მონაცემთა ბაზები

მონაცემთა ბაზების ფართო გამოყენება

დღეისათვის მონაცემთა ბაზები ფართოდ არის გავრცელებული და ისინი ფაქტიურად ყველგან გამოიყენება. თითქმის ყველა გამოყენებით პროგრამას გააჩნია მონაცემთა ბაზებზე დაფუძნებული სტრუქტურა. როდესაც ჩვენ რაიმეს ვყიდულობთ ინტერნეტის საშუალებით "on-line"-რეჟიმში, როდესაც ჩვენ ვახორციელებთ დრაივერების ლიცენზიების განახლებას, ვკითხულობთ თვითმფრინავთა ფრენის განრიგს, ვეძებთ სხვადასხვა სპორტული შეჯიბრერების შედეგებს, ჩვენ ვიყენებთ პროგრამებს რომლებიც მონაცემთა ბაზებზეა დაფუძნებული. მონაცემთა ბაზები უზრუნველყოფენ მონაცემთა შენახვის ეფექტურ, უსაფრთხო და მოქნილ ხერხს, და გამოიყენებიან სხვადასხვა მიმართულების პროგრამებში: ბიბლიოთეკის ბარათების კატალოგებიდან დაწყებული ქარხნების დაზგების ავტომატიზაციით დამთავრებული. მაგრამ მდგომარეობა ყოველთვის ასეთი არ იყო.

1950-იანი წლების ბოლოს, როდესაც მეორე თაობის კომპიუტერები გაჩნდა, მაღალი დონის პროგრამირების ენებისა და დიდი მოცულობის დამაგროვებლების (ძირითადად მაგნიტური ლენტები) გამოყენებამ ძალიან სწრაფად გახადა შესაძლებელი მონაცემთა დიდი კრებულების შექმნა. თავიდან მონაცემები ინახებოდა ცალკეულ ფაილებში, მაგრამ საკმაოდ სწრაფად ცხადი გახდა, რომ ასეთი მიდგომა ქმნიდა მრავალ სიძნელეს.

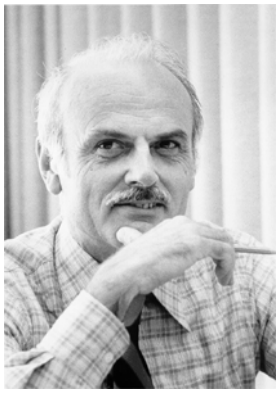
პირველ რიგში, ფაილები რაც უფრო დიდი მოცულობის ხდებოდნენ მით უფრო დიდ დროს მოითხოვდნენ მომხმარებლისათვის საჭირო მონაცემთა ძიებისათვის. გავიხსენოთ ამ წიგნის ის თავი რომელშიც ვიხილავდით ალგორითმებს: თანმიმდევრობითი ძიების ალგორითმი მოითხოვს დროს რომელიც O(n)-ის რიგისაა. ამიტომ რაც უფრო დიდია მოცულობით მონაცემთა ფაილი მით უფრო დიდია მონაცემთა გარკვეული ელემენტის ძიების დრო. ასეთი ვითარება შესაძლებელია არ იყოს პრობლემატური როდესაც საქმე ეხება ჩვენი მეგობრების დაბადების თარიღების შენახვას და ძიებას, მაგრამ როდესაც ჩვენ ვინახავთ და ვამუშავებთ ჩანაწერებს MasterCard-ით განხორციელებული შესყიდვების შესახებ მრავალი მილიონი მომხმარებლისათვის, თანმიმდევრობითი ძიების საშუალებით ინფორმაციის წვდომა მიუღებელი ხდება.

ადგილი აქვთ ასევე სხვა პრობლემებსაც. მაგალითად, თუ ჩვენ ყოველი შესყიდვის თითოეულ ჩანაწერში ვაფიქსირებთ კლიენტის სარეგისტრაციო მისამართს, რაც ფაქტიურად ჭარბი მონაცემების არსებობას ნიშნავს, მაშინ ჩანაწერების ფაილი მონაცემთა მატარებლის საკმაოდ დიდ მოცულობას დაიკავებს. უფრო მეტიც, წარმოვიდგინოთ, რომ რომელიმე კლიენტმა შეიცვალა მისამართი, მაშინ ჩვენ იძულებული ვხდებით ხელახლა გადავწეროთ ამ კლიენტის მიერ განხორციელებული ყველა ტრანზაქცია მისი ახალი მისამართის გათვალისწინებით, ანუ ამ შემთხვევაში სახეზეა მონაცემებში ცვლილებების შეტანის პრობლემატურობა. ჩვენ შესაძლებელია შევეცადოთ ამ კონკრეტული პრობლემის გადაჭრა სარეგისტრაციო მისამართების განთავსებით სხვა გამოყოფილ ფაილში. ეს რა თქმა უნდა, მოახდენს ტრანზაქციების ფაილის მოცულობის შემცირებას, მაგრამ კლიენტის ტრანზაქციათა სრული ანგარიშის შედგენისას ჩვენ იძულებული ვიქნებით განვახორციელოთ თანმიმდევრული ძიება უკვე ერთდროულად ორ ფაილში ერთის ნაცვლად.

მონაცემთა ბაზების ტიპები (სახეები)

1960-იანი წლების ბოლოს ამ და სხვა პრობლემების გადასაჭრელად შეიქმნა მონაცემთა ბაზების სისტემები. მონაცემთა ბაზების პირველი სისტემები წარმოადგენდნენ იერარქიული და ქსელური ტიპის სისტემებს. IBM-ში შეიმუშავეს იერარქიული ტიპის მონაცემთა ბაზა DL/1, სხვა კომპანიებმა წარმოადგინეს ქსელური ტიპის მონაცემთა ბაზები CODASYL-ის (Conference On DAta SYstems Language, Database Task Group) მოდელის საფუძველზე. IDMS-ი განსაკუთრებით წარმატებულ CODASYL-მონაცემთა ბაზას წარმოადგენდა.

იერარქიული და ქსელური ტიპის მონაცემთა ბაზების სტრუქტურები უზრუნველყოფდნენ როგორც მომხმარებლისათვის საინტერესო ინფორმაციის სწრაფ წვდომას, ასევე მონაცემთა მარტივი სახით განახლებას და უსაფრთხო შენახვას. მაგრამ ისინი საკმაოდ რთულ სტრუქტურებს წარმოადგენდნენ. ამასთან ერთად ნებისმიერი ასეთი სტრუქტურა საკმაოდ ძლიერად იყო დამოკიდებული კონკრეტული ფაილური სისტემის იმპლიმენტაციის დეტალებზე, რაც მათ საკმაოდ ხისტ სახეს ანიჭებდა.



1970 წელს IBM-ის თანამშრომელმა ე. კოდმა (E. F. Codd - წარმოშობით ინგლისელი, რომელიც მეორე მსოფლიო ომში მონაწილეობის შემდეგ ამერიკის შეერთებულ შტატებში გადავიდა) შეიმუშავა მონაცემთა ბაზების რელაციური მოდელი. რელაციური მოდელი მკაცრად ეფუძნებოდა მათემატიკურ თეორიას. იმ დროისათვის ამ მოდელს არარეალურ მიდგომად მიიჩნევდნენ, რადგანაც მონაცემების შენახვა მოდელში უნდა მომხდარიყო ცხრილებში („relation“ - ცხრილი). ყოველ ცხრილში განთავსდებოდა ინფორმაცია მხოლოდ ერთი ლოგიკური ობიექტის შესახებ, და ეს ობიექტები ერთმანეთთან დაკავშირებული იქნებოდა არა გარე მაჩვენებლებისა და სხვა ხერხებით არამედ ამ ცხრილებშივე განთავსებული ინფორმაციის საშუალებით.

კოდმა ასევე ჩამოაყალიბა მონაცემთა წვდომის ენა, რომელიც დაფუძნებული იყო სიმრავლეთა თეორიაზე მოგვიანებით, 1980-ან წლებში ფირმა IBM-ი მსოფლიოს წარუდგენს სტრუქტურირებული მიმართვების ენას (Structured Query Language = SQL)). იმ დროის პროფესიონალთა უმეტესობის აზრით კოდის იდეების ეფექტური განხორციელება შეუძლებელ ამოცანას წარმოადგენდა.

ამის მიუხედავად უკვე 1980-ან წლებში კომპანია Oracle პირველად სთავაზობს მომხმარებელს რელაციური ბაზების მართვის სისტემის კომერციულ რეალიზაციას, ხოლო კომპანია IBM-ი იწყებს საკუთარ რელაციურ მონაცემთა ბაზების (სახელად DB2) გაყიდვას. დღეისათვის მონაცემთა რელაციური მოდელი წარმოადგენს ყველაზე უფრო გავრცელებულ მოდელს, და ამ თავში ჩვენ ზუსტად ამ მოდელზე ვიქნებით ფოკუსირებული.

ობიექტებზე ორიენტირებული პროგრამირების განვითარებას მოჰყვა მონაცემთა მართვის ახალი სქემების განვითარება, რომელთაც მონაცემთა ბაზების ობიექტურ-რელაციური ან ობიექტებზე ორიენტირებული მართვა ეწოდება. ეს სისტემები უზრუნველყოფენ მუშაობის ისეთ სახეს, რომელიც მოხერხებულია და შეთანხმებული ობიექტებზე ორიენტირებული პროგრამირების მეთოდებთან.

http://en.wikipedia.org/wiki/Relational_database
http://en.wikipedia.org/wiki/List_of_database_models

მონაცემთა ბაზის გამოყენების უპირატესობა

მონაცემთა ბაზების გამოყენების უპირველეს მოტივაციას წარმოადგენს მონაცემთა წვდომის სიჩქარე (სისწრაფე). თუ მონაცემთა ბაზის სქემა სწორად არის შერჩეული, მაშინ, მიუხედავად მონაცემთა ბაზის ზომებისა და მასში არსებული ჩანაწერების რაოდენობისა, მონაცემთა წვდომა თითქმის მყისიერია. მილიონობით ჩანაწერთა შორის ჩვენთვის საინტერესო ჩანაწერის ზუსტი და მყისიერი მოძებნა გამაოგნებელ შთაბეჭდილებას ახდენს.

კონკრეტულ მონაცემთა წვდომის დრო საკმაოდ მცირე (თითქმის ნულოვანი) და ფაქტიურად მუდმივი სიდიდის ტოლია, თანაც არ არის დამოკიდებული მონაცემთა ბაზაში არსებულ ჩანაწერთა რაოდენობაზე (ასეთი სახის დამოკიდებულება შესაძლებელია გამოიხატოს როგორც O(k), სადაც k არის მცირე სიდიდის მუდმივა). ასეთი სწრაფი მოქმედება შესაძლებელია იმის გამო, რომ მონაცემთა ბაზის მართვის სისტემა ინახავს როგორც თვითონ მონაცემებს ასევე მეტამონაცემებს - მონაცემებს მონაცემების შესახებ (metadata - მეტამონაცემები).

მეტამონაცემების საშუალებით ბაზაში შენახული პირველადი მონაცემების თვითაღწერა ხორციელდება. ეს ნიშნავს, რომ იმ პროგრამებს, რომლებიც ახორციელებენ მონაცემთა წვდომას, არ ესაჭიროებათ მრავალი დეტალის ცოდნა იმის შესახებ თუ როგორ ინახება მონაცემები ბაზაში. თუ პროგრამა კითხულობს მონაცემებს ფაილიდან, მაშინ ამ პროგრამამ უნდა იცოდეს შესაბამისი მონაცემთა ტიპები, ფორმატები და ველების თანმიმდევრობა. მაგრამ როდესაც პროგრამა მიმართავს მონაცემთა ბაზას მას უმეტეს შემთხვევაში ესაჭიროება მხოლოდ მოსაძებნი მონაცემების ზოგადი დახასიათება (ყოველგვარი დეტალების გარეშე).

მონაცემთა ბაზა ასევე უნდა უზრუნველყოფდეს მონაცემთა ფიზიკური მატარებლის ეფექტურ გამოყენებას. თუ მონაცემთა ბაზის სტრუქტურა (მისი შინაგანი სქემა) ადეკვატური სახით არის შერჩეული მაშინ ბაზაში არ უნდა ხდებოდეს ერთი და იგივე მონაცემების განმეორება. იმ დროისათვის, როდესაც მონაცემთა დამაგროვებლის ფასი საკმოდ მაღალი იყო ამ უპირატესობას დიდი მნიშვნელობა ენიჭებოდა. მაგრამ, ბაზაში მონაცემთა სიჭარბის აცილება დღესაც მნიშვნელოვანია ისეთი საკითხებისათვის როგორც მონაცემთა ბაზის მუშაობის ეფექტურობის

გაზრდა, სხვადასხვა სახის შეცდომებისა და ანომალიების (ახალი მონაცემების შეტანის ან ძველი მონაცემების განახლების პროცესში და სხვა) აცილება და მონაცემთა მთლიანობის უზრუნველყოფა.

მონაცემთა ბაზების მართვის სისტემები (Database management systems = (DBMS)) ასევე უზრუნველყოფენ ბაზებში არსებული მონაცემების დაცვას და უსაფრთო შენახვას. მაგალითად, თითქმის ყველა თანამედროვე მონაცემთა ბაზების მართვის სისტემაში არსებობს მონაცემთა ბაზების სარეზერვო ასლების შექმნისა და მათი აღდგენის ინტეგრირებული იარაღები, თანაც სარეზერვო ასლების შექმნა მონაცემთა სარეზერვო ფიზიკურ მატარებლებზე მაშინაც კი არის შესაძლებელი, როდესაც მართვის სისტემა ერთდროულად მრავალ მომხმარებელს ემსახურება.

მონაცემთა ბაზების მართვის სისტემები ასევე უზრუნველყოფენ *ტრანზაქციის* კონცეფციას (concept of a *transaction*). ტრანზაქცია წარმოადგენს მონაცემთა ბაზაში ცვლილებების განხორციელების ოპერაციათა ჯგუფს. ეს ოპერაციები ერთმანეთთან იმყოფებიან მჭიდრო კავშირში, თანაც ან ყველა ოპერაცია უნდა შესრულდეს, ან არცერთი მათგანი. ტრანზაქციის ცნობილი მაგალითია ფონდების მოხსნა სადეპოზიტო ანგარიშიდან (a savings account) და მათი განთავსება მობილურ ანგარიშზე (a checking account). ჩვენ გვჭირდება ორივე ოპერაციის (მოხსნა და განთავსება) წარმატებულად შესრულება, მაგრამ თუ ფონდების გამოტანის ოპერაცია შესრულდა, ხოლო განთავსების ოპერაცია ვერ განხორციელდა, მაშინ ჩვენ ინტერესებში იქნება საწყისი გამოტანის ოპერაციის გაუქმება და ფონდების დაბრუნება უკან სადეპოზიტო ანგარიშზე. ეს ორი ოპერაცია ორივე ერთად წარმოადგენს ერთ ტრანზაქციას, რომელიც ან უნდა შესრულდეს მთლიანად, ან საერთოდ არ უნდა შესრულდეს რათა არ მოხდეს ფულადი თანხების დაკარგვა. მონაცემთა ბაზების მართვის სისტემები იძლევიან მონაცემებზე მოქმედების ოპერაციათა ტრანზაქციების სახით დაჯგუფების საშუალებას. იმ შემთხვევაში, როდესაც ტრანზაქციაში შემავალი ყველა ოპერაცია წარმატებულად შესრულდა, ამბობენ რომ ტრანზაქცია დაფიქსირდა, ხოლო თუ რომელიმე ოპერაცია არ შესრულდა მაშინ ტრანზაქცია მთლიანად უქმდება (ბრუნდება უკან).

მონაცემთა ბაზების მართვის სისტემას (მობამას-ს) შეუძლია ერთდროულად მოემსახუროს ათი ათასობით მომხმარებელს, თანაც ამასთან ერთად უზრუნველყოს ბაზებში არსებული მონაცემების საიმედო უსაფრთხოება და მთლიანობა. მაგალითად ავიღოთ კომერციული ორგანიზაცია www.Amazon.com, რომელთანაც მსოფლიოს სხვადასხვა ადგილებში მყოფი მრავალი მომხმარებელი ერთდროულად აქტიურ დიალოგს ახორციელებს: გზავნიან სხვადასხვა შინაარსის კონკრეტულ მიმართვებს, ახორციელებენ მათთვის საინტერესო ინფორმაციის ძიებას, აფორმებენ შეკვეთებსა და ხელშეკრულებებს, გადარიცხავენ ფულად თანხებს და ა.შ.. მონაცემთა ბაზების მართვის სისტემა ისეთი სახით ახორციელებს ასეთი მრავალ-მომხმარებლური წვდომის კოორდინაციას, რომ მუშაობის პროცესში დაცულია ბაზებში განთავსებული მონაცემების მთლიანობა და მათი ცვლილებების დაფიქსირების ადეკვატურობა. ნებისმიერი კონკრეტული მომხმარებლის მიერ განხორციელებული ნებისმიერი სახის ოპერაციები არვითარ ზეგავლენას არ ახდენენ სხვა მომხმარებლების ურთიერთქმედებაზე მონაცემთა ბაზებთან. მონაცემთა ბაზების მართვის სისტემები უზრუნველყოფენ მონაცემთა ბაზების დროებით ჩაკეტვებს, რითაც ახდენენ მოსალოდნელი კონფლიქტების აცილებას.

ყველა ზემოთ ჩამოთვლილი მიზეზების გამო მონაცემთა ბაზების მართვის სისტემებმა მოიპოვეს ფართო გავრცელება მთელ მსოფლიოში. როგორც ჩვენ ვნახავთ, მონაცემთა ბაზების მართვის სისტემების გამოყენებას ძალიან შეუწყო ხელი SQL-სტანდარტული ენის (**SQL** = **S**tructured **Q**uery **L**anguage, **სტრუქტურირებული მიმართვების ენა**) შემოტანამ და მისმა განვითარებამ. დღეისათვის საკმაოდ რთულია რომელიმე მნიშვნელოვანი გამოყენებითი პროგრამის დასახელება, რომელიც არ მოიცავს მონაცემთა ბაზას ან არ გაჩნია მასთან რაიმე კავშირი.

მონაცემთა დომენის მოდელირება

რელაციური მონაცემთა ბაზის შექმნისას მონაცემთა ბაზების დიზაინერი გადის გარკვეულ ეტაპებს, ერთერთ მათგანს მონაცემთა მოდელირება ეწოდება. მოდელირების ფაზა განსაზღვრავს ერთეულებს (ობიექტებს) (“entities”), თითოეული ობიექტის ატრიბუტებს (თვისებებს) (“attributes”) და სხვადასხვა ობიექტებს შორის კავშირებს, რომლებიც უნდა არსებობდნენ მონაცემთა ბაზაში.

მაგალითად, უნივერსიტეტის საქმიანობის აღმწერ მონაცემთა ბაზის დაპროექტებისას საჭირო ობიექტების სია მოიცავს შემდეგ ობიექტებს: სტუდენტები, პროფესორები, საერთო საცხოვრებლები, სასწავლო კორპუსები, ძირითადი სპეციალობები, დამატებითი სპეციალობები, სხვადასხვა კურსები, და ა.შ.. ობიექტ „სტუდენტს“ ექნება ისეთი ატრიბუტები, როგორიცაა მისი სახელი, გვარი, მისამართი, არჩეული ძირითადი სპეციალობა, ხელმძღვანელი, და ა.შ.. ერთერთ კავშირს სხვადასხვა ობიექტებს შორის წარმოადგენს ხელმძღვანელობასთან დაკავშირებული კავშირი ობიექტ „სტუდენტსა“ და ობიექტ „პროფესორის“ შორის.

ობიექტები წარმოადგენენ საგნებს, არსებით სახელებს, რომლებიც ინახებიან მონაცემთა ბაზაში. ხშირად ეს ობიექტები შეესაბამება ბუნებაში არსებულ რეალურ ობიექტებს, მაგალითად პროფესორები, მანქანები, შენობები და ა.შ.. ზოგჯერ ისინი წარმოადგენენ უფრო აბსტრაქტულ ობიექტებს, მაგალითად კოლეჯი უნივერსიტეტის სტრუქტურაში, წიგნის შეკვეთა ელექტრონულ მაღაზიაში, მომხმარებელთა გარკვეული ჯგუფის პრივილეგიები. მონაცემთა მოდელირების ერთერთ მნიშვნელოვან ნაწილს წარმოადგენს ამ ობიექტების დადგენა-შერჩევა მონაცემთა ბაზის კონკრეტული მოდელისათვის.

ისინი ვინც იცნობენ ობიექტებზე ორიენტირებული პროგრამირების ენების კონცეფციებს, აუცილებლად დაინახავენ მსგავსებას ასეთ ობიექტებსა და ობიექტებზე ორიენტირებული პროგრამირების ენებში არსებულ კლასებს შორის. ობიექტის ყოველი ინდივიდუალური ერთეული (შესაძლებელია ანალოგიის გავლება კლასის ეგზემპლართან) ხასიათდება შესაბამისი ატრიბუტების მნიშვნელობათა კონკრეტული სიმრავლით. ატრიბუტები წარმოადგენენ ზედსართავ სახელებს ან დესკრიპტორებს მონაცემთა ბაზაში არსებული ობიექტების ეგზემპლარებისათვის. მაგალითად რომელიმე კონკრეტული სტუდენტის ატრიბუტებს შესაძლებელია წარმოადგენდნენ შემდეგი კონკრეტული მნიშვნელობები: „ბილ სმიტი“, „კომპიუტერული მეცნიერება“, „პროფესორი ფეინმანი“, და ა.შ..

მონაცემთა ბაზის სტრუქტურა აღიწერება მისი ლოგიკური სქემით (schema). როგორც ვნახავთ მოგვიანებით, იმისათვის რომ მონაცემთა მოდელი გარდავქმნათ რელაციურ მონაცემთა ბაზის სქემაში, ობიექტთა ყოველი ეგზემპლარი უნდა იყოს უნიკალური. ატრიბუტების სიმრავლეთა მიხედვით ყოველი ეგზემპლარი უნდა განსხვავდებოდეს იგივე ობიექტის სხვა ეგზემპლარებისაგან. მაგალითად, ჩვენ შესაძლებელია ვიფიქროთ, რომ მხოლოდ ერთი ბილ სმიტი გვეყოლება, მაგრამ იმ შემთხვევაში თუ ორი ან უფრო მეტი სტუდენტია ერთნაირი სახელითა და გვარით, მაშინ უნდა მოიძებნოს სხვადასხვა ბილ სმიტების განსხვავებულ ეგზემპლარებად გადაქცევის (გარდაქმნის) ხერხი. „სტუდენტი“ ობიექტის თითოეულ ეგზემპლარს ჩვენ უნდა შევუსაბამოთ „გასაღების“ (“key”) ისეთი კონკრეტული მნიშვნელობა, რომელიც განასხვავებს მას ყველა სხვა სტუდენტისაგან.

როდესაც შერჩეულია მონაცემთა ბაზის ობიექტები, შესაძლებელია განხორციელდეს მათ შორის კავშირების დადგენა. მაგალითად ავიღოთ კავშირი ხელმძღვანელი/სტუდენტი (ან მრჩეველი/სტუდენტი), რომელიც შესაძლებელია არსებობდეს პროფესორებსა და სტუდენტებს შორის. აქ ერთერთ ძირითად საკითხს წარმოადგენს იმის განსაზღვრა თუ როგორი სახის იქნება ეს კავშირი: 1:1 (ერთი - ერთთან) (one-to-one), 1:N (ერთი - ბევრთან) (one-to-many), N:M (ბევრი - ბევრთან) (many-to-many). ამ თანაფარდობებს კარდინალობათა თანაფარდობები ეწოდებათ (cardinality ratios).

მრჩეველი/სტუდენტი კავშირის შემთხვევაში მონაცემთა ბაზის დიზაინერმა შესაძლებელია გადაწყვიტოს, რომ ეს კავშირი 1:N ტიპისაა, ანუ ერთი მრჩეველი ემსახურება N სტუდენტს. მეორეს მხრივ, თუ სკოლა ყოველ სტუდენტს მიუჩენს რამდენიმე მრჩეველს (მაგალითად ერთი მრჩეველი ძირითად სპეციალობის საკითხებში და მეორე მრჩეველი საყოფაცხოვრებო საკითხებში) მაშინ კავშირი N:M სახის იქნება, რამდენიმე მრჩეველი თითოეული სტუდენტისათვის და რამდენიმე სტუდენტი თითოეული მრჩევლისათვის.

კავშირის კარდინალობათა თანაფარდობისათვის მნიშვნელოვანია კიდევ ორი დაზუსტება, რომლებიც მინიმალური კარდინალობების განსაზღვრას უკავშირდება. აუცილებლად უნდა ჰყავდეს სტუდენტს მრჩეველი? თუ ასეა მაშინ მრჩეველი/სტუდენტი კავშირის მინიმალური კარდინალობა პროფესორების მხარეს უნდა იყოს 1. ხოლო თუ ასე არ არის მაშინ კავშირის მინიმალური კარდინალობა პროფესორის მხარეს იქნება 0, ანუ შესაძლებელია არსებობდნენ ისეთი ეგზემპლარები სტუდენტ-ობიექტში, რომლებიც არ უკავშირდებიან არცერთ მრჩეველს.

ანალოგიურად, ყველა პროფესორი უნდა ასრულებდეს მრჩევლის ფუნქციას, თუ არა? თუ ასეა მაშინ მრჩეველი/სტუდენტი კავშირის მინიმალური კარდინალობა სტუდენტების მხარეს უნდა იყოს 1-ის ტოლი. ხოლო თუ ასე არ არის, მაშინ კავშირის მინიმალური კარდინალურობა სტუდენტების მხარეს იქნება 0, ანუ არსებობენ პროფესორები ობიექტის ისეთი ეგზემპლარები რომლებიც არ ასრულებენ სტუდენტთა მრჩევლის ფუნქციებს.

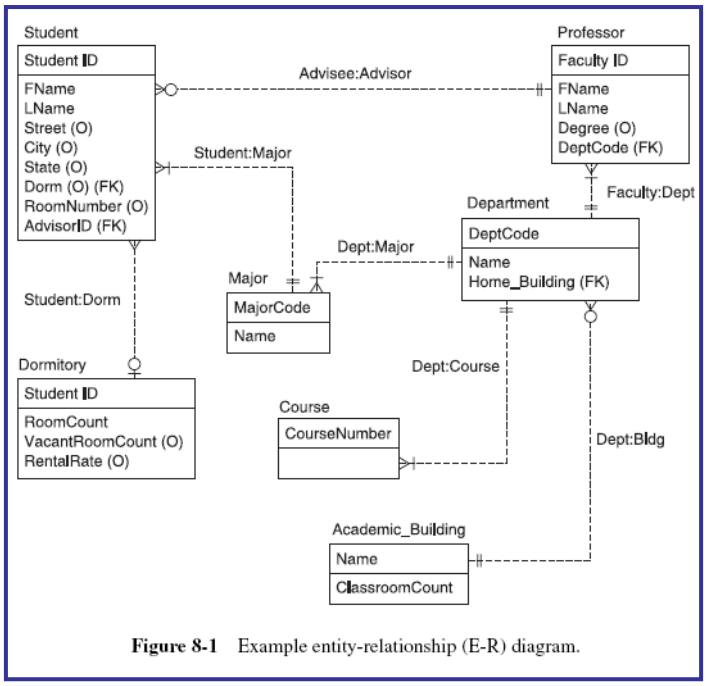


Figure 8-1 Example entity-relationship (E-R) diagram.

კავშირი შესაძლებელია იყოს 1:1. მაგალითად, წარმოვიდგინოთ ისეთი ობიექტი რომელსაც ეწოდება ავტომობილის პარკინგის ნებართვა, და თანაც დავუშვათ რომ ყოველ სტუდენტს ეძლევა ერთი და მხოლოდ ერთი პარკინგის ნებართვა. კავშირს სტუდენტებსა და პარკინგის ნებართვებს შორის შესაძლებელია ვუწოდოთ „სტუდენტი/პარკინგის_ნებართვა“, და რა თქმა უნდა კავშირი 1:1 სახისაა. ამ შემთხვევაშ მინიმალური კარდინალურობა სტუდენტების მხარეს უნდა იყოს 1, რადგანაც სხვა შემთხვევაში მონაცემთა ბაზაში აღმოჩნდება ისეთი გაცემული ნებართვები, რომლებიც არავისზე არ არის გაფორმებული. მინიმალური კარდინალობა პარკინგის ნებართვების მხარეს ალბათ 0-ის ტოლი იქნება, რადგანაც იქნებიან ალბათ ისეთი სტუდენტები რომელთაც არ გააჩნიათ ავტომობილი.

მრავალი-მრავალთან (N:M) კავშირი არსებობს სტუდენტებსა და სალექციო კურსებს შორის. ჩვენ შეგვიძლია ვუწოდოთ ამ კავშირს „კურსები/სტუდენტები“. ყოველი სტუდენტი ირჩევს მრავალ კურს, და ყოველ სალექციო კურს ესწრება მრავალი სტუდენტი. მინიმალური კარდინალობა კავშირის ორივე მხარეს იქნება 1-ის ტოლი, რადგანაც თითოეულ სტუდენტს რასაკვირველია ექნება არჩეული რამდენიმე სხვადასხვა კურსი და თითოეულ სალექციო კურსს ალბათ ესწრება სტუდენტების რაღაც გარკვეული რაოდენობა. მეორეს მხრივ, თუ ჩვენ შევინარჩუნებთ მონაცემთა ბაზაში ისეთ სალექციო კურსებს, რომლებიც აღარ იკითხებიან რაიმე მიზეზების გამო, მაშინ ამ კავშირის მინიმალური კარდინალობა სტუდენტების მხარეს იქნება 0-ის ტოლი.

სურათზე 8.1 გამოსახულია მონაცემთა მოდელი იმ ობიექტებისა და მათ შორის კავშირებისა, რომლებსაც ჩვენ ვიხილავდით ზემოთ. მოცემული დიაგრამა წარმოადგენს „ობიექტები-კავშირები“ დიაგრამას, რომელიც შესრულებულია ერთერთი სტანდარტული გრაფიკული ნოტაციის (აღნიშვნების) საფუძველზე. ეს სურათი შესრულებულია Microsoft Visio-ს გამოყენებით.

მართკუთხედები შეესაბამება ობიექტებს; ატრიბუტი, რომელიც განთავსებულია მართკუთხედის ზემო ნაწილში, შეესაბამება ობიექტის იდენტიფიკატორს, ანუ ობიექტის გასაღებს (key - გასაღები). მაგალითად, სტუდენტური საერთო საცხოვრებლების (dormitory - სტუდენტური საერთო საცხოვრებელი) ობიექტის იდენტიფიკატორს წარმოადგენს dorm-ი; კონკრეტული საცხოვრებლის სახელი განასხვავებს ერთი საცხოვრებლის ჩანაწერებს სხვა საცხოვრებელთა ჩანაწერებისაგან. მართკუთხედის ქვემო ნაწილში ჩამოთვლილი სახელებები შეესაბამება ობიექტის სხვადასხვა ატრიბუტებს. მაგალითად, ობიექტი „სტუდენტური საერთო საცხოვრებლები“ (Dormitory) მოიცავს ყოველი საერთო საცხოვრებლისათვის ისეთ ატრიბუტებს როგორიცაა ოთახების რაოდენობა საერთო საცხოვრებელში (RoomCount), ვაკანტური ოთახების რაოდენობა (VacantRoomCount), საცხოვრებლის ოთახის გაქირავების ფასი (RentalRate).

წყვეტილი ხაზები წარმოადგენენ კავშირებს ობიექტებს შორის, ნიშნები ხაზების ბოლოებში წარმოადგენენ კარდინალობებს. წრეწირი ხაზის ბოლოში ნიშნავს რომ შესაბამისი კავშირი შესაბამისი ობიექტის მიმართ არ არის აუცილებელი (ანუ არის არჩევითი, დამატებითი)(მინიმალური კარდინალობა ნულის ტოლია); ვერტიკალური ხაზი კავშირის ბოლოში აღნიშნავს, რომ კავშირის კონკრეტულ ეგზემპლარებში უნდა არსებობდეს შესაბამისი ობიექტის მხოლოდ ერთი ეგზემპლარი (კარდინალობა ზუსტად ერთის ტოლია); სამი ხაზი კავშირის ბოლოში აღნიშნავს, რომ კავშირის ნებისმიერ კონკრეტულ ეგზემპლარში შესაძლებელია მონაწილეობდეს შესაბამისი ობიექტის რამდენიმე ეგზემპლარი (კარდინალობა ერთზე მეტია).

მაგალითად, ცალკეული დეპარტამენტი სთავაზობს ერთ ან რამდენიმე სალექციო კურს; თითოეული დეპარტამენტი სთავაზობს ერთ კურსს მაინც. კონკრეტული საერთო საცხოვრებელი შესაძლებელია ასოცირებული იყოს ერთ ან მრავალ სტუდენტთან, ხოლო რომელიმე სტუდენტი შესაძლებელია ან არ იყოს დაკავშირებული არც ერთ საერთო საცხოვრებელთან, ან იყოს

დაკავშირებული მხოლოდ ერთ საერთო საცხოვრებელთან. ზოგიერთ სტუდენტს საერთო საცხოვრებელი არ ესაჭიროება, მაგრამ თუ სტუდენტი ასოცირებულია საერთო საცხოვრებელთან მაშინ მას შეესაბამება ყველაზე მეტი ერთი საერთო საცხოვრებელი.

ობიექტების, მათი ატრიბუტების და ობიექტებს შორის კავშირების განსაზღვრის ასეთი პროცესი საკმაოდ ეფექტურია ობიექტებისა და კავშირების უმრავლესობისათვის. მაგრამ არსებობს რამდენიმე სპეციალური შემთხვევა რომლებსაც პრაქტიკაში ხშირად ვხვდებით და რომლებიც მოითხოვენ დამატებით ანალიზს.

ზოგიერთი ობიექტი მხოლოდ იმ შემთხვევაში შეგვიძლია შევიტანოთ მონაცემთა ბაზაში, როდესაც ბაზაში უკვე არსებობს რომელიღაც სხვა მასთან დაკავშირებული ობიექტი. მაგალითად, ჩვენ შეგვიძლია შევიტანოთ მონაცემთა ბაზაში რომელიმე პროფესორის შესახებ გარკვეული მონაცემების განმსაზღვრელი ობიექტის ეგზემპლარი (პროფესორთან დაკავშირებული ობიექტის ეგზემპლარი) მხოლოდ იმ შემთხვევაში როდესაც შესაბამისი პროფესორი უკვე არსებობს მონაცემთა ბაზაში. თუ პროფესორი ტოვებს უნივერსიტეტს და პროფესორის ობიექტის შესაბამისი ეგზემპლარი უნდა ამოღებულ იქნას მონაცემთა ბაზიდან, მაშინ, რა თქმა უნდა, მასთან დაკავშირებული მონაცემების შენახვას აღარ ექნება აზრი. მონაცემთა ბაზის ასეთ დამოკიდებულ ობიექტს სუსტი ობიექტი ეწოდება (დამოკიდებული ობიექტი) (“week entity”). ხოლო იმ ობიექტს, რომელზეც დამოკიდებულია სუსტი ობიექტი, მონაცემთა ბაზის ძლიერი ობიექტი (განმსაზღვრელი ობიექტი, მმართველი ობიექტი, მთავარი ობიექტი) ეწოდება (“strong entity”).

სუსტი ობიექტის განსაკუთრებულ შემთხვევას წარმოადგენს ”ID-დამოკიდებული ობიექტი”. ID-დამოკიდებული ობიექტი არის სუსტი ობიექტი, თანაც ისეთი, რომლისთვისაც შესაბამისი ძლიერი ობიექტის ID (იდენტიფიკატორი) წარმოადგენს მისი (ამ სუსტი ობიექტის) იდენტიფიკატორის ნაწილს. წარმოვიდგინოთ ძლიერი ობიექტი „შენობა“ და ID-დამოკიდებული ობიექტი „ოთახი“. ოთახის ატრიბუტები შესაძლებელია მოიცავდეს ზომებს, მაგიდებისა და სკამების რაოდენობას, ფანჯრების რაოდენობას, და ა.შ., მაგრამ ნებისმიერ ოთახს რეალური აზრი გააჩნია მხოლოდ რაიმე შენობის ფარგლებში (კონტექსტში), და ამიტომ ოთახის იდენტიფიცირება (მისი იდენტიფიკატორი) უნდა ითვალისწინებდეს როგორც ოთახის ნომერს ასევე შესაბამისი შენობის სახელს.

ID-დამოკიდებული ობიექტები გამოიყენება იმ შემთხვევაშიც როდესაც ერთი ეგზემპლარის რომელიმე ატრიბუტს მრავალი მნიშვნელობა გააჩნია (ანუ როდესაც საქმე გვაქვს მრავალმნიშვნელიან ატრიბუტებთან (“multivalued” atribute)). მაგალითად, პროფესორს შესაძლებელია გააჩნდეს ერთზე მეტი სხვადასხვა ხარისხი (სერტიფიკატი), ან რამდენიმე ტელეფონის ნომერი. ასეთი მრავალმნიშვნელიანი ატრიბუტების მოდელირება ხორციელდება ID-დამოკიდებული ობიექტების საშუალებით, როდესაც კავშირი ძლიერ და ID-დამოკიდებულ ობიექტებს შორის არის 1:N სახის.

კავშირი შესაძლებელია იყოს რეკურსიული (recursive relationship). ეს ნიშნავს ერთი გარკვეული ობიექტის ეგზემპლართა შორის კავშირების არსებობას. მაგალითად, ჩვენ შესაძლებელია მოვინდომოთ კავშირების შექმნა იმ სტუდენტებს შორის რომლებიც საერთო საცხოვრებლის ერთსა და იმავე ოთახში ცხოვრობენ. ასეთ შემთხვევაში უნდა განისაზღვროს N:M კავშირი „სტუდენტი/იმავე_ოთახში_მცხოვრები_სტუდენტი“ , რაც განახორციელებს იმ ფაქტის მოდელირებას, რომ ერთ ოთახში შესაძლებელია რამდენიმე სტუდენტი ცხოვრობდეს. თუ ყველა ოთახი ორადგილიანია, მაშინ ეს კავშირი იქნება 1:1, ხოლო თუ ზოგიერთი ოთახი უფრო მეტ ადგილს მოიცავს (სამს, ოთხს და ა.შ.) მაშინ ეს კავშირი N:M სახის იქნება. კავშირის ორივე ბოლოს მინიმალური კარდინალობა შესაძლებელია ნულის ტოლი იყოს, თუ არიან ისეთი სტუდენტები რომელთა საცხოვრებელ ოთახებშიც სხვა სტუდენტები არ ცხოვრობენ.

და ბოლოს ზოგი ობიექტები შესაძლებელია წარმოადგენდნენ ქვეკლასებს ან ზეკლასებს (subclass, superclass). მაგალითად, ობიექტი სტუდენტი მოიცავს როგორც ბაკალავრიატის ასევე მაგისტრატურის სტუდენტებს. ჩვენ შეგვიძლია „სტუდენტის“ მოდელირება ზეკლასის სახით, ხოლო „ბაკალავრიატის სტუდენტის“ და „მაგისტრატურის სტუდენტის“ - ქვეკლასების სახით. ამ შემთხვევაში „სტუდენტის“ ატრიბუტები მოიცავენ იმ ატრიბუტებს, რომლებიც ახასიათებს ყველა სტუდენტს, მაგალითად სახელი, მისამართი, დაბადების თარიღი და სხვა. „ბაკალავრიატის სტუდენტების“ ატრიბუტებში გაერთიანდებიან მხოლოდ ის ატრიბუტები რომლებიც დამახასიათებელია ბაკალავრიატის სტუდენტებისათვის, მაგალითად მრჩეველი საყოფაცხოვრებო საკითხებში (რა თქმა უნდა აქ ჩვენ ვგულისხმობთ, რომ მაგისტრატურის სტუდენტებს ასეთი მრჩევლები არ გააჩნიათ).

8-2 სურათზე გამოსახულია სუსტი და ID-დამოკიდებული ობიექტები, მრავალმნიშვნელიანი ატრიბუტები, ზეკლასური და ქვეკლასური ობიექტები.

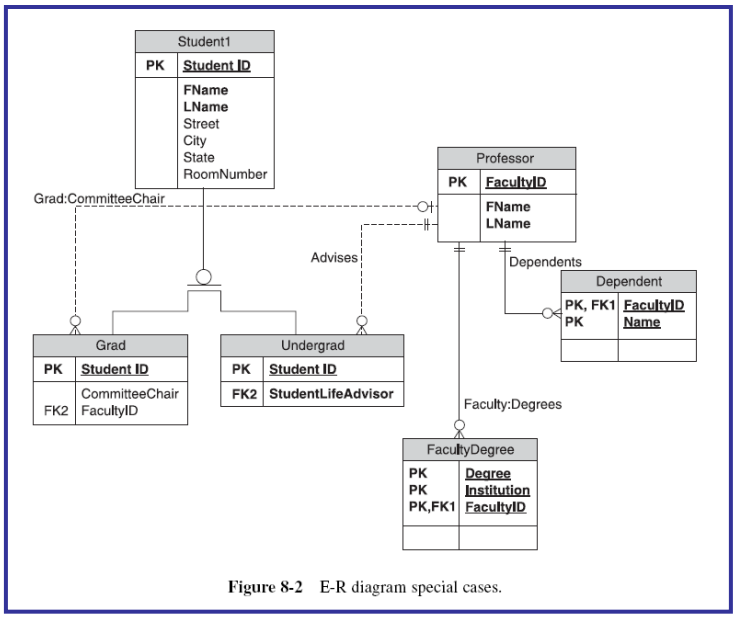


Figure 8-2 E-R diagram special cases.

ამ სურათზე ცხრილი "Dependents" (ოჯახის წევრები) არის id-დამოკიდებული ობიექტი. ამ ცხრილის იდენტიფიკატორი მოიცავს მასთან ასოცირებული "Professor" ძლიერი ობიექტის გასაღებს დამატებული ოჯახის წევრის სახელი (მეუღლის სახელი, შვილის სახელი და ა.შ.). ობიექტი "FacultyDegrees" წარმოადგენს მრავალმნიშვნელიან ატრიბუტს. ფაკულტეტის ცალკეულ წევრს შესაძლებელია გააჩნდეს მრავალი სხვადასხვა ინსტიტუტის მრავალი სხვადასხვა ხარისხი (სერტიფიკატი), და ეს ობიექტი ამ ფაქტის გამოსახვის შესაძლებლობას იძლევა. და ბოლოს, ობიექტი "Student"-ი გვიჩვენებს სტუდენტების ორ ქვეკატეგორიას: მაგისტრატურისა და ბაკალავრიატის სტუდენტებს. ბაკალავრიატის სტუდენტს შესაძლებელია შეესაბამებოდეს ფაკულტეტის წევრი პროფესორი, რომელიც მისი მრჩეველი იქნება სტუდენტური ცხოვრების საკითხებში, ხოლო მაგისტრატურის სტუდენტს შესაძლებელია შეესაბამებოდეს (ან შესაძლებელია არ შეესაბამებოდეს) ფაკულტეტის წევრი პროფესორი, რომელიც მისი თეზისების განმხილველი კომისიის თავმჯდომარე იქნება.

http://en.wikipedia.org/wiki/Data_model

რელაციური მონაცემთა ბაზის აგება მონაცემთა მოდელის საფუძველზე

მონაცემთა მოდელი მოიცავს მონაცემთა ბაზის კონცეპტუალურ სქემას (conceptual schema), ანუ მონაცემთა ბაზის სტრუქტურის აღწერას. ის ერთერთია იმ სამი სქემიდან რომელთანაც საქმე აქვს მონაცემთა ბაზების დიზაინერს (კონსტრუქტორს). დანარჩენ ორ სქემას წარმოადგენენ გარეშე სქემა (external schema), რომელიც ფაქტიურად წარმოადგენს მონაცემთა ბაზის იმ სახეს, რომელსაც აღიქვამს საბოლოო მომხმარებელი, და შინაგანი სქემა (internal schema), რომელიც წარმოადგენს დისკების იმ რეალურ ფიზიკურ ფაილურ სტრუქტურებს რომლებსაც მონაცემთა ბაზების მართვის სისტემები (SQL Server, Oracle და ა.შ.) იყენებენ.

როდესაც მონაცემთა ბაზის კონცეპტუალური სქემა მზად არის, შემდეგი ამოცანაა ამ მონაცემთა სქემის გარდაქმნა ცხრილებში, კავშირებში და მონაცემთა მთლიანობისა და უსაფრთხოების კონკრეტულ წესებში. ასეთი გარდაქმნის შედეგად ობიექტს შეესაბამება ცხრილი, რომლის ყოველი სტრიქონი შეესაბამება მოცემული ობიექტის კონკრეტულ ეგზემპლარს. რელაციური მონაცემთა ბაზების თერმინოლოგიის თანახმად ცხრილს მიმართება (relation) ეწოდება. უნდა მივაქციოთ ყურადღება იმ ფაქტს რომ ინგლისურენოვან ლიტერატურაში სიტყვა "relation" უმეტეს წილად შეესაბამება ცხრილს, და არა კავშირს. კავშირისათვის გამოიყენება სიტყვა "relationship". მოგვიანებით ჩვენ ასევე განვიხილავთ კავშირების დადგენის საშუალებებს. მიმართება შედგება სტრიქონებისაგან (ყოველი მათგანი შეესაბამება მოცემული ობიექტის ერთერთ კონკრეტულ ეგზემპლარს) და სვეტებისაგან (ყოველი მათგანი შეესაბამება მოცემული ობიექტის ერთერთ კონკრეტულ ატრიბუტს (თვისებას, მახასიათებელს)).

მიმართების ყოველ სტრიქონს კორტეჟი (tuple) ეწოდება. პრაქტიკოსები უფრო ხშირად გამოიყენებენ სიტყვას „სტრიქონი" ვიდრე „კორტეჟი", მაგრამ უნდა ვიცოდეთ, რომ კორტეჟი ფაქტიურად ნიშნავს ცხრილის სტრიქონს, ანუ მიმართების ეგზემპლარს, ანუ ობიექტის ეგზემპლარს. ყოველი კორტეჟი შედგება მოცემული ეგზემპლარის შესაბამისი ატრიბუტების მნიშვნელობებისაგან.

ატრიბუტს ხშირად ველსაც უწოდებენ (ველი, field). ამიტომ რელაციურ მონაცემთა ბაზებთან მუშაობისას ჩვენ ყოველთვის უნდა გვახსოვდეს ეს სინონიმები: მიმართება და ცხრილი, კორტეჟი და სტრიქონი, ატრიბუტი და ველი.

პირველ ნაბიჯს წარმოადგენს მონაცემთა მოდელის ყველა ძლიერი (განმსაზღვრელი) ობიექტისათვის შესაბამისი მიმართების შექმნა. ობიექტის თითოეული ატრიბუტი წარმოდგენილი იქნება მიმართების ცალკეული სვეტით, რომელსაც ატრიბუტის შესაბამისი სახელი ენიჭება. ამ დროს საჭიროა შევარჩიოთ ატრიბუტი ან ატრიბუტების ერთობლიობა, რომელსაც პირველადი გასაღები

(primary key) ეწოდება და რომელიც ცალსახად განსაზღვრავს მიმართების ყოველ სტრიქონს (ფაქტიურად პირველადი გასაღების მნიშვნელობა წარმოადგენს სტრიქონის იდენტიფიკატორს ცხრილში).

იდეალური გასაღები უნდა იყოს მოკლე, რიცხვითი სიდიდე და თანაც უცვლელი. იდეალური გასაღების მიღება ყოველთვის არ არის შესაძლებელი, მაგრამ საკმაოდ სასარგებლოა ყოველთვის გვახსოვდეს მისი არსებობის შესაძლებლობა როდესაც ჩვენ ვირჩევთ გასაღებს. მაგალითად, თუ ცხრილი "Student" მოიცავს სხვა ატრიბუტებთან ერთად სახელს, მისამართს და სოციალური დაცვის ნომერს (social security number (SSN)) მაშინ სტუდენტების ცალსახად იდენტიფიცირებისათვის შესაძლებელია ამოვარჩიოთ შემდეგი გასაღებები: შედგენილი გასაღები - ორი ატრიბუტის ერთობლიობა სახელი-მისამართი; ან ერთი ატრიბუტისაგან შედგენილი გასაღები - სოციალური დაცვის ნომერი. სოციალური დაცვის ნომრის შერჩევა უფრო ბრძნული იქნება, რადგანაც ის წარმოადგენს რიცხვითი ტიპის სიდიდეს, რომლის დამუშავება უფრო რაციონალური (ეფექტური) იქნება და თანაც ის გაცილებით უფრო იშვიათად შეიცვლება ვიდრე სახელი ან მისამართი.

ზოგჯერ ცხრილის ატრიბუტებს შორის არ ჩანს ცხადად კარგი გასაღების როლის შემსრულებელი ცალკეული ატრიბუტი. ამ შემთხვევაში შესაძლებელია რამდენიმე ველის მნიშვნელობათა გაერთიანება რათა მივიღოთ ისეთი შედგენილი იდენტიფიკატორი რომელიც ცალსახად განსაზღვრავს ყოველ სტრიქონს. თუ ასეთი გზით მიიღება გრძელი, ანბანურ-რიცხვითი გასაღები, მაშინ მიზანშეწონილია გამოვიყენოთ ხელოვნური გასაღები (სუროგატი გასაღები) (surrogate key, ამ გასაღების მნიშვნელობების გენერირებას ახდენს თვითონ მონაცემთა მართვის სისტემა) (http://en.wikipedia.org/wiki/Surrogate_key). იგი წარმოადგენს რიცხვს, რომელიც გენერირებულია მონაცემთა ბაზების მართვის სისტემის მიერ და ენიჭება ყოველ კორტეჟს. "Student" ცხრილის შემთხვევაში, როდესაც მის ატრიბუტებს შორის არ არის სოციალური დაცვის ნომერი, შესაძლებელია ასეთი გასაღების შექმნა, რომელსაც ვუწოდებთ "StudentID".

შემდეგ ნაბიჯს წარმოადგენს კავშირების შექმნა ყველა სუსტი (დამოკიდებული) და ID-დამოკიდებული ობიექტისათვის. ისევე როგორც ძლიერი (განმსაზღვრელი) ობიექტების შემთხვევაში, სუსტი და ID-დამოკიდებული ობიექტების ატრიბუტები წარმოდგენილი იქნება შესაბამისი მიმართების სვეტებით. სუსტი ან ID-დამოკიდებული ობიექტისათვის საჭიროა დამატებითი სვეტის შემოტანა, რომელშიც შეინახება მისი განმსაზღვრელი ძლიერი ობიექტის კორტეჟის გარე გასაღები (foreign key).

გარე გასაღები წარმოადგენს მიმართების სვეტს, რომელიც ამყარებს ამ მიმართების კავშირს სხვა მიმართებასთან. მაგალითად, წარმოვიდგინოთ, რომ ჩვენი მონაცემთა მოდელი მოიცავს ობიექტს "StudentComputer", და ის წარმოადგენს სუსტ (დამოკიდებულ) ობიექტს, რომელიც დაკავშირებულია მის განმსაზღვრელ "Student" ობიექტთან. ეს ნიშნავს, რომ მონაცემთა ბაზა მოიცავს ინფორმაციას სტუდენტის კომპიუტერის შესახებ მხოლოდ იმ შემთხვევაში, თუ შესაბამისი სტუდენტი უკვე არსებობს მონაცემთა ბაზაში.

სტუდენტის კომპიუტერის ისეთ ატრიბუტებთან ერთად, როგორიცაა კომპიუტერის სერიული ნომერი, მისი სახელი და სხვა, მიმართება "StudentComputer" ასევე უნდა მოიცავდეს დამატებით სვეტს, რომელიც განსაზღვრავს კომპიუტერის მფლობელ სტუდენტს. თუ სოციალური დაცვის ნომერი წარმოადგენს "Student" მიმართების გასაღებს, მაშინ "StudentComputer" მიმართების გარე გასაღები უნდა მოიცავდეს სტუდენტის სოციალური დაცვის ნომრებს. სულაც არ არის სავალდებულო რომ სვეტების სახელები ორივე მიმართებაში ერთნაირი იყოს. ამიტომ, თუ გასაღები-სვეტის სახელი "Student" მიმართებაში არის "SSN" (Social Security Number), მაშინ "StudentComputer" მიმართების სვეტს, რომელიც გარე გასაღებს წარმოადგენს, შესაძლებელია ერქვას "StudentSSN"-ი.

სუსტ ობიექტთან ასოცირებულ ახალ მიმართებაშიც უნდა არსებობდეს საკუთარი პირველადი გასაღები. სუსტი მიმართების პირველადი გასაღების შერჩევა მოითხოვს იგივე სახის პროცედურებს, რომელთაც ადგილი ჰქონდათ ძლიერი ობიექტის მიმართების პირველადი გასაღების შერჩევის დროს. თუ სუსტი ობიექტი ID-დამოკიდებულია ძლიერ ობიექტზე, მაშინ მიზანშეწონილია სუსტი ობიექტის პირველადი გასაღები იყოს შედგენილი და მოიცავდეს როგორც თავის გარე გასაღებს ასევე სხვა, ერთი ან რამდენიმე, ატრიბუტს.

ID-დამოკიდებული ობიექტები ასევე გამოიყენება მრავალმნიშვნელიანი ატრიბუტების მოდელირებისას. მაგალითად, ჩვენ გვინდა ყოველი სტუდენტისათვის მონაცემთა ბაზაში გვქონდეს რამდენიმე მისამართი; მაგალითად ბევრ სტუდენტს ზაფხულში და აკადემიური წლის განმავლობაში სხვადასხვა მისამართები ექნებათ. ასეთ შემთხვევაში ჩვენ ვქმნით ახალ ID-დამოკიდებულ ობიექტს სახელად "Address", რომელსაც შეესაბამება მიმართება ატრიბუტებით "Street", "City", "State" და ასე

შემდეგ. ამასთანავე ეს მიმართება უნდა მოიცავდეს ასევე გარე გასაღების ატრიბუტს, რომელშიც ჩაიწერება “Student” მიმართების პირველად გასაღებში არსებული სისდიდეები.

როდესაც მონაცემთა მოდელის ყველა ობიექტისათვის უკვე შექმნილია შესაბამისი მიმართებები, დგება კავშირების შექმნის დროც. თითოეული 1:1 კავშირისათვის საჭიროა შეირჩეს მიმართებები, რომლებიც შეასრულებენ მშობლისა და შვილის როლებს. კავშირის განსახორციელებლად მიმართებაში, რომელიც შვილის როლს ასრულებს, უნდა შეიქმნას გარე გასაღების სვეტი, რომელიც ამ მიმართების თითოეულ კორტეჟს დააკავშირებს მშობელი მიმართების შესაბამის კორტეჟთან.

თუ 1:1 კავშირის ორივე მხრის მინიმალური კარდინალობა 1-ის ტოლია, მაშინ არა აქვს მნიშვნელობა რომელი მიმართება ასრულებს მშობელი მიმართების როლს. ხოლო იმ შემთხვევაში, როდესაც ერთერთი მხრის მინიმალური კარდინალობა 0-ის ტოლია, მაშინ ის მიმართება, რომელიც მდებარეობს კავშირის მეორე მხარეს, უნდა იყოს მშობელი მიმართებს. მაგალითად, თუ არსებობს 1:1 კავშირი “Room” (ოთახი) და “Projector” (პროექტორი) მიმართებებს შორის, მაგრამ პროექტორი არ არის ყველა ოთახში, მაშინ ჩვენ “Room” მიმართებას უნდა მივანიჭოთ მშობელი მიმართების როლი და ამიტომ გარე გასაღების სვეტი უნდა გავითვალისწინოთ (ჩავსვათ) “Projector” მიმართებაში. ასეთი არჩევანი უფრო ეფექტურია მონაცემთა ციფრული მატარებლების მოცულობის გამოყენების თვალსაზრისით, რადგანაც გარე გასაღების ველი მხოლოდ მაშინ შეივსება როდესაც იარსებებს კორტეჟი “Projector” მიმართებაში, რომელიც უნდა დაკავშირდეს “Room” მიმართების კორტეჟთან.

1:N კავშირისათვის მიმართება, რომელიც 1-ს მხარეს არის განთავსებული, უნდა იყოს მშობელი მიმართება, ხოლო კავშირის მეორე მიმართება, რომელიც N-ს მხარეს მდებარეობს, უნდა იყოს შვილი მიმართება. ამ შემთხვევასი საჭიროა შვილ მიმართებას დავამატოთ გარე გასაღების სვეტი ისე რომ შესაძლებელი იყოს მრავალი შვილი ეგზემპლარის დაკავშირება ერთსა და იმავე მშობელ ეგზემპლართან. მაგალითად, მრჩეველი/სტუდენტი კავშირის განსახორციელებლად მიმართებას “Student” უნდა დაემატოს გარე გასაღების სვეტი, რომლის სახელი, მაგალითად, იქნება “FacultyAdvisor”, და განვათავსოთ ამ სვეტში “Faculty” მიმართების პირველადი გასაღების სიდიდეები.

კავშირი მრავალი-მრავალთან უფრო რთული სახის კავშირია. იმისათვის რომ განვახორციელოთ კავშირი N:M საჭიროა ახალი დამატებითი ცხრილის შექმნა, ანუ ახალი მიმართების შექმნა. ასეთ ახალ მიმართებას ხშირად გადაკვეთის ცხრილი ან კავშირის მიმართება ეწოდება. გადაკვეთის ცხრილი მოიცავს კავშირში მონაწილე ორივე ობიექტის გარე გასაღებთა სვეტებს. გადაკვეთის ცხრილის ყოველი კორტეჟი უნდა მოიცავდეს ორივე მიმართების პირველადი გასაღების სიდიდეებს. ორი ობიექტის ეგზემპლართა შორის არსებულ თითოეულ ასოციაციას უნდა შეესაბამებოდეს გადაკვეთის ცხრილის კონკრეტული ერთადერთი სტრიქონი, რომელიც ამ კონკრეტულ ეგზემპლართა შორის კავშირს ახორციელებს.

მაგალითად, იმისათვის, რომ განვახორციელოთ M:N კავშირი “Student” და “Course” ცხრილებს (მიმართებებს) შორის, უნდა შეიქმნას ახალი მიმართება “StudentCourseIntersection”. “StudentCourseIntersection” მიმართებას უნდა გააჩნდეს გარე გასაღებთა სვეტები, რომლებიც გათვალისწინებული იქნება “Student” და “Course” მიმართებებისათვის, მაგალითად, სვეტი სახელით StudentSSN პირველი მიმართებისათვის და სვეტი სახელით CourseNumber მეორე მიმართებისათვის. StudentCourseIntersection მიმართების ყოველი სტრიქონი რასაკვირველია დააფიქსირებს გარკვეული სტუდენტის მიერ გარკვეული კურსის აღების (არჩევის) ფაქტს. რომელიმე კონკრეტულმა სტუდენტმა შესაძლებელია აირჩიოს მრავალი კურსი, და პირიქით, , შესაძლებელია რომელიმე გარკვეული კურსი აირჩიოს მრავალმა სტუდენტმა.

გადაკვეთის ცხრილის პირველადი გასაღები ჩვეულებრივ წარმოადგენს შედგენილ გასაღებს, რომელიც ფაქტიურად ორი გარე გასაღების სიდიდეთა ერთობლიობაა. რადგანაც გარე გასაღების სიდიდეები შესაბამისი ცხრილების კორტეჟებისათვის წარმოადგენენ უნიკალურ სიდიდეებს, მაშინ, რა თქმა უნდა, ამ ორი გასაღების კომბინაცია აუცილებლად იქნება უნიკალური გადაკვეთის ცხრილში. ეს ზოგადი წესი შესაძლებელია შეიცვალოს მხოლოდ სპეციალურ შემთხვევებში. მაგალითად, თუ ჩვენ გვინდა ჩავიწეროთ სტუდენტის ყველა მცდელობები გარკვეული კონკრეტული კურსის არჩევის მცდელობების შესახებ, მაშინ გადაკვეთის ცხრილის პირველადი გასაღები უნდა გაფართოვდეს, რათა ჩავრთოთ მასში კიდევ ერთი დამატებითი ატრიბუტი, რომელიც ერთი რომელიმე კონკრეტული კურსის ერთი რომელიმე კონკრეტული სტუდენტის მიერ არჩევის სხვადასხვა მცდელობების ერთმანეთისაგან გარჩევის შესაძლებლობას მოგვცემს.

რეკურსიული კავშირების შექმნაც არ წარმოადგენს რთულ საქმეს. დავუშვათ, რომ ზოგიერთი სტუდენტი ასრულებს მრჩევლის ფუნქციებს სხვა სტუდენტებისათვის. კავშირი არის 1:N სახის. ასეთი რეკურსიული კავშირის განხორციელება შესაძლებელია გადაიჭრას ერთი სვეტის დამატებით

მიმართება “Student”-ში, რომლის სახელი იქნება, მაგალითად, “StudentAdvisor”. ფაქტიურად StudentAdvisor სვეტი იქნება გარე გასახების სვეტი, რომელშიც განთავსებული იქნება ამავე მიმართების პირველადი გასაღების სიდიდეები. 1:N რეკურსიული კავშირის შექმნა ფაქტიურად ზუსტად იგივეა რაც სტანდარტული 1:N კავშირის შექმნა, განსხვავება მხოლოდ იმაშია, რომ მშობელი გარე გასაღები უკავშირდება იგივე ცხრილს რომელიც მოიცავს შვილ ეგზემპლარებს. ანალოგიურად იქმნებიან 1:1 ტიპის რეკურსიული კავშირებიც.

M:N რეკურსიული კავშირის განხორციელება მოითხოვს გადაკვეთის ცხრილის შექმნას, ზუსტად ისევე როგორც ამას ჰქონდა ადგილი არარეკურსიული სტანდარტული M:N კავშირების შემთხვევაში. მაგრამ იმ განსხვავებით, რომ რეკურსიული კავშირის შემთხვევაში გარე გასაღებთა ორივე სვეტში განთავსდებიან იმავე მიმართების პირველადი გასაღების სიდიდეები. წარმოვიდგინოთ კავშირი სტუდენტებს შორის რომლებიც ერთსა და იმავე ოთახში ცხოვრობენ. გადაკვეთის ცხრილის თითოეული სტრიქონი წარმოადგენს კავშირს ერთსა და იმავე ოთახში მცხოვრებ ორ სტუდენტს შორის.

ნორმალიზაცია

მონაცემთა მოდელები ერთმანეთისაგან განსხვავდებიან ხარისხით. კერძოდ, თუ ობიექტების განსაზღვრა თავიდანვე არ იყო კარგად გაანალიზებული მაშინ მონაცემთა ბაზაში შესაძლებელია არსებობდეს მონაცემთა სიჭარბე (data redundancy) ან განახლების ანომალიები (update anomalies), ან ორივე ერთად. განახლების ანომალია გულისხმობს ისეთ ქცევას, როდესაც რაიმე კონკრეტული ობიექტის შესახებ (მაგალითად სტუდენტის შესახებ) ინფორმაციის შეტანა მოითხოვს სხვა ობიექტის შესახებ ინფორმაციის ცოდნას (მაგალითად სტუდენტური სასტუმროს შესახებ), ან რაიმე ობიექტის ეგზემპლარის ამოშლა მონაცემთა ბაზიდან (მაგალითად, ბოლო სტუდენტის ამოშლა სასტუმროდან) იწვევს ინფორმაციის დაკარგვას სხვა ობიექტის შესახებ (მაგალითად, იკარგება ინფორმაცია შესაბამისი სტუდენტური სასტუმროს შესახებ). ნორმალიზაცია არის პროცესი რომელიც მიმართებათა ტესტირებას ახორციელებს. ტესტირების ჩატარებამ უნდა გვაჩვენოს რომ მონაცემთა ბაზას გააჩნია ჩვენს მიერ დაგეგმილი თვისებები და ფუნქციონალობა.

ნორმალიზაციის მიზანი მდგომარეობს იმაში, რათა დავრწმუნდეთ, რომ ყველა მიმართება წარმოადგენს ცალკეულ თემას. მაგალითად, მიმართება შესაძლებელია მოიცავდეს ინფორმაციას სტუდენტების შესახებ, მიმართება შესაძლებელია მოიცავდეს ინფორმაციას სტუდენტურ სასტუმროს შესახებ, მაგრამ მიმართება რომელშიც თავმოყრილია ერთდროულად ინფორმაცია სტუდენტებზეც და სტუდენტურ სასტუმროზეც შექმნის პრობლემებს.

რელაციურ მონაცემთა ბაზებისათვის არსებობს რამდენიმე ნორმალური ფორმა. ნორმალიზაციები განსხვავდებიან დონით. რაც უფრო მაღალია ნორმალიზაციის დონე მით უფრო შემცირებულია მონაცემთა სიჭარბე და განახლების ანომალიები. ნებისმიერი უფრო მაღალი დონის ნორმალური ფორმა აკმაყოფილებს ნებისმიერ უფრო დაბალი დონის ნორმალურ ფორმას. მაგალითად, მიმართება მესამე ნორმალურ ფორმაში (third normal form = 3NF) ავტომატურად იმყოფება მეორე (second normal form = 2NF) და პირველ ნორმალურ (first normal form = 1NF) ფორმებშიც.

ნორმალური ფორმების განხილვა ეფუძნება ფუნქციონალური დამოკიდებულების კონცეფციას. როდესაც ერთი ატრიბუტის ან ატრიბუტთა ჯგუფის მნიშვნელობა განსაზღვრავს სხვა ატრიბუტის მნიშვნელობას, ამბობენ, რომ მათ შორის არსებობს ფუნქციონალური დამოკიდებულება, თანაც პირველ (განმსაზღვრელ) ატრიბუტს ან ატრიბუტთა ჯგუფს დეტერმინანტი ეწოდება.

წარმოვიდგინოთ, რომ ჩვენ შევქმენით მიმართება იმ ატრიბუტებით, რომელიც ნაჩვენებია სურათზე Figure 8.3.

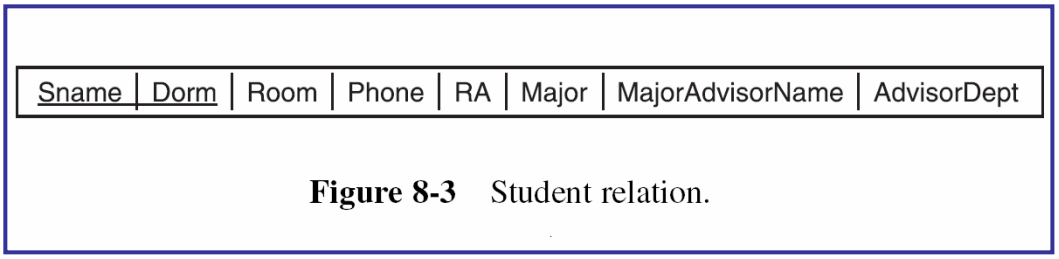


Figure 8-3 Student relation.

ამ Student მიმართების შედგენილი გასაღები მოიცავს ატრიბუტებს Sname და Dorm (ჩვენ ვგულისხმობთ რომ ერთი და იგივე სახელისა და გვარის სტუდენტები არ იცხოვრებენ ერთსა და იგივე სტუდენტურ საერთო საცხოვრებელში).

ჰორიზონტალური მართკუთხედი წარმოადგენს მიმართების გრაფიკულად გამოსახვის ერთერთ ფართოდ გავრცელებულ ხერხს, ვერტიკალური ხაზები გამოყოფენ ატრიბუტების სახელებს და ქვემოდან ხაზგასმული ატრიბუტები შეადგენენ გასაღებს. სულაც არ არის სავალდებულო, რომ გასაღებში შემავალი ატრიბუტები ერთმანეთის თანმიყოლებით ან აუცილებლად სურათის მარცხენა ნაწილში იმყოფებოდნენ, მაგრამ უმეტეს შემთხვევაში მათ ასეთი სახით წარმოადგენენ სურათებზე.

ნებისმიერი მიმართების გასაღები ყოველთვის წარმოადგენს დეტერმინანტს; განმარტების თანახმად გასაღები განსაზღვრავს მთლიან კორტეჟს (ანუ კორტეჟის იდენტიფიცირებას ახდენს). თუ Student მიმართებაში ნებისმიერი კორტეჟისათვის ცნობილია Sname და Dorm ატრიბუტების მნიშვნელობები, მაშინ შესაბამისი კორტეჟის სხვა ატრიბუტების მნიშვნელობები ცალსახადაა განსაზღვრული.

მაგრამ, ყველა დეტერმინანტი არ წარმოადგენს გასაღებს. Student მიმართებაში არსებობს ფუნქციონალური დამოკიდებულება MajorAdvisorName და AdvisorDept ატრიბუტებს შორის. თუ მოცემულია მრჩეველის სახელი, მაშინ დეპარტამენტის მნიშვნელობა განსაზღვრულია.

პირველი ნორმალური ფორმა ფაქტიურად წარმოადგენს მიმართების დეფინიციას (განსაზღვრებას). ყველა ატრიბუტი უნდა იყოს ატომური სახის და ერთმნიშვნელიანი. მაგალითად, თუ Student მიმართების ერთერთ ატრიბუტს წარმოადგენს TelephoneNumber, მაშინ მიმართების თითოეულ კორტეჟს უნდა ჰქონდეს ამ ატრიბუტის მხოლოდ ერთი მნიშვნელობა. იმ შემთხვევაში თუ ჩვენ გვესაჭიროება თითოეული სტუდენტის მრავალი სხვადასხვა ტელეფონის ნომრების ჩაწერა მონაცემთა ბაზაში, მაშინ ამისათვის ჩვენ უნდა შევქმნათ ცალკე PhoneNumber მიმართება. მაშინ ამ ახალი PhoneNumber მიმართების ყოველ კორტეჟს შეუძლია გააჩნდეს ერთადერთი ტელეპჰონის ნომერი, და 1:N კავშირის გამოყენებით ჩვენ შეგვიძლია გარკვეულ სტუდენტთან დავაკავშიროთ PhoneNumber მიმართების მრავალი კორტეჟი.

მეორე ნორმალური ფორმა მოითხოვს, რომ თითოეული არაგასაღებრივი ატრიბუტი (ანუ გასაღებში არ შემავალი ატრიბუტი) უნდა იყოს ფუნქციონალურად დამოკიდებული მთლიან გასაღებზე. სხვა სიტყვებით რომ ვთქვათ, თითოეული არაგასაღებრივი ატრიბუტის მნიშვნელობა უნდა იძლეოდეს ფაქტს მთლიანი ეგზემპლარის შესახებ, და არა ფაქტს გასაღების ნაწილის შესახებ. თუ მიმართების გასაღები წარმოდგენილია ერთადერთი ატრიბუტის მნიშვნელობებით (მაგალითად, მიმართებას გააჩნია ხელოვნური (სუროგატი) გასაღები (surrogate key)) მაშინ მიმართება ავტომატურად იმყოფება მეორე ნორმალურ ფორმაში (second normal form = 2NF). ამ შემთხვევაში ნებისმიერი ატრიბუტი, რომელიც არ შედის პირველად გასაღებში, პირველად გასაღებზე დამოკიდებულ სიდიდეს წარმოადგენს (ანუ განისაზღვრება პირველადი გასაღებით).

ერთი შეხედვით შესაძლებელია ჩავთვალოთ, რომ Student მიმართებას გააჩნია მონაცემთა ბაზისათვის მისაღები სახის სტრუქტურა, რომელშიც განთავსდება ინფორმაცია სხვადასხვა სტუდენტის შესახებ. მაგრამ უფრო დეტალური შესწავლის შედეგად დავინახავთ, რომ ეს მიმართება ერთდროულად მოიცავს ინფორმაციას რეზიდენტური მრჩევლების შესახებ (RA), ფაკულტეტის მრჩევლების შესახებ და სტუდენტების შესახებ.

არის თუ არა Student მიმართების ყოველი არაგასაღებრივი ატრიბუტი დამოკიდებული მთლიანი გასაღების მნიშვნელობაზე? ჩვენს შემთხვევაში პასუხი იქნება „არა“. ვთქვათ ყოველ სტუდენტურ საერთო საცხოვრებელში მუშაობს მხოლოდ ერთი რეზიდენტი მრჩეველი RA, რომელიც ამ სატუმროს ყველა სტუდენტს ემსახურება, მაშინ RA-ს მნიშვნელობა დამოკიდებული იქნება სტუდენტურ საერთო საცხოვრებელზე Dorm და არა სტუდენტზე Sname. იმისათვის, რომ სქემა მოვიყვანოთ მეორე ნორმალურ ფორმაში 2NF ჩვენ უნდა შევქმნათ ახალი მიმართება Dorm (სტუდენტური საერთო საცხოვრებელი) და ამოვიღოთ RA ატრიბუტი მიმართებიდან Student (იხილე სურათი Fig. 8.4).

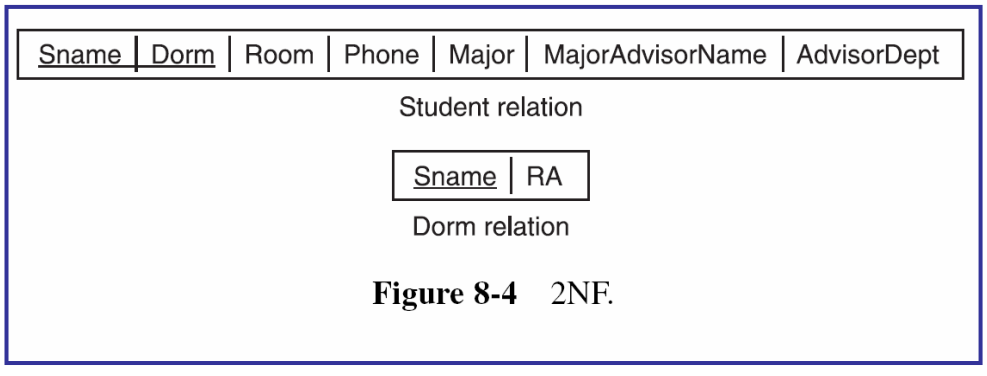


Figure 8-4 2NF.

ეს უკვე პროგრესია, მაგრამ მიმართება Student ჯერ კიდევ მოიცავს ისეთ ინფორმაციას რომელიც არ არის სტუდენტის შესახებ. კერძოდ, მიმართებაში არის ინფორმაცია მრჩევლების

შესახებ: ვინ არის კონკრეტული სტუდენტის მრჩეველი და რომელ დეპარტამენტს მიეკუთვნება იგი. თუ მიმართება უნდა იყოს ფოკუსირებული ერთ თემაზე, მაშინ ასეთი სიტუაცია არ არის კარგი.

მესამე ნორმალური ფორმა მოითხოვს, რომ მიმართებას არ უნდა გააჩნდეს ტრანზიტული დამოკიდებულებები (transitive dependencies). სხვა სიტყვებით შეიძლება ითქვას, რომ ყოველი არაგასაღებრივი ატრიბუტი უნდა იძლეოდეს ფაქტს მთლიანი ეგზემპლარის შესახებ და არა რომელიმე სხვა არაგასაღებრივ ატრიბუტზე. ეს არის ის პირობა რომელიც ჯერ კიდევ ირღვევა ჩვენს მიმართებაში Student. ორი ატრიბუტი MajorAdvisorName და AdvisorDept დამოკიდებულია Student მიმართების გასაღებზე, მაგრამ AdvisorDept ასევე ტრანზიტულად არის დამოკიდებული MajorAdvisorName ატრიბუტზე. მართლაც, თუ ჩვენ ავირჩევთ რომელიმე კონკრეტულ სტუდენტს, მაშინ ჩვენ შეგვიძლია განვსაზღვროთ ამ სტუდენტის მრჩეველი, და თუ ჩვენ ვიცით ვინ არის მრჩეველი მაშინ ჩვენ შეგვიძლია განვსაზღვროთ მრჩევლის დეპარტამენტიც. ეს არის ტრანზიტული დამოკიდებულება: A განსაზღვრავს B-ს, და B განსაზღვრავს C-ს. იმისათვის, რომ სქემა შეესაბამებოდეს მესამე ნორმალურ ფორმას 3NF, ჩვენ უნდა მოვიცილოთ ეს ტრანზიტული დამოკიდებულება Student მიმართებიდან. სურათზე Fig. 8.5 გამოსახულია სქემა რომელიც იმყოფება მესამე ნორმალურ ფორმაში 3NF.

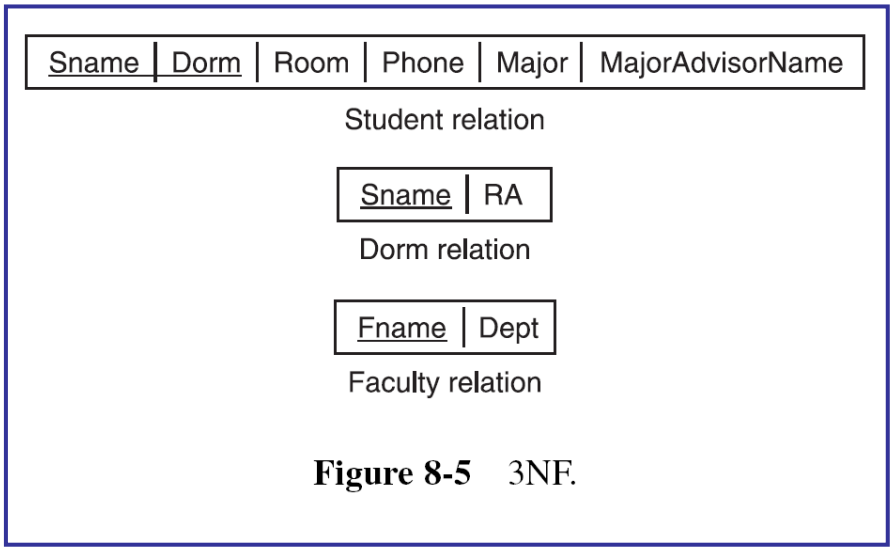


Figure 8-5 3NF.

ახლა საწყისი მიმართება Student გაყოფილია სამ მიმართებად; თითოეული მათგანი აღწერს ცალკეულ ობიექტს (მონაცემთა თემას) - ერთი მოიცავს ინფორმაციას სტუდენტების შესახებ, მეორე მოიცავს ინფორმაციას სტუდენტურ საერთო საცხოვრებლების შესახებ, ხოლო მესამე - ინფორმაციას ფაკულტეტის წევრებზე, რომლებიც მრჩევლებად მუშაობენ.

უფრო დახვეწილ რეალიზაციის შემთხვევაში მიმართებებში Student და Faculty საჭიროა შეირჩეს უფრო კარგი გასაღებრივი მნიშვნელობები. შესაძლებელია შეირჩეს რაიმე სახის ID რიცხვი, იმის ნაცვლად რომ ავირჩიოთ სახელის ატრიბუტი და ვიყოთ იმის იმედად რომ მონაცემთა ბაზაში არ შეგვხვდება ორი ერთნაირი სახელი, მაგალითად, ორი John Smith. ამასთანავე ალბათ Faculty მიმართებაში გვექნება სხვა დამატებითი ატრიბუტებიც, მაგალითად ოფისის მისამართი, ხელფასი, და სხვა.

ბოის-კოდის ნორმალური ფორმა (Boyce-Codd normal form = BCNF) წარმოადგენს მესამე ნორმალური ფორმის რაფინირებას (დახვეწას). მიმართება იმყოფება ბოის-კოდის ნორმალურ ფორმაში, თუ მიმართების ყოველი დეტერმინანტი წარმოადგენს მიმართების პოტენციურ გასაღებს (შესაძლო გასაღებს). ნებისმიერ პოტენციურ გასაღებს შეუძლია შეასრულოს მიმართების პირველადი გასაღების როლი.

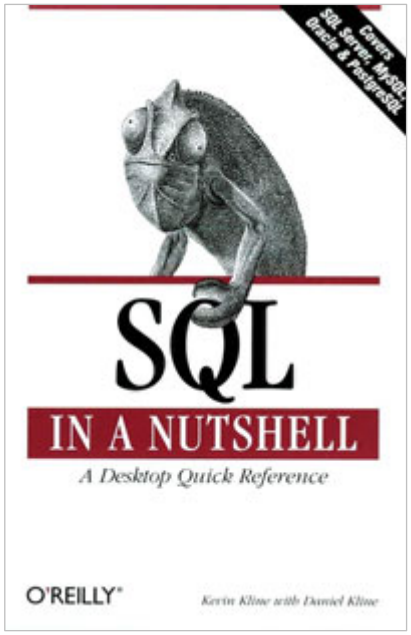
წარმოვიდგინოთ, რომ ჩვენს მაგალითში ოთახების ნუმერაცია სხვადასხვა სტუდენტურ სასტუმროებში განსხვავებულია, თანაც ისეთია რომ ოთახის ნომერი ცალსახად განსაზღვრავს სტუდენტურ სასტუმროს რომელსაც ის მიეკუთვნება (მაგალითად ოთახის ნომრები რომლებიც 100 ნაკლებია მიეკუთვნებოდნენ სტუდენტურ სასტუმროს „საბურთალო“, ხოლო ოთახის ნომრები 100-დან 199-მდე - სტუდენტურ სასტუმროს „ვაკე“, და ასე შემდეგ). მაშინ ატრიბუტი Room იქნება დეტერმინანტი (განმსაზღვრელი ატრიბუტი), მაგრამ ცხადია, რომ ის არ წარმოადგენს პოტენციური გასაღები. ამიტომ ჩვენი მიმართება Student არ აკმაყოფილებს ბოის-კოდის ნორმალურ ფორმას. იმისათვის, რომ მიმართება Student აღმოჩნდეს ბოის-კოდის ნორმალურ ფორმაში, ჩვენ უნდა შევქმნათ კიდევ ერთი დამატებითი მიმართება Room რომელშიც ატუდენტური სასტუმროს ოთახის ნომერი იქნებოდა პირველადი გასაღები, ხოლო სტუდენტური სასტუმრო იქნებოდა არაგასაღებრივი ატრიბუტი.

არსებობენ უფრო მაღალი დონის ნორმალური ფორმებიც: მეოთხე ნორმალური ფორმა, მეხუთე ნორმალური ფორმა. ყოველდღიურ პრაქტიკულ მოღვაწეობაში მაღალი დონის ნორმალური

ფორმის გამოყენების აუცილებლობა საკმაოდ იშვიათია, ამიტომ ეს ნაწილი მთავრდება ბოის-კოდის ნორმალური ფორმის განხილვით. მნიშვნელოვანია ყოველთვის გვახსოვდეს, რომ მონაცემთა ბაზის თითოეული მიმართება უნდა მოიცავდეს მხოლოდ ცალკეულ თემას, ცალკეულ საგანს, ობიექტს.

სტრუქტურირებული მიმართვების ენა (SQL – Structured Query Language)

IBM-მა პირველმა შემოიტანა SQL-ი მონაცემთა ბაზების დამუშავებაში. ის წარმოადგენს მაღალი დონის ენას, რომლითაც შესაძლებელია მონაცემთა ბაზების შექმნა, მონაცემთა მანიპულირება, მონაცემთა გარკვეული სიმრავლეების მოძიება-ამოკრეფა და სხვა. SQL წარმოადგენს არაპროცედურულ ენას (ეს ნიშნავს, რომ SQL-ბრძანებები აღწერენ ჩვენთვის საინტერესო სახის მონაცემებსა და ოპერაციებს, მაგრამ იმავდროულად ისინი არ აკონკრეტებენ დეტალებში მონაცემთა ბაზის მოქმედებებს ჩვენი მოთხოვნების დაკმაყოფილების პროცესში)(http://en.wikipedia.org/wiki/Computer_language,).



ANSI (American National Standards Institute) სტანდარტები SQL-ისათვის გამოქვეყნებულ იქნა 1986, 1989, 1992, 1999 და 2003 წლებში. პრაქტიკულად მონაცემთა ბაზების სხვადასხვა მწარმოებლები გვთავაზობენ SQL-ს, რომლებიც მცირედ განსხვავდებიან ერთმანეთისაგან სინტაქსისა და სემანტიკის მიხედვით. ნებისმიერი კონკრეტული მწარმოებლის ვერსიის SQL-ბრძანებათა უმრავლესობა აკმაყოფილებს სტანდარტებს, მაგრამ იმავდროულად შესაძლებელია მოიცავდეს მრავალ მცირე განსხვავებებსაც. ამის გამო როგორც წესი ჩვენ სტანდარტული SQL-ის ცოდნას ყოველთვის უნდა მივახმაროთ გამოყენებული მონაცემთა ბაზისათვის გათვალისწინებული ინფორმაცია მისი მწარმოებლისაგან. მაგალითად კევინ კლაინის (Kevin Kline) სამაგიდო წიგნში "SQL in a Nutshell" მოყვანილია განცალკევებული თავები სხვადასხვა სტანდარტებისათვის ANSI Standard, DB2 (IBM), MySQL (open source), Oracle, SQL Server (Microsoft).

SQL-ბრძანებებს ხშირად ორ ჯგუფს მიაკუთვნებენ: მონაცემთა განსაზღვრის ენას (Data Definition Language = DDL) და მონაცემთა მანიპულირების (გარდაქმნის) ენა (Data Manipulation Language = DML). DDL გამოსახულებები ქმნიან მონაცემთა ბაზების სტრუქტურებს: ცხრილებს (table), ასახვებს (წარმოდგენებს) (view), ტრიგერებს (trigger) (პროცედურები რომლებიც ინახებიან მონაცემთა ბაზების ობიექტებთან ერთად და რომლებიც ავტომატურად სრულდებიან გარკვეული SQL ბრძანებების გამოყენებისას; ისინი იმართებიან მოვლენებით და უხილავი რჩებიან მომხმარებლისათვის) და პროცედურებს (procedure). DML გამოსახულებები მონაცემთა ბაზაში ასრულებენ შემდეგ მოქმედებებს: მონაცემების შეტანა (insert data), მონაცემების განახლება (update data), მონაცემების ძიება (retrieve data), მონაცემების წაშლა.

SQL-ი არ განასხვავებს ანბანის დიდ და პატარა სიმბოლოებს (მაგალითად, ჩანაწერები MicroSoft, microsoft და MICROSOFT მისთვის ერთსა და იგივეს ნიშნავს). მაგრამ ზოგიერთ მომხმარებელს (პროგრამისტს, Soft-ინჟინერს) დიდი და პატარა ასოების გამოყენების საკუთარი სტილი გააჩნია რომელიც გამოარჩევს SQL-ის დარეზერვირებილ სიტყვებს მონაცემთა ბაზებისა და მისი ობიექტების სახელებისაგან.

მონაცემთა განსაზღვრის ენა (DDL – Data Definition Language)

DDL-ის ერთერთ ბრძანებას წარმოადგენს CREATE. დირექტივა CREATE TABLE ქმნის მიმართვას. SQL-ის თერმინოლოგიის მიხედვით მიმართვას ეწოდება ცხრილი, კორტეჟს - სტრიქონი, ხოლო ატრიბუტს - სვეტი. ქვემოთ მოყვანილია ამ ბრძანების სინტაქსი

```
CREATE TABLE <table_name>
(
<column_name> <dataType> <attributes>
[,<column_name> <dataType> <attributes>]
[CONSTRAINT [<constraintName>] <constraintType>
[,CONSTRAINT [<constraintName>] <constraintType>]]
```

```
);
```

ამ სინტაქსის მიხედვით ბრძანება უნდა იწყებოდეს დარეზერვირებული სიტყვებით CREATE TABLE, რომლის შემდეგაც იწერება ცხრილის სახელი (ცხრილის სახელს ირჩევს პროგრამისტი). ცხრილის სახელი ნაჩვენებია ნაკლებობისა და მეტობის ნიშნებს შორის. ცხრილის სახელის შემდეგ იწერება გახსნილი ფრჩხილი, რომელსაც მოყვება სვეტების სახელების, სვეტების მონაცემთა ტიპების და თითოეული სვეტის მახასიათებლების დეტალიზაცია (სპეციფიკაცია). სვეტების სახელების შემდეგ ჩვენ შეგვიძლია დამატებით განვსაზღვროთ ერთი ან მეტი შეზღუდვის პირობა (CONSTRAINT), მისი სახელი და ტიპი (მაგალითად, PRIMARY KEY ან UNIQUE). ბოლოს უნდა ჩაიწეროს დახურვის ფრჩხილი და წერტილ-მძიმე (semicolon).

მონაცემთა ბაზის დიზაინერი აბსოლუტურად თავისუფალია ცხრილების, სვეტების და შეზღუდვის პირობების სახელების შერჩევაში. SQL-ის სტანდარტი განსაზღვრავს წესებს სახელებისათვის, მაგრამ მონაცემთა ბაზების ყოველ მწარმოებელს გააჩნია თავისი საკუთარი წესები, რომლებიც ცოტა განსხვავებულია სტანდარტისაგან. მაგალითად, სტანდარტი SQL2003 ამბობს, რომ სახელები შესაძლებელია მოიცავდნენ 128 სიმბოლოს, მაგრამ მონაცემთა მართვის სისტემაში MySQL ეს რიცხვი შემოსაზღვრულია 64 სიმბოლოთი, ხოლო Oracle-ში - 30 სიმბოლოთი.

SQL მონაცემთა ტიპები განსხვავდებიან მონაცემთა ბაზების მართვის სისტემების მიხედვით. ზოგადად შესაძლებელია შემდეგი ტიპების არსებობა:

- Integer (მთელი რიცხვითი ტიპი)
- Number/Numeric (decimal floating point)(რიცხვითი ტიპი) (ათობითი წილადი მცურავი წერტილით)
- Varchar(variable length character strings) (ცვლადი სიგრძის ტექსტური სტრიქონების ტიპი)
- Date/DateTime (თარიღისა და თარიღი-დროის ტიპი)
- Char (character string of fixed length) (ფიქსირებული სიგრძის ტექსტური სიმბოლოების სტრიქონი)

მონაცემთა კორექტული ტიპების განსაზღვრელად აუცილებელია ჩვენი კონკრეტული მონაცემთა ბაზის მართვის სისტემის დოკუმენტაციასთან გაცნობა.

ყველაზე უფრო ხშირად სვეტებისათვის მონაცემთა ბაზების დიზაინერები განსაზღვრავენ შემდეგ თვისებებს: NOT NULL (ველი სავალდებულოა შეივსოს), DEFAULT (ავტომატური მნიშვნელობა), CONSTRAINT (მნიშვნელობის შეზღუდვა, პირობა მნიშვნელობისათვის). როდესაც ჩვენ ვირჩევთ სვეტისათვის (მიმართების ატრიბუტისათვის) თვისებას NOT NULL , მაშინ შესაბამის სვეტში ყველა ახალი დამატებული სტრიქონისათვის სავალდებულოა გარკვეული მნიშვნელობის არსებობა, ანუ ეს ამ ველში ყოველთვის არის საჭირო მნიშვნელობის შეტანა. თუ ეს თვისება არ იქნება განსაზღვრული, მაშინ შესაბამის სვეტში შესაძლებელია არსებობდეს ნულოვანი სიდიდე, ანუ სტრიქონის შევსებისას შესაბამის სვეტში შესაძლებელია არავითარი მონაცემი არ შევიტანოთ.

თვისება DEFAULT იძლევა გამოსახულების დაფიქსირებას, რომლის მიხედვითაც ცხრილში ახალი სტრიქონის დამატებისას მონაცემთა ბაზის მართვის სისტემა ავტომატურად შეიტანს შესაბამის მნიშვნელობას იმ შემთხვევაში თუ ჩვენ ხელით არ შეგვაქვს რაიმე სხვა საჭირო მნიშვნელობა. მაგალითად ქვემოთ მოყვანილი დეკლარაცია შტატის შესაბამისი სვეტისათვის განსაზღვრავს “NY” მნიშვნელობას default-რეჟიმში:

```
State Char(2) DEFAULT 'NY',
```

არსებობს ოთხი შეზღუდვა, რომელიც შესაძლებელია განისაზღვროს: PRIMARY KEY, FOREIGN KEY, UNIQUE და CHECK.

შეზღუდვა PRIMARY KEY განსაზღვრავს იმ სვეტს ან სვეტებს რომლებიც შეადგენენ პირველად გასაღებს.

FOREIGN KEY წარმოადგენს მექანიზმს რომელიც ქმნის კავშირებს სხვადასხვა ცხრილების სტრიქონებს შორის. (სტრიქონებს ხშირად ჩანაწერებსაც (record) უწოდებენ).

შეზღუდვა UNIQUE მოითხოვს რომ ცხრილის ყველა სტრიქონში შესაბამის სვეტში ან სვეტებში იმყოფებოდეს უნიკალური (ერთმანეთისაგან განსხვავებული) მნიშვნელობები. ამ შეზღუდვას ზოგჯერ საკანდიდატო (შესაძლო) გასაღებს (candidate key) უწოდებენ, რადგანაც ის სვეტები რომლებიც მოიცავენ უნიკალურ მნიშვნელობებს შესაძლებელია გამოვიყენოთ როგორც ცხრილის პირველადი გასაღები უკვე ცხრილის არსებული პირველადი გასაღების ნაცვლად.

ქვემოთ მოყვანილია ცხრილის შემქმნელი CREATE TABLE ბრძანებები:

```
USE University
GO
CREATE TABLE Dorm
( Dorm NVarChar(20) Not Null,
RA NVarChar(25),
CONSTRAINT DormPK PRIMARY KEY( Dorm )
);
GO
CREATE TABLE Faculty
( Fname NVarChar(25) Not Null,
Dept NVarChar(20),
CONSTRAINT FacultyPK PRIMARY KEY( Fname )
);
GO

CREATE TABLE Student
( Sname NVarChar(25) Not Null,
Dorm NVarChar(20) Not Null,
Room Integer,
Phone Char(12),
Major NVarChar(20),
MajorAdvisorName NVarChar(25),
CONSTRAINT StudentPK PRIMARY KEY( Sname, Dorm ),
CONSTRAINT StudentDormFK FOREIGN KEY( DORM )
REFERENCES Dorm( Dorm ),
CONSTRAINT StudentFacultyFK FOREIGN KEY( MajorAdvisorName )
REFERENCES Faculty( Fname )
      );
```

სხვა სახის შეზღუდვას წარმოადგენს შეზღუდვა CHECK. ეს შეზღუდვა საშუალებას იძლევა დავადოთ სვეტს გარკვეული შემოწმების პირობები. მაგალითად:

```
CONSTRAINT FoundedCheck CHECK ( FoundedDate > 1900),
CONSTRAINT ZipCheck CHECK ( zip LIKE '[0-9][0-9][0-9][0-9][0-9]' ),
```

პირველი შეზღუდვა ადგენს, რომ `FoundedCheck` სვეტის მნიშვნელობა ყოველთვის 1900-ზე მეტია, ხოლო მეორე შეზღუდვა მოითხოვს, რომ `zip` სვეტი უნდა წარმოადგენდეს ხუთი ციფრული სიმბოლოსაგან შემდგარ სიდიდეს. მეორე `CHECK` შეზღუდვის შემთხვევაში სინტაქსი ამბობს, რომ `zip` სვეტის მნიშვნელობები უნდა იყვნენ ხუთ-სიმბოლოიანი სიდიდეები, და თითოეული სიმბოლო წარმოადგენს ციფრს 0-სა და 9-ს შორის. ეს სინტაქსი შესაძლებელია იცვლებოდეს სხვადასხვა მონაცემთა ბაზების მართვის სისტემებისათვის, ამიტომ ჩვენ ყოველთვის უნდა გავეცნოთ ჩვენს ხელთ არსებული მონაცემთა ბაზების მართვის სისტემების დოკუმენტაციას.

ზოგჯერ საჭირო ხდება შექმნილი ცხრილის მოცილება. ჩვენ შესაძლებელია ვიფიქროთ, რომ ასეთი ოპერაციისათვის გამოიყენება ერთერთი დარეზერვირებული სიტყვა “delete”, “dispose” ან “destroy”. მართალია “delete” წარმოადგენს დესკრიპტორს (დარეზერვირებულ სიტყვას) SQL ენაში, მაგრამ ის გამოიყენება მონაცემების წასაშლელად მონაცემთა ბაზიდან და არა ისეთი სტრუქტურის მოსაცილებლად როგორიცაა ცხრილი. მონაცემთა ბაზის ისეთი ობიექტის მოსაცილებლად როგორიც ცხრილია გამოიყენება ბრძანება DROP. ქვემოთ მოყვანილია შესაბამისი სინტაქსი:

```
DROP < object_type > < object_name >;
```

DROP დესკრიპტორის შემდეგ უნდა მოვათავსოთ მონაცემთა ბაზის სტრუქტურული ტიპი და მონაცემთა ბაზის სტრუქტურის სახელი. ობიექტურ სტრუქტურებს მიეკუთვნებიან TABLE, VIEW, PROCEDURE (stored procedure), TRIGGER და კიდევ რამდენიმე სხვა სტრუქტურა.
მაგალითად იმისათვის, რომ მოვიცილოთ ცხრილი Student-ი, უნდა გამოვიყენოთ შემდეგი ბრძანება:

```
DROP TABLE Student;
```

როდესაც ჩვენ გვესაჭიროება მონაცემთა ბაზის ობიექტში, მაგალითად, ცხრილში, ცვლილებების შეტანა, უნდა გამოვიყენოთ ბრძანება ALTER. მაგალითად, თუ ჩვენ გვინდა დავამატოთ სტუდენტის ცხრილს დაბადების თარიღის სვეტი, მაშინ ჩვენ უნდა გამოვიყენოთ შემდეგი ბრძანება

```
ALTER TABLE Student ADD COLUMN Birthdate Date;
```

სვეტების დამატებასთან ერთად, ბრძანება ALTER TABLE შესაძლებელია გამოვიყენოთ სვეტების მოსაცილებლად, შეზღუდვების დამატება-მოსაცილებლად, ან default-მნიშვნელობების შესარჩევად.

DML (Data Manipulation Language) - მონაცემთა მანიპულირების ენა

განვიხილოთ პირველი DML-ოპერატორი SELECT. ეს ოპერატორი წარმოადგენს მონაცემთა ბაზაში ინფორმაციის მოძიების საშუალებას. ეს ბრძანება საკმაოდ მოქნილი ბრძანებაა, რომელსაც მრავალი სხვადასხვა სახით იყენებენ.

ყველაზე მარტივ შემთხვევაში SELECT ოპერატორს გამოიყენებენ ცხრილის გარკვეული სვეტის მნიშვნელობათა გამოსატანად, მაგალითად მიმართვა

```
SELECT Sname, Major FROM Student;
```

გამოიტანს Sname და Major სვეტების მნიშვნელობებს Student ცხრილიდან. ის გამოიტანს ერთ სტრიქონს თითოეული სტუდენტის ჩანაწერისათვის ცხრილში და გვიჩვენებს სტუდენტების სახელებსა და მათ მიერ არჩეულ major სპეციალობებს.

ჩვენ ასევე შეგვიძლია დავამატოთ ფილტრი და გამოვიტანოთ გარკვეული პირობების შესაბამისი ცტრიქონები. ამის განხორციელება შესაძლებელია გამოსახულებაში ოპერატორი WHERE-ს დამატებით:

```
SELECT Sname
FROM Student
WHERE Major = 'Computer Science';
```

ეს მიმართვა (query) დააბრუნებს მხოლოდ და მხოლოდ იმ სტუდენტების სახელებს, რომლებმაც აირჩიეს Computer Science ძირითადი სპეციალობის სახით. არც ერთი სხვა სტუდენტის სახელი არ იქნება გამოტანილი.

თუ ჩვენ გვსურს ცხრილის ყველა სვეტებისათვის ყველა სტრიქონების გამოტანა, მაშინ ჩვენ შეგვიძლია გამოვიყენოთ სიმბოლო “asterisk” (ვარსკვლავი “*”), რომელიც ნიშნავს ყველა სვეტების უპირობოდ გამოტანას:

```
SELECT *
FROM Student
WHERE Major = 'Computer Science';
```

ოპერატორი WHERE საკმაოდ მოქნილი ოპერატორია. ამ ოპერატორთან ერთად ტოლობის ნიშნის მაგივრად ჩვენ შეგვიძლია გამოვიყენოთ სხვა შედარებისა და ლოგიკური ოპერატორები, რომლებიც მოყვანილია ქვემოთ ცხრილში.

ოპერატორი	ოპერატორის შინაარსი	მაგალითი
=	ტოლი equal to	
>	მეტი greater than	
<	ნაკლები less than	
>=	მეტი ან ტოლი greater than or equal to	
<=	ნაკლები ან ტოლი less than or equal to	
!=	არ უდრის (არატოლი) not equal to	
!<	არანაკლები	

	not less than	
!>	არამეტი not greater than	
AND	ჭეშმარიტი თუ ორივე ლოგიკური გამონათქვამი ჭეშმარიტია True if both boolean comparisons are true	
OR	ჭეშმარიტი, თუ რომელიმე (ერთი მაინც) ლოგიკური გამონათქვამი ჭეშმარიტია True if either boolean expression are true	
NOT	ლოგიკური ოპერატორის ჭეშმარიტობის მნიშვნელობის შებრუნება Reverses the truth value of another boolean operator	

IN	ჭეშმარიტი, თუ ოპერანდი ემთხვევა ერთერთ ჩამოთვლილ მნიშვნელობას, ან ემთხვევა ერთერთ მნიშვნელობას დაბრუნებულს ქვემიმართვის საშუალებით
	True if the operand is one of the listed values, or one of the values returned by a subquery
		`USE AdventureWorks;`
		`GO`
		`SELECT FirstName, LastName, e.Title`
		`FROM HumanResources.Employee AS e`
		`JOIN Person.Contact AS c`
		`ON e.ContactID = c.ContactID`
		`WHERE e.Title IN ('Design Engineer', 'Tool Designer', 'Marketing Assistant');`
		`GO`
		`////////////////////////////////`
		`USE AdventureWorks;`
		`GO`
		`SELECT FirstName, LastName`
		`FROM Person.Contact AS c`
		`JOIN HumanResources.Employee AS e`
		`ON e.ContactID = c.ContactID`
		`WHERE EmployeeID IN`
		`(SELECT SalesPersonID`
		`FROM Sales.SalesPerson`
		`WHERE SalesQuota > 250000);`
		`GO`

LIKE	ჭეშმარიტი თუ ოპერანდი ემთხვევა ნიმუშს,შაბლონს (განსაზღვრულ ზოგად სტრუქტურას)
	True if the operand matches a pattern
		`USE AdventureWorks;`
		`GO`
		`CREATE PROCEDURE FindEmployee @EmpLName char(20)`
		`AS`
		`SELECT @EmpLName = RTRIM(@EmpLName) + '%';`
		`SELECT c.FirstName, c.LastName, a.City`
		`FROM Person.Contact c JOIN Person.Address a`
		`ON c.ContactID = a.AddressID`
		`WHERE c.LastName LIKE @EmpLName;`
		`GO`
		`EXEC FindEmployee @EmpLName = 'Barb';`
		`GO`
		`////////////////////////////////////`
		`USE AdventureWorks;`
		`GO`
		`SELECT FirstName, LastName, Phone`
		`FROM Person.Contact`
		`WHERE phone LIKE '415%'`
		`ORDER by LastName;`
		`GO`

%	შეესაბამება ნებისმიერ სიმბოლოს
	matches anything
		`USE AdventureWorks;`
		`GO`
		`SELECT FirstName, LastName`
		`FROM Person.Contact`
		`WHERE FirstName LIKE 'Dan%';`
		`GO`

_	(ქვემო ხაზი) შეესაბამება ნებისმიერ ერთ სიმბოლოს (ჯგუფური სიმბოლო, განზოგადების სიმბოლო); დანარჩენი სიმბოლოები იგივე რჩება
	(underscore)(wildcard) matches
		`USE AdventureWorks;`
		`GO`
		`SELECT FirstName, LastName`
		`FROM Person.Contact`
		`WHERE FirstName LIKE '_an'`
		`ORDER BY FirstName;`

	any one character Other characters match themselves	

BETWEEN	ჭეშმარიტი, თუ ოპერანდი მოთავსებულია განსაზღვრულ არეში
	True if the operand is within the range specified
		`USE AdventureWorks;`
		`GO`
		`SELECT e.FirstName, e.LastName, ep.Rate`
		`FROM HumanResources.vEmployee e`
		`JOIN HumanResources.EmployeePayHistory ep`
		`ON e.EmployeeID = ep.EmployeeID`
		`WHERE ep.Rate BETWEEN 27 AND 30`
		`ORDER BY ep.Rate;`
		`G`

EXISTS	ჭეშმარიტი, თუ ქვემიმართვა აბრუნებს ერთ სტრიქონს მაინც
	True if a subquery returns any rows
		`USE AdventureWorks ;`
		`GO`
		`SELECT a.FirstName, a.LastName`
		`FROM Person.Contact AS a`
		`WHERE EXISTS`
		`(SELECT *`
		`FROM HumanResources.Employee AS b`
		`WHERE a.ContactId = b.ContactID`
		`AND a.LastName = 'Johnson');`
		`GO`

ALL	ჭეშმარიტი, თუ შედარებების მთლიანი სიმრავლე ჭეშმარიტია
	True if all of a set of comparisons are true
		`USE AdventureWorks ;`
		`GO`
		
		`CREATE PROCEDURE DaysToBuild @OrderID int,`
		`@NumberOfDays int`
		`AS`
		`IF`
		`@NumberOfDays >= ALL`
		`(`
		`SELECT DaysToManufacture`
		`FROM Sales.SalesOrderDetail`
		`JOIN Production.Product`
		`ON Sales.SalesOrderDetail.ProductID =`
		`Production.Product.ProductID`
		`WHERE SalesOrderID = @OrderID`
		`)`
		`PRINT 'All items for this order can be manufactured in specified number of days or less.'`
		`ELSE`
		`PRINT 'Some items for this order cannot be manufactured in specified number of days or less.' ;`

ANY	ჭეშმარიტი, თუ ერთი შედარება შედარებათა სიმრავლიდან ჭეშმარიტია True if any one of a set of comparisons is true	იგივეა რაც ოპერატორი SOME

SOME	ჭეშმარიტი, თუ რამდენიმე შედარება შედარებათა სიმრავლიდან ჭეშმარიტია
	True if some of a set of comparisons is ture
		`USE AdventureWorks ;`
		`GO`
		
		`CREATE PROCEDURE ManyDaysToComplete`
		`@OrderID int, @NumberOfDays int`
		`AS`
		`IF`
		`@NumberOfDays < SOME`
		`(`
		`SELECT DaysToManufacture`
		`FROM Sales.SalesOrderDetail`
		`JOIN Production.Product`
		`ON Sales.SalesOrderDetail.ProductID =`
		`Production.Product.ProductID`
		`WHERE SalesOrderID = @OrderID`
		`)`
		`PRINT 'At least one item for this order cannot be manufactured in specified number of days.'`
		`ELSE`
		`PRINT 'All items for this order can be manufactured in the specified number of days or less.' ;`

წარმოვიდგინოთ, რომ ჩვენ გვინდა ვიპოვოთ ყველა ის სტუდენტი, რომელთა სახელია “Jones” და ვისაც არა აქვთ არჩეული ძირითადი სპეციალობის (major) სახით არც მათემატიკა და არც კომპიუტერული მეცნიერებები, თანაც ცხოვრობენ სტუდენტურ სასტუმროებში “Williams” და “Schoelkopf”. შესაბამისი მიმართვა იქნება შემდეგი სახის

```
SELECT *
FROM Student
WHERE Sname LIKE '%m%'
AND Major NOT IN ( 'Math', 'Computer Science' )
AND ( Dorm = 'Williams' OR Dorm = 'Schoelkopf' OR Dorm='vake');
```

მიმართვის შედეგად გამოტანილი მონაცემები შესაძლებელია დავალაგოთ გარკვეული თანმიმდევრობით. ამისათვის საჭიროა მხოლოდ ოპერატორის ORDER BY გამოყენება. მაგალითად,

```
SELECT Sname
FROM Student
WHERE Major = 'Computer Science'
ORDER BY Sname;
```

ეს მიმართვა დაალაგებს სტუდენტებს, რომელთაც აირჩიეს ძირითად სპეციალობათ კომპიუტერული მეცნიერებები, სახელების მიხედვით ანბანური თანმიმდევრობით. ამ მიმართვას თუ დავამატებთ ბოლოში სიტყვას DESC (descending კლებადი, descend - დაკლება, ჩამოსვლა), მაშინ სახელები დალაგდება ანბანის საწინაღმდეგო თანმიმდევრობით.

რას გამოიტანს ქვემოთ მოყვანილი მიმართვა?

```
USE Northwind
GO
SELECT UnitPrice FROM Products
GO
```

ის გამოიტანს ერთ სტრიქონს თითოეული პროდუქტისათვის, და თითოეულ სტრიქონში ჩაწერილი იქნება შესაბამისი პროდუქტის ფასი. სტრიქონების რაოდენობა იქნება 77, რადგანაც Northwind მონაცემთა ბაზის ცხრილში Products მთლიანობაში 77 ჩანაწერია (სტრიქონია). ასეთი ანგარიში შესაძლებელია მთლად ის არ იყოს რაც გვინდა. ჩვენ შესაძლებელია გვაინტერესებდეს მხოლოდ ერთმანეთისაგან განსხვავებული ფასების გაგება. მაშინ ჩვენ უნდა გამოვიყენოთ სიტყვა DISTINCT, და მიმართვა მიიღებს სახეს

```
USE Northwind
GO
SELECT DISTINCT UnitPrice FROM Products
GO
```

ამ მიმართვის ანგარიშში უკვე გვექნება 62 ჩანაწერი (სტრიქონი) პროდუქტების ფასების შესახებ, თანაც ყველა ერთმანეთისაგან განსხვავებული იქნება.

როგორ უნდა მოვიქცეთ, თუ ჩვენ გვაინტერესებს ინფორმაცია, რომელიც რამდენიმე სხვადასხვა ცხრილშია განაწილებული? შესაბამისი მიმართვა მოითხოვს ცხრილების გაერთიანებას (დაკავშირებას) JOIN ოპერატორის გამოყენებით. JOIN ოპერატორისათვის არსებობს რამდენიმე ერთმანეთისაგან განსხვავებული სახე სხვადასხვა შემთხვევისათვის. წარმოვიდგინოთ ჩვენ გვაინტერესებს სტუდენტების სახელებისა და მათი რეზიდენტი მრჩევლების სია. ქვემოთ მოყვანილია ერთერთი გზა Student და Dorm ცხრილების გაერთიანება ოპერატორით JOIN ჩვენი ინფორმაციის მოსაძიებლად

```
SELECT Sname, RA
FROM Student, Dorm
WHERE Student.Dorm = Dorm.Dorm;
```

კონცეპტუალურად ოპერატორი JOIN მუშაობს ორი ცხრილის სტრიქონების გაერთიანების საშუალებით, თანაც გადაბმა ხდება იმ სტრიქონებისა სადაც პირობა WHERE სრულდება. ამ

მიმართვის კონკრეტულ შემთხვევაში ყოველი სტრიქონი Student ცხრილიდან გაერთიანდება (გადაიბმევა) Dorm ცხრილის იმ სტრიქონთან სადაც Dorm ცხრილის Dorm სვეტის მნიშვნელობა (Dorm.Dorm) უდრის Student ცხრილის Dorm სვეტის მნიშვნელობას (Student.Dorm – Student ცხრილის Dorm სვეტის აღნიშვნას წარმოადგენს). ამის შემდეგ გადაბმული სტრიქონებიდან ხდება Sname და RA სვეტების მნიშვნელობათა ამოკრეფა.
იგივე მიმართვის სხვა გზით ჩაწერა შესაძლებელია შემდეგი სინტაქსის გამოყენებით

```
SELECT Sname, RA
FROM Student JOIN Dorm
ON Student.Dorm = Dorm.Dorm;
```

ორივე სინტაქსი მისაღებია. ორივე შემთხვევაში სვეტი სახელად Dorm, რომელიც არსებობს ორივე ცხრილში, მიმართვის შედგენისას ცალსახად უნდა იყოს განსაზღვრული შესაბამისი ცხრილის სახელის მითითებით.

ჩვენ შეგვიძლია დავამატოთ University მონაცემთა ბაზას კიდევ ორი ცხრილი, რომლებშიც განთავსდება ინფორმაცია კლუბებისა და მათში სტუდენტების მონაწილეობის შესახებ. ნებისმიერი კლუბის წევრი შესაძლებელია იყოს მრავალი სტუდენტი, და ნებისმიერი სტუდენტი შესაძლებელია მრავალი კლუბის წევრი იყოს, ამიტომ შესაბამისი კავშირი იქნება M:N სახის. ჩვენ უნდა შევქმნათ ორი ცხრილი: ერთი კლუბებისათვის (Club), მეორე კი გადაკვეთის ცხრილი იქნება (ClubMembership), რომელიც დააფიქსირებს სტუდენტების წევრობას სხვადასხვა კლუბებში:

```
USE University
GO
CREATE TABLE Club
( Cname NVarChar(20) Not Null,
Dues Int,
Building NVarChar(20),
Room Int
CONSTRAINT ClubPK PRIMARY KEY (Cname)
);
CREATE TABLE ClubMembership
( Cname NVarChar(20) Not Null,
MemberName NVarChar(25) Not NULL,
Dorm NVarChar(20) NOT Null
CONSTRAINT ClubMembershipPK PRIMARY KEY (Cname,
MemberName),
CONSTRAINT ClubMembership_Club_FK
FOREIGN KEY (Cname) REFERENCES Club(Cname),
CONSTRAINT Member_FK
FOREIGN KEY (MemberName, Dorm) REFERENCES Student(Sname,Dorm)
);
```

იმისათვის, რომ მივიღოთ სია, რომელშიც იქნება ჩამოთვლილი სტუდენტები, მათი major (ძირითადი) სპეციალობა და მათი წევრობის კლუბები, ჩვენ შეგვიძლია შევასრულოთ შემდეგი მიმართვა, რომელიც აერთიანებს (აკავშირებს) ცხრილებს Student და ClubMembership

```
SELECT Sname, Major, Cname
FROM Student JOIN ClubMembership
ON Student.Sname = ClubMembership.MemberName;
```

ბოლო სტრიქონში წერტილი მიუნიშნებს იმ ფაქტს, რომ ჩვენ გვაინტერესებს სტრიქონების ისეთი დაკავშირება (გაერთიანება), როდესაც Student ცხრილის Sname სვეტის შედარება ხდება ClubMembership ცხრილის MemberName სვეტთან. ამ მიმართვის შედეგი მოიცავს სტიქონს ყველა იმ სტუდენტისათვის ვინც მონაწილეობს რომელიმე კლუბში, და თუ სტუდენტი ერთზე მეტი კლუბის წევრია, მაშინ გამოტანილ შედეგში ასეთი სტუდენტისათვის რამდენიმე სტრიქონი გვექნება. როგორ მოვიქცეთ თუ ჩვენ ასევე გვაინტერესებს ის სტუდენტები, რომლებიც არ არიან წევრები არც ერთი კლუბის? ამ საკითხის გადაჭრა შესაძლებელია მიმართვაში ”გარე გაერთიანების” (outer join) გამოყენებით.

სტანდარტული, ანუ ”შიდა” (“inner join”), გაერთიანება აგროვებს ინფორმაციას ორი ცხრილიდან ახალი სტრიქონის შექმნით ორივე ცხრილის ატრიბუტებით, თანაც ახალი სტრიქონი

იქმნება ყოველთვის, როდესაც ადგილი აქვს ორივე ცხრილის იმ სვეტების მნიშვნელობების დამთხვევას, რომლებიც მითითებულია გაერთიანების პირობაში. ზემოთ მოყვანილ მაგალითში როდესაც Student ცხრილის Sname სვეტის მნიშვნელობა დაემთხვევა ClubMembership ცხრილის MemberName სვეტის მნიშვნელობას, გაერთიანება ქმნის ახალ სტრიქონს რომელიც მოიცავს ორივე ცხრილში არსებულ ყველა ატრიბუტს. ამის შემდეგ SELECT ოპერატორი ამოარჩევს Sname, Major და Cname სვეტების მნიშვნელობებს შედგენილი სტრიქონიდან.

გარე გაერთიანება (outer join) მოიცავს ყველა სტრიქონს ერთი ან ორივე ცხრილიდან, იმის და მიუხედავად ადგილი აქვს თუ არა დამთხვევას გაერთიანების პირობის თანახმად. მაგალითად, იმისათვის რომ ვაჩვენოთ ყველა სტუდენტი იმის და მიუხედავად წევრია ის რომელიმე კლუბის თუ არა, საჭიროა ზემოთ მოყვანილი მიმართვის მოდიფიცირება LEFT (მარცხენა) სიტყვის გამოყენებით:

```
SELECT Sname, Major, Cname
FROM Student LEFT JOIN ClubMembership
ON Student.Sname = ClubMembership.MemberName;
```

სიტყვა LEFT მიუთითებს, რომ მარცხენა ცხრილი, ამ შემთხვევაში Student და არა ClubMembership, ზუსტად ის ცხრილია, რომლიდანაც მოხდება ყველა სტრიქონის გამოტანა. ეს ახალი მიმართვა დააბრუნებს ერთ სტრიქონს ყველა სტუდენტისათვის (ან უფრო მეტს იმ სტუდენტებისათვის, რომლებიც ერთზე მეტ კლუბს მიეკუთვნებიან), და თუ სტუდენტი არცერთი კლუბის წევრი არ არის, მაშინ გამოტანილი ანგარიშის შესაბამის სტრიქონში Cname სვეტის მნიშვნელობა იქნება NULL (NULL - ნიშნავს რომ შესამამის ადგილში არავითარი მონაცემი არა გვაქვს).

სიტყვა RIGHT (მარჯვენა) იგივე სახის მოქმედებას ახდენს, მხოლოდ ამ შემთხვევაში გამოდის მარჯვენა ცხრილის ყველა სტრიქონი. თუ LEFT გაერთიანების პირობაში ჩვენ გადავაადგილებთ ცხრილებს და LEFT სიტყვას შევცვლით RIGHT სიტყვით, მაშინ მიღებული მიმართვა საწყისი მიმართვის ეკვივალენტური იქნება:

```
SELECT Sname, Major, Cname
FROM ClubMembership RIGHT JOIN Student
ON Student.Sname = ClubMembership.MemberName
```

თუ ჩვენ გვაქვს სურვილი ორივე ცხრილის ყველა სტრიქონი გამოვიტანოთ, იმის და მიუხედავათ არის თუ არა შესრულებული გაერთიანების პირობა, მაშინ უნდა გამოვიყენოთ სიტყვა FULL (LEFT და RIGHT სიტყვების ნაცვლად).

მხოლოდ იმ სტუდენტების გამოსატანად, რომლებიც არცერთი კლუბის წევრები არ არიან, ჩვენ უბრალოდ უნდა დავამატოთ WHERE გამონათქვამი:

```
SELECT Sname, Major, Cname
FROM Student LEFT JOIN ClubMembership
ON Student.Sname = ClubMembership.MemberName
WHERE Cname IS NULL;
```

უნდა შევნიშნოთ, რომ ჩვენ NULL მნიშვნელობის არსებობის დასადგენად ვიყენებთ სიტყვას IS ტოლობის ნიშნის მაგივრად. უნდა ითქვას, რომ NULL მნიშვნელობას რეალურად არავითარი მნიშვნელობა არ გააჩნია. რადგანაც არავითარი მნიშვნელობა არ არსებობს, ჩვენ არ შეგვიძლია ასეთი სიდიდის შედარება სხვა ნებისმიერ სიდიდესთან, თუნდაც იგივე NULL-თან. ამიტომ როდესაც ჩვენ ვახდენთ NULL სიდიდის შემოწმებას (ტესტირებას), ჩვენ ყოველთვის უნდა გამოვიყენოთ IS NULL ან IS NOT NULL.

დავუშვათ, რომ ჩვენ გვაინტერესებს სულ რამდენი სტუნდენტი მონაწილეობს თითოეულ კლუბში. SQL ენას გააჩნია ჩაშენებული ფუნქციები (built-in functions) მარტივი და საკმაოდ ხშირად გამოყენებადი მათემატიკური ოპერაციების შესასრულებლად. სტანდარტულ ხუთ ფუნქციას წარმოადგენენ COUNT (count - დათვლა), SUM (sum - ჯამი), AVG (average - საშუალო), MIN (minimum - მინიმალური) და MAX (maximum - მაქსიმალური). გარდა ამისა მონაცემთა ბაზების მართვის სიტემებს შესაძლებელია ასევე დამატებით გააჩნდეთ სხვა არასტანდარტული ფუნქციებიც. სტუდენტების დათვლა თითოეულ კლუბში შესაძლებელია შემდეგი გზით:

```
SELECT Cname, COUNT(*) AS Membership
FROM ClubMembership
```

```
GROUP BY Cname
ORDER BY Cname;
```

`COUNT(*)` მიუთითებს სტრიქონების ეგზემპლართა დათვლას. ფრაზა `AS Membership` ადგენს ანგარიშში რაოდენობათა სვეტის სათაურისათვის Membership მნიშვნელობას. ფრაზა `GROUP BY` მიუთითებს SQL-ს როგორ განახორციელოს დათვლა ჩაშენებული ფუნქციის გამოყენებით; ჩვენ შემთხვევაში დათვლა ხორციელდება თითოეული კლუბისათვის.

თუ არსებობენ ისეთი კლუბები, რომლებშიც არცერთი სტუდენტი არ მონაწილეობს, და ჩვენ გვინდა ეს ინფორმაცია ჩავრთოთ ანგარიშში, მაშინ ჩვენ ჩაშენებული დათვლის ფუნქციასთან ერთად უნდა გამოვიყენოთ გარე გაერთიანება:

```
SELECT Club.Cname, COUNT(ClubMembership.Cname) AS Membership
FROM Club LEFT JOIN ClubMembership
ON Club.Cname = ClubMembership.Cname
GROUP BY Club.Cname
ORDER BY Club.Cname;
```

ამ მიმართვაში ცხრილების Club და ClubMembership `LEFT JOIN` გაერთიანება იძლევა ანგარიშში ყველა კლუბების გამოტანის გარანტიას იმ შემთხვევაშიც, როდესაც კლუბში არცერთი წევრია. გარდა ამისა Cname სვეტი არსებობს ორივე ცხრილში, ამიტომ Cname-ის ყოველი მოხსენიება მიმართვაში ზუსტად უნდა განსაზღვრავდეს რომელ ცხრილზეა საუბარი. ზემოთ მოყვანილი მიმართვა ამბობს: შესრულდეს გარე გაერთიანება ცხრილებზე Club და ClubMembership ისე რომ გამოტანილ იქნას ყველა კლუბი, და შემდეგ დაითვალოს ჩანაწერების ყველა შემთხვევა თითოეული კლუბისათვის, ხოლო დათვლის დროს დაჯგუფება განხორციელდეს კლუბების სახელების მიხედვით.

თუ ჩვენ გვჭირდება თითოეული კლუბის შემოსავლების (revenue - შემოსავლები) ანგარიში, მაშინ მიმართვაში უნდა გამოვიყენოთ ფუნქცია SUM :

```
SELECT CM.Cname, SUM( C.Dues ) AS Revenue
FROM ClubMembership CM JOIN Club C
ON CM.Cname = C.Cname
GROUP BY CM.Cname;
```

ამ მიმართვაში გამოყენებულია ცხრილებისათვის ფსევდონიმი სახელები (გამოგონილი სახელები). FROM გამოსახულებაში ცხრილის სახელის შემდეგ ჩვენ შეგვიძლია ჩავწეროთ შესაბამისი აბრევიატურა (ან სხვა გამოგონილი სახელი). ასეთ შემთხვევაში ეს აბრევიატურა ჩვენ შეგვიძლია გამოვიყენოთ ნებისმიერ ადგილას სადაც ცხრილის სახელი დაგვჭირდება, თანაც ამის გაკეთება შესაძლებელია SELECT გამოსახულების დასაწყისშიც მანამ FROM გამოსახულებას შევხვდებით! SQL-ში გამოცდილი მრავალი სპეციალისტი ენერგიულად გამოიყენებს ცხრილების გამოგონილ სახელებს (ფსევდონიმებს), რის შედეგადაც SQL-მიმართვები ზომით მცირდებიან და წასაკითხად უფრო მარტივები ხდებიან.

ერთი SQL-მიმართვა შესაძლებელია მეორე SQL-მიმართვის შიგნით იმყოფებოდეს, ამ შემთხვევაში საუბრობენ ჩასმულ მიმართვებზე (nested query). დავუშვათ ჩვენ გვინდა ყველა იმ სტუდენტის მოძებნა ვისი მრჩეველიც ძირითადი სპეციალობის მიხედვით არ არის მათემატიკის ფაკულტეტიდან (mathematics). შესაბამისი მიმართვის შედგენის ერთერთ გზას წარმოადგენს ფაკულტეტთან დაკავშირებული მიმართვის ჩასმა მეორე მიმართვაში, რომელიც სტუდენტებთან არის დაკავშირებული:

```
SELECT Sname, Dorm
FROM Student
WHERE MajorAdvisorName
IN (
SELECT Fname
FROM Faculty
WHERE Dept != 'Mathematics'
      );
```

ამას ეწოდება ჩასმული მიმართვა ან ქვემიმართვა. ხშირად შესაძლებელია გამოვიყენოთ გაერთიანების (JOIN) ოპერაცია ჩასმული მიმართვის ნაცვლად, მაგრამ ამის გაკეთება ყოველთვის არ არის შესაძლებელი. მოგვიანებით ჩვენ განვიხილავთ კორელირებულ ქვემიმართვებს (correlated subqueries), რომელთა გადაწერა გაერთიანების ოპერაციის გამოყენებით შეუძლებელია. აქ კი ჩვენ მოვიყვანთ მიმართვას, რომელიც ზემოთ განხილული მიმართვის ეკვივალენტურია:

```
SELECT Sname, Dorm
FROM Student JOIN Faculty
ON MajorAdvisorName = Fname
WHERE Dept != 'Mathematics';
```

ამ მიმართვაში არავითარი ორი აზრი არ არსებობს იმის შესახებ თუ რომელი ცხრილის სვეტებზეა საუბარი, ამიტომ სვეტების დასახელებისას არ არის საჭირო შესაბამისი ცხრილების სახელების მითითება.

როდესაც ჩვენ ვიყენებთ ქვემიმართვას ყველა გამოტანილი სვეტი მიიღება იმ ცხრილიდან რომელიც დასახელებულია პირველი SELECT გამოსახულების FROM ოპერატორის შესაბამის ნაწილში. ამის გამო ჩვენ შეგვიძლია გამოვიყენოთ როგორც ქვემიმართვა ასევე JOIN ოპერატორი.

SELECT ოპერატორს ჩვენ დავუბრუნდებით მოგვიანებით, ხოლო ეხლა განვიხილოთ კიდევ სხვა სახის SQL DML ოპერატორები. როდესაც მონაცემთა ბაზაში ცხრილი უკვე შექმნილია, დგება ამ ცხრილის მონაცემებით შევსების საკითხი. SQL ოპერატორი INSERT ზუსტად ის ოპერატორია, რომელიც ამატებს სხრილს ახალ სტრიქონებს. ქვემოთ მოყვანილია ამ ოპერატორის ზოგადი სინტაქსი:

```
INSERT INTO <table_name> ( <columnX>, <columnY>..., <columnZ>)
VALUES( <valueX>, <valueY>..., <valueZ>)
```

ამ სინტაქსიდან კარგად ჩანს, რომ INSERT INTO (insert - ჩასმა) ძირითადი სიტყვების შემდეგ უნდა ჩაიწეროს ცხრილის სახელი. შემდეგ უნდა გაიხსნას ფრჩხილი, ჩაიწეროს ერთი ან მეტი სვეტის დასახელებები და დაიხუროს ფრჩხილი. ხოლო ამის შემდეგ უნდა ჩაიწეროს ძირითადი სიტყვა VALUES (value - მნიშვნელობა), გაიხსნას ფრჩხილი, ჩაიწეროს სვეტების მნიშვნელობების სია, რომელიც შეესაბამისობაშია სვეტების სახელებთან, და დაიხუროს ფრჩხილი. ქვემოთ მოყვანილია მაგალითი:

```
INSERT INTO Student( Sname, Dorm, Room, Phone )
VALUES ('manjavidze mari', 'varketili', 142, '774566');
```

უნდა აღინიშნოს, რომ ზოგიერთი სვეტის მნიშვნელობა Student ცხრილიდან არ არის განსაზღვრული. სტუდენტს ჯერ არ აურჩევია ძირითადი სპეციალობა (major). რა თქმა უნდა ამ სვეტების (Major, MajorAdvisorName) მნიშვნელობები შესაბამის ცხრილში იქნება NULL.

სულაც არ არის საჭირო, რომ სვეტების სახელების თანმიმდევრობა მიმართვაში ემთხვეოდეს სვეტების თანმიმდევრობას მონაცემთა ბაზის ცხრილში, მაგრამ მნიშვნელობათა თანმიმდევრობა VALUES კონსტრუქციაში შიგნით უნდა ემთხვეოდეს შესაბამისი სვეტების თანმიმდევრობას INSERT კონსტრუქციაში.

იმ შემთხვევაში როდესაც ჩვენ ცხრილის ყველა სვეტში შეგვაქვს მნიშვნელობები, მაშინ არ არის სავალდებულო სვეტების სახელების სიის ჩაწერა INSERT კოსტრუქციაში. ამ შემთხვევაში შესატანი მნიშვნელობების თანმიმდევრობა უნდა ემთხვეოდეს სვეტების თანმიმდევრობას ცხრილში. ქვემოთ მოყვანილია კიდევ ერთი სხვა მაგალითი:

```
INSERT INTO Student VALUES ('tsertsvadze giorgi', 'saburtalo', 399,
'322335', 'Philology', 'nutsubidze irakli');
```

მონაცემთა ბაზებთან მუშაობისას ხშირად საჭიროა მონაცემების განახლება (უკვე არსებულ მონაცემებში ცვლილებების შეტანა). ამისათვის იყენებენ ოპერატორს UPDATE (update - განახლება). ქვემოთ მოყვანილია ამ ოპერატორის სინტაქსი:

```
UPDATE <table_name>
SET <columnX> = <valueX>, <columnY> = <valueY>...,
<columnZ> = <valueZ>
```

```
WHERE <condition>;
```

და მისი გამოყენების მაგალითი:

```
UPDATE Student
SET Major = 'Biology', MajorAdvisorName = 'metreveli lali'
WHERE Sname = 'manjavidze mari';
```

მანჯავიძე მარიმ აირჩია ძირითადი (major) სპეციალობა და ეს მიმართვა დაამატებს შესაბამის ინფორმაციას Student ცხრილის შესაბამის სტრიქონს.

WHERE ოპერატორი UPDATE გამონათქვამის შიგნით ისეთივეა და იგივე სახის მოქნილობა გააჩნია, როგორც ამას ადგილი ჰქონდა მისი SELECT გამონათქვამის შიგნით გამოყენების შემთხვევაში. WHERE ოპერატორის შიგნით (მის ფარგლებში) ჩვენ შეგვიძლია გამოვიყენოთ ქვემიმართვებიც. WHERE ოპერატორი განსაზღვრავს იმ სტრიქონებს, რომელთა სვეტების მნიშვნელობების შეცვლა ხდება. ზემოთ მოყვანილ მაგალითში ცვლილებები ხდება მხოლოდ ერთ სტრიქონში, მაგრამ სხვა შემთხვევებში UPDATE ოპერატორს შეუძლია შეცვალოს სტრიქონთა მთლიანი ჯგუფები.

მაგალითად, წარმოვიდგინოთ, რომ ComputerScience დეპარტამენტის სახელი შეიცვალა ახალი სახელით InformationTechnology. ქვემოთ მოყვანილი UPDATE მიმართვა შეცვლის Faculty ცხრილის ყველა იმ სტრიქონს, რომელშიც მითითებულია Dept (department) სვეტის მნიშვნელობა ComputerScience, ისე რომ მათში ჩაიწერება ახალი მნიშვნელობა InformTechnology:

```
UPDATE Faculty
SET Dept = 'InformTechnology'
WHERE Dept = 'ComputerScience';
```

ეს ერთი მიმართვა შეასწორებს ყველა შესაბამის სტრიქონს.

სტრიქონების ამოშლა მონაცემთა ბაზიდან ხდება მარტივად DELETE ოპერატორის საშუალებით. ქვემოთ მოყვანილია შესაბამისი მიმართვის სინტაქსი:

```
DELETE FROM <table_name>
WHERE <condition>;
```

ამ შემთხვევაში ამოსაშლელი სტრიქონების იდენტიფიცერებისათვის WHERE ოპერატორი უზრუნველყოფს იგივე სახის მოქნილ და მძლავრ მექანიზმს.

მანჯავიძე მარის ამოსაშლელად მონაცემთა ბაზიდან უნდა გამოვიყენოთ შემდეგი DELETE მიმართვა:

```
DELETE FROM Student WHERE Sname = 'manjavidze mari';
```

იმისათვის რომ ამოვშალოთ ყველა სტრიქონი ცხრილიდან საჭიროა WHERE ოპერატორის მოშორება ამ მიმართვაში, ოღონდ უნდა გავფრთხილდეთ, რათა შემთხვევით არ წავშალოთ ჩვენთვის მნიშვნელოვანი ინფორმაცია. ასევე გავიხსენოთ, რომ ბაზიდან მთლიანი ცხრილის ამოსაშლელად ჩვენ ვიყენებთ DROP ოპერატორს.

ეხლა დავუბრუნდეთ SELECT ოპერატორს, რათა განვიხილოთ კორელირებული ქვემიმართვები (correlated subqueries). როდესაც ქვემიმართვის გამოთვლა (შესრულება) ეფუძნება (ეყრდნობა) რომელიმე ატრიბუტს გარე მიმართვიდან, მაშინ ჩასმულ მიმართვას ან ქვემიმართვას ეწოდება კორელირებული ქვემიმართვა.

იმისათვის რომ გავაანალიზოთ კორელირებული ქვემიმართვის ფუნქციონირება (მუშაობა), უნდა წარმოვიდგინოთ, რომ ეს ქვემიმართვა სრულდება თითოეული სტრიქონისათვის გარე მიმართვიდან. გარე მიმართვა უნდა შესრულდეს ცხრილის ყველა სტრიქონისათვის, და შიდა მიმართვას უნდა გააჩნდეს ინფორმაცია შესაბამისი სტრიქონის შესახებ თავისი სამუშაოს განსახორციელებლად.

მაგალითად, წარმოვიდგინოთ რომ ჩვენ გვაინტერესებს არის თუ არა მონაცემთა ბაზაში ისეთი სტუდენტური სასტუმროები, რომლებშიც არცერთი სტუდენტი არ ცხოვრობს. იმისათვის რომ პასუხი გავცეთ ასეთ კითხვას საჭიროა გავიაროთ Dorm ცხრილის თითოეული სტრიქონი და შევამოწმოთ

არსებობენ თუ არა სტუდენტები რომლებიც ამ სასტუმროში ცხოვრობენ. შესაბამისი მიმართვა შესაძლებელია ჩაიწეროს შემდეგი სახით:

```
SELECT Dorm
FROM Dorm
WHERE NOT EXISTS (
                  SELECT *
                  FROM Student
                  WHERE Student.Dorm = Dorm.Dorm
                  );
```

გარე მიმართვა (outer query) შესრულებისას გაივლის Dorm ცხრილის თითოეულ სტრიქონს და ამოწმებს Dorm სვეტის (Dorm.Dorm) მნიშვნელობებს. Dorm.Dorm-ის თითოეული მნიშვნელობისათვის შიდა მიმართვა ამოარჩევს Student ცხრილიდან ყველა იმ სტუდენტს, რომელთა სტრიქონების Dorm სვეტში (Student.Dorm) ამ სასტუმროს სახელია ჩაწერილი.

ეს მიმართვა გამოიყენებს NOT EXISTS (არ არსებობს) ფუნქციას (რა თქმა უნდა, არსებობს EXISTS (არსებობს) ფუნქციაც). EXISTS და NOT EXISTS ფუნქციები გამოიყენება კორელირებულ ქვემიმართვებში მათი შესრულების შედეგად მიღებული რეზულტატის არსებობის ან არარსებობის შესამოწმებლად. ჩვენ შემთხვევაში როდესაც არცერთი სტუდენტი არ არის ნაპოვნი (რეზულტატი NOT EXISTS (არ არსებობს)), მაშინ Dorm ცხრილის შესაბამისი სტრიქონი აკმაყოფილებს ჩვენი ძიების კრიტერიუმებს. შედეგი იქნება იმ სატუმროების სია, რომლებშიც არცერთი სტუდენტი არ ცხოვრობს.

ანალოგიურად, ჩვენ შეგვიძლია შევქმნათ ყველა იმ სასტუმროს სია, რომლებშიც ცხოვრობენ სტუდენტები. ამისათვის NOT EXISTS ფუნქცია უნდა შეიცვალოს EXISTS ფუნქციით.

NOT EXISTS ფუნქციას შეუძლია საკმაოდ რთული ლოგიკის განხორციელება. დავუშვათ მაგალითისათვის, რომ ჩვენ გვინდა ვიცოდეთ არსებობს თუ არა ისეთი კლუბი, რომლის წევრებიც არიან ყველა ის სტუდენტი ვისი ძირითადი სპეციალობა მათემატიკაა. ჩვენ შეგვიძლია გამოვიყენოთ NOT EXISTS ფუნქცია ისეთი კლუბების მოსაძებნად, რომლებშიც არცერთი ასეთი სტუდენტია, და შემდეგ ისევ გამოვიყენოთ NOT EXISTS ფუნქცია იმ კლუბების მოსაძებნად, რომელსაც ყველა მიეკუთვნება, რომლებმაც აირჩიეს მათემატიკა ძირითად სპეციალობათ. შესაბამისი მიმართვა ასეთია:

```
SELECT Cname
FROM Club
WHERE NOT EXISTS (
SELECT *
FROM Student
WHERE Student.Major = 'Math' AND NOT EXISTS (
SELECT *
FROM ClubMembership
WHERE Student.Sname = ClubMembership.MemberName
AND Club.Cname = ClubMembership.Cname));
```

ეს მიმართვა შესრულებისას გაივლის ყველა სტრიქონს Club ცხრილიდან. თითოეული სტრიქონისათვის Club ცხრილიდან ის გაივლის Student ცხრილის ყველა სტრიქონს. შემდეგ თითოეულ სტრიქონთა კომბინაციისათვის Club row/Student row, ეს მიმართვა გაივლის ყველა სტრიქონს ClubMembership ცხრილიდან.

ყველაზე უფრო შიგნით მდებარე მიმართვა მოძებნის იმ კლუბებს, რომლებშიც ზოგიერთი მათემატიკოსი სტუდენტი არ იღებს მონაწილეობას. მაგრამ, თუ რომელიმე კონკრეტულ კლუბში მონაწილეობს ყველა მათემატიკოსი სტუდენტი, მაშინ ეს ყველაზე უფრო შიგნით მდებარე მიმართვა დააბრუნებს მნიშვნელობას NULL. თუ ამ მიმართვის შედეგი NULL-ია მაშინ ყველაზე უფრო შიგნით მდებარე NOT EXISTS ფუნქციის მნიშვნელობა იქნება TRUE (ის არის NULL; ის არ არსებობს; ამიტომ NOT EXISTS არის TRUE). თუ ყველაზე უფრო შიგნით მდებარე NOT EXISTS ფუნქცია არის TRUE, მაშინ შესაბამისი კლუბი აკმაყოფილებს ძიების ლოგიკურ კრიტერიუმს და ეს კლუბი გამოიტანება საბოლოო ანგარიშში, როგორც კლუბი რომელშიც ყველა მათემატიკოსი სტუდენტია გაერთიანებული. ამ იდეის გაანალიზება გარკვეულ დროს მოითხოვს!

შენახული პროცედურები (Stored Procedures)

მონაცემთა ბაზების მართვის სისტემათა უმრავლესობა იძლევა შენახული პროცედურების (stored procedure) შექმნის საშუალებას. შენახული პროცედურები წარმოადგენენ პროგრამებს, რომლებიც წინასწარ არიან კომპილირებული და ინახებიან თვითონ მონაცემთა ბაზის შიგნით. მომხმარებლებს შეუძლიათ გამოიყენონ შენახული პროცედურები მონაცემთა ბაზაში ცვლილებების შესატანა ან გარკვეული ინფორმაციის მოსაძიებლად, თანაც მათ შეუძლიათ ამის გაკეთება როგორც ინტერაქტიურად ბრძანებების გამოყენებით, ასევე გარკვეული პროგრამების გამოყენებით, რომლებიც გამოიძახებენ შენახულ პროცედურებს და მიიღებენ შესაბამის შედეგებს.

შენახული პროცედურები იწერება ენაზე, რომელიც სტანდარტული SQL ენის გაფართოებას წარმოადგენს და რომელიც მოიცავს ისეთ დამატებით დაპროგრამების კონსტრუქციებს, როგორიცაა პირობითი განშტოება (conditional branching), ციკლები (looping), შეტანა/გამოტანა (I/O), შეცდომების დამუშავება (error handling). ეს გაფართოებული ენები სხვადასხვა მონაცემთა ბაზების მართვის სიტემებისათვის განსხვავებულია ერთმანეთისაგან, ამიტომ პრაქტიკაში საჭიროა ვიცნობდეთ კონკრეტული მონაცემთა ბაზების მართვის სისტემის დოკუმენტაციას.

შენახულ პროცედურებს, როგორც მონაცემთა ბაზების წვდომის ერთერთ ხერხს, გააჩნიათ რამდენიმე გარკვეული უპირატესობა. პირველ რიგში პროცედურა შესაძლებელია იყოს საკმაოდ რთული სტრუქტურის, მაგრამ საკმაოდ მარტივი მოსახმარი ნებისმიერი არაპროფესიონალი მომხმარებლისათვის. მონაცემთა ბაზების გამოცდილი პროგრამისტს შეუძლია შექმნას ისეთი შენახული პროცედურები, რომლებიც მონაცემთა ბაზის ყოველდღიურ გამოყენებას უფრო მოხერხებულ და კომფორტულს გახდის. მაგალითად, მომხმარებელს შესაძლებელია უნდოდეს კლიენტთა შესყიდვების შესახებ ინფორმაციის ჩაწერა ბაზაში და სულაც არ იყოს კურსში იმის თაობაზე, რომ ყოველი ჩაწერა რეალურად მოითხოვს ორ, Customer და Product, ცხრილში, , ცვლილებების განხორციელებას. შენახულ პროცედურას შეუძლია მიიღოს გარკვეული ფაქტები (კლიენტის სახელი, ფასი, რაოდენობა, პროდუქტის დასახელება) და შემდეგ ყველა საჭირო ცვლილებების განხორციელება სცენის გაღმა (ფარულად მომხმარებლის თვალსაზრისით).

შენახული პროცედურების გამოყენების მეორე უპირატესობას წარმოადგენს მონაცემთა ბაზის მართვის სისტემის მოქმედების დახვეწა (სისწრაფის თვალსაზრისით). შენახული პროცედურების გარეშე SQL ბრძანება უნდა წარედგინოს მონაცემთა ბაზების მართვის სისტემას, მონაცემთა ბაზის მართვის სისტემამ უნდა შეამოწმოს მიმართვები შეცდომების დასადგენად, მოახდინოს ბრძანებების კომპილირება, შეიმუშაოს შესრულების გეგმები, შეასრულოს გეგმები და დააბრუნოს საბოლოო შედეგები. მეორეს მხრივ, თუ პროცედურა წინასწარ არის კომპილირებული და შენახული, მაშინ საჭიროა გაცილებით ნაკლები სამუშაოს შესრულება შეცდომების დასადგენად, ხოლო შესრულების გეგმა უკვე გამზადებულია. მონაცემთა ბაზებთან მომუშავე პროგრამებში, რომლებიც სწრაფად უნდა მუშაობდნენ, შენახული პროცედურების გამოყენება წარმატების მიღწევის ძირითად სტანდარტულ სტრატეგიას წარმოადგენს.

მესამე უპირატესობას წარმოადგენს ის ფაქტი, რომ შენახული პროცედურების გამოყენება იძლევა ერთხელ დაწერილი პროგრამული კოდის გამოყენებას სხვადასხვა პროგრამებისა და მომხმარებლების მიერ. ეს შესაძლებლობა იძლევა პროგრამირების დროის ეკონომიას და ამასთან ერთად სვადასხვა პროგრამებისერთმანეთთან თავსებადობის გარანტიას.

მეოთხე, შენახული პროცედურები დაცულია იგივე დაცვის მექანიზმებით, რომლითაცაა დაცული მონაცემთა ბაზების მონაცემები. ზოგჯერ პროცედურებში ინკაფსულირებულია მნიშვნელოვანი ბიზნეს-წესები (business rules) ან კერძო (კონფიდენციალური) მონაცემების დამუშავება, ამიტომ მათი უსაფრთოების საკითხი საკმაოდ მნიშვნელოვან ფაქტორს წარმოადგენს. შენახული პროცედურები ინახება მონაცემთა ბაზის შიგნით და მათი წვდომა დაცულია ზუსტად ისევე როგორც მონაცემთა წვდომა. ასეთი სიტუაცია შესაძლებელია უფრო მომგებიანი იყოს ვიდრე პროგრამული კოდის ცალკე დაცვა.

ერთადერთ დისკომფორტს შენახული პროცედურების შექმნისას და გამოყენებისას წარმოადგენს ის ფაქტი, რომ საჭიროა დაპროგრამების ენების ცოდნის უფრო დიდი გამოცდილება. მაგალითად, პროგრამისტმა შესაძლებელია კარგად იცის Java და SQL, მაგრამ იმავე დროს არ გააჩნდეს Oracle -ის PL/SQL ენასთან მუშაობის გამოცდილება. შენახული პროცედურების გამოყენება იძლევა საბოლოო პროდუქტის მოქმედების ყველაზე მაღალი სიჩქარეების მიღწევას, მაგრამ კონკრეტული პროგრამული პროდუქტის შექმნაზე დახარჯული დროის შესამცირებლად პროგრამისტთა ჯგუფმა შესაძლებელია გადაწყვიტოს SQL ოპერატორების პირდაპირი ჩართვა პროგრამულ კოდში (რომელიც

შესრულებულია მაგალითად Java ან C++ ენაზე), იმის ნაცვლად რომ დაწეროთ შესაბამისი შენახული პროცედურები.

შენახული პროცედურის შესაქმნელად გამოიყენება SQL-ოპერატორი CREATE. იმისათვის რომ რეალურად დავინახოთ შენახული პროცედურის კოდი, ქვემოთ მოყვანილია შენახული პროცედურა Oracle მონაცემთა ბაზებისათვის (ეს კოდი მოყვანილია მხოლოდ საილუსტრაციო მიზნებისათვის):

```
Create or Replace Procedure Record_sale
(
v_CustomerName IN Customer.Name%TYPE,
v_Artist IN Artist.Name%TYPE,
v_Title IN Work.Title%TYPE,
v_Copy IN Work.Copy%TYPE,
v_Price IN NUMBER,
v_Return OUT varChar2 --Return message to caller
)
AS
recCount int;
v_TransactionFound Boolean;
v_CustomerID Art_Customer.CustomerID%TYPE;
v_WorkID Work.WorkID%TYPE;
v_ArtistID Artist.ArtistID%TYPE;
v_SalesPrice Transaction.SalesPrice%TYPE;
v_testSalesPrice Transaction.SalesPrice%TYPE;
CURSOR TransactionCursor IS
SELECT SalesPrice
FROM Transaction
WHERE WorkID = v_WorkID
FOR UPDATE OF SalesPrice, CustomerID, PurchaseDate;
BEGIN
/*
Selecting and then looking for NULL does not work
because finding no qualifying records results in
Oracle throwing a NO_DATA_FOUND exception.
So, be ready to catch the exception by creating
an 'anonymous block' with its own EXCEPTION clause.
*/
BEGIN
SELECT CustomerID INTO v_CustomerID
FROM Art_Customer
WHERE Art_Customer.Name = v_CustomerName;
EXCEPTION
WHEN NO_DATA_FOUND THEN
SELECT CustomerSeq.nextval into v_CustomerID from Dual;
INSERT INTO Art_Customer (CustomerID, Name)
VALUES ( v_CustomerID, v_CustomerName );
END;
SELECT ArtistID into v_ArtistID
FROM Artist
WHERE Artist.Name = v_Artist;
SELECT WorkID INTO v_WorkID
FROM Work
WHERE Work.Title = v_Title
AND Work.Copy = v_Copy
AND Work.ArtistID = v_ArtistID;
--We need to use a cursor here, because a work can re-enter the
-- gallery, resulting in multiple records for a given WorkID.
--Look for a Transaction record with a null for SalesPrice:
v_TransactionFound:= FALSE;
FOR Trans_record in TransactionCursor LOOP
IF( Trans_Record.SalesPrice is null) THEN
v_TransactionFound:= TRUE;
UPDATE Transaction SET
SalesPrice = v_Price,
```

```
CustomerID = v_CustomerID,
PurchaseDate = SYSDATE
WHERE CURRENT OF TransactionCursor;
END IF;
EXIT WHEN v_TransactionFound;
END LOOP;
IF( v_TransactionFound = FALSE ) THEN
v_Return:= 'No valid Transaction record exists.';
ROLLBACK;
RETURN;
END IF;
COMMIT;
v_Return:= 'success';
EXCEPTION
WHEN NO_DATA_FOUND THEN
v_Return:= 'Exception: No data found';
ROLLBACK;
WHEN TOO_MANY_ROWS THEN
v_Return:= 'Exception: Too many rows found';
ROLLBACK;
WHEN OTHERS THEN
v_Return:= ( 'Exception: ' || SQLERRM );
ROLLBACK;
END;
```

თქვენ შესაძლებელია იცნოთ ზოგიერთი SQL ოპერატორი ამ პროცედურაში, ამასთანავე თქვენ ხედავთ ოპერატორებს რომლებიც საერთოდ არ გვანან იმ SQL ოპერატორებს, რომლებიც ჩვენ ზემოთ განვიხილეთ. PL/SQL წარმოადგენს გაცილებით რთულ ენას ვიდრე SQL-ი. სხვა მონაცემთა ბაზების მართვის სისტემებს გააჩნიათ SQL-ის პროცედურული სახით გაფართოების თავისი საკუთარი ენები. Microsoft-ის შემთხვევაში, მაგალითად, შესაბამის ენას Transact-SQL -ი ჰქვია. შემდეგ პარაგრაფში, რომელიც ეხება ტრიგერებს (trigger), ჩვენ გავეცნობით Transact-SQL გამოყენების მაგალითს.

ტრიგერები (Triggers)

ტრიგერი წარმოადგენს სპეციალური ტიპის შენახულ პროცედურას, რომელიც სრულდება მონაცემთა ბაზის რაიმე მონაცემური პირობების ცვლილებებისას. ტრიგერები გამოიყენება გარკვეული წესების დასაცავად მონაცემთა ბაზაში. მაგალითად წარმოვიდგინოთ, რომ სხვადასხვა სტუდენტური სასტუმროების ოთახების ნომრებს გააჩნიათ განსხვავებული დომენები (domain). ეს ნიშნავს, მაგალითად, რომ ერთ სასტუმროში გამოყენებულია ორნიშნა ნომრები, მეორეში - სამნიშნა ნომრები, მესამეში - ოთხნიშნა ნომრები, ... ყველა ოთახის ნომრების ავტომატური დადასტურება შეუძლებელია სტანდარტული CHECK შეზღუდვის გამოყენებით, რადგანაც CHECK შეზღუდვა ვერ უზრუნველყოფს ასეთ რთულ ლოგიკას. ამ შემთხვევაში შესაძლებელია ტრიგერის დაწერა, რომელსაც ექნება ნებისმიერი დონის სირთულის ლოგიკა.

შენახული პროცედურებისაგან განსხვავებით, რომლებიც სრულდებიან როდესაც მათ გამოიძახებს მომხმარებელი ან პროგრამა, ტრიგერები სრულდებიან მაშინ როდესაც ცხრილებში ხდება ცვლილებები INSERT, UPDATE ან DELETE ოპერატორების საშუალებით. ტრიგერების შესრულება შესაძლებელია დაიწყოს ამ ოპერატორების შესრულებამდე, შესრულების შემდეგ ან მათ მაგივრად, ანუ არსებობენ ტრიგერები BEFORE (INSERT, UPDATE ან DELETE ოპერატორების შესრულების დაწყებამდე), AFTER (INSERT, UPDATE ან DELETE ოპერატორების შესრულების შემდეგ), INSTEAD OF (INSERT, UPDATE ან DELETE ოპერატორების შესრულების ნაცვლად).

ქვემოთ მოყვანილია AFTER ტრიგერი დაწერილი Microsoft Transact-SQL ენაზე. ყოველთვის, როდესაც Student ცხრილში ხდება სტრიქონის დამატება ან სტრიქონის განახლება, ჩვენი ტრიგერი სრულდება Student ცხრილში ცვლილებების განხორციელების შემდეგ. მონაცემთა ცვლილება იწვევს შესაბამისი კოდის შესრულების დაწყებას (გაშვებას). ეს კოდი მხოლოდ საილუსტრაციო მიზნებისათვის არის გათვალისწინებული და ამიტომ ჩვენ არ დავიწყებთ მისი სინტაქსის დაწვრილებით გარჩევას.

```
CREATE TRIGGER RoomCheck ON Student
FOR INSERT, UPDATE
```

```
AS
declare @Dorm varchar(20)
declare @Room int
IF UPDATE (Room)
Select @Dorm = Dorm from inserted
Select @Room = Room from inserted
IF @Dorm = 'Williams' and (@Room > 999 or @Room < 100)
BEGIN
PRINT 'Williams dorm has 3 digit room numbers.'
ROLLBACK TRAN
RETURN
END
IF @Dorm = 'Appleby' and (@Room > 9999 or @Room < 1000)
BEGIN
PRINT 'Appleby dorm has 4 digit room numbers.'
ROLLBACK TRAN
RETURN
END
IF @Dorm = 'Arpers' and (@Room > 99 or @Room < 10)
BEGIN
PRINT 'Arpers dorm has 2 digit room numbers.'
ROLLBACK TRAN
RETURN
END
```

მონაცემთა მთლიანობა (Data Integrity)

მონაცემთა ბაზების მართვის სისტემები უზრუნველყოფენ ინსტრუმენტებს მონაცემთა მთლიანობის შენარჩუნებისათვის. საბაზისო წესების მნიშვნელოვან სიმრავლეს, რომელიც უზრუნველყოფს ბაზის მონაცემთა მთლიანობასა და არაწინააღმდეგობრივობას, კავშირთა (მითითებათა) დონეზე მთლიანობის შეზღუდვები ეწოდება (*referential integrity constraints*). მონაცემთა თანმიმდევრულობისა და მთლიანობის დამცავი წესებს მიეკუთვნებიან შემდეგი ჩაშენებული წესები:

1. ახალი სტრიქონის დამატება მშობელ-ცხრილში ყოველთვის ნებადართულია.
2. ახალი სტრიქონის დამატება შვილ-ცხრილში ნებადართულია მხოლოდ იმ შემთხვევაში თუ შესაბამისი გარე გასაღები არსებობს მშობელ-ცხრილში.
3. მშობელი-ცხრილიდან სტრიქონის ამოშლა ნებადართულია მხოლოდ იმ შემთხვევაში თუ მას არ გააჩნია შვილი სტრიქონები შვილ-ცხრილში.
4. შვილი-ცხრილიდან სტრიქონის ამოშლა ყოველთვის არის ნებადართული.
5. პირველადი გასაღების განახლება მშობელ-ცხრილში ნებადართულია მხოლოდ მაშინ თუ არ არსებობენ შვილი სტრიქონები.
6. შვილი სტრიქონის გარე გასაღების განახლება ნებადართულია იმ შემთხვევაში, თუ გარე გასაღების ახალი მნიშვნელობა არსებობს მშობელ-ცხრილში.

თუ ჩვენ მონაცემთა ბაზის დიზაინერი ვართ, მაშინ ჩვენ შეგვიძლია ეს საბაზისო შეზღუდვები სავალდებულო გავხადოთ, რასაც დიდი მნიშვნელობაა აქვს მონაცემთა მთლიანობის შენარჩუნებისათვის. ხშირ შემთხვევებში ბიზნეს-წესების უზრუნველსაყოფად საჭირო ხდება დამატებითი შეზღუდვების შემოტანა.

მონაცემთა ბაზების მართვის სისტემები იძლევიან კიდევ ერთ შესაძლებლობას მონაცემთა მთლიანობის უზრუნველსაყოფად. ამ შესაძლებლობას ტრანზაქცია (transaction) ეწოდება. ტრანზაქცია წარმოადგენს მონაცემთა ბაზის ერთმანეთთან დაკავშირებული ცვლილებების დაჯგუფების სპეციალურ მექანიზმს, რომელიც გამოიყენება იმ შემთხვევაში როდესაც ან ყველა ცვლილება უნდა განხორციელდეს, ან საერთოდ არავითარი ცვლილება არ უნდა მოხდეს.

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

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

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

ზოგიერთი მონაცემთა ბაზის მართვის სისტემა იყენებს წინასწარ რეგისტრაციას (ცვლილებების წინსწრებით რეგისტრაციას), მაგრამ ამასთან ერთად ახორციელებს მონაცემთა ბაზის ფაქტიურ ცვლილებას ტრანზაქციის ფორმალურ დაფიქსირებამდე. ასეთ სისტემებში ავარიული სიტუაციის შემდეგ აღდგენა შესაძლებელია განხორციელდეს მონაცემთა ბაზების მართვის სისტემის გადატვირთვით და სარეგისტრაციო ჟურნალში არსებული ყველა იმ ტრანზაქციების გაუქმებით, რომლებიც არ დაფიქსირდა. ასეთ მიდგომას დაუფიქსირებელი ტრანზაქციების გაუქმებით აღდგენას უწოდებენ (rollback recovery).

ტრანზაქციის იზოლირების დონეები (Transaction Isolation Levels)

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

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

კიდევ ერთი განსხვავებული სახის პრობლემაა არაგანმეორებადი წაკითხვა (nonrepeatable read), რომელიც ხდება მაშინ როდესაც ტრანზაქციას თავისი მუშაობის შესრულების დროს ერთსა და იგივე მონაცემებს ორჯერ კითხულობს და ადგენს რომ მონაცემები შეცვლილია მეორე წაკითხვის დროს. ეს შესაძლებელია მოხდეს მაშინ, როდესაც ამ ტრანზაქციის მუშაობის დროს სხვა ტრანზაქცია ახდენს მონაცემების ცვლილებას.

იგივე პრობლემას წარმოადგენს მოჩვენებითი წაკითხვა (phantom read). თუ თავისი მუშაობის დროს ერთი ტრანზაქცია კითხულობს ჩანაწერების რომელიმე სიმრავლეს ორჯერ, მაშინ მან შესაძლებელია წაიკითხოს ახალი ჩანაწერები მეორე წაკითხვისას. ეს შესაძლებელია მოხდეს იმ შემთხვევაში, თუ პარალელურად მონაცემთა ბაზაში სხვა ტრანზაქციას ახალი ჩანაწერები შეაქვს.

ამ ტიპის ყველა პრობლემის გადაჭრა ხორციელდება ჩაკეტვის მექანიზმებით (locking mechanisms), რომლებიც უზრუნველყოფენ მომხმარებლებისა და ტრანზაქციების ერთმანეთისაგან იზოლირებას. ძველ დროს პროგრამისტებს საჭირო დაცვის განსახორციელებლად ჩაკეტვების მართვის საკუთარი სისტემების შემოტანა სჭირდებოდათ, მაგრამ დღეს ჩაკეტვების მართვას მთლიანად მონაცემთა ბაზების მართვის სიტემა ახორციელებს. დაკარგული განახლებების პრობლემას

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

ტრანზაქციათა იზოლაციის დონეებია: read uncommitted, read committed, repeatable read, serializable.

Read uncommitted დონე არ უზრუნველყოფს დაცვას იმ დამთხვევათა პრობლემებისაგან, რომლებიც განვიხილეთ, მაგრამ უზრუნველყოფს მოქმედების მაქსიმალურ სისწრაფეს.

Read committed დონე გარანტირებულად არიდებს თავს შეცდომითი წაკითხვების პრობლემას, რადგანაც იზოლაციის ამ დონეზე ნებადართულია მხოლოდ დაფიქსირებული ტრანზაქციებით შეცვლილი მონაცემების წაკითხვა.

Repeatable read დონე იცილებს ასევე არაგანმეორებადი წაკითხვების პრობლემასაც.

Serializable დონე უზრუნველყოფს ერთდროული ტრანზაქციების სრულ განცალკევებას, რაც ხორციელდება ტრანზაქციაში მონაწილე ყველა სტრიქონის ჩაკეტვით. ასეთი დონის უსაფრთხოება ახდენს მნიშვნელოვან ზეგავლენას იმ პროგრამების მოქმედების სისწრაფეზე, რომლებიც ერთდროულად ემსახურებიან მრავალ მომხმარებელს.

ჩვეულებრივ ყველაზე უფრო ხშირად გამოიყენებენ read committed იზოლაციის დონეს.