The database search methods for databases and appropriate examples and Analysis of this methods. Designed for " Combined " model, which represents the gradual application of the model. Model of the search process is treated as a three-level process is preceded identifying, request and the results of the ranking stages.

შესავალი

ოცდამეერთე საუკუნეში თანამედროვე ტექნოლოგიების განვითარების პარალელრად, სულ უფრო მეტად იზრდება შესანახი ინფორმაციის მოცულობა, რაც ბუნებრივია ართულებს ჩვენთვის საჭირო ინფორმაციის ადვილად და დროულად მოძიებას. უკანასკნელი ათწლეულის განმავლობაში ინფორმაციის ძიების ოპტიმიზაციის პროცესი განსაკუთრებით მნიშვნელოვანი გახდა, რადგან ასეთი დიდი ზომის ინფორმაციის დამუშავება არა ოპტიმალური მეთოდებით თითქმის შეუძლებელია და უზარმაზარ დროს და რესურსს მოითხოვს .ინფორმაციის ძებნისთვის ძირითადად გამოიყენება ვებ სივრცე, რომელიც მალე გახდა ყველაზე ხშირად გამოყენებადი სივრცე, დღეს ინტერნეტი არის ყველაზე კარგი საშუალება ყოველდღიური ინფორმაციის მოსაძიებლად.

მიუხედავად იმისა, რომ ინფორმაციული ტექნოლოგიების სამომხმარებლო სივრცეში მუდმივად ჩნდებიან სხვადასხვა საძიებო სისტემები ინფორმაციის ძებნის ეფექტური ალგორითმების შექმნა კვლავ აქტუალურია. განსაკუთრებით აქტუალურია ინფორმაციის ნაკადის მზარდ მოცულობაში მომხმარებლის მოთხოვნის შესაბამისი (რელევანტური) ინფორმაციის მიღების შესაძლებლობა. აქ იკვეთება ორი ძირითადი მიმართულება:

1. პასუხი მოთხოვნაზე იყოს რაც შეიძლება ზუსტი (სემანტიკური თვალსაზრისით)
2. მოთხოვნაზე პასუხი იყოს რაც შეიძლება სწრაფი.

ინფორმაციის ძებნა

ინფორმაციული ძებნა ზოგადად ეს არის არასრუქტურირებლი ინფორმაცის ძებნის

პროცესი, რომელიც აკმაყოფილებს ინფორმაციულ მოთხოვნებს, ამიტომ მიშვნელოვანია ინფრმაცილი ძებნის შედგი იყოს რაც შეიძლება ზუსტი. როგორც წესი ძებნის პროცესი შეიძლება წარმოვიდგინოთ ოთხი ეტაპისაგან შემდგარი პროცედურის სახით, რომლის სისრულეც დამოკიდებულია ყოველი ეტაპის წარმატებულობაზე:

1. მოთხოვნის ინფორმაციის განსაზღვრა და ინფორმაციული მოთხოვნის ფორმულირება;
2. ინფორმაციის მასივების მფლობელების (წყაროების)განსაზღვრა;
3. ინფორმაციის ამოკრება გამოვლენილი მასივებიდან;
4. მიღებული ინფორმაციის გაცნობა და ძებნის შედეგის შეფასება.

ინფორმაციული ძებნის კლასიკურ ამოცანად მიჩნეულია დოკუმენტების ძებნა, რომელიც აკმაყოფილებს მოთხოვნას, დოკუმენტების რაღაც კოლექციის ფარგლებში. ძებნის მეთოდებიდან გამოიყოფა ოთხი ძირითადი:

1. ძებნა მისამართით
2. სემანტიკური ძებნა
3. დოკუმენტური ძებნა
4. ფოტოგრაფული ძებნა.

მონაცემთა ბაზაში ძებნა განიხილება, როგორც ძებნა მისამართით. მაგრამ ტექსტების ძებნის დროს აუცილებელია სემანტიკური ძებნის ელემენტების დამატებაც.

ინფორმაციის ძებნის, როგორც პროცესის, აქტუალურობა ზრდის მოთხოვნას შეიქმნას ისეთი ტიპის პროგრამული სისტემები და ინსტრუმენტები (მათ შორის დანართებიც) რომლებიც მომხმარებლის მოთხოვნაზე დააბრუნებენ ადეკვატურ პასუხს რაც შეიძლება მცირე დროში.

თანამედროვე საძიებო სისტემები (Search engines) არის უდიდესი მონაცემთა მართვის სისტემები მსოფლიოში. ამ საძიებო სისტემებს შეუძლია დაამუშაოს 3 მილიარდზე მეტი დოკუმენტი, ერთი ბრძანებით 10 ტერაბაიტი მონაცემების დამუშავება შეუძლია, საძიებო სისტემას 150 მილიონი მოთხოვნის დღეში და რამოდენიმე ათასი მოთხოვნის 1 წამში დაამუშავება შეუძლია.

ეს რეტროსპექტივა დაფუძნებულია თითქმის ცხრა წლიან (1994-2003) ნამუშევარზე, Inktomi საძიებო სისტემაზე. ის აგრეთვე ასახავს ზოგიერთ ზოგად საკითხებს და მიდგომებს, რომლებიც გამოიყენება ძირითად საძიებო სისტემებში, კერძოდ Altavista, Infoseek და Google-ში, თუმცა მათი სპეციფიკაცია შეიძლება დიდად განსხვავებულიც იყოს. ასეთ სისტემებს უწევს ძებნა განსხვავებულ ენებზე რომლებიც დაახლოვებით 10 მილიონ სიტყვას მოიცავს, ამასთან ერთად გასათვალისწინებელია რამდენიმე სიტყვის გაერთიანებით მიღებული სიტყვები, მათი მნიშვნელობები (წონები, მაგ. სათაური სიტყვა) და სხვა. საძიებო სისტემა უნდა იყოს ხელმისაწვდომი ადვილად მოსახმარი და თანამედროვე ინფორმაციას მორგებული რათა გადაწყვიტოს მონაცემთა მართვის რთული საკითხები.

ძებნა მონაცემთა ბაზებში

ვებ სივრცეში ინფორმაციის მოძიებასთან ერთად არანაკლებ მნიშვნელოვანია ძებნის ისეთი მოდელების შემუშავება, რომელიც საშუალებას მოგვცემს დიდ

მონაცემთა ბაზებში საჭირო (რელევანტური) ინფორმაციის მოძიებას, ნაკლები რესურსებით და ოპტიმალურ დროში.

მონაცემთა ბაზის მართვის სისტემაში ძებნის უზრუნველყოფის სერვისი ერთ-ერთ მნიშვნელოვანი პრინციპია, რომლებზეც დამოკიდებულია ბაზის ძირითადი ფუნქციონალი: მომხმარებლისათვის მოთხოვნის შესაბამისი ინფორმაციის მიწოდება.

მონაცემთა ბაზისათვის საძიებო სისტემის შემუშავება ცალკე პროგრამის სახით თითქმის არ განიხილება. უმთავრესად ძებნის პროცედურა წარმოდგება, როგორც მონაცემთა ბაზის მართვის სისტემის დანართი. ამ ფუნქციონალური დანართის მუშაობის ეფექტურობა დამოკიდებულია ბაზის მოდელზე, ბაზის სტრუქტურაზე, სისტემურ რეალიზაციაზე და ასევე მის ფიზიკურ განლაგებაზე (განლაგებულია ერთ სერვერზე, თუ დაცილებული წვდომის საცავებში).

მონაცემთა ბაზების თეორია იცნობს იერარქიულ, ქსელურ, რელაციურ და ჰიბრიდულ მონაცემთა სტრუქტურებს, რომლებიც ხანგრძლივი დროის განმავლობაში ვითარდებოდა და კვლავაც ვითარდება. მრავალი მონაცემთა ბაზების მართვის სისტემა ისტორიას ჩაბარდა, ბევრი ახალიც გამოჩნდა, რომელთა შორის ლიდერებიცაა, კერძოდ Oracle, MsSQL Server, MySQL, Intebase, №სცცესს, dBase, Visual_FoxPro, Paradox და სხვ. ყველა ეს „გადარჩენილი“ თუ „ახალდაბადებული“ ბაზების მართვის სისტემები რელაციურ მოდელზეა აგებული (ან სუბრელაციურია).

მონაცემთა ბაზაში ძებნის ინსტრუმენტები

ბულის ძებნა

ხშირად დეველოპერებს და მონაცემთა ბაზის ადმინისტრატორებს სჭირდებათ ბაზაში მონაცემების და ობიექტების ძებნა.

ინფორმაციის ძებნას, მონაცემთა ბაზის მართვის სისტემებში, ერთ-ერთი ძირითადი ადგილი უკავია. არსებულ სისტემებში უფრო ხშირად გამოიყენება ძებნის ინსტრუმენტები, მაგ. ბულის ძიება, რომელსაც იყენებს მონაცემთა ბაზა ინდექსაციისთვის, რაც შემდეგ ამარტივებს ძიებას. ეს ალგორითმი ხშირად გამოყენებადია, რადგან მას გააჩნია დიდ ინფორმაციასთან გამკლავების საშუალება, ასევე იყენებენ ისეთი სტრუქტურირებული ინფორმაციის დასამუშავებლად, სადაც შესაძლო ვარიანტები მცირეა. განვიხილოთ მაგალითი: ვთქვათ, გვაქვს პერსონათა

ცხრილი, რომელშიც ერთ-ერთი ველი უჭირავს სქესს, ხოლო სტრიქონის უნიკალური იდენტიფიკატორია მაგ. გამოგონილი ID სვეტი (შესაძლებელია პირადი ნომერიც).

ID/სქესი	1	2	3	4	5	6	7	8	9
მდედრობითი	0	1	1	0	0	0	1	1	1
მამრობითი	1	0	0	1	1	1	0	0	0

აღნიშნულ ცხრილში მოვძებნოთ მდედრობითი სქესის პიროვნებები. ამისთვის ვირჩევთ ყველა იმ იდენტიფიკატორს, რომელსაც მდედრ. სტრიქონში აქვს ერთიანი, ასეთებია, 2,3,7,8,9. ამ იდენტიფიკატორების შესაბამისი სტრიქონები, ძირითად ცხრილში არის მდედრობითი სქესის. ეს ალგორითმი გამოიყენება სტრუქტურირებული მონაცემების ძებნის ინსტრუმენტებში, სემანტიკური მონაცემებისთვის კი არ იძლევა კარგ შედეგს. თუ გვაქვს ტექსტური ჩანაწერი, ამ ველში სემანტიკური ძიება ბულის ალგორითმით შედეგიანი არ იქნება. ბულის ძებნაში გამოიყენება ტექსტის სტრუქტურირებულ მონაცემებად წარმოდგენა.

მაგ: განვიხილოთ დოკუმენტები.

1. პირველ რიგში ეს დოკუმენტები უნდა იყოს დასათაურებული, ან დავასათაუროთ ჩვენით.
2. შევადგინოთ ცხრილი, სადაც სვეტების სახელები იქნება დოკუმენტების სათაურები,ხოლო სტრიქონები იქნება ამ დოკუმენტებში გამოყენებული განსხვავებული სიტყვები.

მაგ:

	დოკ 1	დოკ 2	დოკ 3	დოკ 4	დოკ 5
სიტყვა 1	1	0	0	0	0
სიტყვა 2	1	1	1	0	0
სიტყვა 3	0	1	0	1	1
სიტყვა 4	0	1	0	0	0

ეს ცხრილი შედგენილია ზემოთ მოყვანილი მაგალითის მიხედვით. თუ გვინდა მოვძებნოთ დოკუმენტები, რომელშიც გვხვდება სიტყვა 1 და სიტყვა 2 ერთად, უნდა ავიღოთ შესაბამის სვეტებში ჩაწერილი 1-იანები, ანუ, დოკ 1 არის მხოლოდ ასეთი დოკუმენტი. 10000 და 11100 =10000

მონაცემთა ბაზებში იგივე სტრუქტურა შეგვიძლია ავაგოთ. ბულის ძებნის ალგორითმი არ იძლევა ძებნის ჩატარების საშუალებას სემანტიკური თვალსაზრისით. იმისთვის რომ ბულის ძებნა გამოყენებული იყოს სრულყოფილად, უფრო მეტი აქცენტი უნდა გაკეთდეს, ტექსტური მონაცემის სემანტიკურად სტრუქტურირებულ მონაცემად წარმოდგენაზე.

ინდექსები

ცხრილში, რომელშიც ათასობით სტრიქონია, მონაცემების ძებნა დიდ დროს იკავებს. ძებნის დაჩქარების მიზნით გამოიყენება ინდექსები. მონაცემების ძებნის დაჩქარება მიიღწევა მათი ფიზიკური ან ლოგიკური მოწესრიგების ხარჯზე. ინდექსი წარმოადგენს მიმართვების ნაკრებს, რომლებიც მოწესრიგებულია გარკვეული სვეტის მნიშვნელობების მიხედვით. ასეთ სვეტს ინდექსირებული სვეტი ეწოდება.

ინდექსი მონაცემთა ბაზის დამოუკიდებელ ობიექტს წარმოადგენს. ის შეიძლება მოიცავდეს ცხრილის ერთ ან მეტ სვეტს.

ფიზიკურად ინდექსი წარმოადგენს ინდექსირებული სვეტის მონაცემების მოწესრიგებულ ნაკრებს ცხრილის სტრიქონების ფიზიკური განლაგების ადგილმდებარეობაზე მიმთითებლებით. როცა სრულდება მოთხოვნა, რომელიც ინდექსირებულ სვეტს მიმართავს, მაშინ სერვერი ავტომატურად აანალიზებს ინდექსს საჭირო მნიშვნელობის მოძებნის მიზნით. ამჟამად, შემუშავებულია ეფექტური მათემატიკური ალგორითმები მოწესრიგებულ მიმდევრობაში მონაცემების მოსაძებნად. ერთ-ერთი მათგანი, რომელიც წარმატებით გამოიყენება, არის „შუაზე გაყოფის'' მეთოდი.

ინდექსების გამოყენების დაგეგმვა

თუ ცხრილიდან მონაცემების ამორჩევა დიდ დროს იკავებს, ეს იმას ნიშნავს, რომ საჭიროა ინდექსის შექმნა. ბუნებრივია, ვიფიქროთ, რომ კარგი იქნება ინდექსის შექმნა თითოეული სვეტისთვის. მაგრამ, საქმე ის არის რომ, როცა სრულდება

სტრიქონების ცვლილება, მაშინ საკუთრივ მონაცემების განახლების გარდა, სრულდება ყველა ინდექსის განახლება. ეს კი, თავის მხრივ, გარკვეულ დროს იკავებს. სასურველია, რომ ინდექსების რაოდენობა არ აღემატებოდეს 4-5-ს. ამრიგად, ინდექსების გამოყენების მთავარი უპირატესობაა მონაცემების ამორჩევის მნიშვნელოვანი დაჩქარება, ხოლო მთავარი ნაკლია - მონაცემების განახლების (დამატების, შეცვლის და წაშლის) პროცესის შენელება.

ინდექსისათვის სვეტის ამორჩევისას უნდა გავაანალიზოთ, თუ მოთხოვნების რა ტიპები შესრულდება ყველაზე ხშირად და რომელი სვეტები არის საკვანძო. საკვანძოა ის სვეტები, რომლებიც განსაზღვრავენ მონაცემების ამორჩევის კრიტერიუმებს. არ უნდა მოვახდინოთ იმ სვეტების ინდექსირება, რომლებიც არ თამაშობენ არანაირ როლს მოთხოვნის შესრულების დროს. არ უნდა მოვახდინოთ ძალიან გრძელი სვეტების ინდექსირება, რომელთა სიგრძეა რამდენიმე ათეული სიმბოლო. საქმე ის არის, რომ ინდექსში შედის არა მარტო სტრიქონზე მიმართვა, არამედ თვით სვეტის შემცველობაც. უკიდურეს შემთხვევაში შეგვიძლია შევქმნათ გრძელი სვეტის შემოკლებული ვარიანტი, რომელშიც მოვათავსებთ პირველ ათ სიმბოლოს და მხოლოდ ამის შემდეგ შევასრულებთ ინდექსირებას. არსებობს ინდექსების სამი ტიპი: კლასტერული, არაკლასტერული და უნიკალური.

არაკლასტერული ინდექსი

არაკლასტერული ინდექსები ახდენენ მიმართვების ორგანიზებას შესაბამის სტრიქონებზე. ცხრილში საჭირო სტრიქონის იდენტიფიცირებისათვის არაკლასტერული ინდექსი ახდენს სპეციალური მიმთითებლების (rowlocator) ორგანიზებას. მიმთითებლები შეიცავენ:

- ინფორმაციას იმ ფაილის საიდენტიფიკაციო ნომრის (IDfile) შესახებ, რომელშიც სტრიქონი ინახება
- საძებნი სტრიქონის ნომერს შესაბამის გვერდზე
- სვეტის შემცველობას

ერთ ცხრილში შეიძლება განისაზღვროს 249-მდე არაკლასტერული ინდექსი. მაგრამ, ხშირ შემთხვევაში უნდა შემოვიფარგლოთ 4-5 ინდექსით.

კლასტერული ინდექსი

კლასტერული ინდექსის (clustered index) გამოყენების დროს ცხრილში მონაცემების ფიზიკური განლაგება გადაეწყობა ინდექსის სტრუქტურის შესაბამისად. ამ შემთხვევაში ცხრილის ლოგიკური სტრუქტურა ლექსიკონს უფრო წააგავს.

კლასტერული ინდექსები მნიშვნელოვნად ზრდიან მონაცემების ძებნის მწარმოებლურობას. არაკლასტერული ინდექსის გამოყენების შემთხვევაში, სერვერი თავდაპირველად მიმართავს ინდექსს, შემდეგ კი პოულობს საჭირო სტრიქონს ცხრილში. კლასტერული ინდექსის გამოყენების შემთხვევაში, მონაცემების მომდევნო პორცია განლაგდება უშუალოდ მოძებნილი მონაცემების შემდეგ. შედეგად, აღარ შესრულდება ინდექსთან მიმართვის და სტრიქონის ძებნის ზედმეტი ოპერაციები.

კლასტერულ ინდექსად უნდა ავირჩიოთ ყველაზე ხშირად გამოყენებადი სვეტები. კლასტერული ინდექსი შეიძლება შეიცავდეს რამდენიმე სვეტს, მაგრამ ასეთი სვეტების რაოდენობა შეძლებისდაგვარად მცირე უნდა იყოს.

კლასტერული ინდექსი არ უნდა გამოვიყენოთ ხშირად ცვალებადი სვეტებისათვის, რადგან სერვერმა უნდა შეასრულოს ყველა მონაცემის ფიზიკური გადაადგლება ცხრილში, რათა ისინი იმყოფებოდნენ მოწესრიგებულ მდგომარეობაში, როგორც ამას ითხოვს კლასტერული ინდექსი. ხშირადცვალებადი სვეტებისათვის უმჯობესია არაკლასტერული ინდექსის შექმნა.

ცხრილში პირველადი გასაღების შექმნისას სერვერი მისთვის ავტომატურად ქმნის კლასტერულ ინდექსს, თუ ის არ იყო ადრე შექმნილი ან გასაღების განსაზღვრისას არ იყო აშკარად მითითებული ინდექსის სხვა ტიპი.

თუ ცხრილში განსაზღვრულია კლასტერული და არაკლასტერული ინდექსები, მაშინ არაკლასტერული ინდექსის მიმთითებელი მიმართავს სტრიქონის არა ფიზიკურ მდებარეობას, არამედ კლასტერული ინდექსის შესაბამის ელემენტს, რომელიც ამ სტრიქონს აღწერს. ეს საშუალებას გვაძლევს არ გადავაწყოთ არაკლასტერული ინდექსების სტრუქტურა ყოველთვის, როცა კლასტერული ინდექსი ცვლის სტრიქონების ფიზიკურ განლაგებას ცხრილში. იცვლება მხოლოდ კლასტერული ინდექსი, ხოლო არაკლასტერული ინდექსები ანახლებენ მხოლოდ ინდექსირებულ მნიშვნელობებს და არა მიმთითებელს. თუ არაკლასტერული ინდექსის აგებისას კლასტერული ინდექსი არ არის უნიკალური, მაშინ სერვერი ავტომატურად უმატებს მას დამატებით მნიშვნელობებს, რომლებიც მას უნიკალურს ხდის. მომხმარებლისათვის ეს დამატებითი მნიშვნელობები უხილავია. რადგან, არაკლასტერული ინდექსები შეიცავენ კლასტერულ ინდექსზე მიმართვას, უნდა ვეცადოთ შეძლებისდაგვარად მინიმუმამდე დავიყვანოთ კლასტერული ინდექსში შემავალი სვეტების რაოდენობა და ზომა. წინააღმდეგ შემთხვევაში, სერვერის მწარმოებლურობა მკვეთრად მცირდება. ცხრილიდან კლასტერული ინდექსის წაშლისას, სერვერი გადააწყობს ყველა არაკლასტერულ ინდექსს, განსაზღვრავს რა მათთვის მიმართვებს სტრიქონების ფიზიკურ განლაგებაზე.

ცხრილში მხოლოდ ერთი კლასტერული ინდექსი შეიძლება იყოს.

უნიკალური ინდექსი

უნიკალური ინდექსები (unique indexes) იძლევიან ინდექსირებულ ცხრილში მნიშვნელობების უნიკალურობის გარანტიას. ის შეიძლება რეალიზებული იყოს როგორც კლასტერული, ისე არაკლასტერული ინდექსისათვის. ერთ ცხრილში შეიძლება არსებობდეს ერთი უნიკალური კლასტერული ინდექსი და რამდენიმე უნიკალური არაკლასტერული ინდექსი.

მონაცემების მთლიანობის უზრუნველსაყოფად უმჯობესია UNIQUE ან PRIMARY KEY შეზღუდვების და არა უნიკალური ინდექსის გამოყენება.

შევსების ფაქტორი

შევსების ფაქტორი (fill factor) განსაზღვრავს გვერდზე მონაცემების ჩაწერის სიმჭიდროვეს. ის განსაზღვრავს საინდექსო გვერდების თავისუფალი სივრცის რამდენი პროცენტი შეივსება მონაცემებით. დარჩენილი სივრცე თანდათანობით შეივსება ცხრილში მონაცემების დამატებასთან ერთად. იგი საშუალებას გვაძლევს უფრო მოქნილად ვმართოთ სერვერის მუშაობა ცხრილების ინდექსირების დროს. რაც მეტია შევსების ფაქტორი, მით ნაკლები იქნება გვერდზე თავისუფალი სივრცე და მით უფრო კომპაქტურად იქნება განთავსებული ინფორმაცია ინდექსების შესახებ. შევსების ფაქტორის არჩევისას უნდა შევაფასოთ რამდენად ინტენსიურად შესრულდება ცხრილში მონაცემების შეცვლა და დამატება. თუ ცხრილი ძირითადად გამოიყენება მხოლოდ წაკითხვისათვის, მაშინ უმჯობესია შევსების ფაქტორი დავაყენოთ 100%- თან ახლოს. ეს საშუალებას მოგვცემს უფრო ეკონომიურად გამოვიყენოთ მონაცემთა ბაზის სივრცე.

თუ ცხრილში მონაცემები ხშირად იცვლება, მაშინ შევსების ფაქტორი არ უნდა იყოს დიდი როგორც მონაცემებისათვის, ისე ინდექსებისათვის. მონაცემთა ბაზის სივრცე ამ შემთხვევაში არ იქნება გამოყენებული ეკონომიურად, სამაგიეროდ მონაცემების შეცვლისა და დამატების ოპერაციების შესრულების მწარმოებლურობა მაქსიმალური იქნება. თუ ინტენსიურად გამოყენებად ცხრილში დავაყენებთ მაღალ შეფასების ფაქტორს, მაშინ სერვერი იძულებული იქნება ხშირად შეასრულოს გვერდების დახლეჩა (page split) ახალი მონაცემების ჩასმის მიზნით. გვერდების დახლეჩა იკავებს დიდ დროს, ამიტომ შეძლებისდაგვარად უნდა მოვერიდოთ მის გამოყენებას.

გვერდების შევსების ფაქტორი განისაზღვრება ინდექსის შექმნის დროს. ცხრილში მონაცემების ცვლილებასა და დამატებასთან ერთად იცვლება გვერდების შევსების ხარისხი. ამიტომ, ინდექსის ზომის შემცირების მიზნით, პერიოდულად საჭირო ხდება ინდექსის გადაწყობა. ინდექსების გადაწყობა შრომატევადი და ხანგრძლივი პროცესია, ამიტომ შეძლებისდაგვარად ის უნდა შევასრულოთ მომხმარებლების მცირე აქტიურობის დროს. ინდექსების გადაწყობა შეიძლება, აგრეთვე, შესრულდეს ხელით. ინდექსების გადაწყობის დროს სერვერი ახდენს მონაცემების ბლოკირებას და მომხმარებლები ვერ შეძლებენ მონაცემებთან მიმართვას გადაწყობის დამთავრებამდე.

ინდექსის შექმნა

ინდექსის შექმნის უფლება აქვს მხოლოდ ცხრილის მფლობელს და ამ უფლების გადაცემა სხვა მომხმარებლისათვის არ შეიძლება. არსებობს ინდექსის შექმნის რამდენიმე საშუალება:

- ინდექსის ავტომატურად შექმნა პირველადი გასაღების შექმნისას;
- ინდექსის ავტომატურად შექმნა UNIQUE შეზღუდვის განსაზღვრისას;
- ინდექსის განსაზღვრა ცხრილის შექმნისას;
- ინდექსის შექმნა CREATE INDEX ბრძანებით.

ჩვენ განვიხილავთ ინდექსის შექმნას CREATE INDEX ბრძანების საშუალებით. მისი სინტაქსია:

CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED]

INDEX ინდექსის_სახელი ON { <ობიექტი>} (სვეტის_სახელი [ASC | DESC] [,...n])

[WITH

[PAD_INDEX]

[[,] FILLFACTOR = შევსების_ფაქტორი]

[[,] IGNORE_DUP_KEY]

[[,] DROP_EXISTING]

[[,] STATISTICS_NORECOMPUTE = { ON | OFF }]

<ობიექტი> კონსტრუქციის სინტაქსია:

<ობიექტი>::=[მონაცემთა_ბაზის_სახელი.[სქემის_სახელი].]ცხრილის_ან_წარმოდგენის_სახელი

განვიხილოთ არგუმენტების დანიშნულება:

UNIQUE არგუმენტის მითითების შემთხვევაში შეიქმნება უნიკალური ინდექსი. მისი შექმნისას სერვერი ასრულებს სვეტის წინასწარ შემოწმებას მნიშვნელობების უნიკალურობაზე. თუ სვეტში არის ორი ერთნაირი მნიშვნელობა, მაშინ ინდექსი არ შეიქმნება და გაიცემა შეტყობინება შეცდომის შესახებ. ამ შემთხვევაში სერვერის ქცევა არ არის დამოკიდებული IGNORE_DUP_KEY საკვანძო სიტყვის არსებობაზე. სერვერი არ წაშლის დუბლირებულ მნიშვნელობებს. თუ ინდექსი შედგენილია, ანუ შეიცავს ორ ან მეტ სვეტს, მაშინ უნიკალური უნდა იყოს ყველა ინდექსირებული სვეტის მნიშვნელობების ერთობლიობა ცხრილის ან წარმოდგენის თითოეულ სტრიქონში.

ინდექსირებულ სვეტში სასურველია, აგრეთვე, NULL მნიშვნელობის შენახვის აკრძალვა, რათა ავიცილოთ მნიშვნელობების უნიკალურობასთან დაკავშირებული პრობლემები. თუ სვეტში გვხვდება ორი NULL მნიშვნელობა, მაშინ სერვერი მას ორ ერთნაირ მნიშვნელობად ჩათვლის.

უნიკალური ინდექსის განსაზღვრის შემდეგ სერვერი არ შეგვასრულებინებს ისეთ INSERT ან UPDATE ბრძანებას, რომელიც გამოიწვევს ორი ერთნაირი მნიშვნელობის არსებობას.

– CLUSTERED მიუთითებს, რომ შესაქმნელი ინდექსი კლასტერული იქნება, ე.ი. ფიზიკურად მონაცემები განლაგებული იქნება ინდექსში განსაზღვრული რიგითობით.

- NONCLUSTERED მიუთითებს, რომ შესაქმნელი ინდექსი იქნება არაკლასტერული.
- ინდექსის_სახელი უნიკალური უნდა იყოს ცხრილის ფარგლებში.
- სვეტის_სახელი განსაზღვრავს იმ სვეტებს, რომლებიც ჩართული იქნება შესაქმნელ ინდექსში. თუ მითითებულია ერთზე მეტი სვეტი, მაშინ ინდექსი იქნება შედგენილი (composite index). ერთი შედგენილი ინდექსი შეიძლება შეიცავდეს 16-მდე სვეტს. ინდექსის აგება დაუშვებელია text, ntext ან image ტიპის მქონე სვეტების ბაზაზე. შედგენილ ინდექსში ჩართული ყველა სვეტის ზომა არ უნდა აღემატებოდეს 900 ბაიტს.

სვეტების სახელების მითითების მიმდევრობა გავლენას ახდენს მოთხოვნების შესრულების მწარმოებლურობაზე. ინდექსის საკვანძო ელემენტი შეიცავს სვეტების მნიშვნელობებს იმ მიმდევრობით, რა მიმდევრობითაც ისინი არიან ჩამოთვლილი. მაქსიმალური მწარმოებლურობის უზრუნველსაყოფად რეკომენდებულია თავდაპირველად იმ სვეტების სახელების მითითება, რომლებსაც მინიმალური სიგრძე აქვთ, შემდეგ კი იმ სვეტების სახელების მითითება, რომლებსაც მაქსიმალური სიგრძე აქვთ.

– [ASC | DESC] არგუმენტი ინდექსში განსაზღვრავს საკვანძო ელემენტების დახარისხების მეთოდს. ASC შემთხვევაში ელემენტები დალაგდება ზრდადობის მიხედვით, DESC შემთხვევაში კი - კლებადობის მიხედვით. ავტომატურად იგულისხმება ASC.

– PAD_INDEX არგუმენტის მითითების შედეგად სერვერი ინდექსის თითოეული გვერდზე მოახდენს თავისუფალი სივრცის რეზერვირებას ახალი სტრიქონების ჩასმისათვის. ეს არგუმენტი უნდა გამოვიყენოთ მხოლოდ FILLFACTOR არგუმენტთან ერთად.

– FILLFACTOR = შევსების_ფაქტორი აგრუმენტი განსაზღვრავს საინდექსო გვერდების შევსების ხარისხს და აზრი აქვს მხოლოდ ინდექსის შექმნის დროს. სერვერი ავტომატურად არ უზრუნველყოფს შევსების ფაქტორისს გარკვეულ მნიშვნელობას. თუმცა ჩვენ შეგვიძლია შევასრულოთ ინდექსის გადაწყობა (reindexing) შევსების ფაქტორის დასაყვანად გარკვეულ მნიშვნელობაზე.

– IGNORE_DUP_KEY არგუმენტი გამოიყენება მხოლოდ უნიკალური ინდექსის შექმნისას და განსაზღვრავს სერვერის ქცევას სტრიქონების დამატების ან შეცვლის ოპერაციის შესრულების მცდელობის შემთხვევაში, რომლებიც იწვევენ გამეორებადი მნიშვნელობების შექმნას. თუ უნიკალური ინდექსის შექმნისას (როგორც კლასტერული, ისე არაკლასტერული) მითითებული იყო IGNORE_DUP_KEY არგუმენტი, მაშინ იმ ოპერაციის შესრულებისას, რომელიც იწვევს დუბლირებული მნიშვნელობების შექმნას, სერვერი გასცემს შეტყობინებას შეცდომის შესახებ და გააუქმებს მხოლოდ იმ სტრიქონების შეცვლას, რომლებმაც დუბლირება გამოიწვიეს. თუ IGNORE_DUP_KEY არგუმენტი არ იყო მითითებული, მაშინ სერვერი, ასევე გასცემს შეტყობინებას შეცდომის შესახებ, მაგრამ აუქმებს ყველა ცვლილებას.

– DROP_EXISTING. თუ ცხრილში განსაზღვრული იყო ინდექსი იმავე სახელით, როგორიც შესაქმნელ ინდექსს ექნება, მაშინ ამ არგუმენტის გამოყენება გამოიწვევს არსებული ინდექსის ახლით შეცვლას და შეასრულებს მის გადაწყობას. უმჯობესია ეს არგუმენტი გამოვიყენოთ, ვიდრე ინდექსი ჯერ წავშლოთ და შემდეგ შევქმნათ. ეს არგუმენტი განსაკუთრებით ეფექტურია კლასტერული ინდექსების გადაწყობის

დროს მაშინ, როცა ცხრილში არაკლასტერული ინდექსების დიდი რაოდენობაა. თუ ჯერ წავშლით კლასტერულ ინდექსს, სერვერს მოუწევს ყველა არაკლასტერული ინდექსის სტრუქტურის გადაწყობა, რათა შეცვალოს ცხრილის სტრიქონებზე მიმართვები. შემდეგში, ახალი კლასტერული ინდექსის შექმნა ისევ გამოიწვევს არაკლასტერული ინდექსების გადაწყობას. DROP_EXISTING არგუმენტის მითითების შემთხვევაში სერევერი არ შეასრულებს არაკლასტერული ინდექსების შუალედურ გადაწყობას, რაც მკვეთრად ზრდის ინდექსის შექმნის სიჩქარეს. ამ არგუმენტის გამოყენება ნებადართულია მაშინ, როცა მონაცემთა ბაზაში ცხრილისთვის ან წარმოდგენისთვის უკვე არსებობს ამავე სახელის მქონე ინდექსი.

– STATISTICS_NORECOMPUTE. თუ არგუმენტი იღებს ON მნიშვნელობას, მაშინ ვადაგასული სტატისტიკა ავტომატურად არ გამოითვლება. თუ არგუმენტი იღებს OFF მნიშვნელობას, მაშინ შესაძლებელია სტატისტიკის ავტომატური გაახლება. ინდექსი ყოველთვის იქმნება მხოლოდ მიმდინარე მონაცემთა ბაზის ცხრილის ან წარმოდგენისათვის.

ინდექსების მართვა

ინდექსის შეცვლა ცხრილის ან წარმოდგენის ინდექსის შესაცვლელად ALTER INDEX ბრძანება გამოიყენება. მისი 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 }

}

განვიხილოთ არგუმენტების დანიშნულება.

– ALL. მიუთითებს, რომ უნდა შეიცვალოს ცხრილთან ან წარმოდგენასთან დაკავშირებული ყველა ინდექსი.

– REBUILD [WITH (<ხელახალი_აგების_რეჟიმი>)]. მიუთითებს, რომ ინდექსის ხელახალი აგება შესრულდება იმავე სვეტების, ინდექსის ტიპის, უნიკალურობის ატრიბუტებისა და დახარისხების მიმდევრობის გამოყენებით. ამ არგუმენტის გამოყენება DBCC DBREINDEX. ბრძანების გამოყენების ეკვივალენტურია.

– SORT_IN_TEMPDB. თუ არგუმენტი იღებს ON მნიშვნელობას, მაშინ tempdb მონაცემთა ბაზაში შეინახება დახარისხების შუალედური შედეგები, რომლებიც გამოყენებული იქნება ინდექსის ასაგებად. თუ არგუმენტი იღებს OFF მნიშვნელობას, მაშინ დახარისხების შუალედური შედეგები მოთავსდება იმავე მონაცემთა ბაზაში, რომელშიც ინდექსი. OFF არის ნაგულისხმევი მნიშვნელობა. თუ დახარისხების ოპერაცია არ არის მოთხოვნილი ან თუ დახარისხება ოპერატიულ მეხსიერებაში სრულდება, მაშინ ეს არგუმენტი უარიყოფა.

ინდექსების გადაწყობა

დროთა განმავლობაში საინდექსო ცხრილების შევსების ხარისხი მნიშვნელოვნად იცვლება, რასაც მივყავართ ცხრილში მონაცემების დამატებისა და შეცვლის ოპერაციების მწარმოებლურობის დაქვეითებასთან. ასეთ შემთხვევებში უნდა შევასრულოთ ინდექსის გადაწყობა საინდექსო გვერდებზე თავისუფალი ადგილის მოწესრიგების მიზნით. გარდა ამისა, ინდექსის გადაწყობა შეიძლება საჭირო გახდეს ინდექსში შემავალი სვეტების შემადგენლობის შეცვლისას.

ინდექსის გადაწყობის რამდენიმე გზა არსებობს. ერთი გზაა ინდექსის წაშლა და მისი ხელახლა შექმნა. თუ ასეთი გზით შევეცდებით კლასტერული ინდექსის გადაწყობას, მაშინ სერვერს მოუწევს ყველა არაკლასტერული ინდექსის გადაწყობა, ეს კი დიდ დროს იკავებს. ამიტომ, ეს არც თუ მისაღები გზაა. ინდექსის გადაწყობის მეორე, უფრო ეფექტური გზაა CREATE INDEX ბრძანებაში DROP_EXISTING არგუმენტის მითითება. ამ შემთხვევაში არ მოხდება ყველა კლასტერული ინდექსის გადაწყობა. ინდექსის გადაწყობის მესამე გზაა DBCC DBREINDEX ბრძანების გამოყენება.

მონაცემების ძებნა ცხრილებში

SQL-ის ძებნის გამოყენება სპეციფიკური მონაცემების მოსაძებნად ყველა ცხრილში და ყველა სვეტში არ არის ოპტიმალური ვარიანტი, SQL-ში არსებობს

სხვადასხვა სკრიპტები და მიდგომები რომლებიც შეიძლება გამოვიყენოთ ბაზიდან ინფორმაციის ამოსაღებად, მაგრამ ისინი ყველა იყენებენ კურსორს და ობიექტებს.

მაგალითად:

```
DECLARE
   @SearchText varchar(200),
   @Table varchar(100),
   @TableID int,
   @ColumnName varchar(100),
   @String varchar(1000);
--modify the variable, specify the text to search for SET @SearchText = 'John';
DECLARE CursorSearch CURSOR
    FOR SELECT name, object_id
        FROM sys.objects
      WHERE type = 'U';
--list of tables in the current database. Type = 'U' = tables(user-defined) OPEN
CursorSearch;
FETCH NEXT FROM CursorSearch INTO @Table, @TableID;
WHILE
       @@FETCH_STATUS
       =
       0
    BEGIN
        DECLARE CursorColumns CURSOR
            FOR SELECT name
                  FROM sys.columns
                WHERE
                       object_id
                       =
                       @TableID AND system_type_id IN(167, 175, 231, 239);
        -- the columns that can contain textual data
--167 = varchar; 175 = char; 231 = nvarchar; 239 = nchar
OPEN CursorColumns;
        FETCH NEXT FROM CursorColumns INTO @ColumnName;
        WHILE
               @@FETCH_STATUS
               =
               0
            BEGIN
                SET @String = 'IF EXISTS (SELECT * FROM '
                            + @Table
                            + ' WHERE '
                            + @ColumnName
                            + ' LIKE ''%'
                            + @SearchText
                            + '%'') PRINT '''
                            + @Table
                            + ', '
                            + @ColumnName
                            + '''';
                EXECUTE (@String);
                FETCH NEXT FROM CursorColumns INTO @ColumnName;
            END;
        CLOSE CursorColumns;
        DEALLOCATE CursorColumns;
        FETCH NEXT FROM CursorSearch INTO @Table, @TableID;
    END;
CLOSE CursorSearch;
DEALLOCATE CursorSearch;
```

ამ გადაწყვეტილების ნაკლია კურსორების (cursors) გამოყენება რომელიც არაეფექტურია არის რთული და დიდი დრო ჭირდება გაშვებისთვის პატარა მონაცემთა ბაზაზეც კი. მისი გამოყენება შესაძლებელია მხოლოდ ტექსტის საძებნად

რაც კიდევ ერთი მინუსია, სხვა ტიპის მონაცემების მოსაძებნად, როგორიცაა „datetime“ საჭიროა ახალი კოდის დაწერა.

ობიექტების ძებნა

მონაცემთა ბაზაში ობიექტის სახელის ან მისი განმარტების მოძებნა უპრო მარტივია ვიდრე ტექსტის, ობიექტის მოსაძებნად არსებობს რამოდენიმე მეთოდი რომლის გამოყენებაც შესაძლებელია, თუმცა ყველა ამ მეთოდს მოიცავს ქვერი სისტემის ობიექტები (querying system objects)

INFORMATION_SCHEMA.ROUTINES

SQL მოთხოვნა INFORMATION_SCHEMA.ROUTINES გამოიყენება კონკრეტული პარამეტრის მოსაძებნად ყველა პროცედურაში. INFORMATION_SCHEMA.ROUTINES შეიცავს ინფორმაციას ყველა ფუნქციის და შენახული პროცედურის შესახებ მონაცემთა ბაზაში. ROUTINE_DEFINITION სვეტი შეიცავს კოდის ანგარიშგებას რომელიც გამომდინარეობს ფუნქციიდან ან შენახული პროცედურიდან.

კოდი:

```
SELECT ROUTINE_NAME, ROUTINE_DEFINITION
    FROM INFORMATION_SCHEMA.ROUTINES
    WHERE ROUTINE_DEFINITION LIKE '%@StartproductID%'
    AND ROUTINE_TYPE='PROCEDURE'
```

შედეგი არის:

Results | Messages

	ROUTINE_NAME	ROUTINE_DEFINITION
1	uspGetBillOfMaterials	CREATE PROCEDURE [dbo].[uspGetBillOfMaterials] ...
2	uspGetWhereUsedProductID	CREATE PROCEDURE [dbo].[uspGetWhereUsedProductI...

არ არის რეკომენდირებული INFORMATION_SCHEMA views გამოყენება იმ ობიექტების მოსაძებნად რომლებიც ინახება ROUTINE_SCHEMA სვეტში, მის ნაცვლად შეგვიძლია გამოვიყენოთ sys.objects კატალოგი.

sys.syscomments view

sys.syscomments view მოთხოვნა, რომელიც შეიცავს მონაცემებს ყველა view, rule, default, trigger, CHECK და DEFAULT შეზღუდვებზე მონაცემთა ბაზაში, მოთხოვნა (query) ამოწმებს კონკრეტულ ტექსტს ექსტების სვეტში რომელიც შეიცავს object DDL-ს.

მაგალითი :

```
SELECT OBJECT_NAME( id )
  FROM SYSCOMMENTS
  WHERE text LIKE '%@StartProductID%' AND OBJECTPROPERTY(id , 'IsProcedure') = 1
  GROUP BY OBJECT_NAME( id );
```

შედეგი არის :

	(No column name)
1	uspGetBillOfMaterials
2	uspGetWhereUsedProductID

ეს მეთოდი არ არის რეკომენდირებული, რადგან sys.syscomments ცხრილი ამოღებულ იქნება SQL Server-ის მომავალში ვერსიები.

sys.sql_modules

sys.sql_modules მოთხოვნა, რომელიც შეიცავს სახელს, ტიპს და მნიშვნელობას (განსაზღვრებას) ყველა მოდულისა მონაცემთა ბაზაში.

```
SELECT OBJECT_NAME( object_id )
  FROM sys.sql_modules
WHERE
       OBJECTPROPERTY(object_id , 'IsProcedure')
       =
       1 AND definition LIKE '%@StartProductID%';
```

შედეგი იგივეა რაც წინა შემთხვევაში

	(No column name)
1	uspGetBillOfMaterials
2	uspGetWhereUsedProductID

სხვა sys schemaviews

მოთხოვნები sys.syscomments, sys.schemas and sys.objects views. sys.schemas შეიცავს რიგს (row) ყველა მონაცემთა ბაზის სქემისთვის. sys.objects შეიცავს რიგის ყველა განსაზღვრული მომხმარებლს და სქემის ფარგლებში მოქცეულ ობიექტს მონაცემთა ბაზაში, უნდა აღინიშნოს ის რომ ის არ შეიცავს ტრიგერის ინფორმაციას, ამიტომ უნდა გამოვიყენოტ sys.triggers -ები რომ მოვძებნოთ ობიექტის სახელები ან ობიექტის აღწერა ტრიგერებში.

```
DECLARE
 @searchString nvarchar( 50 );
SET@searchString = '@StartProductID';
SELECT DISTINCT
    s.name AS Schema_Name , O.name AS Object_Name , C.text AS Object_Definition
FROM
     syscomments C INNER JOIN sys.objects O
                     ON
     C.id
     =
     O.object_id
                   INNER JOIN sys.schemas S
                   ON
     O.schema_id
     =
     S.schema_id
WHERE
    C.text LIKE
     '%'
   + @searchString
   + '%'
 OR O.name LIKE
     '%'
   + @searchString
   + '%'
ORDER BY
       Schema_name , Object_name;
```

დაბრუნებული პასუხია :

	Schema_Name	Object_Name	Object_Definition
1	dbo	uspGetBillOfMaterials	CREATE PROCEDURE [dbo].[uspGetBillOfMaterials] ...
2	dbo	uspGetWhereUsedProductID	CREATE PROCEDURE [dbo].[uspGetWhereUsedProductI...

მთავარი სისუსტე ამ მეთოდისა არის რომ, ობიექტის ტიპის ძებნისას ყოველ ცვლილებაზე საჭიროა SQL ის კოდის შეცვლაც. იმისათვის რომ ჩვენ შევძლოთ ცვლილების შეტანა ჩვენ უნდა ვიცნობდეთ კარგად სისტემის ობიექტის სტრუტურას (SQL Server system object). შედგენილი ობიექტის ტიპით ან დამატებითი კრიტერიუმებით ძებნისას როგორიცაა, შეიცავს/გარდა ობიექტის სახელს და ტანს (bodie) საჭირო ხდება უფრო რთული SQL-ის კოდის გამოყენება, რაც იწვევს უფრო მეტ შეცდომას და მისი ტესტირებაც შრომატევადია.

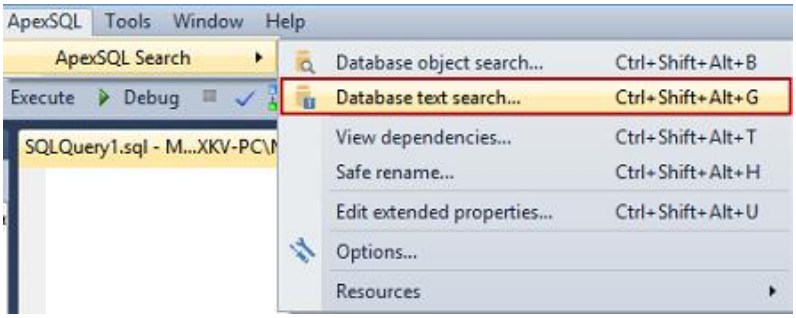
თუ არ ხართ გამოცდილი დეველოპერი უმჯობესია გამოიყენოთ დატესტილი და ერორებისგან გასფთავებული გადაწყვეტა (solution) მონაცემთა ბაზაში ძებნისთვის და მართვისთვის, თუ არ იცნობთ კარგად სისტემ ობიექტს(SQL Server system object) რომელსაც აქვს DDL ინფორმაცია მონაცემთა ბაზის ბიექტებზე გამოიყენეთ ApexSQL Search.

ApexSQL Search

ApexSQL Search არის SQL search add-in for SSMS and Visual Studio. მას შეუძლია მოძებნოს ტექსტი მონაცემთა ბაზის ობიექტებში (ობიექტის სახელის ჩათვლით) , ცხრილში შენახულ მონაცემებში და წარმოდგენებში (თუნდაც დაშიფრული), მას შეუძლია წინა ძებნის გამეორებაც ერთი დაკლიკებით.

ცხრილებში და view-ებში მონაცემების მოძებნა

1. In SQL Server Management Studio or Visual Studio's მთავარ მენიუში, დააკლიკეთ ApexSQL Search
2. Select Database text search…option:

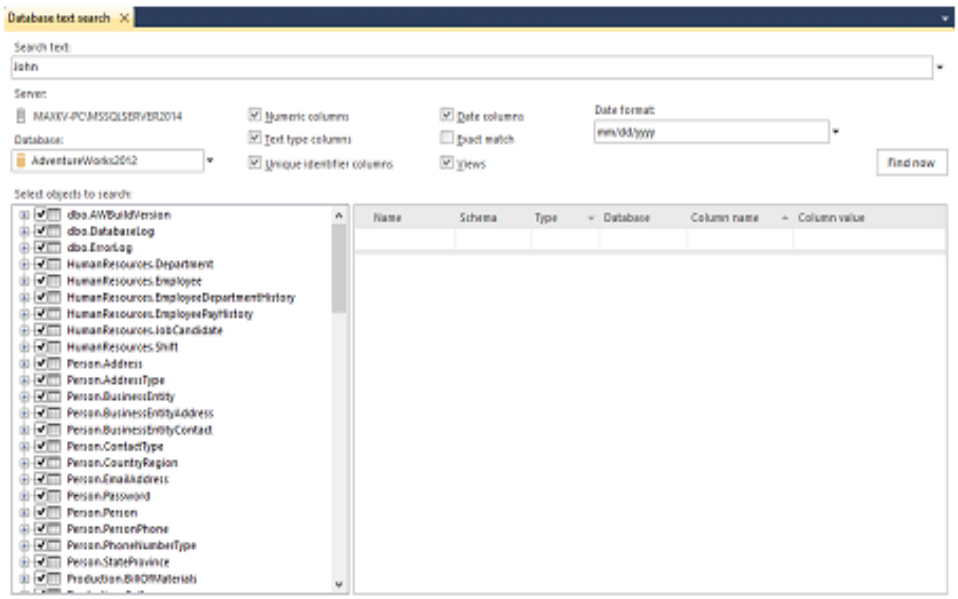


3. მოსაძებნ ველში ჩაწერეთ მონაცმის მნიშვნელობა რომელის მოძებნაც გსურთ

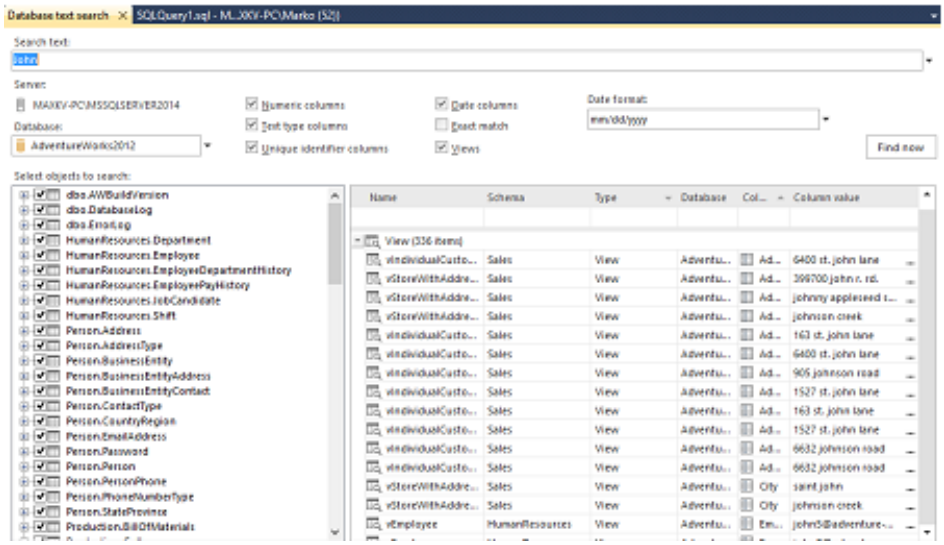
4. From the Database drop-down menu-დან ამოირჩიეთ მონაცემთა ბაზე რომელსიც სურთ მოძებნა

5. მონიშნეთ ობიექტი ძებნის ხეში, მონიშნეთ სასურველი ცხრილი ან დატოვეთ ყველა მონიშნული

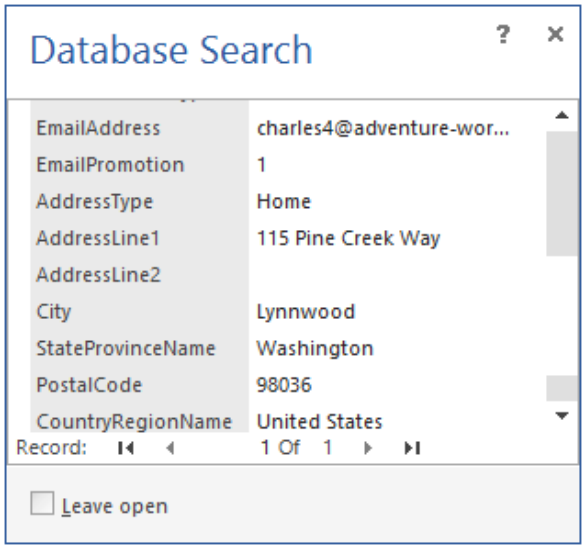
6. შეარჩიეთ სასურველი ძებნის კომპონენტები რიცხვითი, ტექსტური, უნიკალური აიდით, თარიღის სვეტით და მონიშნეთ შესაბამისი ველები, რათა ძებნა იყოს ზუსტი, თუ ეძებთ თარიღის სვეტით მაშინ მიუთითეთ თქვენთვის სასურველი თარიღის ფორმატი



7. დააკლიკეთ ღილაკს Find now და გრიდში გამოვა ცხრილები რომლებიც შეესაბამება ჩვენს მიერ მონიშნულ ველებს

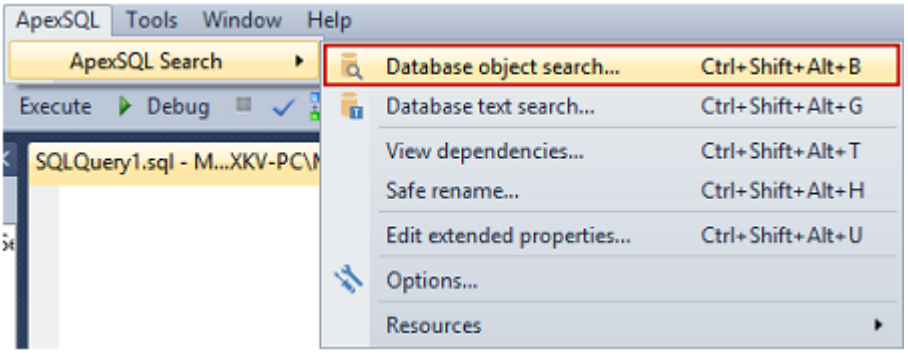


8. ობიეტის დეტალების (აღწერილობის სანახავად) დააჭირეთ პატარა ღილაკს Column value სვეტში



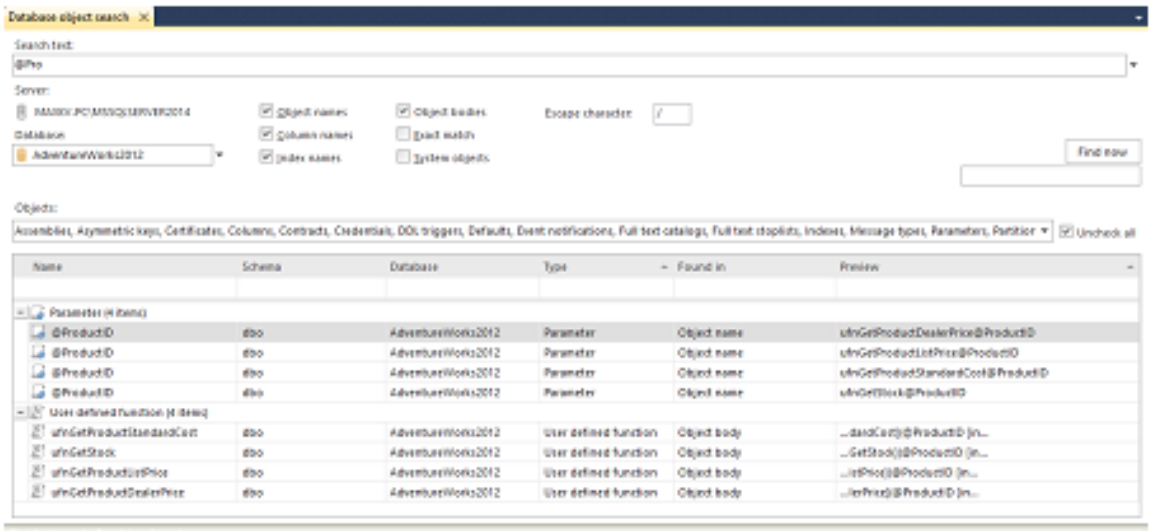
ობიექტების ძებნა

1. SQL Server Management Studio-ში ან Visual Studio-ს Main menu-ში , ApexSQL ის მენიუდან დავაკლიკოთ ApexSQL Search
2. დავაკლიკოთ Database object search-ს



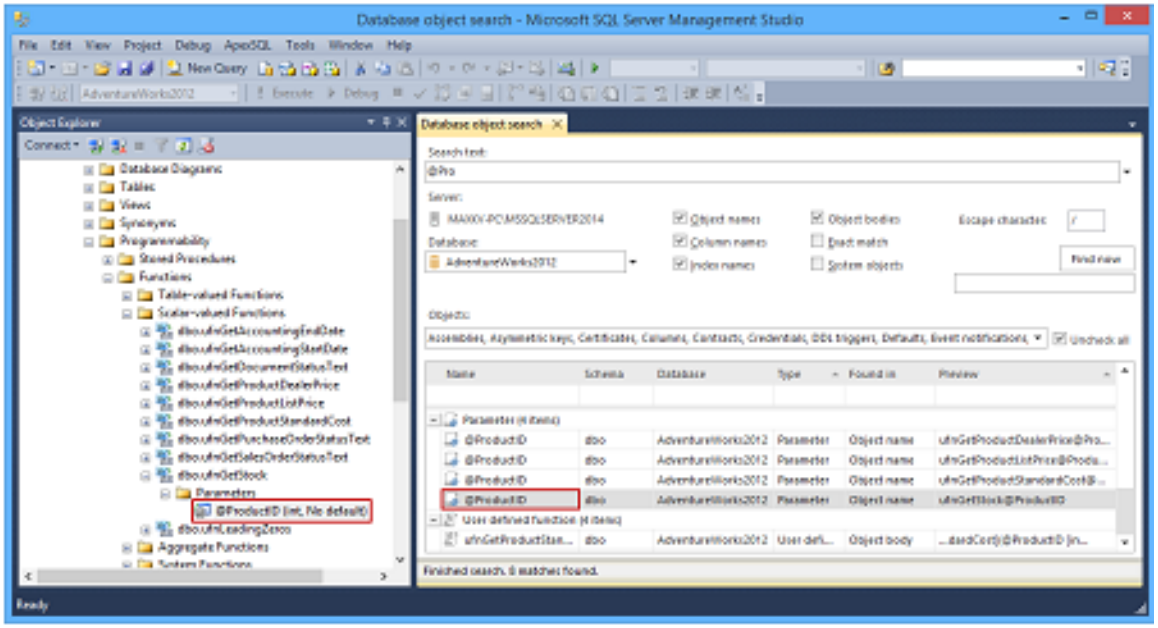
3. საძიებო ველში (Search text) ჩავწეროთ ტექსტი რომლის მოძებნაც გვინდა (მაგ: ცვლადის სახელი)
4. მონაცემთა ბაზის ჩამოსაშლელი მენიუდან მოვნიშნოთ სასურველი ბაზა
5. ობიექტების ჩამოსაშლელი მენიუდან მოვნიშნოთ ობიექტის ტიპი ა ყველა დავტოვოთ მონისნული
6. მოვნიშნოთ შესაბამისი ჩექ ბოქსები რათა უკეთ განისაზღვროს საძიებო ობიექტი

7. დავაკლიკოთ Find now ღილაკს



გრიდი გამოიტანს მონაცემთა ბაზის ობიექტებს რომლებიც შეიცავს ჩვენს საძებნ კონკრეტულ ობიეტს.

8. ორმაგი დაკლიკებით გრიდში მოინიშნება კონკრეტული ობიექტი (რომელსაც ვეძებდით)



SQL Server Management Studio და Visual Studio არ უზრუნველყოფს ძებნას მონაცემთა ბაზაში ობიექტის სახელის, ობიექტის განსაზღვრების და ინფორმაციის მიხედვით, SQL მოთხოვნა რომელსაც ამ ყველაფრის გაკეთება შეუძლია არის კომპლექსური, ნელი და მოითხოვს SQL Server system object-ის

ცოდნას, გამოიყენეთ ApexSQL მონაცემთა ბაზაში ობიექტების და ტექტების საძებნელად, ეს ბევრად ნაკლებ დროს და ცოდნას მოითხოვს.

სრული ტექსტის ძებნა (Full Text Search)

SQL Server საშულებას იძლევა ცხრილებში Full Text მოთხოვნების გამოყენების. ეს შესაძლებლობა ხელსაყრელია როდესაც გვჭირდება დიდი ზომის ტექსტის მოძიება, მაგალითად როგორიცაა შენიშვნები, წიგნის აღწერილობა, კინოს მიმოხილვა და ა.შ. Full Text მოთხოვნები შეიძლება შეიცავდეს მთლიან სიტყვებს და ფრაზებს, აგრეთვე სიტყვებისა და ფრაზების მრავალჯერად ფორმებს. Full Text Search შესაძლებლობას SQL Server-ში უზრუნველყოფს Microsoft Full Text Engine. მისი გამოყენებისას ვახდენთ ცხრილის ან მთლიანი მონაცემთა ბაზის მონიშვნას. Full Text ინდექსები იგება SQL Server მონაცემთა ბაზის ფაილების გარეთ Windows file system-ში სპეციალური Full Text ინდექსების სახით. შესაძლებელია მოხდეს სპაციფიკაცია, თუ რა სიხშირით განახლდეს ეს ინდექსები, რათა დაბალანსდეს მონაცემების გამოყენების საკითხები დროსთან მიმართებაში.

SQL Server Database Engine-ის მეშვეობით შესაძლებელია ძირითადი ტექსტის ძებნა (base text search). მაგალითად, იმისათვის რომ მოვნახოთ ყველა სტრიქონი, რომლის ტექსტური სვეტი შეიცავს სიტყვას “Geo”, შესაძლებელია დავწეროთ შემდეგი SQL ბრძანება:

```
Select * from resume where description like '%Geo%';
```

ეს მოთხოვნა იპოვის resume ცხრილიდან ყველა იმ სვეტს, რომლის description ველი შეიცავს სიტყვას “**Geo**”. შევნიშნოთ, რომ ამ მეთოდს გააჩნია გარკვეული პრობლემა: პირველი, ძებნა ხორციელდება საკმაოდ ნელა, რადგანაც Database Engine-ს არ შეუძლია სტრიქონების ინდექსირება რის გამოც Database Engine-ს უხდება ძებნა მთელ ცხრილში . მეორე, თუნდაც მონაცემები ყოფილოყო varchar ტიპის და არა text ტიპის, ინდექსირება ვერ გვიშველიდა ვერც ამ შემთხვევაში, რადგან ჩვენ უნდა მოგვეძებნა „**Geo**”მთელ სტრიქონში და არა მხოლოდ მის დასაწყისში. შესაბამისად ინდექსის გამოყენება არ იქნებოდა შედეგიანი. გარდა ამისა როგორ უნდა ვიმოქმედოთ თუ გვინდა ამ სიტყვის მოძებნა მთლიან ცხრილში და არა მხოლოდ სტრიქონში, ან თუ გვინდა მოვძებნოთ კონკრეტული აღწერილობა, მაგ თუ ვეძებთ “SQL” და “ability to work independently”. Full Text ინდექსები უზრუნველყოფს ამ პრობლემის გადაჭრას. იმისათვის რომ გაგვეხორციელებინა ძებნა ზემოთ აღნიშნული მაგალითის შესაბამისად, უნდა დაგვეწერა:

```
Select * from resume where contains (description, 'Geo');
```

მონაცემთა ბაზაში ინფორმაციის ძებნა

(კომბინირებული)მოდელი

ინფორმაციის ძებნის თანამედროვე ალგორითმები, რომლებიც ძირითადად გამოიყენება სხვადასხვა საძიებო სისტემებში ერთი მხრივ აპრობირებულია და ფართოდ გამოყენებადია, მაგრამ მეორეს მხრივ მისი გამოყენება მონაცემთა ბაზებისათვის სემანტიკურად რელევანტური ძებნის თვალსაზრისით თითქმის არ განიხილება. ძებნა სტრუქტურირებულ მონაცემთა ბაზაში, არააქტუალურად ითვლება, რადგან არსებობს ძებნის კლასიკური ალგორითმები და მათი სხვადასხვა მოდიფიკაციები, რომლებიც უფრო სწრაფია.

მონაცემთა ბაზებში ინფორმაცის ძებნის მოდულის შესამუშავებლად მოხდა არსებული თანამედროვე ძებნის ალგორითმების (ლოგიკური მოდელი „IR Boolean model“, ვექტორული სივრცის მოდელი „Vector Space Model“ , ალბათური მოდელი „Probabilistic Model“) კომბინირება და მის საფუძველზე მივიღეთ ეგრეთწოდებული ინფორმაციის ძებნის „ჰიბრიდული“ მოდელი.

ინფორმაციის ძებნის ლოგიკური მოდელი
IR Boolean model

პირველი ინფორმაციის ძებნის მოდელი არის ინფორმაციის ლოგიკური ძებნის მოდელი. ეს მოდელი ახდენს მოთხოვნის შინაარსობრივ ანალიზს და მას უსადაგებს შესაბამის რელევანტურ დოკუმენტს. მაგალითად მოთხოვნა სახელწოდებით economic მარტივად განსაზღვრავს, იმ დოკუმენტებს რომლების ინდექსირებული არიან განმარტებით economic.

ლოგიკური მოდელი იყენებს George Bool-ის მათემატიკურ ლოგიკას. იგი მოთხოვნის ტერმინებს და დოკუმენტის ინდექსირებულ აღწერებს უსადაგებს ერთმანეთს, რათა შექმნას ახალი ტიპის დოკუმენტები. ამ მოდელში განსაზღვრულია სამი საბაზისო ოპერატორი. ესენია: ოპერატორები “AND“ , “OR” და “NOT”

დადებითი მხარეები:

- არის ადვილად დასანერგი და კომპიუტერულად ეფექტური/ქმედითი
- საშუალებას აძლევს მომხმარებელს გადმოსცეს სტრუქტურული და კონცეპტუალური შეზღუდვები, რათა შეძლოს მნიშვნელოვანი ლინგვისტური თვისებების აღწერა. (მოთხოვნაში შესაძლებელია გამოყენებულ იქნას სინონიმური სპეციფიკაციები და ფრაზები).
- ლოგიკური ძებნის მეთოდს გააჩნია მკაფიოდ და ნათლად გამოხატვის შესაძლებლობა. (მოთხოვნა უნდა იყოს ამომწურავი და არაორაზროვანი).
- ეს მეთოდი გვთავაზობს ბევრ მეთოდს მოთხოვნის გასაფართოებლად ან დასაკონკრეტებლად.

- აქვს მეტი ეფექტი ძებნის შემდეგ ეტაპზე, მეტი სიზუსტე ცნებებსა და დამოკიდებულებებს შორის.

უარყოფითი მხარეები:

- მომხმარებელს უჭირს ეფექტური ლოგიკური მოთხოვნის შემუშავება (მომხმარებლები იყენებენ ენის კავშირებს “AND“ „„OR“,“NOT“, რომლებსაც მოთხოვნის შემთხვევაში გააჩნია განსხვავებული მნიშვნელობა, ვინაიდან აღიქვამენ, როგორც ინგლისური ენის კავშირებს)

ინფორმაციის ძებნის ვექტორული სივრცის მოდელი „Vector Space Model“

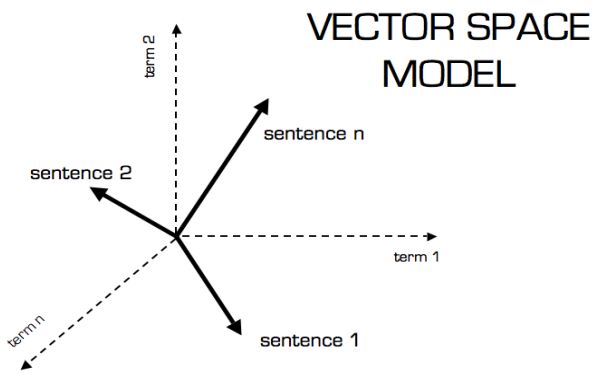
ამ მოდელით ძებნისთვის მომხმარებელმა საჭირო დოკუმენტების მოსაძებნად უნდა მოამზადოს ისეთი დოკუმენტი რომელიც მაქსიმალურად მსგავსი იქნება საჭირო დოკუმენტებისა. მსგავსების ხარისხი, მომზადებულ დოკუმენტსა და დოკუმენტების კოლექციას შორის გამოიყენება ძიების რეზულტატის რეიტინგის განსასაზღვრად.

რაც უფრო მეტი იქნება თანხვედრა/შეთანხმება წარმოდგენებსა და მოცემულ ელემენტებს და მის განაწილებას შორის, მით მეტი იქნება ალბათობა იმისა, რომ მოძიებული იქნება საძიებო ინფორმაციის ანალოგიური/მსგავსი.

აღნიშნულ მოდელში დოკუმენტები და მოთხოვნები წარმოდგება მრავალგანზომილებიან სივრცეში, რომლის განზომილებები გამოიყენება ინდექსის შესაქმნელად, დოკუმენტის აღწერისათვის. ინდექსის შექმნის მიზანია მნიშვნელოვანი ასპექტების ლექსიკური სკანირება, მორფოლოგიური ანალიზი საძიებო სიტყვის „აზრობრივად“ მსგავსი ალტერნატივების შერჩევისათვის. დოკუმენტის ან მისი სუროგატის/შემცვლელის მოთხოვნის დროს შედარება ხდება ვექტორების გამოყენებით, მაგალითად გამოიყენება კოსინუსის მსგავსი გაზომვა.

აღნიშნული მეთოდი კარგად მუშაობს დოკუმენტებთან, რომლებიც შესაძლოა შეიცავდეს მოთხოვნის რამდენიმე ზედმეტ ტერმინს, თუ ეს ტერმინები ხვდებიან იშვიათად კოლექციაში, მაგრამ ხშირად დოკუმენტში. ვექტორულ სივრცის მოდელი შედგება შემდეგი მოსაზრებებისაგან:

- უფრო მეტად დოკუმენტთან მიახლოებული მოთხოვნის ვექტორი ზრდის იმის ალბათობას, რომ დოკუმენტი ეკუთვნის/შეესაბამება მომხმარებლის მოთხოვნას.
- საძიებო სიტყვა გამოიყენება დოკუმენტის სივრცის ზომის განსასაზღვრად (როდესაც მოხდება საძიებო სიტყვის მიახლოებულ მნიშვნელობის მოძიება, ითვლება რომ ეს სიტყვა არ არის რეალური, იდენტური)



ინფორმაციის ძებნის ალბათური მოდელი
Probabilistic Model

ინფორმაცის ძებნის ალბათური მოდელი დაფუძნებულია მონაცემების ალბათური/რეიტინგული შეფასების პრინციპზე, რომელიც გულისხმობს, რომ ინფორმაციის საძიებო სისტემას უნდა შეეძლოს მოთხოვნის და დოკუმენტის საუკეთესო შესაბამისობის მოძიება. მეთოდში შესაძლოა იყოს განსხვავებული მტკიცებულების წყაროები, რომლებსაც იყენებს მოცემული მეთოდი. უმეტეს შემთხვევაში დოკუმენტში არსებობს როგორც რელევანტური ასევე არა რელევანტური ინფორმაციის სტატისტიკური განაწილება.
განვიხილოთ ინფორმაციის ძებნის სტატისტიკური მოდელის ძლიერი და სუსტი მხარეები:
უპირატესობები:

- მეთოდები აწვდიან მომხმარებლებს რელევანტური რეიტინგით მოძიებულ დოკუმენტს.
- მოთხოვნა შესაძლებელია იყოს მარტივად ფორმულირებადი, რადგან მომხმარებელს არ მოუწიოს მოთხოვნის ენის შესწავლა.
- მოთხოვნის კონცეფციის უჩვეულოდ დამახასიათებელ არჩევა შეიძლება იყოს წარმოდგენილი.

უარყოფითი მხარეები:

- არ ყოფნის სტრუქტურა, რათა მოძებნოს საჭირო აზრობრივად მიახლოებული ფრაზები.
- რელევანტური გამოთვლა შესაძლოა შეფასებული იყოს როგორც ძვირადღირებული(მოითხოვს მეტ რესურსს) გამოთვლა.

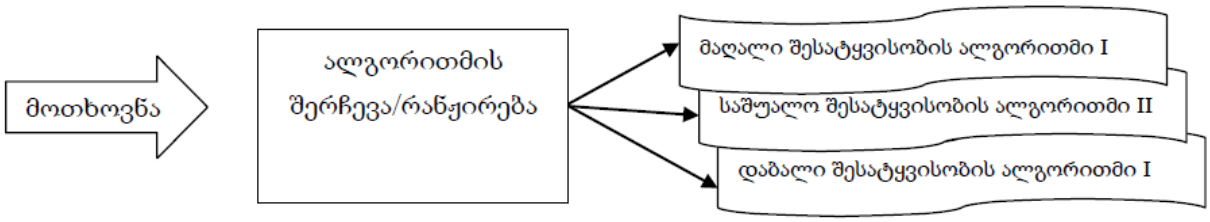
ზემოთ აღნიშნული ძებნის ალგორითმების ანალიზის საფუძველზე ირკვევა, რომ სხვადასხვა ტიპის მონაცემების ძებნისათვის (სკალარული, ვექტორული, სიმბოლური, ტექსტური, ლოგიკური და ა.შ) სხვადასხვა ალგორითმს სხვადასხვა

შედეგები აქვს. ჩვენ აქცენტს ვაკეთებთ სემანტიკურ ძებნაზე ამიტომ ალგორითმების პრიორიტეტულობა ლაგდება სემანტიკურად ზუსტი შედეგის მიხედვით (დროის ფაქტორი ამ ეტაპზე არ განიხილება).

ჩვენს მიერ შექმნილი მოდელი წარმოადგენს სამი(ლოგიკური მოდელი „IR Boolean model“, ვექტორული სივრცის მოდელი „Vector Space Model“ , ალბათური მოდელი „Probabilistic Model“) მოდელის ეტაპობრივ გამოყენებას. მოდელში ძებნის პროცესი განიხილება, როგორც სამ დონიანი პროცესი, რომელსაც წინ უძღვის მოთხოვნის ტიპის იდენტიფიცირების ეტაპი და შედეგების რანჟირების ეტაპი.

საწყისი ეტაპი - მოთხოვნის ტიპის იდენტიფიცირება

ამ ეტაპზე ხდება მოთხოვნის ტიპის იდენტიფიცირება. ეს გულისხმობს - საწყისს ეტაპზე იმისდა მიხედვით თუ მომხმარებელი რა ტიპის (სკალარული, ვექტორული, სიმბოლური, ტექსტური, ლოგიკური და ა.შ) მონაცემის მოძიებას სურს, სისტემა ირჩევს მონაცემს ტიპის მიხედვით სამი წარმოდგენილი ალგორითმიდან ყველაზე მისადაგებულ ალგორითმს.



ძებნის პირველი ეტაპი

საწყის ეტაპზე შერჩეული ალგორითმით ხორციელდება ძებნა მონაცემთა ბაზაში. გამოიყოფა ცალკე მოძიებული ინფორმაცია.

ძებნის მეორე ეტაპი

საწყის ეტაპზე რანჟირების საფუძველზე მეორე ალგორითმით ხორციელდება ძებნა მონაცემთა ბაზაში. გამოიყოფა ცალკე მოძიებული ინფორმაცია.

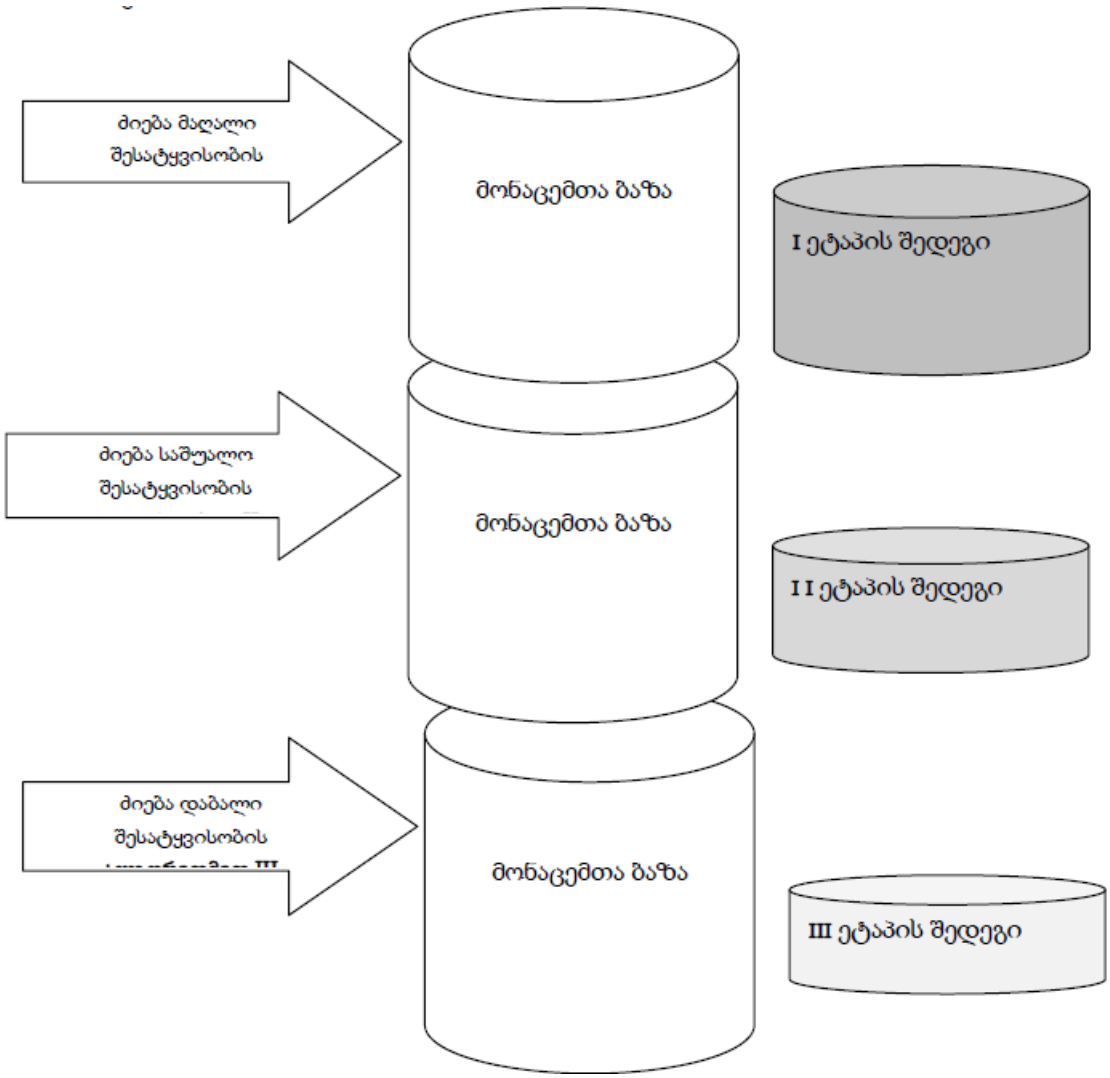
ძებნის მესამე ეტაპი

დარჩენილი ალგორითმით ხორციელდება ძებნა მონაცემთა ბაზაში. გამოიყოფა ცალკე მოძიებული ინფორმაცია.

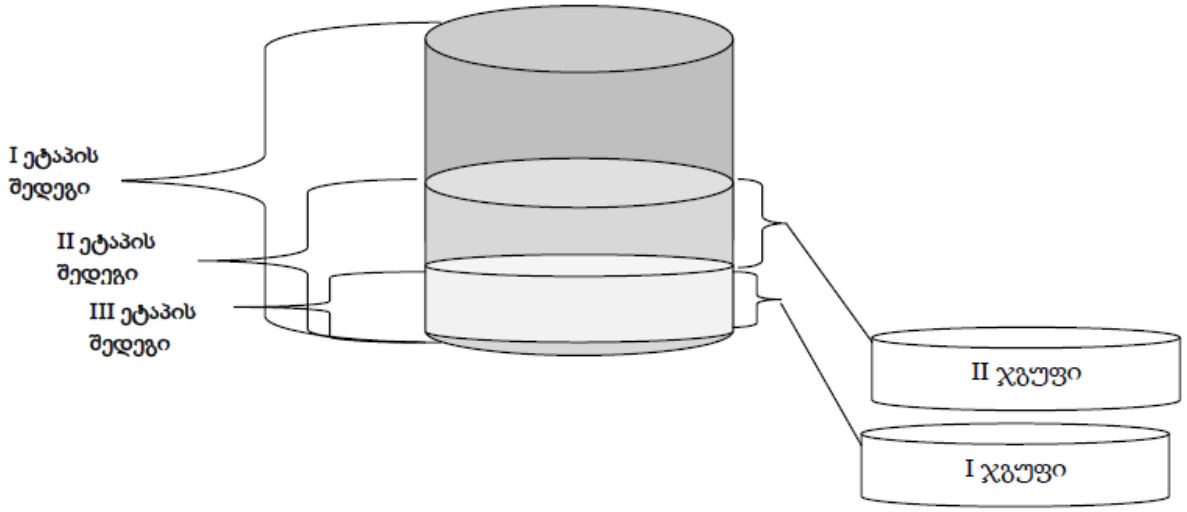
შედეგების რანჟირების ეტაპი

ძებნის სამი დონის გავლის შედეგად მიიღება მონაცემთა ბაზიდან ამოკრეფილი სამი ქვესიმრავლე, რომლებსაც ლოგიკურად შესაძლებელია ჰქონდეთ თანაკვეთა. ამ შედეგთა გაერთიანებით მიღებული მონაცემები რანჟირება ხდება შემდეგი პრინციპით:

- პირველი ჯგუფი - მონაცემები რომელიც სამივე ალგორითმით მოიძებნა.
- მეორე ჯგუფი - მონაცემები რომელიც პირველი და მეორე ალგორითმით მოიძებნა.
- მესამე ჯგუფი - მონაცემები რომელიც პირველი და მესამე ალგორითმით მოიძებნა.



- მეოთხე ჯგუფი - მონაცემები რომლებიც დარჩა წინა ამორჩევების შემდეგ პირველ ჯგუფში
- მეხუთე ჯგუფი - მონაცემები რომელიც მეორე და მესამე ალგორითმით მოიძებნა.
- მეექვსე ჯგუფი - მონაცემები რომლებიც დარჩა წინა ამორჩევების შემდეგ მეორე ჯგუფში
- მეშვიდე ჯგუფი - მონაცემები რომლებიც დარჩა წინა ამორჩევების შემდეგ მესამე ჯგუფში.



ჩვენს მიერ შემუშავებული მეთოდი იმის გამო, რომ მოითხოვს ბაზაზე სამჯერ ერთი და იმავე ინფორმაციის მოძიებას, ძებნის სისწრაფის თვალსაზრისით არაეფექტურია, მაგრამ მოთხოვნაზე იძლევა პასუხს მაღალი სიზუსტით, ასევე მომხმარებელს საშუალება ეძლევა მიიღოს პრიორიტეტულობის მიხედვით იერარქიულად დალაგებული სხვა ალტერნატიული პასუხებიც.

შემუშავებული მოდელის რეალიზაციისათვის თუ ჩვენ გამოვიყენებთ ისეთ პროგრამულ ინსტრუმენტს, რომელიც ნებისმიერ მონაცემთა ბაზის მართვის სისტემასთან იქნება თავსებადი, მიღებული პროდუქტი იქნება უნივერსალური და მოსახერხებელი ყველა მომხმარებლისათვის.

დასკვნა

სამაგისტრო ნაშრომის ფარგლებში ჩატარებული სამუშაოების შედეგად გამოიკვეთა, რომ არ არსებობს ინფორმაციის ძებნის უნივერსალური მეთოდი, რომელიც ერთნაირად ეფექტური იქნება მონაცემთა ბაზებში და ასევე სხვადასხვა მონაცემთა საცავებში. გარდა ამისა მონაცემთა ბაზებში ინფორმაციის ძებნისათვის ძირითადად გამოიყენება სტანდარტული გადარჩევის ალგორითმები, რომლებიც უზრუნველყოფენ ძებნას ველების მიხედვით.

შემუშავებული მოდელის დახვეწის და მისი პროგრამული რეალიზაციის შედეგად შესაძლებელი იქნება შეიქმნას ისეთი პროგრამულ ინსტრუმენტი, რომელიც ნებისმიერ მონაცემთა ბაზის მართვის სისტემასთან იქნება თავსებადი, მიღებული პროდუქტი იქნება უნივერსალური და მოსახერხებელი ყველა მომხმარებლისათვის.

გამოყენებული ლიტერატურა

1. Eric A. Brewer Combining Systems and Databases: A Search Engine Retrospective
2. Milena Petrovic http://solutioncenter.apexsql.com/quickly-search-for-sql-database-data-and-objects
3. http://en.wikipedia.org/wiki/Database_search_engine
4. The University of Queensland www.library.uq.edu.au