БАЗИ ОТ ДАННИ

Задължителен курс за специалност Математика и информатика

Изборен курс за специалности Математика, Приложна математика и Статистика

гл. ас. д-р Моника Филипова

Катедра Изчислителни системи, ФМИ

летен семестър на уч. 2009/2010г.

СЪДЪРЖАНИЕ

1. Архитектура на система за управление на бази данни

1.1. Понятие за база данни

База от данни (БД) е интегрирана съвкупност от взаимосвързани данни, съхранявани на електронен носител, които представят всички факти за дадена предметна област, представляващи интерес и използвани от много потребители посредством приложни програми. Предметна област (ПО) е тази част от реалния свят, за която се съхранява информация в БД. Това може да е промишлено предприятие, държавно ведомство, университет, болница, малка фирма или голяма корпорация. Основните характеристики на данните в БД, които следват от това определение са:

- Интегрирани – В БД са събрани данни за ПО, обединени от различни източници, например различни подразделения или дейности на ПО.
- Общи – Едни данни, в едно и също време са достъпни за много потребители за различни цели.

Какви преимущества дава използването на БД? Ще разгледаме други важни характеристики на БД, които ни позволяват да разглеждаме БД като нещо повече от група файлове на диска.

- **Централизирано управление на данните за ПО**

Преди появата на БД, за различните нужди на ПО (приложни програми или системи) се създават и поддържат собствени файлове и никой не се грижи за синхронизиране на данните, използвани в различните приложения. В системите за управление на БД има лице/група хора, наречено администратор на БД (АБД, DBA), който отговаря за цялостното управление на данните. В групата на АБД влизат технически специалисти в областта на информационните технологии, които определят съдържанието на БД така, че тя най-точно да представя ПО и поддържат БД в коректно състояние.

- **Намалено и контролирано излишество на данните**

Преди появата на БД е характерно съществуването на неконтролирано излишество на данните в ПО, една и съща информация за ПО се съхранява за различни нужди и цели в различни файлове от различни поделения на предприятието. Например персонални данни за служителите в предприятието, като имена, адрес, дата на раждане, и др. са необходими както в отдел кадри така и във финансово-счетоводен отдел. Всеки от тези отдели има собствена информационна система, която използва собствени файлове, съдържащи необходимата й информация. Недостатъците от такава повторяемост са много: разход на външна памет; многократно и скъпо обновяване на данните; възможна противоречивост на информацията, получавана от различните приложения. В БД с интегриране на данните се намалява повторението, но все още е възможно да съществува минимално излишество с цел повишаване на ефективността или надеждността на системата. Но дори и да съществува, това е управлявано излишество, т.е. АБД знае за него и системата за управление на базата данни (СУБД) го поддържа.

- **Осигуряване на сигурност на данните**

Сигурност на данните (Data Security) е понятие, свързано със защитата на данните в БД. Под сигурност се разбира защита на данните от неправомерен достъп, изменение или унищожаване. АБД определя правата на потребителите за достъп до БД, като описва за всеки потребител към кои данни какъв достъп има право. Това са така наречените ограничения за сигурност. СУБД следи за спазването на тези ограничения, като извършва проверка при всеки опит за достъп до БД.

- **Осигуряване на цялостност на данните**

Цялостност на данните (Data Integrity) също е понятие, свързано със защитата на данните в БД. Под цялостност се разбира защита на данните от неправилни изменения, т.е. целта е в БД да се съхраняват само правилни (правдоподобни) и непротиворечиви данни. С отстраняване на неуправляемото излишество на данни, се намалява и опасността за противоречивост на данните, но въпреки това данните в БД може да не са верни. АБД определя ограничения за цялостност – условия, на които трябва да отговарят данните, за да се счита че са правдоподобни. Например, сумата в банкова сметка да е положително число или годината на раждане на служител да е преди текущата година. Тези ограничения се проверяват при всеки опит за обновяване на БД – добавяне, изтриване или изменение на данни, и съответната операция се изпълнява само ако не е нарушено нито едно ограничение за цялостност. Но както се досещате това не винаги гарантира правилни данни, затова използваме думата правдоподобни.

- **Независимост на данните**

Под независимост на данните (Data Independence) се разбира изолиране на програмите от изменения в структурата на данните, които обработват. За да обясним какво е независимост ще разгледаме противоположен пример. Нека имаме програма, написана на някакъв език за програмиране, без използване на база данни, която обработва файл съдържащ записи за служители. Записите във файла са в хронологичен ред (в реда на добавяне). След време се оказва, че програмата работи бавно и се налага да се промени организацията на файла, напр. записите да са сортирани по ЕГН на служителя. Или се появява ново приложение, за което е необходимо да се добави ново поле в записа на служителя, напр., e-mail адрес. Такива промени в начина на съхраняване или структурата на данните най-вероятно ще наложат преработка на програмата, т.е. програмата зависи от данните. Защо е желателно да има независимост, особено при използване на БД?

- БД се развива

При създаване на БД е трудно да се предвидят всички възможни приложения. Освен това самата ПО се развива, появяват се нови приложения, изменя се приоритета на съществуващите, променят се нормативни актове и закони, което се отразява на дейността в ПО и естествено на БД. Всичко това налага изменения във физическата организация на данните или добавяне на нови типове данни.

- Данните в БД се използват от много потребители

Едни и същи данни се използват и е възможно да се представят различно от различните приложни програми. Например, отдел кадри иска да вижда датата на раждане на служителя, а ФСО вижда същите данни като възраст на служителя.

Има две нива на независимост:

- Физическа независимост – Осигурява изолиране на програмите от изменения във физическото представяне на данните.
- Логическа независимост – Означава измененията в логическата структура на данните, в резултат от развитието на БД, да не оказват влияние на разработените приложни програми.

Независимостта на данните е относително понятие, различните СУБД осигуряват независимост в различна степен или не я осигуряват въобще. Съвременните системи са по-развити в това отношение. В следващия раздел ще разгледаме архитектурата на ANSI/SPARC, която е основа за постигане на по-висока степен на независимост.

1.2. Компоненти в система за управление на БД и архитектура на ANSI/SPARC

Една система за управление на БД включва различни видове компоненти:

- **Програми**
  1. Ядро на СУБД (database server, database engine, сървър) – сложна програмна система, осигуряваща съхранение и достъп до данните в БД, сигурност и цялостност на данните в условията на многопотребителски, конкурентен достъп.
  2. Сервизни програми (утилити) – средства за АБД, с които той изпълнява дейности по поддържане на БД, като:
     - настройка и тестване на системата;
     - създаване на резервни копия на БД;
     - възстановяване след аварийни ситуации и повреда;
     - реорганизация на БД;
     - събиране на статистики за функционирането и производителноста.
  3. Инструментални средства - средства за разработка на приложения, като:
     - процесори на езици за програмиране;
     - процесори на езици за БД;
     - генератори на отчети и др.
  4. Приложни програми – програми, създадени чрез горните средства, които реализират конкретните обработки на данните в БД, осигуряващи удобен потребителски интерфейс за работа с БД, създаването на отчети и др.
- **Потребители**
  1. АБД – Това е групата отговорна за управлението на БД на техническо ниво, т.е. осигурява създаването, функционирането и развитието на БД.
  2. Приложни програмисти (DB application programmers) – Това са програмистите, които разработват приложните програми.
  3. Крайни потребители (end users) – Това са потребителите, за които се създава БД.
- **Модели на данните**

Модел на данните (Data model) е съвкупност от абстрактни обекти данни, които представят обектите, свойствата и връзките в определена ПО. Един от ключовите въпроси в областта на БД е: Какви да са тези абстрактни обекти данни? Отговорът е важен, тъй като той определя понятията и начина, по който потребителят си представя данните в БД. В една система съществуват различни нива на абстрактно представяне на данните между компютъра и човека. През 1975г. ANSI/SPARC публикува документ, в който въведе три нива на представяне на БД и съответно три вида модели на данните:

1. Вътрешен модел на данните (физически). На това ниво БД се представя чрез понятия близки до физическите, напр., файлове на операционната система с определена организация, индекси и др. И вътрешният модел е абстрактен, защото не се използват машинните понятия, като бит, байт, сектор. Вътрешният модел на БД е един.

2. Концептуален модел на данните (логически). На това ниво БД се представя чрез понятия на по-високо ниво на абстракция, без да се конкретизира физическото представяне на данните. Концептуалният модел на БД е един.

3. Външни модели на данните. Това са моделите на потребителите. Всеки потребител или група потребители със сходни интереси, има свой външен модел, който определя начина, по който той вижда част от БД. Понятията на външния модел са на същото ниво на абстракция, както и в концептуалния. Разликата е, че

концептуалният модел е глобален, а външните са локални в смисъл, че съответстват на индивидуалните представи на потребителите за БД.

- **Езици за БД**

Езикът е интерфейс на потребителите към БД. Някои автори използват термина подезик на данните, с което показват, че това не е пълноценен език за програмиране, а само осигурява инструмент за работа с данните. По-често се използва наименованието Език на заявките или Query Language като общо наименование на всички езикови средства за БД (макар, че това наименование не е съвсем коректно). Езиковите средства обикновено се разделят в две основни групи:

1. **Език за описание на данните** (Data Definition Language)

Това са езиковите средства, с които се описват моделите данни. Формалното и пълно описание на модел на данните се нарича схема (schema). Следователно според архитектурата на ANSI/SPARC всяка БД се описва с една концептуална схема, една вътрешна схема и няколко външни схеми (използва се и термина подсхема). Освен тези три вида схеми описанието включва и два вида изображения:

- Изображението концептуален-вътрешен определя по какъв начин обектите в концептуалния модел се материализират от тези във вътрешния модел.
- Изображението външен-концептуален аналогично определя начина, по който външните обекти се конструират от концептуалните.

Наличието на междинния концептуален модел в архитектурата е основа за постигане на по-висока степен на независимост на данните. Изменение във физическото представяне на данните ще доведе до промяна във вътрешната схема и евентуално изображението концептуален-вътрешен, което означава, че концептуалната схема остава непроменена и следователно това няма да се отрази на съществуващите приложни програми. Аналогично добавянето на нови обекти в концептуалния модел ще се реализира чрез разширение на концептуалната схема и изменения в изображенията външен-концептуален, но външните модели ще останат непроменени, което означава логическа независимост.

Обикновено езикът за описание на данните включва и средства за описание на ограниченията за цялостност и ограниченията за сигурност.

2. **Език за манипулиране на данните** (Data Manipulation Language)

Това са езиковите средства за обработка на данните в термините на външния модел, а именно за търсене и извличане, добавяне, изтриване и изменение на данни. Езиковите средства се реализират в СУБД по два начина, а именно като:

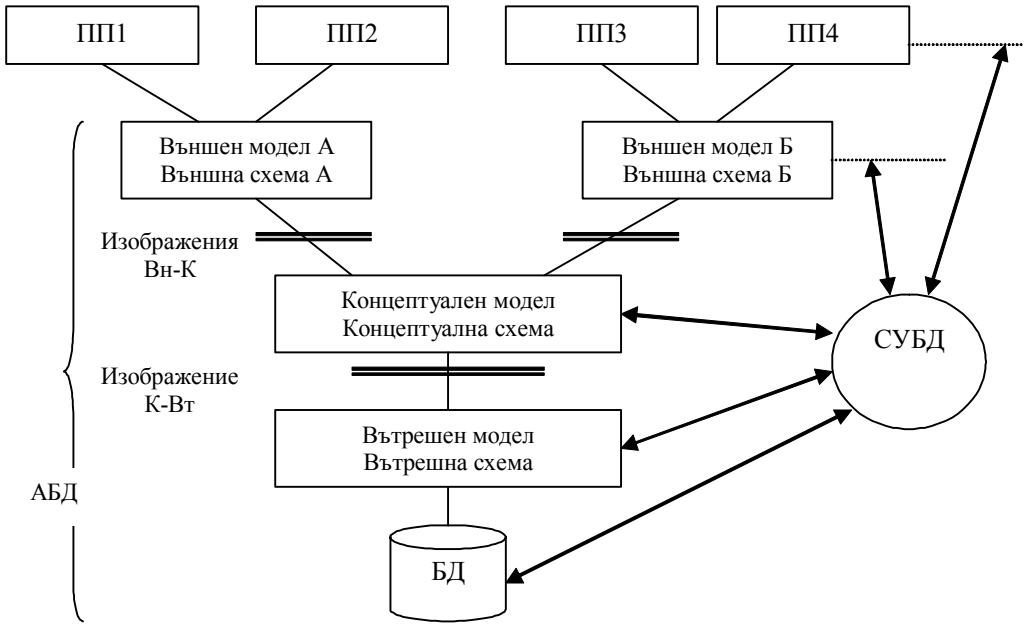
- Независим език. Самостоятелни програми на езика на заявките се изпълняват от специални процесори на езика, които са част от инструменталните средства доставяни със СУБД.
- Вграден (Embedded Query Language) или програмен език на заявките. Програмата на езика на заявките се съдържа в друга програма, написана на традиционен език за програмиране (най-често C, C++, Java).

Да се върнем към ролята на АБД. Неговите отговорности по управление на данните включват следните основни задачи:

- Проектира БД – проектира концептуалния модел, вътрешния модел и описва концептуалната, вътрешната схема и изображението концептуален-вътрешен.
- Взаимодейства с потребителите, за да определи информационните им потребности и да опише външните схеми и изображенията външен-концептуален.
- Определя ограниченията за сигурност.

- Определя и изпълнява процедурите за резервно копиране и възстановяване на БД след повреда.
- Отговаря за настройка на системата при изменения на изискванията към нея, т.е. на етапа на експлоатация отговаря за ефективното функциониране на системата.

Фиг.1 представя основните компоненти на една система за БД и ролята на АБД.

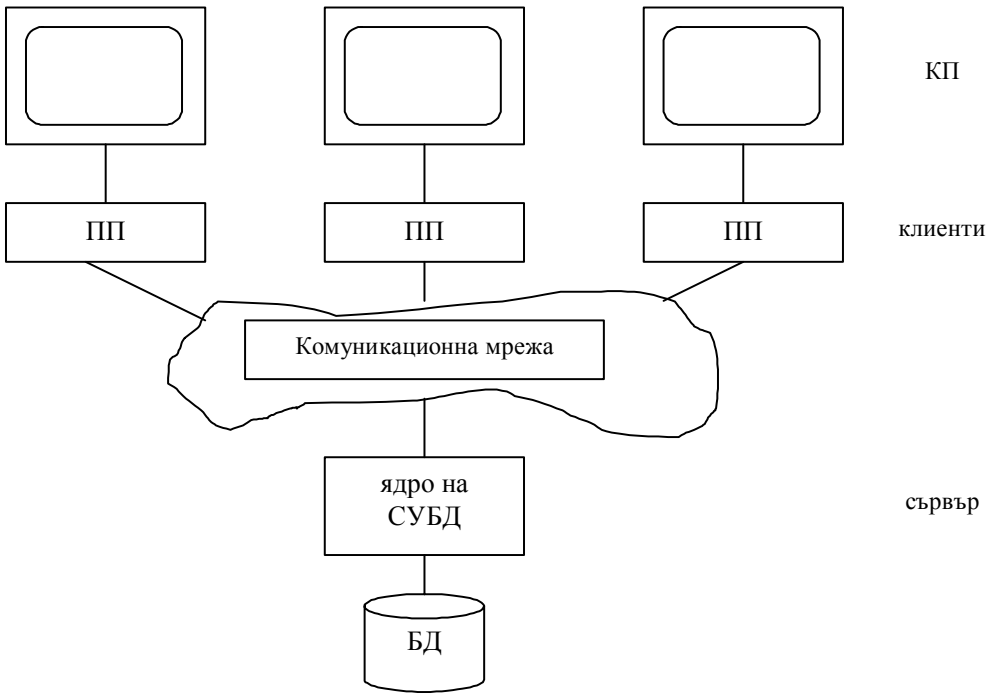


Фиг. 1. Архитектура на Система за БД

1.3. Архитектура клиент/сървер

Ако разглеждаме архитектурата на система за БД на най-високо ниво, то тя е представена на Фиг.2 и е система, включваща два типа елементи:

- Сървър – ядро на СУБД
- Клиенти – приложни програми, инструментални средства, утилити



Фиг. 2. Архитектура клиент/сървър на Система за БД

2. ER модел (Entity Relationship Model)

При проектиране на база от данни много често се използва низходящ подход. Започва се на по-високо ниво на абстракция с изясняване на семантиката на нещата в предметната област и след това се преминава на сравнително по-ниско ниво на абстракция, представено от модела на данните. Съществува група от модели, известни като "семантични" или "информационно-логически", чиято основна роля при проектиране на БД е да изясни смисловото значение на данните, които ще се съхраняват в БД и трябва възможно най-точно да представят фактите в ПО. Един от най-популярните такива модели е ER модела, предложен от Чен през 1976г. и след това нееднократно усъвършенстван от Чен и от други изследователи в областта на БД. Характерно за този модел е използването на графична нотация, т.е. представяне на ER модела чрез ER диаграма. В литературата се използват леко различаващи се графични нотации за ER диаграмите. Ние ще използваме нотация, която е близка до оригиналната, предложена от Чен.

ER моделът е неформален модел, т.е. използва неформални, интуитивни понятия. Следва изложение на основните понятия в модела.

Entity / същност / обект

Това е нещо, което съществува реално, различимо е, представлява интерес за ПО и за което ще се съхранява информация в БД. Обектът може да е нещо конкретно – служител, студент, банкова сметка, клиент, доставчик, стока или абстрактно понятие.

Entity set / множество същности / клас обекти / тип обекти

Това е множеството от всички подобни обекти в ПО, т.е. обекти притежаващи общи свойства. Например, всички служители в предприятието, всички клиенти на фирмата, всички студенти във факултета. В ER диаграма клас обекти се представя чрез именован правоъгълник.

Attribute / атрибут / свойство

Обектите се характеризират чрез свойства, т.е. свойството е някакъв факт за обекта. Напр., обект служител може да има свойствата – име, ЕГН, пол, длъжност, адрес и т.н. Свойство (свойства), чието значение еднозначно идентифицира обект в рамките на определен клас обекти се нарича ключово свойство или **ключ**. За всеки клас обекти трябва да може да се определи поне един ключ, тъй като обектите по дефиниция са различими. Напр., за клас обекти служители ключ може да е ЕГН или някакъв вътрешен (служебен) номер на служителя, който се използва в предприятието; ключ на банкова сметка е номера, ключ за студента е факултетния номер. Могат да се различават следните видове свойства (в зависимост от значенията, които могат да приемат):

- Ключово или не ключово свойство
- Просто или съставно. Ако значението на свойството е неделимо от гледна точка на ПО, то е просто, в противен случай е съставно. Напр., свойството "име на служител" може да се разглежда като съставно, тъй като се състои от простите свойства "първо име", "презиме" и "фамилия".
- Еднозначно или многозначно. Свойството е многозначно, ако може да има няколко значения за един обект.

В ER диаграма свойство се представя чрез именована елипса, която е свързана чрез линия с правоъгълника на класа обекти. Ключът за всеки клас обекти се показва чрез подчертаване.

Relationship / връзка

Връзката свързва или асоциира обекти (2, 3 или повече) от различни класове или от един и същи клас. Най-често се използват бинарни връзки, т.е. всяка връзка свързва два обекта. Напр., връзка между служител и отдел се установява, когато служителят бъде назначен в отдела, връзка между клиент и сметка - когато клиентът си открие сметка. Връзките също могат да имат свойства, т.е. факт, който характеризира връзката между обектите, а не самите обекти. Напр., свойство на връзката между служител и отдел може да е дата на назначаване в отдела. Връзките също характеризират обектите.

Relationship set / множество връзки / клас връзки / тип връзки

Това е множеството от всички подобни връзки, които съществуват между два или повече класа обекти и имат еднаква семантика.

Вид на връзка

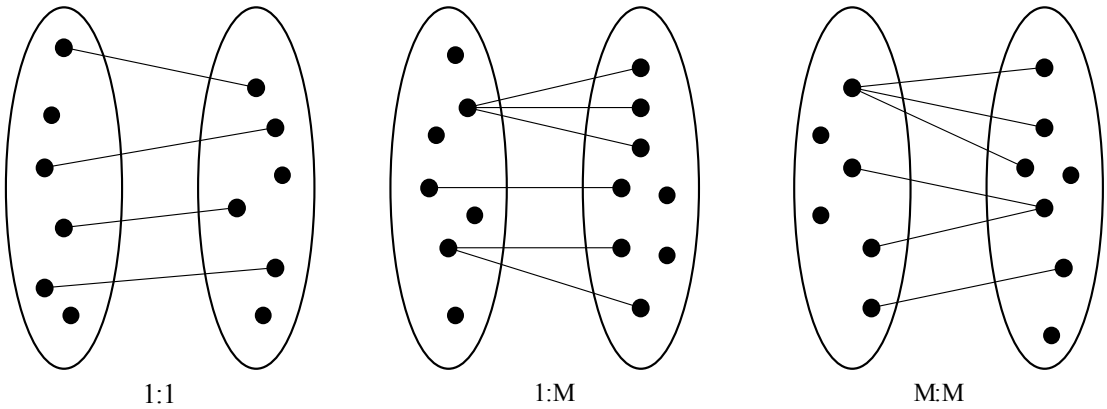
Класовете връзките имат различен вид в зависимост от това, обект от единия клас с колко обекта от другия може да бъде свързан в определен момент и обратно, и дали участието на обект във връзка от определен клас е задължително или не. Според първия критерий връзките биват от вид 1:1 (едно към едно), 1:M (едно към много) или M:M (много към много). Нека R е клас връзки между класовете обекти E1 и E2.

R е от вид **1:1** ако всеки обект от E1 е свързан най-много с един обект от E2 и обратно.

R е от вид **1:M** ако всеки обект от E1 може да е свързан с много обекти от E2, но всеки обект от E2 е свързан най-много с един обект от E1.

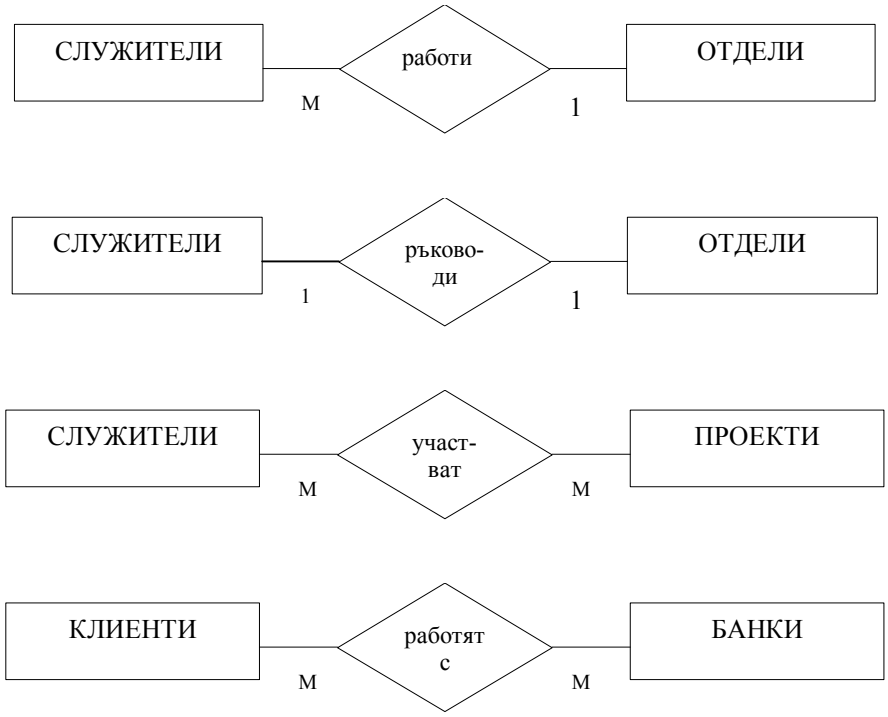
R е от вид **M:M** ако всеки обект от E1 може да е свързан с много обекти E2 и обратно, т.е. няма ограничения за броя на връзките, в които обект може да участва.

Според втория критерий (Participation constraint) участието на клас обекти в клас връзки може да е **пълно** (total) или **частично** (partial). Ако всеки обект от E1 участва поне в една връзка от клас R, се казва че участието на E1 в R е пълно, иначе участието е частично.



В ER диаграма клас връзки се представя чрез именован ромб, който е свързан с класовете обекти чрез линии, а вида на връзката се обозначава чрез надпис над линиите. Когато участието на класа обекти е пълно линията е двойна.

Следват примери за класове връзки от различни видове.



Слаб клас обекти / Weak entity set

Класовете обекти в ПО могат да бъдат класифицирани на независими класове (наричат ги още силни) и на слаби класове. Обектите от независим клас имат независимо съществуване в ПО. Слаб обект се нарича обект, който не съществува самостоятелно в ПО, т.е. съществуването му зависи от съществуването на друг обект. Някои ги наричат още характеризиращ клас, защото често целта на тяхното съществуване в ПО е да характеризират обектите от някой друг клас.

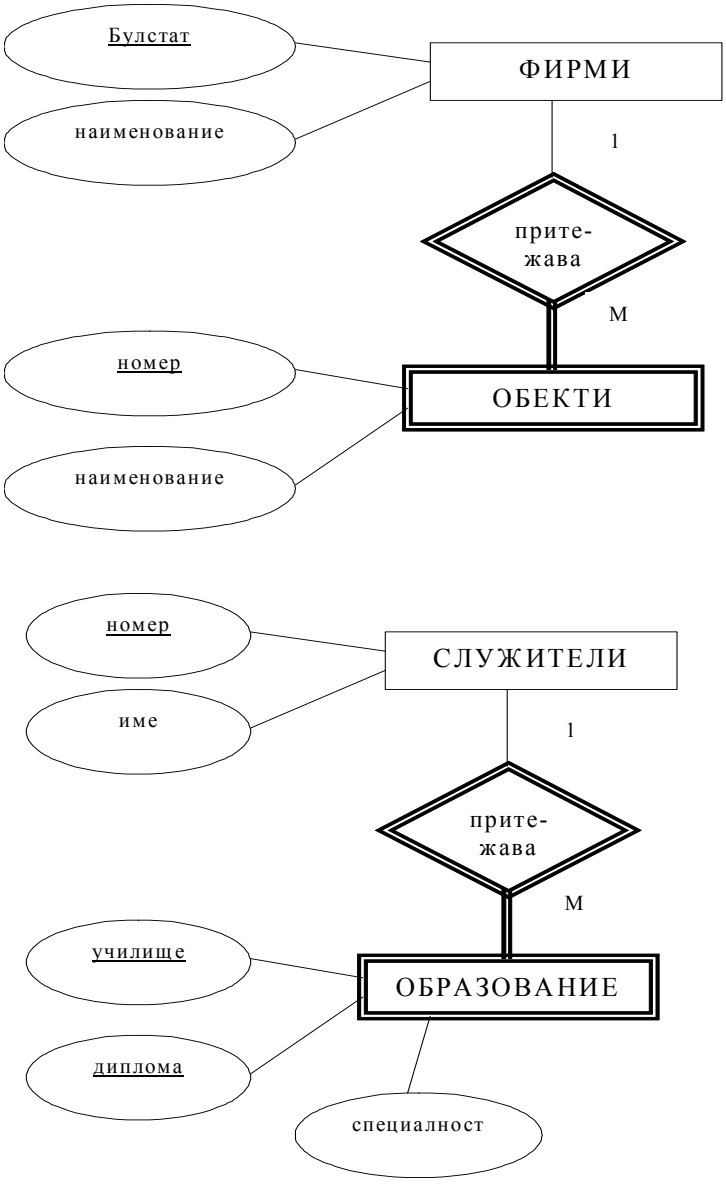
Например, в една данъчна БД независим клас обекти са фирмите. Всяка фирма има ред свойства, като Булстат, наименование, адрес на регистрация и т.н. Фирмите притежават различни обекти – магазини, заведения, хотели и др., като всяка фирма може да притежава няколко обекта, но всеки обект е собственост на една фирма и няма независимо съществуване.

Един друг пример, където може да се използва слаб клас обекти, е за представяне на многозначно и съставно свойство. Напр., образованието или квалификацията на служителите, може да се разглежда като слаб клас обекти със свойства – учебно заведение, дата на завършване, специалност, степен, и др.

Много често ключът на слабите обекти се формира от ключа на обекта, от когото зависи плюс собствени свойства на самия слаб обект. Напр., ключ на обект от клас ОБЕКТИ се състои от Булстат на фирмата и номер на обекта във фирмата.

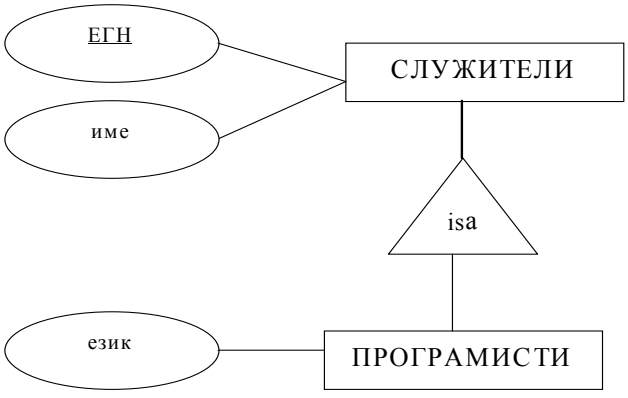
В ER диаграма слаб клас обекти се изобразява чрез двойна линия на правоъгълника. Връзката между слабия клас и класа, от когото той зависи, се изобразява чрез ромб с двойна линия.

Следват ER диаграмите на двата примера, в които могат да се използват слаби класове обекти.



Връзка isa / подклас обекти

Ако обектите в един клас могат да бъдат групирани в подкласове, всеки от които има собствени специфични свойства в допълнение към свойствата характеризиращи целия клас, тогава в ER модела може да се изгради йерархия от класове обекти, свързани чрез специална връзка “isa”.

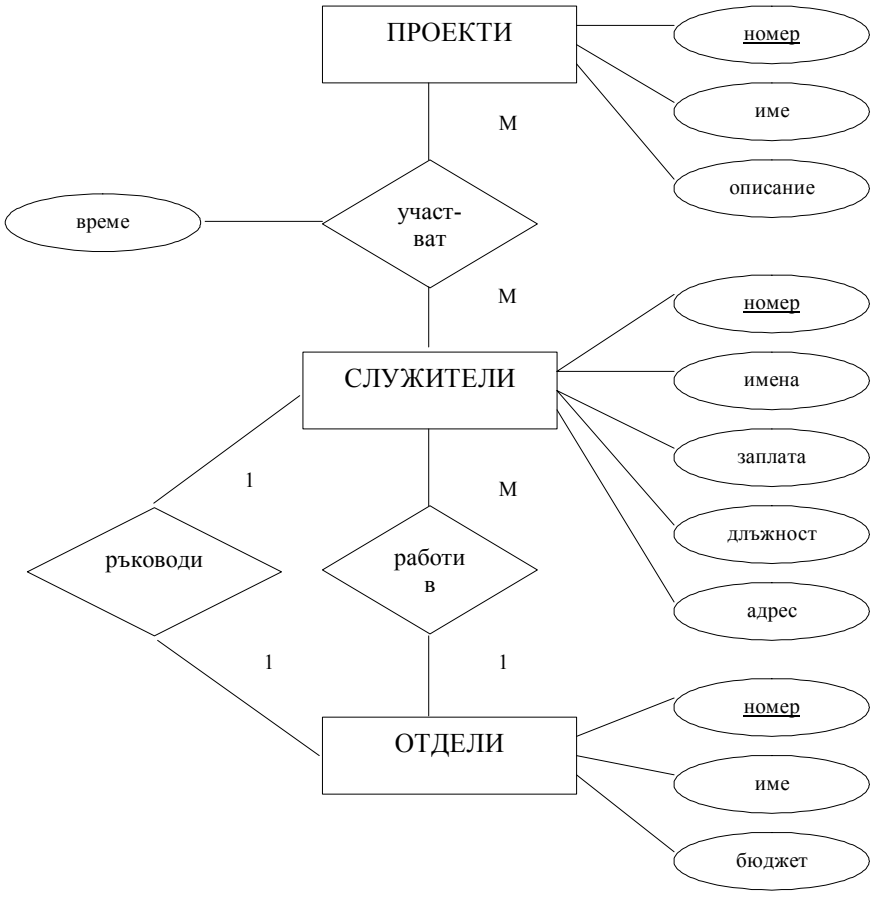


Всеки програмист е и служител, т.е. притежава всички свойства на класа СЛУЖИТЕЛИ, може да участва във всички връзки, в които участват служители. Казано по друг начин множеството на програмистите е подмножество на служителите. Това е концепция позната от обектно-ориентирания подход и според терминологията там СЛУЖИТЕЛИ е суперклас, а ПРОГРАМИСТИ е негов подклас. Подкласът наследява всички свойства и връзки от суперкласа.

ER моделът и диаграмата на определена ПО може да бъде изграден по два начина – възходящ и низходящ. При низходящо изграждане най-напред се определят основните класове обекти. След това се определят връзките, които съществуват между обектите и вида им. Най-накрая се детайлизират свойствата на обектите и връзките, като обикновено това не е еднократно действие. Възможно е по-късно да се добавят нови свойства с цел удоволетворяване на нови информационни потребности.

Една и съща концепция от реалния свят може да се възприема като обект от един потребител и като връзка от друг. Например, какво е "брак": връзка между двама човека, която притежава ред свойства, като дата и място на сключване, номер на свидетелство, или е обект. Аналогично, дали образованието на служител да се възприема като слаб клас обекти или като многозначно свойство на класа обекти служители. Когато се изгражда ER модел на ПО, решението се взема от АБД и следователно е субективно.

На Фиг.3 е представена ER диаграма на БД-служители, която по-нататък ще се използва като пример.



Фиг. 3. ER диаграма на БД-служители

3. Релационен модел на данните

3.1. Понятия в релационния модел

Релационният модел е предложен от Е. Код през 1970, когато излиза историческата му статия в CACM, която поставя нов период в областта на базите данни. Този модел има две важни характеристики, които го издигат до положението на основен подход в съвременната технология на БД, а именно:

- прост и естествен начин за представяне на данните – таблици.
- строга математическа основа на модела – теория на множествата и предикатната логика.

Нека разгледаме пример на две таблици, съдържащи съответно данни за отделите и служителите в предприятие.

dep

dno	**dname**	**budget**
1	Административен	20000
2	ФСО	10000
3	Продажби	50000

emp

eno	**ename**	**salary**	**dno**
100	Иванов	1500	1
120	Петрова	900	1
200	Антонов	800	2
210	Георгиева	900	2

Всяка таблица се състои от редове и колони, като значенията във всяка колона са сравними, т.е. принадлежат на едно множество от допустими значения. Обикновено всяка колона има наименование, което показва какъв е смисъла на значенията в колоната. Сега да разгледаме формалните понятия в релационния модел, които приблизително съответстват на неформалните понятия.

Неформални понятия	**Формални понятия**
table / таблица	relation / релация
row / ред	tuple / n-орка / ред
column / колона	attribute / атрибут
множество от допустими значения	domain / домен

Домен е множество от атомарни (неделими) значения от един и същи тип. Атомарни означава, че те са най-малката семантична единица данни за модела, т.е. ако разложим значението, то ще загуби смисъла си. Напр., множеството от всички имена на хора, всички имена на отдели и т.н.

Релация R над домените D1, D2, … Dn се състои от:

- Заглавие – фиксирано множество от имена на атрибути A1, A2, …, An, такива че Ai съответства на домена Di, за i=1,2,…n.

- Тяло – изменящо се във времето множество от редове (tuples, n-торки).

Ред е множество от двойки от вида <име-на-атрибут: значение-на-атрибут>, т.е. { <A1:v_{i1}>, <A2: v_{i2}>,… <An: v_{in}>} където v_{ij} принадлежи на домена Dj за j=1,…n. Броят n на атрибутите на релацията се нарича **степен на релацията**. Броят на редовете на релацията се нарича **кардинално число**.

Свойства на релациите, които са следствие от тази дефиниция:

- В релациите няма еднакви редове.
- Редовете в една релация са ненаредени.
- Атрибутите в релацията са ненаредени.
- Стойността на всеки атрибут е атомарна (неделима).

Релация, която притежава последното свойство, се нарича нормализирана. Релационият модел използва само нормализирани релации, за разлика от математическите релации, които не е задължително да са нормализирани.

Много често таблицата се използва като изображение на абстрактното понятие релация. При това представяне се внасят някои свойства, които не са верни според формалната дефиниция. В таблицата редовете са наредени отгоре надолу, също както и колоните – отляво надясно. Но когато говорим неформално понятията релация и таблица са синоними, както и понятията атрибут и колона. Това е едно от неоспоримите преимущества на релационния модел – едно формално понятие има просто неформално представяне. Още повече, че в езика SQL се използват неформалните понятия – таблица, колона, ред.

Възможен и първичен ключ

Нека R{A1, A2, … An} е релация, а K⊆{A1, A2,…An}, т.е. K е подмножество от атрибути на R. Ще казваме, че K е възможен ключ (candidate key) на R, ако притежава свойствата:

- уникалност – не съществуват два реда на R, които имат еднакви значения за всички атрибути от K;
- без излишество – никое подмножество на K не притежава свойството уникалност.

Всяка релация има поне един възможен ключ. Ако K се състои от един атрибут го наричаме прост, в противен случай - съставен. Един от възможните ключове се избира за първичен ключ (primary key), а останалите ще наричаме алтернативни ключове (alternate keys) или възможни ключове.

Първичният ключ осигурява инструмент за адресиране на редове в релационния модел.

Външен ключ

Нека R1{PK, A1, … } е релация с първичен ключ PK и R2{B1, … FK, …} е релация. FK ще наричаме външен ключ (Foreign key) на R2 ако съответства на първичния ключ на R1, което означава следното:

- FK и PK са сравними атрибути – включват еднакъв брой атрибути и съответните атрибути са определени са над един и същи домен.
- Всяко значение на FK в ред от R2 съвпада с някое значение на PK в ред от R1.

Чрез външен ключ се създават връзки между редове в различни релации, т.е. външният ключ е нещо като указател, който сочи към ред от друга релация. Всъщност R1 и R2 не е задължително да са различни релации.

Пример. Нека релациите са:

```
dep{dno, dname, budget}
emp{eno, ename, job, sal, addr, dno}
```

Атрибутът dno в релацията emp е външен ключ, който съответства на първичния ключ на релацията dep. Релацията emp, съдържаща външния ключ, се нарича referencing relation, а релацията dep, съдържаща съответния първичен ключ, се нарича referenced relation.

Релационна БД (РБД) е съвкупност от изменящи се във времето релации от различна степен.

3.2. Общи правила за цялостност

Във всеки момент БД съдържа данни, които моделират ПО. Следователно описанието на БД трябва да включва правила за цялостност, които да информират СУБД за различни ограничения в ПО и по този начин да се предотвратява въвеждането в БД на неверни данни. Има правила за цялостност, които са специфични за конкретната ПО и се описват чрез ограничения за цялостност, но има две общи правила, които се прилагат към всяка БД.

Entity integrity

Това правило е свързано с понятието първичен ключ и с едно специално значение, наречено `NULL`, което се поддържа от почти всички съвременни релационни СУБД. Това значение може да бъде приемано от всеки тип данни и означава неизвестно или неприложимо значение на атрибута в съответния ред. Чрез използване на значение `NULL` в БД се решава проблема с отсъстващата информация, нещо което се среща често в реалния живот. Правилото за цялостност гласи: първичният ключ е уникален и не може никога да е `NULL`. Това е така, защото първичният ключ е уникален идентификатор на реда и в БД не може да се въвежда информация, която не може да бъде идентифицирана.

Referential integrity

Всяко значение на външен ключ трябва или да е равно на някое значение на съответния му първичен ключ или да е напълно `NULL`. Понякога е необходимо да се разреши външният ключ да приема значение `NULL`. Напр., ако определен служител не е към никой отдел, то атрибутът dno на релацията emp в съответния ред трябва да е `NULL`. В други случаи външният ключ не може да приема значение `NULL`.

Всяко състояние на БД, което не удоволетворява тези правила е некоректно и следователно не бива да се допуска. Но как да се поддържа БД в коректно състояние? Това не се казва в правилата за цялостност.

Подържането на ограничението за първичния ключ е сравнително ясно и лесно. А именно всяка операция за добавяне или изменение на ред, при която първичният ключ ще стане `NULL`, трябва да бъде отхвърлена, т.е. да не се изпълни.

Поддържането на ограничението за външен ключ е по-сложно, тъй като то засяга две релации. За всеки външен ключ трябва да се отговори на следните въпроси:

1. Може ли външният ключ да приема значение `NULL` или не?

2. Какво да се прави при опит да се изтрие ред, към който сочи външен ключ?

Възможни отговори на втория въпрос са:

Restricted – операцията изтриване се разрешава само, ако с изтривания ред няма свързани редове със същото значение на външния ключ (ограничително правило).

Cascades – каскадно изпълнение на операцията изтриване и за всички редове, съдържащи съответното значение на външния ключ (каскадно правило).

Set null / nullifies – операцията изтриване се изпълнява, като в съответните редове на външния ключ се дава значение `NULL`.

3. Какво да се прави при опит да се измени значението на първичен ключ, към който сочи външен ключ?

И на този въпрос възможните отговори са същите: Restricted, Cascades, Set null.

3.3. Проектиране на РБД на основата на ER модел

След като е проектиран ER модела на ПО трябва да се проектира релационния модел, т.е. трябва да се изберат релациите и техните атрибути, които ще представят класовете обекти, връзки и техните свойства. Ще формулираме няколко полезни правила.

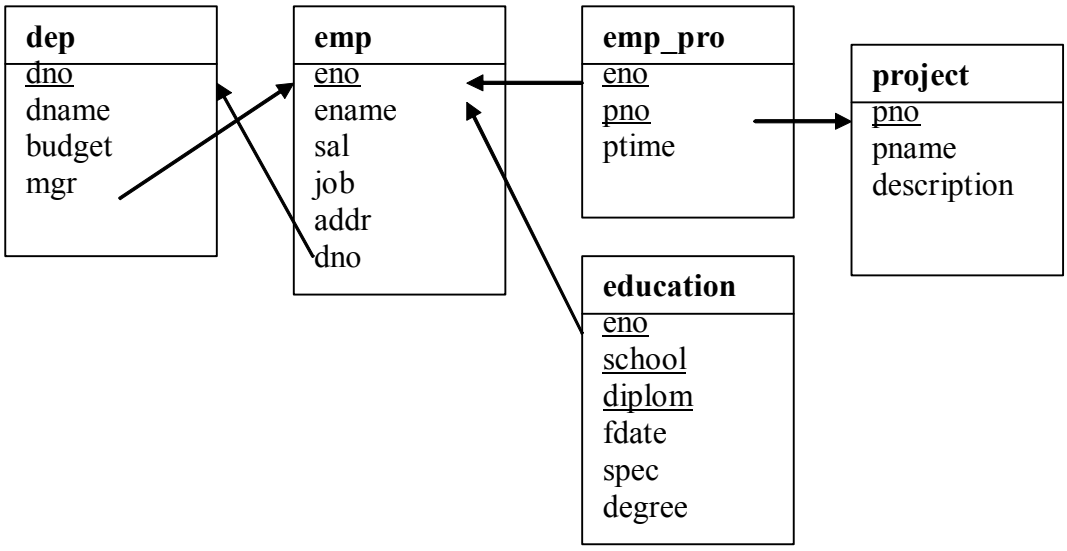
1. Независим клас обекти се представя чрез релация, атрибутите на която представят свойствата на обектите. Избира се най-подходящият първичен ключ.

2. Слаб клас обекти се представя чрез релация, в която освен атрибутите за свойствата на класа се включва и атрибут(и), идентифициращ обекта от когото зависи слабия обект. Този атрибут е външен ключ и може да е част от първичния ключ на релацията.

3. Клас връзки M:M между класовете обекти E1 и E2 се представя чрез отделна релация, в която има два външни ключа, съответстващи на първичните ключове на релациите за E1 и E2. Най-често първичният ключ в тази релация е комбинацията от двата външни ключа. Ако връзката има свойства за тях се добавят допълнителни атрибути в същата релация.

4. Клас връзки 1:M между класовете обекти E1 и E2 може да се представи както връзка M:M, но има и по-икономичен начин - чрез външен ключ в релацията за E2, който съответства на първичния ключ на релацията за E1. Ако връзката има свойства за тях се добавят допълнителни атрибути в същата релация. Връзка 1:1 може да се разглежда като частен случай на връзка 1:M, т.е. външният ключ може да е във всяка от двете релации.

На Фиг.4 е представен релационния модел на БД-служители. Атрибутите, които са първични ключове на релациите са подчертани, а връзката външен първичен ключ е представена чрез стрелка.



Фиг. 4. Релационен модел на БД-служители

4. Релационна алгебра

Релационните езиците за работа с БД се делят на няколко типа в зависимост от теоретичната си основа.

- Релационна алгебра

Пример за реализиран език от този тип е ISBL (Information System Base Language) в PRTV на IBM.

- Релационно смятане

Базира се на предикатното смятане от математическата логика. Първият език от този тип е предложен от Код и е наречен ALPHA, но никога не е реализиран. По-късно е разработен и реализиран език QUEL в INGRES на Университета в Беркли.

- Език SQL (Structured Query Language)

Притежава елементи на релационното смятане и релационната алгебра.

Всички релационни езици, колкото и да са различни, имат една обща черта. Всеки оператор действа не върху един, а върху много редове и резултатът е много редове (често е релация). Освен това всички релационни езици са еднакво мощни, т.е. всяка операция, която може да се изрази на един език може да бъде изразена и в другите, което се нарича релационна пълнота.

Ще започнем разглеждането на релационните езици с релационната алгебра, която понастоящем има основно теоретично значение. Началната релационна алгебра, предложена от Код още в първата му статия, включва осем операции. За всяка от тях важи следното: операндите са релации и резултатът е релация. Това свойство се нарича затвореност и от него следва, че резултатът от една операция може да се използва като операнд за друга, т.е. може да се пишат вложени изрази.

Според дефиницията всяка релация се състои от заглавие и тяло. Ако разглеждаме затвореността строго, трябва за всяка операция да определим как се образува заглавието и тялото на резултатната релация. Следователно необходим е набор от правила за наследяване на имената на атрибутите в резултата и специална допълна операция за преименуване на атрибути.

```
R rename A as B [, . . .]
```

Осемте операции на Код са разделени в две групи:

- традиционни операции от теорията на множествата – обединение, сечение, разлика и произведение;
- специални релационни операции – селекция, проекция, съединение и деление.

4.1. Операции от теорията на множествата

Нека `R` и `S` са релации с еднакви заглавия, което означава следното:

- Имат еднакви множества от имена на атрибути и следователно еднаква степен.
- Съответните атрибути (атрибутите с еднакви имена) са определени над едни и същи домени.

Ще казваме, че такива релации са **съвместими по тип**. Ако релациите са почти съвместими по тип, т.е. има само различия в имената на някои атрибути, то може да се приложи операцията `rename`, за да станат напълно съвместими. Изискването за съвместимост се поставя при операциите обединение (`union`), сечение (`intersect`) и разлика (`minus`), защото резултатът трябва да е релация, а не просто какво да е множество.

```
R union S
```

Заглавието на резултата е същото като на `R` и `S`. Тялото е множество от всички редове, които принадлежат на `R` или на `S`.

`R intersect S`

Заглавието на резултата е същото като на `R` и `S`. Тялото е множество от всички редове, които принадлежат на `R` и на `S`.

`R minus S`

Заглавието на резултата е същото като на `R` и `S`. Тялото е множество от всички редове, които принадлежат на `R` и не принадлежат на `S`.

`R times S`

Операцията произведение е всъщност разширено декартово произведение, отново заради затвореността. Тук няма изисквания за съвместимост по тип на операндите, дори се изисква имената на атрибутите в `R` и `S` да са различни, за да може резултатът да има правилно формирано заглавие. Нека схемите на операндите са: `R{A1, A2, ..., An}` и `S{B1, B2, ..., Bm}`. Резултатът е релация със:

- заглавие, което е теоретико-множествено обединение на заглавията на `R` и `S`, т.е. заглавието е `{A1, A2, ..., An, B1, B2, ..., Bm}`.

- тяло, което е множеството от всички редове, всеки от които е обединение (теоретико-множествено) на ред от `R` и ред от `S`, т.е редове са множества от вида
`{<A1:a1>,<A2:a2>, ..., <An:an>,<B1:b1>,<B2:b2>, ..., <Bm:bm>}`
такива, че `{<A1:a1>,< A2:a2>, ..., <An:an>}` ∈ `R` и
`{<B1:b1>,<B2:b2>, ..., <Bm:bm>}` ∈ `S`.

Следователно степента на резултата е сума от степените на операндите, а кардиналното число е произведение от кардиналните им числа.

Операциите `union`, `intersect` и `times` са **комутативни**, т.е изразите

`R union S` и `S union R` са еквивалентни.

Същите операции са и **асоциативни**, т.е изразите

`(R union S) union T` и `R union (S union T)`

са еквивалентни и следователно за удобство може да се записват без скоби.

От четирите операции само сечението не е примитивна операция, защото може да се изрази чрез разлика, т.е. в сила е следното тъждество:

`R intersect S ≡ R minus (R minus S)`

4.2. Специални релационни операции

Селекция (restrict, select)

Нека `X` и `Y` са атрибути на релацията `R`, определени над един и същи домен и за него операцията за сравнение $\theta \in \{=, \neq, <, \leq, >, \geq\}$ има смисъл.

`R where X θ Y`

Тази операция се нарича **θ-селекция**. Заглавието на резултата е същото като на `R`. Тялото е множество от всички редове, които принадлежат на `R` и за които сравнението `X θ Y` е истина. Вместо `X` или `Y` може да има константа, т.е.,

`R where X θ literal`

При операцията θ-селекция в `where` има просто сравнение, но може да се определи по-обща операция селекция, която вече не е примитивна.

`R where условие`

В `where` има произволен логически израз, съставен от сравнения от двата вида, логическите операции `and`, `or`, `not` и скоби. Такова условие, приемащо значение истина или лъжа за всеки отделен ред се нарича restriction predicate.

Основание за въвеждане на общата операция селекция ни дава свойството затвореност и следните тъждества:

```
R where c1 and c2 ≡ (R where c1) intersect (R where c2)
R where c1 or c2 ≡ (R where c1) union (R where c2)
R where not c ≡ R minus (R where c)
```

Проекция

Нека X, Y, . . . , Z са атрибути на релацията R.

```
R[X, Y, . . ., Z]
```

Заглавието на резултата е {X, Y, ..., Z}. Тялото е множество от всички редове {<X:x>,<Y:y>, ..., <Z:z>}, за всеки от които съществува ред в R със същите значения на атрибутите X, Y, . . . , Z.

Съединение (join)

Има няколко вида операции съединение и всички те са с два операнда. Ще започнем с **естественото съединение.**

```
R join S
```

Нека схемите на операндите са :

R{A1, A2, ..., An, X} и S{B1, B2, ..., Bm, X}, т.е. X е общ атрибут (единствен) за операндите и е определен над един и същи домен. Резултатът е релация със:

- заглавие, което е {A1, A2, ..., An, X, B1, B2, ..., Bm}
- тяло, което е множеството от всички редове от вида

{<A1:a1>,<A2:a2>,...,<An:an>,<X:x>,<B1:b1>, ...,<Bm:bm>}

такива, че {<A1:a1>,<A2:a2>, ..., <An:an>,<X:x>} ∈ R и

{<B1:b1>,<B2:b2>, ..., <Bm:bm>,<X:x>} ∈ S.

θ-съединение е по-обща операция, при която редове от две релации се съединяват (слепват) на базата на условие различно от равенство на атрибути. Нека R и S нямат общи имена на атрибути, но X ∈ R и Y ∈ S са атрибути, определени над един и същи домен, за който операцията сравнение θ има смисъл.

```
R join S where X θ Y
```

Резултатът е релация със:

- заглавие, което е същото като при операцията произведение, т.е. теоретико-множествено обединение на заглавията на R и S. Следователно степента на резултата е сума от степените на операндите.
- тяло, което е множество от всички редове, всеки от които е обединение (теоретико-множествено) на ред от R и ред от S, за които условието X θ Y е истина.

Ако операцията сравнение θ е равенство (=) имаме съединение по равенство (equijoin). Съединението е комутативна и асоциативна операция.

Съединението не е примитивна операция, защото може да се изрази чрез произведение и селекция, т.е. в сила е следното тъждество:

```
R join S where X θ Y ≡ (R times S) where X θ Y
```

Деление

```
R divide by S
```

Нека схемите на операндите са: R{X, Y} и S{Y}, т.е. Y е общ атрибут (единствен) за операндите и е определен над един и същи домен. Резултатът е релация със:

- заглавие, което е {X}

- тяло, което е множество от всички редове {<X:x>}, такива че, съществува ред {<X:x>,<Y:y>} ∈ R за всеки ред {<Y:y>} ∈ S.

Делението също не е примитивна операция, защото може да се изрази чрез проекция, произведение и разлика.

R divide by S ≡ R[X] minus((R[X] times S) minus R)[X]

Примери за релационните операции

R

A	B
1	aa
2	aa
2	bb
3	bb

S

A	B
1	cc
2	aa

T

C	D
1	cc
2	aa

P

B
aa
bb

R union S

A	B
1	aa
2	aa
2	bb
3	bb
1	cc

R intersect S

A	B
2	aa

R minus S

A	B
1	aa
2	bb
3	bb

R times T

A	B	C	D
1	aa	1	cc
1	aa	2	aa
2	aa	1	cc
2	aa	2	aa
2	bb	1	cc
2	bb	2	aa
3	bb	1	cc
3	bb	2	aa

R where B='aa'

A	B
1	aa
2	aa

R[B]

B
aa
bb

R divide by P

A
2

R join T where A=C

A	B	C	D
1	aa	1	cc
2	aa	2	aa
2	bb	2	aa

R join T where A>C

A	B	C	D
2	aa	1	cc
2	bb	1	cc
3	bb	1	cc
3	bb	2	aa

Примери на релационни изрази, които извличат от БД-служители

♦ Пълните данни за служителите, които работят в отдел 1 и получават заплата по-малка от 500.

```
emp where dno = 1 and sal < 500
```

♦ Номерата, имената и длъжностите на всички служители.
```
emp[eno, ename, job]
```

♦ Номерата, имената и длъжностите на служителите, които работят в отдел 1 и получават заплата по-малка от 500.
```
(emp where dno=1 and sal < 500)[eno, ename, job]
```

♦ Номерата на отделите, в които работят служители със заплата по-малка от 500.
```
(emp where sal < 500)[dno]
```

♦ Номерата на отделите, в които няма служители.
```
dep[dno] minus emp[dno]
```

♦ Номер, име на служител и име на отдела, в който работи за всички служители (за служителите със заплата по-малка от 500).
```
(emp join dep)[eno, ename, dname]
((emp join dep) where sal < 400)[eno, ename, dname]
((emp where sal < 500) join dep)[eno, ename, dname]
```

♦ Номерата и имената на отделите, в които няма служители.
```
((dep[dno] minus emp[dno]) join dep)[dno, dname]
```

♦Номер, име на служител, длъжност и наименование на проекта, в който участва служителя за служителите от отдел 1.
```
(((emp where dno=1) join emp_pro) join project)[eno,ename,job, pname]
```

♦ Номерата (и имената и длъжностите) на служителите, които участват във всички проекти.
```
emp_pro[eno, pno] divide by project[pno]
((emp_pro[eno, pno] divide by project[pno]) join emp)[eno, ename, job]
```

♦ Имената на служителите, които не участват в проект с номер 100.
```
((emp[eno] minus (emp_pro where pno = 100)[eno]) join emp)[ename]
```

4.3. Релационно присвояване

Релационни изрази, включващи тези осем операции, ни дават инструмент за извличане на данни от БД. Те обаче не са достатъчни за да се изразят всички функции над БД. Минималното, което е необходимо още е операцията присвояване.
```
R := S
```
S може да е произволен релационен израз, като R и S трябва да са съвместими по тип. Тази операция е основата за реализиране на функциите по обновяване на БД, а именно добавяне, изтриване и изменение на редове.

Добавяне на редове към релация може да се изрази посредством операцията обединение. Напр., добавяне на нов служител в релацията emp:

```
emp:=emp union {<eno:200>,<ename:'Ivanov'>,<sal:300>,<job:'ac'>,<dno:2>}
```

Изтриване на редове от релация може да се изрази посредством операцията разлика. Напр., изтриване на отдел 6 от релацията dep:

```
dep:= dep minus {<dno:6>,<dname:'xxx'>,<budget:2000>,<mgr:null>}
```

Използването на `union` и `minus` вместо специални оператори insert, delete и update не е много удобно. Затова в една релационна СУБД, използваща релационната алгебра като език, трябва да има по-удобни, явни оператори за обновяване. Напр., операторът за добавяне на редове може да изглежда така:

```
insert релационен-израз into R
```

5. Език SQL

Езикът SQL (Structured Query Language) притежава елементи на релационното смятане и релационната алгебра. Създаден е в IBM, където през 1974-75г. се разработва една от първите релационни СУБД System/R. Езикът, създаден за тази система, е наречен SEQUEL (Structured English Query Language). След опитна експлоатация, тестване и развитие на езика, той е преименуван в SQL и проекта System/R завършва. Така започва историята на езика SQL, който сега де-факто е стандарт за релационните СУБД. Списъкът на СУБД, реализиращи SQL като език за БД, е много дълъг, само някои от тях са: SQL/DS, DB2, Oracle, Informix, Sybase, Ingres, Postgres, SQL Server, MS Access.

През 1982г. към ANSI е сформиран комитет за стандартизация на SQL, а през 1986г. е приет първия стандарт SQL X3.135, който през 1987г. е приет и от ISO. Тъй като езикът се развива и стандартизацията му продължава. По-късно са приети и следващи стандарти: SQL/89 или SQL1, SQL/92 или SQL2 и SQL/99 или SQL3. Въпреки многото стандарти има и много промишлени диалекти, които се различават един от друг. Но все пак повечето съвременни промишлени СУБД реализират езика в съответствие с SQL2, като го разширяват и с нови възможности. Ще разгледаме основните средства на езика, като ще се придържаме към SQL2.

5.1. Общи елементи

Всеки оператор започва с ключова дума-глагол и включва една или повече фрази. Всяка фраза започва с ключова дума, като някои фрази са задължителни, а други не. Ключовите думи в SQL2 са около 300 и в повечето реализации могат да се пишат с малки или главни букви, без да се прави разлика.

5.1.1. Имена на обекти

Обектите в БД, които се именуват, са таблици, колони, потребители, ограничения за цялостност, индекси, синоними, view и др. Характерно за SQL е, че се използва нематематическата терминология за обектите в релационния модел, а именно – таблица, колона и ред. Според SQL1 име на обект в БД е символен низ с максимална дължина 18 символа, който започва с буква и съдържа букви, цифри и символа "_". В SQL2 се въвеждат дълги имена до 128 символа.

Някои обекти имат и пълни имена. Пълното име на таблица се състои от име на потребителя, който е собственик и собственото име на таблицата и се записва:

```
име_собственик.име_таблица
```

Аналогично понякога се налага използване на пълно име на колона. Ако в един оператор се използват няколко таблици и в тях има колони с еднакви собствени имена, тогава собственото име е нееднозначно и се уточнява с името на таблицата:

```
име_таблица.име_колона
```

5.1.2. Типове данни

Тип на данните представлява примитивна реализация на понятието домен от релационния модел. Типовете данни според SQL2 са:

- цели числа
- числа с фиксирана точка
- числа с плаваща точка
- символни низове с фиксирана дължина
- символни низове с променлива дължина
- дата, време и временен интервал
- парична величина

В някои СУБД се реализират и допълнителни типове, наречени BLOB:

- поток от байтове

В колона от такъв тип може да се съхраняват графични изображения, изпълним код и всяка друга неструктурирана последователност от байтове.

- дълъг текст – символни низове с променлива дължина, която е максимално 32000, 65000 или повече байта.

Има различия в типовете данни, реализирани в различните СУБД, и дори в наименованията им, което е съществено препятствие за преносимостта на приложенията. Например в Oracle има тип `DATE`, в който се съхранява дата и време. В Informix има тип `DATE`, в който се съхранява само дата, тип `DATETIME` за дата и време и тип `INTERVAL` за временен интервал. Различията продължават и по отношение на реализираните операции и вградените функции за тези типове.

5.1.3. Константи

Числови константи се записват като десетично число със или без знак и евентуално с дробна част или като десетично число с плаваща точка (както в повечето езици за програмиране). Например:

21 -123 421.35 1.55E3 (което е $1.55*10^3$)

Низова константа трябва да се ограничава с единични кавички, но в някои СУБД са разрешени и двойни кавички.

`'низ'` или `"низ"`

Константи съдържащи календарна дата, време и временен интервал се записват като символен низ. Форматът на низа е различен за различните СУБД.

Пример в Informix: ‘3/30/2003’, ‘2003-03-30 12:10:15.20’, ‘20-5’

Пример в Oracle: ‘30-MAR-2003’

Много често дори в една СУБД се поддържат няколко формата на запис на константите в зависимост от страната.

Съществуват специални символни константи, чиито значения се определят от СУБД. Символна константа може да се използва на всяко място в оператор, където е разрешена константа. Например:

`USER` е името на потребителя, който работи с БД

`SYSDATE` е текущата дата в Oracle

`TODAY` е текущата дата в Informix

5.1.4. Изрази

В аритметични изрази могат да се използват операциите: +, -, *, / и скоби. Според стандарта се извършва автоматично преобразуване между различните числови типове, когато това се налага.

В някои СУБД се реализират и символни изрази. Например, в Informix и Oracle:

```
'Mr.' || emp.ename || ', ' || emp.job
```

Изрази над дата и време са включени в SQL2 и се реализират. Например в Informix и Oracle:

```
education.fdate + 20
```

Ако `education.fdate` е колона, съдържаща дата на завършване на образование, резултатът е от тип дата, която е 20 дена след датата, съдържаща се в колоната.

`TODAY - education.fdate` или `SYSDATE - education.fdate`

Резултатът е цяло число, което е брой дни между двете дати.

Много често се реализират и **вградени функции**, които изпълняват различни преобразования над типове, форматиране или други изчисления. Такава функция може да се използва на всяко място в оператор, където може да стои име на колона от съответния тип, следователно може да се използва и в изрази. Списъкът на функциите в различните диалекти на SQL е доста различен.

Примери от Oracle:

`trunc(sal, 1), round(sal, 1)`

Връщат число, което е значението в аргумента, съответно отрязано или закръглено до един знак след десетичната точка.

`length(addr)`

Връща цяло число, което е дължината на низа в аргумента.

`trim(str)`

Изключва от двата края на `str` всички срещания на символа интервал.

`last_day(fdate)`

Връща дата, която е последния ден от месеца на датата `fdate`.

Примери от Informix:

`year(fdate), month(fdate), day(fdate), weekday(fdate)`

Връщат цяло число, което е съответно годината, месеца, деня и деня от седмицата на датата в аргумента.

`trunc(sal, 1), round(sal, 1)` и `length(addr)`

Имат същото действие както в Oracle.

В много случаи значението `NULL` изисква специална обработка. Напр., ако се изчислява израза `sal + 10`, какво да се прави ако значението в `sal` е `NULL`. Отговорите на подобни въпроси дават набор от правила за обработка на значения `NULL` в различни оператори и фрази на езика. Заради необходимостта от такива правила някои теоретици на релационния модел са против това значение. Но независимо от теоретичните спорове значението `NULL` е част от стандартите и се реализира от повечето промишлени СУБД.

При изчисляване на израз, ако някой от операндите е `NULL` и резултатът има значение `NULL`.

5.2. Извличане на данни – оператор SELECT

Ще започнем с операторите за манипулиране на данни (Data Manipulation Language или DML) и най-напред с оператора за извличане на данни от БД. В промишлените СУБД много често се допуска таблицата да съдържа еднакви редове, което означава че тя не е множество, а мултимножество. Ако приемем такова понятие за таблица, то може да кажем, че резултатът от изпълнение на оператор SELECT е таблица, чиито редове и колони са изчислени от данните в други таблици. Резултатната таблица е временна, т.е. тя не остава трайно в БД, а само докато е необходима, напр. докато потребителят я разглежда на екрана на терминала или до завършване на приложната програма. Обикновено резултатната таблица е неименована. Операторът SELECT включва следните фрази, от които само първите две са задължителни:

SELECT *целеви-списък*
: Определя колоните на резултната таблица.

FROM *таблици*
: Определя таблиците, от които ще се извличат данни (наричаме ги изходни таблици).

WHERE *условие*
: Определя редовете от изходните таблици, които ще участват в резултата.

GROUP BY *списък-от-колони-при-групиране*
: Задава колоните, по които ще се формират групи от редове.

HAVING *условие*
: Определя групите, които ще участват в резултата.

ORDER BY *списък-от-колони-за-сортиране*
: Определя колоните на резултата, по които се сортират редовете в резултата.

5.2.1. Прости еднотаблични заявки

Да разгледаме по-подробно синтаксиса на първите две фрази, които задължително присъстват във всеки оператор.

SELECT [ALL|DISTINCT|UNIQUE] *израз* [[AS] *етикет*] [, . . .]

FROM *име-на-таблица* [*псевдоним*] [, . . .]

Целевият списък в SELECT е списък от елементи, разделени със запетая. Всеки елемент определя една колона в резултатната таблица, т.е. броят на елементите определя степента на резултата. *Израз* задава начина, по който да се изчисляват значенията в колоната от значенията на колони в изходни таблици и константи. *Етикет* определя името на резултатната колона. Ако не е зададен и *израз* е просто име на колона се използва това име, иначе резултатната колона е безименна. Ключовите думи след SELECT определят дали да се премахват еднаквите редове в резултата (DISTINCT и UNIQUE) или не (ALL). По премълчаване дубликатите не се премахват, защото тази операция е скъпа.

Фразата FROM съдържа списък от имена на таблици, от които ще се изчисляват редовете в резултата според останалите фрази на оператора. В някои случаи се налага определянето на синонимно име на таблица - *псевдоним*, което важи само в съответния оператор. В този раздел ще предполагаме, че фразата FROM включва една таблица.

♦ Списък на всички служители, включващ име, заплата и длъжност на служителя.

```
SELECT ename, sal, job FROM emp;
```

Ако в резултата искаме да включим всички колони на определена таблица, може да използваме символа "*" или "*име_таблица*.*" вместо списък от колони. Това означава всички колони на указаната таблицата или на таблиците от фразата FROM. Това е удобно при интерактивна работа, но използването във вграден SQL е потенциално опасно, тъй като смисълът на "*" с времето може да се измени.

♦ Пълни данни за всички отдели.
```
SELECT * FROM dep;
SELECT dep.* FROM dep;
```

♦ Списък на всички служители, включващ име, предшествано от обръщение и годишна заплата на служителя.
```
SELECT 'Mr./Mrs.'||ename AS titename, sal*12 AS ysal FROM emp;
```

Фразата WHERE определя редовете, които ще бъдат извлечени от изходните таблици и ще участват при изчисляване на резултата.

```
WHERE условие
```

Условието се конструира от няколко типа прости условия, логическите операции AND, OR, NOT и скоби. Всички редове, за които условието е истина, са избрани и ще участват в резултата. Съществуването на значение NULL влияе при изчисляването на условията, като в същност се прилага тризначна логика. Значението на всеки условен израз може да е истина (true), лъжа (false) или незвестно (unknown). Да разгледаме основните прости условия.

- **Сравнения**

```
израз1  { = | != | <> | < | <= | > | >= } израз2
```

Ако поне един от сравняваните изрази има значение NULL, то резулатът от сравнението е unknown.

- **Проверка за принадлежност на затворен интервал**

```
израз1  [NOT] BETWEEN израз2 AND израз3
```

Това условие е еквивалентно на израза:

```
[NOT] (израз1 >= израз2 AND израз1 <= израз3)
```

Ако *израз1* е NULL или ако *израз2* и *израз3* са NULL, то условието е unknown.

Ако *израз2* (долната граница) е NULL, то условието е false ако *израз1>израз3* и unknown в противен случай.

Ако *израз3* (горната граница) е NULL, то условието е false ако *израз1<израз2* и unknown в противен случай.

- **Проверка за принадлежност на множество**

```
израз [NOT] IN ( списък-от-значения )
```

Списък-от-значения е списък от константи от един и същи тип, разделени със запетая. Условието е истина ако значението на *израз* е равно на някоя от константите. Ако *израз* е NULL, то условието е unknown.

Проверката за принадлежност на множество също не разширява възможностите на езика, защото условието:

```
X IN (A, B, C, D)
```

е еквивалентно на:

```
(X = A) OR (X = B) OR (X = C) OR (X = D)
```

- **Проверка за съответствие на шаблон**

```
име-на-колона [NOT] LIKE 'низ' [ ESCAPE 'символ' ]
```

Колоната *име-на-колона* трябва да е от символен тип. *Низ* представлява шаблон на значението на колоната. В него освен обикновени символи може да има метасимволи (символи, имащи специално значение в шаблона), които са:
% - разширява се до произволен брой произволни символи;
_ - разширява се до точно един на брой произволен символ;
\ - означава отмяня на специалното значение на символа след него (това е escape символа по премълчанване, но може да се смени чрез фраза ESCAPE).

Условието е истина ако значението на *име-на-колона* съответства на шаблона, т.е. равно е на някой от низовете, генерирани от шаблона. Ако значението на колоната е NULL, то условието е unknown.

- **Проверка за неопределеност**

име-на-колона IS [NOT] NULL

При операция IS NULL условието е истина ако значението на колоната е неопределено, а при операцията IS NOT NULL условието е истина ако значението е определено. За разлика от останалите видове прости условия, това условие никога не може да е unknown.

- **Съставни условия**

NOT условие
условие1 AND *условие2*
условие1 OR *условие2*

Както и при простите условия, значението NULL влияе на истинността и на съставните условия.

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

NOT	TRUE	FALSE	UNKNOWN
	FALSE	TRUE	UNKNOWN

Наредбата на редовете в резултата е непредсказуема, докато явно не се укаже чрез фразата ORDER BY.

ORDER BY *име-на-колона* [ASC | DESC] [, . . .]

Редовете в резултата се сортират по нарастващите (ASC) или намаляващи (DESC) значения на указаните резултатни колони. Това означава, че може да се сортира само по колони или изрази явно или неявно включени в целевия списък. Вместо име на колона може да се използва етикет или пореден номер в целевия списък. Правилата за сортиране на значение NULL може да са различни в различните СУБД. Например в Informix значението NULL се счита за по-малко от всяко друго значение. В Oracle фразата ORDER BY има следния по сложен вид:

ORDER BY *израз* [ASC | DESC] [NULLS FIRST | NULLS LAST]

Може да се сортира по значението на *израз* върху колони от целевия списък и колони на изходните таблици. При сортиране по нарастващи значения NULLS LAST е по премълчаване, а за намаляващ ред NULLS FIRST.

В някои СУБД резултатът от оператор `SELECT` може да се съхрани в БД като временна таблица, която ще съществува до края на сесията. В Informix това може да се укаже чрез фразата `INTO TEMP`.

`INTO TEMP` *име-на-резултатна-таблица*

Създава се таблица с указаното име и в нея се записва резултата от оператора `SELECT`. Не трябва да има таблица със същото име в БД. Таблицата се унищожава автоматично при завършване на приложната програма (сесията с БД) или явно с оператор `DROP TABLE`.

♦ Списък на служителите, включващ името, заплатата и длъжността, за служителите които работят в отдел с номер 1 и получават заплата по-малка от 500. Резултатът да се сортира по азбучен ред на длъжността и по намаляващи значения на заплатата.

```
SELECT ename, sal, job FROM emp
WHERE dno=1 AND sal < 500
ORDER BY job, sal DESC;
```

♦ Списък от имената на служителите, които работят в отдел с номер 1 и получават заплата по-голяма от 500 и по-малка от 1000.

```
SELECT ename FROM emp
WHERE dno=1 AND sal BETWEEN 500 AND 1000;
```

♦ Списък от имената на служителите, които работят в отдели с номера 1, 3 или 5.

```
SELECT ename FROM emp
WHERE dno IN (1, 3, 5);
```

♦ Списък от имената на служителите с неизвестен адрес.

```
SELECT ename FROM emp
WHERE addr IS NULL;
```

♦ Списък от имената на служителите с адрес в София.

```
SELECT ename FROM emp WHERE addr LIKE 'София%';
```

♦ Списък от номерата на отделите, в които работят служители със заплата по-малка от 500.

```
SELECT DISTINCT dno FROM emp WHERE sal < 500;
```

Правила при изпълнение на еднотабличен оператор `SELECT`

Ще опишем правилата при изпълнение на еднотабличен оператор `SELECT`, които по-нататък ще бъдат допълвани и разширявани. Тези правила определят какво означава оператора, а не как СУБД го изпълнява.

1. Взема се таблицата от фразата `FROM`.
2. Ако има фраза `WHERE`, то се проверява условието в нея за всеки ред на таблицата и се избират редовете, за които условието е true, останалите редове (за които условието е false или unknown) се игнорират.
3. За всеки от избраните редове се изчисляват последователно елементите в целевия списък и се формира един ред за резултата.
4. Ако има ключова дума `DISTINCT`, то се изключват повтарящите се редове.
5. Ако има фраза `ORDER BY`, то се сортират редовете в резултата.

5.2.2. Многотаблични заявки – съединение и произведение

На практика повечето заявки извършват търсене и извличане на данни от няколко таблици, т.е. резултатната таблица се формира като се използват редове от няколко таблици. Един от случаите когато това се случва, е ако във фразата `FROM` има няколко имена на таблици. Тогава ако във фразата `WHERE` има достатъчно условия, в които се сравняват колони от различните таблици (join условия), това е съединение.

Ако няма такова условие, то се прави произведение на таблиците. Какво означава да има достатъчно join условия. Например, ако във фразата `FROM` има две таблици, то в `WHERE` трябва да има едно join условие:

```
SELECT T1.*, T2.*
FROM T1, T2
WHERE T1.X = T2.Y
```

Колоните, участващи в join условието `T1.X = T2.Y`:

- Трябва да са сравними, като операцията за сравнение може да е всяка друга.
- Могат да са външен и съответния му първичен ключ, но може и да са произволни сравними колони.
- Могат да участват или не в целевия списък.
- Могат да са съставни колони.

Ако във фразата `FROM` има три таблици, то в `WHERE` трябва да има две join условия, които свързват последователно колони от трите таблици. Освен това в `WHERE` може да има и други условия, които избират редове от всяка една от таблиците.

Ако в таблиците от фразата `FROM` има колони с еднакви имена, тогава се налага използване на пълни имена на колоните, т.е. името на колоната се уточнява с името на таблицата. Общото правило относно използване на пълни или собствени имена на колони в SQL оператор е, че собствени имена могат да се използват ако това не предизвиква неоднозначност. На някои места името задължително трябва да е собствено, напр., в `ORDER BY`.

♦ Списък на всички служители, включващ пълните данни за служителя и за отдела, в който работи.

```
SELECT emp.*, dep.*
FROM emp, dep
WHERE emp.dno = dep.dno;
```

♦ Списък на всички служители, включващ името, заплатата и името на отдела, в който работи. Резултатът да се сортира по азбучен ред на името на отдела и по намаляващи значения на заплатата.

```
SELECT ename, sal, dname
FROM emp, dep
WHERE emp.dno = dep.dno
ORDER BY dname, sal DESC;
```

♦ Списък на началниците, включващ името, заплатата и името на отдела, който ръководи, за началниците със заплата по-голяма от 1000.

```
SELECT ename, sal, dname
FROM emp, dep
WHERE eno = mgr AND sal > 1000;
```

Когато при съединение се използват връзки между редовете на една таблица се казва, че това е съединение на таблица със себе си (selfjoin). В този случай е задължително използването на псевдоними на имената на таблиците, за да се различават различните екземпляри на една таблица. Тогава в пълните имена на колони се използват псевдонимите.

♦ Списък от имената на всички служители, които получават заплата еднаква с (по-голяма от) тази на Иван Иванов от отдел 1.

```
SELECT e.ename
FROM emp e, emp i
WHERE e.sal = i.sal AND i.ename = 'Иван Иванов' AND i.dno = 1;
```

♦ Списък на всички служители, включващ името и длъжността, които участват в проект с наименование ”ABC”.

```
SELECT ename, job
```

```
FROM emp e, emp_pro ep, project p
WHERE e.eno = ep.eno AND ep.pno = p.pno AND pname = 'ABC';
```

Използването на псевдоними в този случай не е задължително, но е удобно тъй като пълните имена на колоните са по-кратки.

♦ Списък на всички служители, включващ името, заплатата и името на началника му, за служителите със заплата по-малка от 500.

```
SELECT e.ename, e.sal, m.ename
FROM emp e, emp m, dep
WHERE e.dno = dep.dno AND dep.mgr = m.eno AND e.sal < 500;
```

Правила при изпълнение на оператор SELECT над няколко таблици

Ще опишем правилата при изпълнение на този по-сложен вариант на оператора `SELECT`, при който може да има няколко таблици във фразата `FROM`.

1. Формира се произведението на таблиците от фразата `FROM`. Ако във `FROM` има само една таблица, то самата тя ще е произведението.
2. Ако има фраза `WHERE`, то се проверява условието в нея за всеки ред на произведението и се избират редовете, за които условието е true, останалите редове (за които условието false или unknown) се игнорират.
3. За всеки от избраните редове се изчисляват последователно елементите в целевия списък и се формира един ред за резултата.
4. Ако има ключова дума `DISTINCT`, то се изключват повтарящите се редове.
5. Ако има фраза `ORDER BY`, то се сортират редовете в резултата.

Външно съединение

При разгледаната операция съединение, аналог на join от релационната алгебра, се съединява ред от една таблица с ред от друга таблица, за които join условието е истина. Следователно, ако ред от едната таблица няма нито един съответен ред в другата, то той няма да участва в резултата. В някои случаи това може да доведе до неочаквани резултати. Напр., в задачата:

♦ Списък на всички служители, включващ името на служителя и името на отдела, в който работи.

```
SELECT ename, dname
FROM emp, dep
WHERE emp.dno = dep.dno;
```

Резултатът няма да включва служителите, за които колоната `dno` има значение `NULL`, а може би ние очакваме да видим и тях в резултата.

Външното съединение е разширение на обикновеното, наричано още вътрешно. Правилата при изпълнението му за две таблици са следните:

1. Създава се вътрешното съединение на двете таблици.
2. Всеки несвързан ред от първата таблица се включва в резултата, като на всички колони от втората таблица се присвояват `NULL` значения.
3. Всеки несвързан ред от втората таблица се включва в резултата, като на всички колони от първата таблица се присвояват `NULL` значения.

Всъщност съединението, създадено по тези правила се нарича **пълно външно** съединение, има още **ляво външно** съединение (стъпки 1 и 2) и **дясно външно** съединение (стъпки 1 и 3).

Начините за обозначаване на външните съединения са доста различни в различните СУБД, тъй като те се реализират преди да са включени в стандартите. При лявото външно съединение първата (лява) таблица във фразата `FROM` се нарича главна, втората (дясна) - подчинена в съединението. В Oracle лявото външно съединение се записва:

```
SELECT *
FROM T1, T2
WHERE T1.A = T2.B(+)
```

Знак плюс в скоби, се записва от дясно на колоната от таблицата, която е подчинена (допълва се с празен ред).

```
SELECT *
FROM T1, T2
WHERE T1.A = T2.B (+)
```

Други начини за записване на лявото външно съединение.

Informix

```
SELECT *
FROM T1, OUTER T2
WHERE T1.A = T2.B
```

SQL Server

```
SELECT *
FROM T1, T2
WHERE T1.A *= T2.B
```

В Informix ключовата дума `OUTER` се поставя пред името на таблицата, която се допълва с празен ред. В SQL Server се поставя знак * в ляво на операцията за сравнение.

В Informix и Oracle няма пълно външно съединение. В SQL Server пълното външно съединение се записва:

```
SELECT *
FROM T1, T2
WHERE T1.A *=* T2.B
```

В SQL2 стандарта е избран метод, който не съответства на никоя от популярните СУБД. Разширена е фразата `FROM` с обозначаването на вида на съединението и условието за съединение, което се пренася от фразата `WHERE`. За вътрешното съединение е запазен и записа според SQL1 стандарта.

```
SELECT *                                        SELECT *
FROM T1 INNER JOIN T2 ON T1.A = T2.B            FROM T1, T2
                                                WHERE T1.A = T2.B
```

```
SELECT *
FROM T1 LEFT [OUTER] JOIN T2 ON T1.A = T2.B
```

```
SELECT *
FROM T1 RIGHT [OUTER] JOIN T2 ON T1.A = T2.B
```

```
SELECT *
FROM T1 FULL [OUTER] JOIN T2 ON T1.A = T2.B
```

Външното съединение може да се разшири за три и повече таблици, но тогава резултатът зависи от реда, в който се изпълняват съединенията. Затова в разширената фраза `FROM` се добавят скоби, т.е. резултатът от едно съединение се използва като операнд на друго.

♦ Списък на **всички** служители, включващ името, длъжността и името на отдела, в който работи. Решение със синтаксиса на Oracle, Informix и SQL2 стандарта .

```
SELECT ename, job, dname                    -- SQL2
FROM emp LEFT OUTER JOIN dep ON emp.dno = dep.dno;
```

```
SELECT ename, job, dname                    -- Oracle
FROM emp, dep
WHERE emp.dno = dep.dno(+);
```

```
SELECT ename, job, dname                    -- Informix
FROM emp, OUTER dep
WHERE emp.dno = dep.dno;
```

♦ Списък на **всички** служители, включващ името, длъжността, името на проекта и времето, които имат заплата по-голяма от 500. Резултатът от вътрешното съединение на `emp_pro и project` се съединява външно с `emp`.

Първи вариант – със синтаксиса според стандарта.

```
SELECT ename, job, pname, ptime
FROM emp LEFT JOIN
      (emp_pro INNER JOIN project ON emp_pro.pno=project.pno)
      ON emp.eno=emp_pro.eno
WHERE sal > 500;
```

Втори вариант – със синтаксиса на Informix.

```
SELECT ename, job, pname, ptime
FROM emp, OUTER (emp_pro, project)
WHERE emp.eno=emp_pro.eno AND emp_pro.pno=project.pno AND sal > 500;
```

5.2.3. Вложени заявки - вложен оператор SELECT

Друг случай, при който резултатната таблица се формира като се използват данни от няколко таблици, е когато се използа вложен оператор `SELECT`. Вложен оператор `SELECT` или подзаявка се нарича оператор `SELECT`, съдържащ се в друг такъв оператор. Местата, на които това се реализира, са условията във фразите `WHERE` или `HAVING`. Условията с вложен `SELECT` са следните.

израз `[NOT] IN (` *оператор-select* `)`

Това е същото условие, проверяващо за принадлежност на множество и описано преди, с тази разлика, че списъкът от значения се изчислява посредством вложен оператор `SELECT`.

израз `{ = | != | <> | < | <= | > | >= } [ANY|ALL](` *оператор-select* `)`

Условието θ`ALL` е истина, ако сравнението (означено с θ) е истина за всяко значение, върнато от вложения оператор. Ако вложеният оператор не върне нито едно значение, то условието е истина. Ако сравнението не е false за нито едно върнато значение, но има и значения `NULL`, то условието е unknown. Условието `!=ALL` е еквивалентно на `NOT IN`.

Условието θ`ANY` е истина, ако сравнението е истина за поне едно значение, върнато от вложения оператор. Ако вложеният оператор не върне нито едно значение, то условието е лъжа. Ако сравнението не е true за нито едно върнато значение, но има и значения `NULL`, то условието е unknown. Условието `=ANY` е еквивалентно на `IN`.

Ако сме сигурни, че вложеният оператор ще върне едно значение, то може да не указваме ключовите думи `ALL` или `ANY`.

`[NOT] EXISTS (` *оператор-select* `)`

Условието `EXISTS` е истина, ако вложеният оператор върне поне един ред, иначе условието е лъжа. Не е възможно значението на условието да е unknown.

Вложеният оператор `SELECT` не може да включва фразите `ORDER BY` и `UNION`.

♦ Списък на имената и заплатите на началниците, които получават заплата по-голяма от 1000.

```
SELECT ename, sal
FROM emp
WHERE eno IN (SELECT mgr FROM dep) AND sal > 1000;
```

♦ Списък от имената на всички служители, които получават заплата еднаква с (по-голяма от) тази на Иванов от отдел 1 (предполагаме, че има само един служител Иванов в отдел 1).

```
SELECT ename
FROM emp
WHERE sal = (SELECT sal FROM emp
             WHERE ename = 'Иван Иванов' AND dno = 1);
```

♦ Списък на всички служители, включващ името и длъжността, които участват в проект с наименование ”ABC”.

```
SELECT ename, job
FROM emp
WHERE eno IN (SELECT eno FROM emp_pro
              WHERE pno IN (SELECT pno FROM project
                            WHERE pname = 'ABC'));
```

♦ Списък от имената и заплатите на всички служители, които получават заплата по-голяма от максималната заплата в отдел 3.

```
SELECT ename, sal
FROM emp
WHERE sal >ALL (SELECT sal FROM emp WHERE dno = 3);
```

В това решение може да възникне проблем, ако заплатата на някой от служителите в отдел 3 е NULL, т.е. ако вложеният SELECT върне поне едно значение NULL.

Във всеки оператор SELECT имената на колоните са неявно уточнени с името на таблица от съответната фраза FROM. Затова ако външният и вложеният SELECT са върху една и съща таблица, не се налага да използваме псевдоними (освен ако не са корелирани).

5.2.4. Агрегатни функции и групиране

Понякога се изисква извличане на обобщена информация от БД. Например, искаме да получим отговор на въпроси от типа: Каква е средната заплата и броят на служителите в отдел 1? В SQL решението за въпроси от този тип се изразява чрез оператор SELECT с агрегатни функции и фрази GROUP BY и HAVING.

Агрегатни функции

Агрегатната функция приема като аргумент съвкупност от значения на една колона, съдържащи се във всички редове, избрани от условието в WHERE (и принадлежащи на една група, ако има групиране) и връща едно значение, което е обобщение на значенията по определен начин. Аргументът може да е колона на изходна таблица или израз (производна колона). Агрегатните функции са:

```
SUM([ALL|DISTINCT|UNIQUE] израз)
AVG([ALL|DISTINCT|UNIQUE] израз)
MIN([ALL|DISTINCT|UNIQUE] израз)
MAX([ALL|DISTINCT|UNIQUE] израз)
COUNT([ALL|DISTINCT|UNIQUE] израз)        COUNT(*)
```

Функцията SUM изчислява сумата на всички значения на колоната, която трябва да е от числов тип и връща числов тип.

Функцията AVG изчислява средното аритметично на всички значения на колоната, която трябва да е от числов тип и връща числов тип.

С функциите MIN и MAX се намира най-малкото или най-голямото значение в колоната, която може да е от числов, символен или тип дата/време. Сравненията се извършват по правилата на съответния тип, напр. дати се сравняват хронологически.

Функцията COUNT изчислява броя на всички значения на колоната, която може да е от произволен тип и връща цяло число. Вариантът COUNT(*) брои редовете.

С ключовата дума DISTINCT или UNIQUE се указва да се премахнат еднаквите значения на колоната преди да се изчисли агрегатната функция. При функциите MIN

и MAX това е безмислено, но някои СУБД го разрешават. В стандарта SQL1 DISTINCT може да се указва само пред име на проста колона, но не и пред израз. В стандарта SQL2 много от ограниченията са снети, но в различните СУБД правилата може да се различават.

Значението NULL се отразява и при изчисление на агрегатните функции. Правилата за обработка на NULL по стандарта са:

- Значението NULL се игнорира при изчисляване на функциите. Изключение прави COUNT(*), тъй като тя брои редове, независимо от NULL.
- Ако всички значения в колоната са NULL, то SUM, AVG, MIN и MAX връщат NULL, а COUNT връща 0.
- Ако в колоната няма нито едно значение, т.е броят на редовете е 0, то SUM, AVG, MIN и MAX и връщат NULL, а COUNT връща 0.

В някои случаи значението NULL може да доведе до проблеми, например:

SUM(A) + SUM(B) и SUM(A+B)

могат да върнат различни резултати, ако в някоя от колоните в някой ред има NULL.

Агрегатни функции могат да стоят непосредствено само във фразите SELECT и HAVING. Ако агрегатна функция е във фразата SELECT и няма групиране, то резултатът е таблица от един ред и целевият списък не може да включва имена на колони. Агрегатни функции не могат да се влагат, тъй като при определяне на аргументите им роля играят фразите FROM и WHERE.

♦ Средната заплата и брой служители в предприятието (за отдел 1).

```
SELECT AVG(sal) avgsal, COUNT(*) empcount
FROM emp;

SELECT AVG(sal) avgsal1, COUNT(*) empcount1
FROM emp WHERE dno = 1;
```

♦ Списък от имената и заплатите на всички служители, които получават заплата по-голяма от средната заплата в предприятието.

```
SELECT ename, sal
FROM emp
WHERE sal > (SELECT AVG(sal) FROM emp);
```

За да поставим агрегатна функция в условието на фразата WHERE, тя трябва да е във вложен SELECT. Тогава той винаги връща едно значение и може да изпуснем ANY или ALL при сравнението.

♦ Списък от имената и заплатите на всички служители, които получават заплата по-голяма от максималната заплата в отдел 3.

```
SELECT ename, sal
FROM emp
WHERE sal > (SELECT MAX(sal) FROM emp WHERE dno = 3);
```

♦ Списък от имената и заплатите на всички служители, които получават заплата по-голяма от средната заплата за отдела си.

```
SELECT ename, sal
FROM emp x
WHERE sal > (SELECT AVG(sal) FROM emp WHERE dno = x.dno);
```

Това е решение с **корелиран (свързан)** вложен SELECT, тъй като в него участва колоната x.dno от външния SELECT. При проверката на условието във външния SELECT, за всеки ред се променя значението на тази колона, следователно вложеният SELECT ще връща различни значения за всеки проверяван ред и трябва да се изчислява многократно.

♦ Списък от номерата и имената на отделите, в които работят повече от 5 служители със заплата по-голяма 500.

```
SELECT dno, dname
FROM dep
WHERE 5 < (SELECT COUNT(*) FROM emp
            WHERE dno = dep.dno AND sal > 500);
```

При това решение с корелиран вложен `SELECT` в резултата не може да участва броя на служителите за всеки отдел. По-нататък ще разгледаме друго решение, което позволява в резултата да се включи и броя на служителите за всеки отдел.

Групиране - фраза `GROUP BY`

Групата е съвкупност от редове, които са избрани от фразата `WHERE` и имат еднакви значения за определени колони. Групиране в оператор `SELECT` се указва чрез фразата `GROUP BY`, задаваща имената на колоните, по значенията на които ще се формират групите.

```
GROUP BY израз [, ...]
```

Указаните *израз* са изрази върху колони от таблиците във фразата `FROM`. В някои СУБД е позволено да се групира само по значения на колони на изходните таблици. Групиране може да се прави по повече от една колона (до 8 колони в Informix), но е на едно ниво, т.е. не може групата да се разбие на подгрупи.

Между агрегатните функции в целевия списък и фразата `GROUP BY` има връзка. Ако има групиране, то за всяка група в резултата се включва един ред. От това следват ограничения за възможните значения на елемент от целевия списък. Разрешените значения са следните:

- константа
- агрегатна функция, която ще връща едно значение за всяка група
- *израз* от фразата `GROUP BY`
- израз, включващ горните елементи.

Това означава, че с един оператор `SELECT` не могат да се получат детайлни и сумарни данни едновременно. Това е така защото резултатът трябва да е таблица, а не комбинация от редове от различен тип.

Ако има фраза `GROUP BY`, но в целевия списък няма агрегатни функции, то операторът може да се изрази без `GROUP BY`, а чрез `DISTINCT`.

Когато колоната, по която се групира, в някои редове приема значения `NULL`, това отново изисква допълнително правило. Правилото за фразата `GROUP BY` в стандарта (в Oracle и Informix) е: ако два реда имат значение `NULL` в една колона за групиране и еднакви значения в другите колони за групиране, то те попадат в една група. Това правило не е много логично, тъй като то на практика приема, че две значения `NULL` са еднакви, но е по-удобно.

♦ Списък на отделите, включващ номер на отдела, средна заплата и брой служители в отдела.

```
SELECT dno, AVG(sal) avgsal, COUNT(*) empcount
FROM emp
GROUP BY dno;
```

♦ Списък на отделите, включващ номер на отдела и брой служители в отдела със заплата по-малка от 500.

```
SELECT dno, COUNT(*) empcount
FROM emp
WHERE sal < 500
GROUP BY dno;
```

♦ Списък на отделите, включващ номер, име на отдела и брой служители в отдела със заплата по-малка от 500.

```
SELECT dep.dno, dep.dname, COUNT(*) empcount
FROM emp, dep
WHERE dep.dno = emp.dno AND sal < 500
GROUP BY dep.dno, dep.dname;
```

При анализ на коректността на оператор `SELECT` с групиране се игнорира информацията за първични и външни ключове. Затова в горното решение трябва да включим и двете колони в `GROUP BY`, ако искаме да участват в резултата. Това решение може да върне един ред по-малко в сравнение с предходната задача, ако колоната `emp.dno` има значение `NULL` в някои редове.

Условия за групи - фраза `HAVING`

Фразата `HAVING` избира някои от групите, формирани от `GROUP BY`, които да участват при изчисляване на резултата, т.е. има същата роля за групите както `WHERE` за редовете.

`HAVING` *`условие`*

За всяка група се проверява *`условие`* и ако то е истина, групата се избира и от нея се формира един ред за резултата. Условието в тази фраза се конструира както и условието в `WHERE`, но има някои особености. В простите условия могат да участват имена на колони, по които се групира, агрегатни функции и константи, т.е. в условието се проверяват свойства на групата като цяло. Ако в `HAVING` има агрегатна функция, то тя се изчислява за всяка група при проверка на условието за нея. В `HAVING` може да има и условие с вложен оператор `SELECT`, който може да е корелиран по колони на групиране или не.

Ако в `HAVING` няма агрегатна функция, то условието може да бъде пренесено във фразата `WHERE`.

Правилата за обработка на значения `NULL` в условието на `HAVING` са същите както при условието във фразата `WHERE`.

Фразата `HAVING` почти винаги се използва в съчетание с `GROUP BY`, но синтаксисът на `SELECT` не изисква това. Ако `HAVING` е без `GROUP BY`, то всички редове, избрани от `WHERE`, образуват една група.

♦ Списък от номер, име на отдела и брой служители в отдела със заплата по-голяма от 500, който включва отделите със повече от 5 служители със заплата по-голяма 500.

```
SELECT d.dno, d.dname, COUNT(*) cnthighsal
FROM emp e, dep d
WHERE d.dno = e.dno AND e.sal > 500
GROUP BY d.dno, d.dname
HAVING COUNT(*) > 5
ORDER BY cnthighsal DESC;
```

♦ Списък от номер, име на отдела и средна заплата за отдела, който включва отделите със средна заплата по-голяма от средната заплата за предприятието.

```
SELECT d.dno, d.dname, AVG(sal) avgsal
FROM emp e, dep d
WHERE d.dno = e.dno
GROUP BY d.dno, d.dname
HAVING AVG(sal) > (SELECT AVG(sal) FROM emp)
ORDER BY 3 DESC;
```

Правила при изпълнение на многотабличен оператор `SELECT` с групиране и агрегатни функции

Ще опишем правилата при изпълнение на този по-сложен вариант на оператора `SELECT`, при който може да има няколко таблици във фразата `FROM`, вложена заявка, групиране и агрегатни функции.

1. Формира се произведението на таблиците от фразата `FROM`. Ако във `FROM` има само една таблица, то самата тя ще е произведението.
2. Ако има фраза `WHERE`, то се проверява условието в нея за всеки ред на произведението и се избират редовете, за които условието е true, останалите се игнорират. Ако има вложен `SELECT`, той се изпълнява за всеки проверяван ред.
3. Ако има фраза `GROUP BY`, то от избраните редове се формират групи, така че в една група да са редове с еднакви значения във всички колони за групиране.
4. Ако има фраза `HAVING`, то за всяка група се проверява условието в нея и се избират групите, за които е условието е true. Ако има `SELECT`, вложен в `HAVING` той се изпълнява за всяка проверявана група.
5. За всеки избран ред или за всяка избрана група се изчисляват последователно елементите в целевия списък и се формира един ред за резултата. Ако има фраза `GROUP BY` и агрегатни функции, то като аргумент се използват значенията на колоната от всички редове в групата. Ако има агрегатни функции без `GROUP BY`, като агрумент се използват значенията на колоната във всички редове.
6. Ако има ключова дума `DISTINCT`, то се изключват повтарящите се редове.
7. Ако има фраза `ORDER BY`, то се сортират редовете в резултата.

5.2.5. Теоретико-множествени операции

Операцията обединение, позната ни от релационната алгебра, позволява да се обединят редовете на две таблици в една. Тази операция се поддържа в SQL и чрез нея могат да се обединят резултатите от два оператора `SELECT` в една таблица.

select-оператор UNION [ALL] *select-оператор* [...];

За да могат да се обединят резултатите от операторите `SELECT` е необходимо резултатите от тях да са сравними:

- Целевите им списъци трябва да включват еднакъв брой елементи.
- Съответните елементи да са от един тип.
- Не се изискват еднакви имена на съответните елементи. Прието е резултатната колона да взема името от първия оператор `SELECT`.
- Никой от резултатите на оператор `SELECT` не може да е сортиран. Сортиране може да се прилага към резултата от обединението, т.е. ако е необходимо фразата `ORDER BY` се указва след последния оператор `SELECT`.

Какви са правилата за обработка на повтарящите се редове? При операцията `UNION` се премахват повтарящите се редове в резултата след обединението. Ако е указана думата `ALL`, повтарящите се редове не се премахват след обединението. Ако искаме да се изключат дубликатите преди обединението, е необходимо в съответния оператор `SELECT` да зададем ключова дума `DISTINCT`.

♦ Списък от имената и адресите на всички студенти и преподаватели, сортиран по азбучен ред на името. Това е пример от БД-студенти.

```
SELECT ime, adres, 's' AS who
FROM student
UNION ALL
SELECT ime, adres, 't'
FROM prepod
```

```
ORDER BY 1;
```

Операциите сечение и разлика от релационната алгебра са включени директно в стандарта SQL2.

```
select-оператор INTERSECT select-оператор;
select-оператор MINUS select-оператор;
```

В някои СУБД тези релационните операциите не се поддържат директно от езика SQL, тогава те могат да се изразят чрез оператор `SELECT`. Ако `T1` и `T2` са сравними таблици с първични ключове `T1.X` и `T2.X`, които също са сравними по тип, то операцията на релационната алгебра `T1 INTERSECT T2` може да се изрази чрез следния оператор `SELECT`:

```
SELECT *
FROM T1
WHERE X IN (SELECT X FROM T2);
```

или

```
SELECT *
FROM T1
WHERE EXISTS (SELECT * FROM T2 WHERE T1.X = T2.X);
```

Операцията на релационната алгебра `T1 MINUS T2` може да се изрази чрез:

```
SELECT *
FROM T1
WHERE X NOT IN (SELECT X FROM T2);
```

или

```
SELECT *
FROM T1
WHERE NOT EXISTS (SELECT * FROM T2 WHERE T1.X = T2.X);
```

5.3. Обновяване на данни

5.3.1. Оператор INSERT

Добавяне на нови редове към таблица се извършва чрез оператор INSERT, който има два варианта. Първият вариант добавя винаги един ред към таблица.

```
INSERT INTO име-на-таблица [( списък-от-имена-на-колони )]
VALUES ( списък-от-константи );
```

Фразата INTO задава името на таблицата, към която ще се добави новия ред (целевата таблица). Фразата VALUES съдържа значенията на колоните в новия ред. Значенията може да са константи, включително символни константи (USER, SYSDATE, TODAY) и NULL. Списъкът от имена на колони определя коя константа на коя колона е значение. Следователно, трябва да има позиционно съответствие между двата списъка. Ако името на някоя колона на целевата таблица отсъства от списъка, тя ще има значение по премълчаване в новия ред, най-често NULL. За удобство се разрешава *списък-от-имена-на-колони* да не се задава, тогава се подразбира списъка от всички колони на целевата таблица (в същия ред както при SELECT *).

Редовете във всяка таблица са ненаредени (както е и според теорията на релационния модел) и затова в оператора не се указва къде да се добави новия ред по отношение на съществуващите редове.

♦ Да се добавят данни за нов служител.

```
INSERT INTO emp (eno, ename, job, sal, addr, dno)
VALUES (301, 'Иван Петров', 'ас.', 300, 'София, ул.Латинка 6', NULL);
```

или

```
INSERT INTO emp (eno, ename, job, sal, addr)
VALUES (301, 'Иван Петров', 'ас.', 300, 'София, ул.Латинка 6';
```

Вторият вариант на оператор INSERT добавя няколко реда към целевата таблица, които се изчисляват от данни в други таблици на БД чрез вложен оператор SELECT.

```
INSERT INTO име-на-таблица [( списък-от-имена-на-колони )]
оператор-select ;
```

Вложеният *оператор-select* може да е произволно сложен, но се налагат някои ограничения:

- Целевият списък на вложения *оператор-select* и списъкът от имена на колони трябва да са съвместими по брой и по тип на данните за съответните елементи.
- Вложеният оператор не трябва да съдържа фразата ORDER BY, тъй като няма смисъл да се сортират редовете преди да се добавят към таблица, където редовете са ненаредени.
- Вложеният оператор трябва да е върху таблици различни от целевата.
- Вложеният оператор не трябва да е с обединение UNION.

В новите стандарти последните две ограничения са отменени, но е възможно някои СУБД все още да ги изискват.

♦ Към таблица losalemp (eno, name, job, dreamsal) да се добавят данни за служители със заплати по-малки от 500.

```
INSERT INTO losalemp (eno, name, job, dreamsal)
      SELECT eno, ename, job, sal*2
      FROM emp
      WHERE sal < 500;
```

5.3.2. Оператор UPDATE

Оператор UPDATE изменя значенията на една или повече колони в избрани редове на една таблица.

```
UPDATE име-на-таблица [ псевдоним ]
SET    { име-на-колона  =  израз [, ... ]
       | (списък-от-колони ) =  (оператор-select )
       }
[ WHERE условие ];
```

Първата фразата задава името на целевата таблица. Фразата WHERE избира редовете, които ще бъдат изменени. Условието се конструира по същия начин, както във фразата WHERE на оператор SELECT. Това означава, че може да има условие с вложен SELECT. Това е полезно когато изборът на редовете за изменение се основава на данни от други таблици. Вложеният SELECT може да е независим, но може и да е корелиран с целевата таблица. В някои системи има ограничение върху вложения оператор SELECT да е върху таблици, различни от целевата. Ако няма фраза WHERE, то измененията се извършват във всички редове на целевата таблица.

Фразата SET определя колоните, които ще бъдат изменяни и начина, по който това ще стане, т.е представлява списък от присвоявания. Значението на *израз* се изчислява като се използват значенията в текущия ред преди изменението. Типът на израза трябва да съответства на типа на съответната му колона. В някои СУБД, напр. Oracle, е възможно *израз* да е вложен SELECT, връщащ едно значение. Там е реализирано и допълнението към синтаксиса, където *оператор-select* е вложен оператор, връщащ един ред с необходимия брой значения. И в двата случая може да има ограничение върху вложеният SELECT да е върху таблици, различни от целевата, но може да е колелиран с нея.

♦ Да се измени длъжността и заплатата на служител с номер 120.

```
UPDATE emp SET sal = 350, job = 'ac.'
WHERE eno = 120;
```

♦ Да се увеличи заплатата на всички служители с 10%.

```
UPDATE emp SET sal = sal*1.10;
```

♦ Да се увеличи с 10% заплатата на всички служители от отдел 1 и те да се преместят в отдел 10.

```
UPDATE emp SET sal = sal * 1.10, dno = 10
WHERE dno = 1;
```

♦ Да се увеличи с 50лв. заплатата на всички служители, работещи в отдел с началник служител с номер 100.

```
UPDATE emp SET sal = sal + 50
WHERE dno IN (SELECT dno FROM dep WHERE mgr = 100);
```

♦ Да се увеличи с 50лв. заплатата на всички служители, които участват в повече от 3 проекта.

```
UPDATE emp SET sal = sal + 50
WHERE eno IN (SELECT eno FROM emp_pro
              GROUP BY eno
              HAVING COUNT(*) > 3);
```

или друг вариант с корелиран вложен SELECT

```
UPDATE emp SET sal = sal + 20
WHERE 3 < (SELECT COUNT(*) FROM emp_pro
           WHERE emp_pro.eno = emp.eno);
```

♦ Да се увеличи с 50лв. заплатата на всички служители, които са работели по проекти повече от 100 дена общо.

```
UPDATE emp SET sal = sal + 50
WHERE eno IN (SELECT eno FROM emp_pro
              GROUP BY eno
              HAVING SUM(ptime) > 100);
```

♦ Да се измени длъжността и заплатата на служителите, за които има редове в `losalemp`, като новите данни се вземат от там.

```
UPDATE emp
SET (sal, job) = (SELECT dreamsal, job FROM losalemp
                  WHERE  losalemp.eno = emp.eno)
WHERE eno IN (SELECT eno FROM losalemp);
```

♦ Да се измени заплатата на служител с номер 120 със средната заплата за отдела му.

```
UPDATE emp e
SET sal = (SELECT avg(sal) FROM emp x
           WHERE x.dno = e.dno)
WHERE eno = 120;
```

5.3.3. Оператор DELETE

Операторът `DELETE` изтрива избрани редове от една таблица.

```
DELETE FROM име-на-таблица [ псевдоним ]
[ WHERE условие ];
```

Фразата `FROM` задава името на целевата таблица. Фразата `WHERE` избира редовете, които ще бъдат изтрити. Условието се конструира по същите правила, както условието във фразата `WHERE` на оператор `UPDATE` (може да има условие с вложен `SELECT`, като и ограниченията са същите както при оператор `UPDATE`). Ако няма фраза `WHERE`, то се изтриват всички редове от целевата таблица.

♦ Да се изтрият всички данни от таблица `losalemp`.

```
DELETE FROM losalemp;
```

♦ Да се изтрият всички данни за проект с номер 20.

```
DELETE FROM emp_pro WHERE pno = 20;
DELETE FROM project WHERE pno = 20;
```

♦ Да се изтрият всички данни за проект с наименование ABC.

```
DELETE FROM emp_pro
  WHERE pno IN (SELECT pno FROM project WHERE pname = 'ABC');
DELETE FROM project WHERE pname = 'ABC';
```

♦ Да се изтрият всички данни за отдел с номер 10.

```
UPDATE emp SET dno = NULL
  WHERE dno = 10;
DELETE FROM dep WHERE dno = 10;
```

♦ Да се изтрият всички данни за служител с номер 120.

```
DELETE FROM emp_pro WHERE eno = 120;
UPDATE dep SET mgr = NULL
  WHERE mgr = 120;
DELETE FROM emp WHERE eno = 120;
```

♦ Да се изтрият всички данни за служител с име Иван Иванов.

```
DELETE FROM emp_pro
  WHERE eno IN (SELECT eno FROM emp WHERE ename = 'Иван Иванов');
UPDATE dep SET mgr = NULL
  WHERE mgr IN (SELECT eno FROM emp WHERE ename = 'Иван Иванов');
DELETE FROM emp WHERE ename = 'Иван Иванов';
```

5.4. Описание на данните

Ще разгледаме операторите на езика SQL за описание и създаване на БД. Наборът от оператори, които определят структурата на БД, се нарича Data Definition Language (DDL). Макар, че DDL и DML са две отделни части на езика SQL, в повечето СУБД такова разделяне е само на абстрактно ниво. Те са напълно равноправни и могат да се използват смесено, както при интерактивна работа, така и във вграден SQL. Това е едно от преимуществата на релационния модел в сравнение с по-ранните модели на данните. Структурата на БД е динамична, т.е. могат да се създават, унищожават и изменят обекти едновременно с достъпа на потребителите до БД. Ядрото на DDL се състои от три глагола, с които започва почти всеки оператор:

- `CREATE` – Описва и създава обект в БД.
- `ALTER` – Изменя описанието на съществуващ обект в БД.
- `DROP` – Унищожава обект в БД.

След всяка от тези думи в оператора следва ключова дума, определяща вида на обекта, за който се отнася оператора – таблица, синоним, виртуална таблица (view), индекс и др.

5.4.1. Базови таблици

5.4.1.1. Оператор CREATE TABLE

Чрез този оператор се определя и създава най-важният обект в БД - таблица. Така създадена таблица ще наричаме **базова таблица**, защото е постоянна и реална и за да я различаваме от таблиците, създадени чрез оператор `SELECT` или чрез оператор `CREATE VIEW`. Операторът работи с абстрактните понятия на релационния модел, но не може да не се засегне и физическото представяне на таблиците. Има няколко фрази, които определят параметри на физическото преставяне. Те са различни в различните СУБД и ще ги разгледаме в раздела за вътрешно представяне на БД. Потребителят, изпълнил оператора `CREATE`, става собственик на таблицата.

```
CREATE TABLE име-на-таблица
( име-на-колона тип-данни [DEFAULT значение] [NOT NULL]
                    [ ограничение-за-цялостност-за-колоната ]
  [, ...]
[, ограничение-за-цялостност-за-таблицата ] [, ...]
)
[ параметри-на-физическото-представяне ];
```

Името на таблицата трябва да е уникално. Според стандарта се изисква уникалност на пълното име на таблица, т.е. "*име-на-собственик.име-на-таблица*", но в някои СУБД се изисква уникалност за собственото име на таблица. След името на таблицата следва описанието на колоните и ограниченията за цялостност, което е заградено в кръгли скоби. Накрая са параметрите, управляващи физическото представяне на таблицата, които ще разгледаме по-нататък.

Описанието на всяка колона включва следната информация:

- Име на колоната. Името трябва да е уникално в таблицата.
- Тип на данните.
- Значение по премълчаване - `DEFAULT` *значение*.

Това е значението, което се използва когато в оператор `INSERT` не е зададено значение за колоната в добавяните редове. Значението може да е константа, `NULL` или символна константа – `USER`, `SYSDATE` и др. По премълчаване, ако няма фраза `DEFAULT`, се използва значение `NULL`.

- Ограничение, забраняващо неопределено значение – `NOT NULL`.

- Ограничения за цялостност, отнасящи се само до описваната колона.

Всъщност това е "съкратена" форма, която може да се използва когато ограничението за цялостност се отнася само до една колона на таблицата. Тогава то може да се добави в края на описанието на колоната.

След описание на колоните на таблицата могат да се определят и ограничения за цялостност, които се отнасят до всички описани вече колони. Това е "пълната" форма при задаване на ограничения за цялостност.

Типове данни в Oracle

`CHAR(`*n* `[BYTE|CHAR])`	символен низ с фиксирана дължина *n* (1<=*n*<=2000).
`VARCHAR2(`*n* `[BYTE|CHAR])`	символен низ с променлива дължина, най-много *n* (1<=*n*<=4000)
`NUMBER(`*p*`[,`*s*`])`	число с *p* десетични цифри и *s* след десетичната точка (1<=*p*<=38, -84<=*s*<=127, по премълчаване *s* е 0)
`DATE`	календарна дата
`TIMESTAMP`	календарна дата и време
`INTERVAL` *точност*	период от време, измерван с определена точност и мерни единици

При типа `INTERVAL,` *точност* е от вида:

`YEAR(`*n*`) TO MONTH`	период от време в години и месеци
`DAY (`*n*`) TO SECOND(`*m*`)`	период от време в дни, часове, минути, секунди

където *n* и *m* са максималният брой десетични цифри, използвани при съхраняването.

Типове данни в Informix

`INTEGER, INT`	цяло число, съхранявано в двоично представяне в 4 байта
`SMALLINT`	цяло число, съхранявано в двоично представяне в 2 байта
`SERIAL[(`*s*`)]`	цяло число, представяно като `INT`, чиито значения се генерират автоматично от СУБД като последователни цели числа, започвайки от *s* или 1 по премълчаване
`DECIMAL[(`*p*`[,`*q* `])]`	десетично число с *p* цифри (16 по премълчаване), от които *q* са след десетичната точка. Ако *q* не е зададено, се счита, че точката може да плава. (0≤*q*≤*p*, 1≤*p*≤32)
`MONEY[(`*p* `[,`*q* `])]`	десетично число с *p* цифри (16 по премълчаване), от които *q* са след десетичната точка (2 по премълчаване)
`FLOAT,DOUBLE PRECISION`	число, съхранявано в представяне с плаваща точка в 8 байта
`REAL, SMALLFLOAT`	число, съхранявано в представяне с плаваща точка в 4 байта
`CHAR(`*n*`)`	символен низ с фиксирана дължина *n* байта (1≤ *n*≤ 32767)
`VARCHAR(`*n*`)`	символен низ с променлива дължина, най-много *n* байта (1≤ *n*≤ 255)
`DATE`	календарна дата, съхранявана като цяло число, което е брой дни след 31/12/1899
`DATETIME` *точност*	календарна дата и време с определена точност
`INTERVAL` *точност*	времеви интервал, измерван с определена точност и мерни единици

При типа `DATETIME,` *точност* е от вида: *first* `TO` *last*, където *first* и *last* са от списъка:

```
YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, FRACTION(n)
```

При типа `INTERVAL, точност` е от вида: `first[(n)] TO last[(m)]`, където `first` и `last` са и двете от един от двата списъка:

```
YEAR, MONTH
DAY, HOUR, MINUTE, SECOND, FRACTION
```

а `n` и `m` са максималният брой десетични цифри, използвани при съхраняването.

Ограничения за цялостност

Типовете ограничения, които могат да бъдат зададени в описанието на таблица са:

- **Първичен ключ - `PRIMARY KEY`**

Една таблица може да има най-много един първичен ключ. Всяка СУБД налага ограничения за максималната дължина на първичния ключ (напр., в Oracle е до 32 колони с максимална обща дължина до 1 блок, а в Informix е до 16 колони с максимална обща дължина 255 байта). Обикновено за първичния ключ не се разрешава значение `NULL`.

- **Ограничение за уникалност – `UNIQUE`**

Това е ограничение чрез което може да се поддържа понятието възможен ключ от релационния модел. Една колона не може да е определена едновременно като първичен ключ и като `UNIQUE`. Ограниченията за максималната дължина са същите, както за първичен ключ. За възможния ключ се разрешава значение `NULL`.

- **Външен ключ - `FOREIGN KEY`**

Определя колоните, които са външен ключ в описваната таблица и връзката, която той създава с друга таблица *име-на-таблица-pk* и съответния му първичен ключ *име-на-колона-pk* (може да е и `UNIQUE`). Когато се изпълнява операторът СУБД сравнява определяния външен ключ със съответния му първичен ключ по брой колони и тип на данните. Следователно таблицата, към която сочи външния ключ, трябва вече да съществува освен ако не е създаваната в изпълнявания оператор.

- **Условие за проверка - `CHECK`**

Това условие налага ограничения на съдържанието на всеки ред от таблицата, разглеждайки го независимо от останалите. Следователно условието е логически израз, в който се сравняват изрази върху колони на описваната таблица и константи. Условието се проверява при всеки опит да се добави или измени ред и ако не е истина операцията не се изпълнява (условието може да е unknown, ако има значение `NULL` в реда).

В синтаксиса има два варианта за определяне на ограничения за цялостност:

ограничение-за-цялостност-за-колоната

Може да се използва, когато ограничението се отнася до една колона и се добавя в края на описанието на колоната.

```
[ CONSTRAINT име-на-ограничение ]
{ UNIQUE | PRIMARY KEY
| REFERENCES име-на-таблица-pk ( име-на-колона-pk )
            [ON DELETE {CASCADE | SET NULL}]
| CHECK ( условие ) }
```

ограничение-за-цялостност-за-таблицата

Ограничението е зададено след описанието на колоните.

```
[ CONSTRAINT име-на-ограничение ]
{ { UNIQUE | PRIMARY KEY }(име-на-колона-pk [, ...])
| FOREIGN KEY(име-на-колона-fk [,...])
            REFERENCES име-на-таблица-pk (име-на-колона-pk [,...])
            [ON DELETE {CASCADE | SET NULL}]
| CHECK ( условие ) }
```

На всяко ограничение за цялостност може да се даде име чрез фразата CONSTRAINT, което е удобно когато впоследствие се наложи да се отменят ограничения. Името на ограничението е идентификатор. Ако не е зададено име на ограничението, то СУБД генерира по свои правила уникално име.

5.4.1.2. Оператор DROP TABLE

Чрез оператор DROP TABLE се унищожава таблица. Това означава, че се унищожават данните в нея, ограниченията за цялостност върху нея и други обекти в БД, свързани с таблицата – синоними, views, индекси, ограничения за достъп.

```
DROP TABLE име-на-таблица [CASCADE CONSTRAINTS];
```

Този оператор може да промени и описанието на други таблици. Например, ако с унищожаваната таблица има свързан външен ключ в друга таблица, то ако е зададено CASCADE CONSTRAINTS, се унищожава и ограничението за съответния външен ключ. Ако унищожаваната таблица се използва в определението на view, то в Oracle не се унищожава, а става невалидно (не може да се използва, докато не се създаде отново) в други системи виртуалната таблица може да се унищожава.

5.4.1.3. Оператор ALTER TABLE

Чрез оператор ALTER TABLE се изменя описанието на таблица. Някои изменения могат да предизвикат и изменения на съдържанието ѝ, ако не е празна или са невъзможни, ако не е празна. Най-общият вид на оператора е следния:

```
ALTER TABLE име-на-таблица
{
  ADD фраза
| DROP фраза
| MODIFY фраза
| MODIFY CONSTRAINT фраза
};
```

Всяка от фразите определя различен вид изменение на описанието на таблицата с име *име-на-таблица*.

Добавяне на нова колона

```
ADD (име-на-нова-колона тип-данни [DEFAULT значение] [NOT NULL]
                    [ограничение-за-цялостност-за-колоната]
      [,...])
```

Добавят се една или повече нови колони към съществуваща таблица. Описанието на всяка нова колона включва същите елементи както в оператора CREATE TABLE. Ако таблицата не е празна, във всичките съществуващи редове новата колона ще има значение, зададено в DEFAULT или NULL. Това определя някои ограничения в случай, че таблицата не е празна. Ограничение UNIQUE може да се зададе заедно с DEFAULT, ако таблицата има най-много един ред. Ограничение PRIMARY KEY може за празна таблица или за таблица с един ред и DEFAULT значение различно от NULL. За непразна таблица NOT NULL може заедно с DEFAULT.

Изменение на описанието на колона

```
MODIFY (име-на-стара-колона тип-данни [DEFAULT значение] [[NOT] NULL]
                    [ограничение-за-цялостност-за-колоната]
        [,...])
```

Чрез тази фраза може да се измени описанието на съществуваща колона, а именно:

- типа на данните;

- да се добави или измени значение по премълчаване;
- да се забрани или разреши неопределено значение;
- да се добави или отмени ограничение за цялостност.

При смяна на типа на данните е позволено почти всичко, стига съществуващите данни в таблицата да могат да бъдат преобразувани към новия тип и след преобразуването да не бъде нарушено ограничение `UNIQUE`. Следователно, ако таблицата не е празна, в някои случаи изменението може да не се извърши.

Когато се променя описанието на колона, какво правим за характеристиките от старото определение, които искаме да запазим. В Informix те трябва отново да се зададат в оператора, а в Oracle всяка от характеристиките, която не е зададена в оператора, не се променя.

Унищожаване на колона

`DROP (`*`име-на-стара-колона`* `[, ...]) [CASCADE CONSTRAINTS]`

Унищожават се колоните на таблицата и всички свързани с тях ограничения за цялостност, индекси по колоните и други обекти в БД. Следователно това изменение може да промени и описанието на други таблици. Ако унищожаваната колона е първичен ключ или `UNIQUE`, към който сочи външен ключ от друга таблица, то се унищожава и ограничението за съответния външен ключ, ако е зададено `CASCADE CONSTRAINTS`, в противен случай е грешка.

Добавяне на ограничение за цялостност

`ADD` *`ограничение-за-цялостност-за-таблица`* `[, ...]`

Добавят се нови ограничения за цялостност. Използва се пълният формат на синтаксиса, както в оператора `CREATE TABLE`. Ако таблицата не е празна, данните в нея трябва да не нарушават добавяните ограничения за цялостност.

Унищожаване на ограничение за цялостност

`DROP CONSTRAINT` *`име-на-ограничение`* `[CASCADE]`

Унищожават се ограничението за цялостност, идентифицирано чрез името си. Следователно, както при унищожаване на колона, това изменение може да промени и описанието на други таблици, напр. ако се унищожи ограничение за първичен (възможен) ключ, то се унищожават и ограниченията за съответните му външни ключове, ако е зададено `CASCADE`.

♦ Да се опишат таблиците на БД-служители.

```
CREATE TABLE dep
(  dno NUMBER(4) CONSTRAINT dep_pk PRIMARY KEY,
   dname VARCHAR2(30) NOT NULL UNIQUE,
   budget NUMBER(7,2) NOT NULL,
   mgr NUMBER(6)
);
CREATE TABLE emp
(  eno NUMBER(6) CONSTRAINT emp_pk PRIMARY KEY,
   ename VARCHAR2(50) NOT NULL,
   sal NUMBER(6,2)  DEFAULT 0  NOT NULL,
   job VARCHAR2(20),
   addr VARCHAR2(50),
   dno NUMBER(4),
   CONSTRAINT dno_fk FOREIGN KEY (dno) REFERENCES dep(dno)
);
CREATE TABLE projects
(  pno NUMBER(4) CONSTRAINT pro_pk PRIMARY KEY,
   pname VARCHAR2(30) NOT NULL,
```

```
   description VARCHAR2(100)
);
CREATE TABLE emp_pro
(  eno NUMBER(6),
   pno NUMBER(4),
   ptime INTERVAL YEAR TO MONTH,
   CONSTRAINT ep_pk PRIMARY KEY(eno, pno),
   CONSTRAINT eno_fk FOREIGN KEY(eno) REFERENCES emp(eno),
   CONSTRAINT pno_fk FOREIGN KEY(pno) REFERENCES project(pno)
);

ALTER TABLE dep
   MODIFY (budget NUMBER(10,2) DEFAULT 0);

ALTER TABLE dep
   ADD CONSTRAINT budget_chk CHECK (budget >= 0);

ALTER TABLE emp
   ADD (comm NUMBER(7,2) DEFAULT 0 NOT NULL,
        email VARCHAR2(25));

ALTER TABLE emp
   ADD CONSTRAINT sal_chk CHECK (sal + comm < 5000);

ALTER TABLE emp
   DROP (comm) CASCADE CONSTRAINTS;

ALTER TABLE dep DROP UNIQUE (dname);

ALTER TABLE dep
   ADD CONSTRAINT dname_chk CHECK (dname = UPPER(dname));
```

5.4.2. Синоними

Много СУБД изискват използване на пълни имена на таблици, когато потребителят работи с чужди таблици както е и по стандартите. Това води до неудобството, че имената стават дълги и трябва да се знае и помни собственика. За решаване на тези проблеми в някои СУБД се въвежда понятието синоним. Синоним е алтернативно име на таблица (базова или view), което се създава с оператора `CREATE SYNONYM` и съществува докато явно не се унищожи с оператор `DROP SYNONYM`. Синоним може да се използва в SQL оператори като име на таблица.

`CREATE [PUBLIC] SYNONYM` *`синоним`* `FOR` *`име-на-таблица`* `;`

Името на синонима трябва да е уникално. Пространството от имена на базови таблици, виртуални таблици, временни таблици и синоними е едно. Потребителят, изпълнил `CREATE SYNONYM` става собственик на синонима. В някои системи има два вида синоними - `PUBLIC` и `PRIVATE`. Синоним `PUBLIC` може да се използва от всички потребители, имащи достъп до БД. Синоним, който не е `PUBLIC` (`PRIVATE`), може да се използва само от собственика си, другите потребители трябва да използват пълно име на синонима. Синоним се унищожава с оператора `DROP SYNONYM`.

`DROP [PUBLIC] SYNONYM` *`синоним`* `;`

5.4.3. Създаване и унищожаване на база данни

Преди да започне създаването на таблици, синоними и други обекти е необходимо да се създаде самата база данни, която първоначално ще е празна. В стандарта SQL1 не е определен начинът, по който това се извършва, затова в СУБД има различни подходи. Няколко примера:

В Oracle БД се създава в процеса на инсталиране на системата, тъй като се предполага, че на всеки компютър има една база данни и създаването е най-често

еднократно действие. БД може да се създаде и впоследствие, но процедурата е доста сложна.

В Ingres и Postgres има специални сервизни програми: `createdb` - за създаване на нова БД, `destroydb` - за унищожаване на БД. Това не са оператори на езика SQL.

В Informix, SQL Server, Sybase има оператори на езика SQL, част от DDL, чрез които се създава нова БД, унищожава се БД или се извършват други действия с БД като цяло.

В същност в СУБД се реализират два основни подхода за организация на базите данни в една компютърна система.

Еднобазова архитектура

СУБД поддържа работата с една обща БД, която може да съдържа таблици за различни информационни системи. Преимуществото на този подход е, че БД не трябва да се идентифицира и достъпът до нея е по-прост. Недостатък тук е опасността БД да стане прекалено голяма и трудно управляема. Освен това много често се налага за едно приложение и на един компютър да се поддържат две БД – една за промишлено използване и друга за развитие и тестване на нови приложения.

Многобазова архитектура

СУБД поддържа работа с няколко БД, всяка предназначена за независимо приложение. Например БД за заплати, счетоводна БД, БД за обработка на поръчки. Всяка БД, съхранявана в компютърната система, има уникално име. Преимуществото при тази организация се състои в разделянето на задачата за управление на данните на по-малки и леки задачи. Всяка БД може да има свой АБД. Проблемът е, че трябва да има средства за свързване с БД.

От казаното до тук става ясно, че съществува разнообразие по отношение на това как са организирани базите данни, как се създават и как се осъществява достъп към тях. Тази част от езика SQL е една от най-нестандартизираните.

5.4.4. Виртуални таблици (views)

Виртуална таблица (view) е таблица, която за разлика от базовата таблица е:

- Производна – Редовете в нея се изчисляват от данните в базови таблици.
- Виртуална – Данните не се съхраняват отделно от данните на базовите таблици, от които е производна.

По друг начин погледнато view е съхранен `SELECT` оператор, който има име и чрез който се виждат данните в изходните базови таблици. От тук и името му view (изглед), тъй като е прозорец, през който се гледат данните в БД. Предимствата, които дава използването му, са няколко:

- Дава възможност различните потребители да виждат едни и същи данни по различен начин. Така за всеки потребител може да се изгради най-подходящата за него структура на БД.
- Използва се при осигуряване на сигурност на данните за ограничаване достъпа на потребителите само до производни данни.
- Използва се и за осигуряване на цялостност при view с режим на контрол.

5.4.4.1 Създаване и унищожаване

Нова виртуална таблица се създава с оператор `CREATE VIEW`. Ядрото на този оператор е вложен `SELECT` оператор, който определя начина на изчисляване на редовете и колоните на виртуалната таблица. Вложеният `SELECT` оператор може да е произволно сложен, но без фразата `ORDER BY`. Потребителят, изпълнил оператора `CREATE VIEW`, става собственик на новата виртуална таблица.

```
CREATE [OR REPLACE] VIEW име-на-view [( списък-от-имена-на-колони )]
AS оператор-select
[WITH CHECK OPTION];
```

Името *име-на-view* трябва да е уникално в БД. Правилата са същите както при имена на базови таблици. Ако е зададена ключовата дума `OR REPLACE`, то се изменя дефиницията на съществуващо view, като така се избягва последователното изпълнение на `DROP` и `CREATE`. След името на виртуалната таблица в скоби се задават имената на виртуалните колони. Ако този списък е пропуснат, тогава имената се наследяват от имената на колоните от `SELECT` оператора. Следователно, ако в целевия списък на `SELECT` оператора има израз или колони с еднакви собствени имена, е задължително да се дадат имена на колоните на view. Между целевия списък на оператора `SELECT` и *списък-от-имена-на-колони* има позиционно съответствие. Това означава, че ако искаме да дадем име на една виртуална колона, трябва да дадем на всички.

Виртуална таблица може да се създава от базови и други виртуални таблици.

Фразата `WITH CHECK OPTION` определя създаване на виртуална таблица с режим на контрол, което е свързано с използването й. Този режим не позволява да се изпълни оператор `INSERT` или `UPDATE` чрез виртуалната таблица, който нарушава условието във фразата `WHERE` на `CREATE VIEW`.

Това, което се случва при изпълнение на оператора `CREATE VIEW`, е че се съхранява определението на виртуалната таблица. Не се изпълнява вложения оператор `SELECT` и следователно не се материализира виртуалната таблица.

Виртуална таблица се унищожава чрез оператор `DROP VIEW`.

```
DROP VIEW име-на-view;
```

Унищожава се определението на виртуалната таблица *име-на-view*. Ако на базата на *име-на-view* са създадени други виртуални таблици, тогава в Oracle те стават невалидни. В Informix чрез ключва дума `CASCADE` или `RESTRICT` се указва дали да се унищожат и всички други виртуални таблици, определени чрез унищожаваната, или операторът да завърши с грешка ако има други виртуални таблици, определени чрез унищожаваната.

5.4.4.2. Използване на виртуални таблици

Виртуална таблица може да се използва само в операторите за манипулиране на данни – `SELECT`, `INSERT`, `UPDATE` и `DELETE`. Когато СУБД срещне името на виртуална таблица в оператор, тя преобразува потребителския оператор в еквивалентен оператор върху изходните базови таблици и го изпълнява. По такъв начин системата поддържа илюзията, че виртуалната таблица съществува, като я материализира когато има обръщение към нея.

Когато потребител се опита да изпълни обновяващ оператор (`INSERT`, `UPDATE` и `DELETE`) върху виртуална таблица, системата трябва да преобразува тази операция в операция по обновяване на изходните базови таблици. Това обаче не винаги е възможно и затова се налагат някои ограничения на възможните начини за използване на виртуална таблица. Виртуална таблица наричаме напълно обновяема, ако са разрешени всички обновяващи оператори. Това е така, ако операторът `SELECT` в определението й отговаря на следните изисквания:

- Не съдържа ключовата дума `DISTINCT`.
- Във фразата `FROM` има само едно име на таблица.
- Няма групиране – фрази `GROUP BY` и `HAVING`.
- Всеки елемент от целевия списък е име на колона, т.е. няма изрази или агрегатни функции.

- Условието в `WHERE` не съдържа подзаявка.
- Не съдържа `UNION`, `INTERSECT`, `MINUS`.

Някои виртуални таблици са частично обновяеми, т.е. някои обновяващи оператори са възможни, а други не. Например, в Oracle ако има изрази в целевия списък, то операторът `UPDATE` е разрешен при условие, че не се изменят виртуалните колони, получавани чрез израз. Също така, ако има съединение, в някои случаи може частични обновявания.

♦ Да се създаде виртуална таблица, съдържаща пълните данни за служителите от отдел с номер 1.

```
CREATE VIEW vemp1
AS SELECT * FROM emp WHERE dno = 1
WITH CHECK OPTION;
```

Таблица като `vemp1` представлява хоризонтално подмножество на една базова таблица и може да се използва, за да се ограничи достъпа на потребители само до част от редовете на базова таблица. Тя е напълно обновяема. Това, че е създадена с режим на контрол, означава невъзможност да се изпълни оператор:

```
UPDATE vemp1 SET dno = 2 WHERE eno = 100;
```

Чрез този оператор би се изменил ред във виртуалната таблица, който след това няма да е видим чрез нея. Това ще е допустимо, ако е създадена без `WITH CHECK OPTION`.

♦ Да се създаде виртуална таблица, съдържаща за всеки служител номер, име, длъжност и адрес на служителя.

```
CREATE VIEW vemp
AS SELECT eno, ename, job, addr FROM emp;
```

Таблица като `vemp` представлява вертикално подмножество на една базова таблица и може да се използва, за да се ограничи достъпа на потребители само до част от колоните на базова таблица. Такъв вид виртуална таблица може да е частично обновяема. Операторът `INSERT` ще е разрешен само ако не се нарушава ограничение `NOT NULL` за колона, не участваща във виртуалната таблица. В този случай `INSERT` е разрешен (виж описанието на таблица `emp`). Операторите `DELETE` и `UPDATE` са разрешени.

♦ Да се създаде виртуална таблица, съдържаща за всеки служител номер, име, месечна и годишна заплата на служителя.

```
CREATE VIEW vpay (nom, name, msal, ysal)
AS SELECT eno, ename, sal, sal*12 FROM emp;
```

Тук има виртуална колона - `ysal`, на която съответства израз в целевия списък. Затова `vpay` е частично обновяема. Операторът `DELETE` е разрешен, но операторите `INSERT`, `UPDATE` в някои случаи може да са, а в други да не са разрешени. Например, възможно е в някои СУБД да е разрешен `UPDATE`, ако не изменя колоната `ysal`.

♦ Да се създаде виртуална таблица, съдържаща номер, име, месечна заплата за служителя и номер на началника му, за служителите със заплата по-малка от 500.

```
CREATE VIEW vlopay (nom, name, msal, mgr)
AS SELECT eno, ename, sal, mgr FROM emp, dep
WHERE emp.dno = dep.dno AND sal < 500;
```

Това е напълно необновяема виртуална таблица, тъй като се изчислява от две базови таблици. Какво значи, например “да се измени значението на колоната `mgr` за служител с номер 100”? Дали това означава, че служителят преминава към друг

отдел или се сменя началникът на отдела му. Невъзможно е такъв оператор `UPDATE` да бъде преобразуван еднозначно в оператори върху базовите таблици `emp` и `dep`. В други системи това може да е частично обновяема виртуална таблица. Може да се измени заплатата или името на служителя, но не може да се изменя колоната `mgr`. Може да се добави нов ред чрез view, ако не се задава значение за колоната `mgr`.

♦ Да се създаде виртуална таблица, съдържаща за всеки отдел номер на отдела, минимална, максимална и средна заплата за отдела.

```
CREATE VIEW vdep (dno, losal, hisal, avgsal)
AS SELECT dno, MIN(sal), MAX(sal), AVG(sal)
FROM emp GROUP BY dno;
```

Това също е напълно необновяема виртуална таблица, тъй като съдържа групиране и агрегатни функции. Причината е очевидна. Какво значи, например "да се измени средната заплата за отдел 1"? Невъзможно е такъв оператор `UPDATE` да бъде преобразуван в оператор върху базовата таблица `emp`. Освен това тук съществуват и ограничения за операторите `SELECT` върху `vdep`. Тъй като е недопустимо влагането на агрегатни функции е недопустим и следният оператор:

```
SELECT MIN(avgsal) FROM vemp;
```

Преобразуваният оператор ще изглежда така:

```
SELECT MIN(AVG(sal)) FROM emp GROUP BY dno;
```

Предимства и недостатъци на виртуалните таблици

Използването на виртуални таблици е полезно в най-различни ситуации и по различни причини. Основните преимущества, които дава използването им са:

Сигурност на данните. Да се ограничи достъпът на потребител до част от БД чрез виртуални таблици, до които той има достъп, а няма такъв за базовите.

Простота на структурата. За всеки потребител може да се създаде собствена "структура" на БД.

Цялостност на данните. Когато обновяването на данни се извършва чрез обновяема виртуална таблица с режим на контрол.

Независимост на данните. При реструктуриране на БД, за потребителя "структурата" може да остане непроменена.

Но наред с преимуществата използването на виртуални таблици има два основни недостатъка:

Производителност. Преобразуването на оператора и материализирането на виртуалната таблица при всяка заявка към нея изисква време, което зависи от сложността на определението на виртуалната таблица.

Ограничения при обновяване. Сложните виртуални таблици са необновяеми, което ограничава тяхното използване.

5.5. Вътрешно представяне на БД

Как обектите на релационния модел – базови таблици, виртуални таблици, синоними и други се съхраняват в паметта? Различните СУБД реализират различно физическо представяне на БД, което определя сложността на системата и за какви цели може да се използва. СУБД, които работят на персонални компютри, са предназначени за малки БД и реализират по-проста физическа организация. Например, в MS Access цялата БД се съхранява като един файл на операционната система. Всъщност в по-новите версии организацията малко е усъвършенствана и БД може да се разположи в няколко файла. В системи, работещи на сървери, като Oracle, Infomix, Ingres и др. физическата организация е доста по-сложна. Това позволява на АБД да управлява разполагането на данните, разпределяйки ги на различни дискове и така да повиши производителността при големи бази данни.

Въпреки съществуващите различия, СУБД имат една обща характеристика: каквото и да е физическото представяне на таблиците, и данните и метаданните се съхраняват като таблици. Освен таблиците създавани с оператор `CREATE TABLE` съществуват и така наречените **системни таблици**. Те се създават автоматично когато се създава БД и съдържат метаданни, т.е. данни за обектите в БД. Обикновено тази група от таблици се нарича с общото име **системен каталог**. Какво е предимството при такъв подход? Едно важно предимство е, че информация за БД може да се получава чрез оператор `SELECT`. Но обновяването на данните в системните таблици обикновенно се извършва само от СУБД при изпълнение на различните DDL оператори.

5.5.1. Вътрешно представяне в Oracle

Ще разгледаме физическото представяне на БД в Oracle, понятията свързани с него и елементите на SQL операторите, където се задават параметри на вътрешното представяне.

Физически понятия

Това са понятия, използвани при управлението на дисковата памет и свързани с организацията на дисковата памет. Връзката между физическите понятия е представена на Фиг. 5.

Файл с данни (datafile)

Това е най-голямата физическа единица дискова памет, която се разпределя от АБД за съхраняване на данни от БД. Може да е файл на операционната система или цял неформатиран дял на диск. При втория вариант, целият дял на диска е отделен за БД, на него не се изгражда файлова система и не съдържа други данни, което осигурява по-висока надеждност и по-добра производителност.

Блок (block)

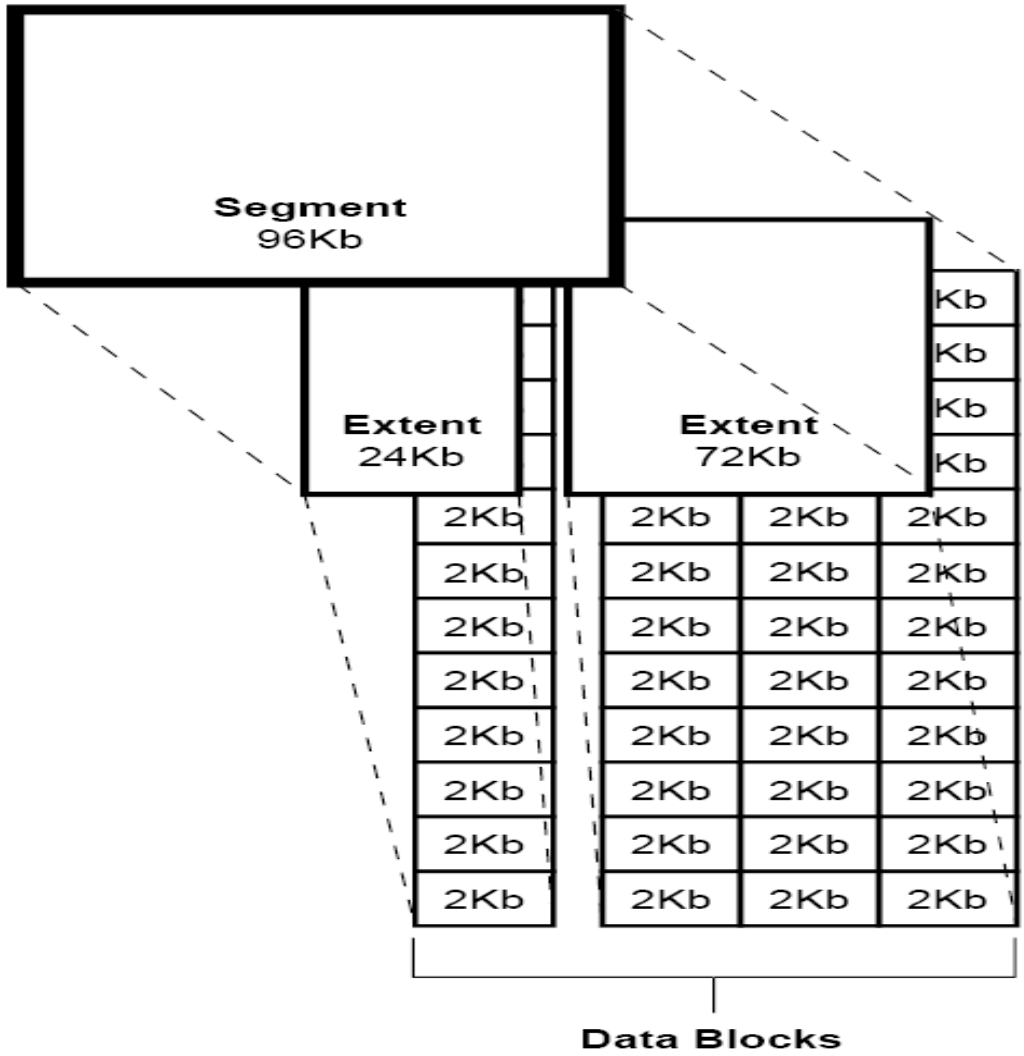
Това е най-малката единица дискова памет, разпределяна за таблица. Всички блокове на сървера са с фиксирана дължина - 2KB или 4KB в зависимост от платформата, на която работи СУБД - операционната система и хардуера.

Екстент (extent)

Екстент е непрекъсната последователност от блокове в един файл, които са разпределени за един обект в БД - таблица, индекс. Разпределянето на екстент за таблица или индекс се извършва един път при създаването и динамично при нарастване. Данните на всяка таблица (индекс) се съхраняват в един или повече екстента. С всяка таблица са свързани два параметра – размер на началния екстент и размер на следващите екстенти.

Сегмент (segment)

Сегмент е множество от екстенти, съдържащи всички данни на определен обект на БД - таблица или индекс.



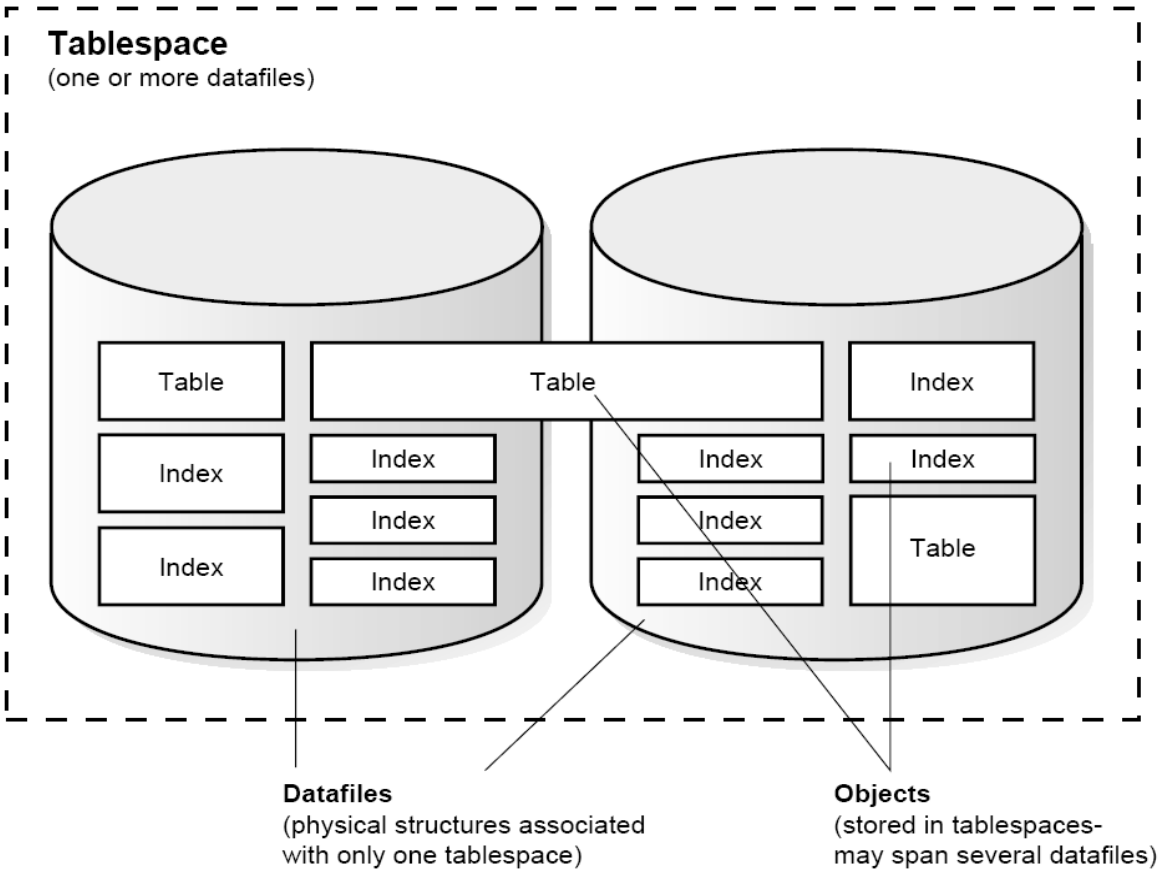
Фиг. 5. Физически понятия в Oracle.

Логически понятия

Едно от основните задължения на АБД е да управлява разполагането на данните на БД по дисковете на сървера, да следи за наличната свободна памет и да се грижи за ефективната работа на системата. Понятието, което дава тази възможност на АБД е **област** (**tablespace**). То осигурява връзка между понятията на релационния модел и физическите единици. Връзката между логическите и физически понятия е представена на Фиг 6.

Всяка област има име и включва един или повече файла. На всеки сървер има една основна област с име SYSTEM, която се създава при създаването на БД. Тогава се разпределя и първият файл за нея. След това при необходимост могат да се разпределят допълнителни файлове за основната област. Създаване на други области, добавяне на файл към област, изменение и други операции над област се извършва с SQL оператори:

```
CREATE TABLESPACE / ALTER TABLESPACE / DROP TABLESPACE
```



Фиг. 6. Логически понятия в Oracle.

Кои DDL оператори са свързани с управлението на разполагането на данните, т.е. в тях се задават параметри на физическото представяне? Това са операторите за създаване на обекти в БД, като `CREATE TABLE` и др.

Всеки потребител има област по премълчаване (`DEFAULT TABLESPACE`). Всички създавани от него таблици (индекси) по премълчаване се разполагат във файловете на тази област. В оператора `CREATE TABLE` има незадължителни фрази, в които се задават параметри на физическото представяне.

```
CREATE TABLE име-на-таблица (  ...  )
[TABLESPACE име-на-tablespace]
[STORAGE(
  [INITIAL първи {K|M}]
  [NEXT следващ {K|M}]
  [MINEXTENTS n] [MAXEXTENTS {m|UNLIMITED}]
  [PCTINCREASE процент]
 )];
```

Фразата `TABLESPACE` определя областта, в която се разполага създаваната таблица. Ако не е указана, се подразбира областта по премълчаване на потребителя. Това означава, че една таблица се свързва с една област, но различните екстенти на таблицата могат да са в различни файлове на областта. Във фразата `STORAGE` могат да се зададат:

- размер на началния екстент - `INITIAL` *първи*
- размер на следващ екстент - `NEXT` *следващ*
- максимален брой екстенти - `MAXEXTENTS`
- минимален брой екстенти - `MINEXTENTS`
- процент за нарастване на размера на следващия сегмент, спрямо предишния - `PCTINCREASE` *процент*.

Размер на екстент се задава в брой KB или MB, но трябва да е кратно на размера на блок. Зачеренията по премълчаване за *първи* и *следващ* са 5 блока, а за *процент* е 50.

5.5.2. Вътрешно представяне в Informix

Физическото представяне на БД в Infomix използва понятия аналогични на тези в Oracle, но има някои различия в реализацията им и в елементите на SQL операторите, където се задават параметри на вътрешното представяне.

Физически понятия

- **Файл (chunk)**

Това е най-голямата физическа единица дискова памет, която се разпределя от АБД за съхраняване на данни от БД. Също може да е файл на операционната система (cooked disk space) или цял неформатиран дял на диск (row disk space).

- **Страница (page)**

Това е най-малката единица дискова памет, разпределяна за таблица. Всички страници на сървера са с фиксирана дължина - 2KB или 4KB.

- **Екстент (extent)**

Екстент е непрекъсната последователност от страници в един файл, които са разпределени за една таблица. Разпределянето на екстент за таблица също се извършва един път при създаването й и динамично при нарастване на таблицата. Данните на всяка таблица се съхраняват в един или повече екстента.

- **Tblspace**

Това е един или повече екстента, съдържащи всички данни и индекси на една таблица. С всяка таблица са свързани два параметра – размер на началния екстент и размер на следващите екстенти, които могат да се задават в оператора `CREATE TABLE`, но имат и значения по премълчаване.

Логически понятия

Понятието, което осигурява връзка между понятията на релационния модел и физическите единици е **област** (**Dbspace**). Всяка област има име и включва един или повече файла. На всеки сървер има една основна област (root dbspace), която се създава след инсталиране, при инициализиране на СУБД с име определено от АБД. Тогава се разпределя и първият файл за нея. След това при необходимост могат да се разпределят допълнителни файлове за основната област. Създаването на други области, добавянето на файл към област, унищожаването на области или на някои техни файлове се извършва чрез специална сервизна програма (не с оператори на SQL, както е в Oracle).

В кои DDL оператори се задават параметри на физическото представяне?

```
CREATE DATABASE име-на-бд IN име-на-област;
```

Фразата `IN` определя областта, в която ще бъдат създадени системните таблици на създаваната БД и областта по премълчаване, където ще се създават впоследствие базовите таблици. Ако тази фраза не е зададена, БД се създава в основната област.

```
CREATE TABLE име-на-таблица (  ...  )
[IN име-на-област ]
[EXTENT SIZE първи ] [NEXT SIZE следващ ];
```

Фразата `IN` определя областта, в която се разполага създаваната таблица. Ако не е указана тук, се подразбира областта от оператора `CREATE DATABASE`. И тук всяка

една таблица се свързва с една област, но различните екстенти на таблицата могат да са в различни файлове на областта.

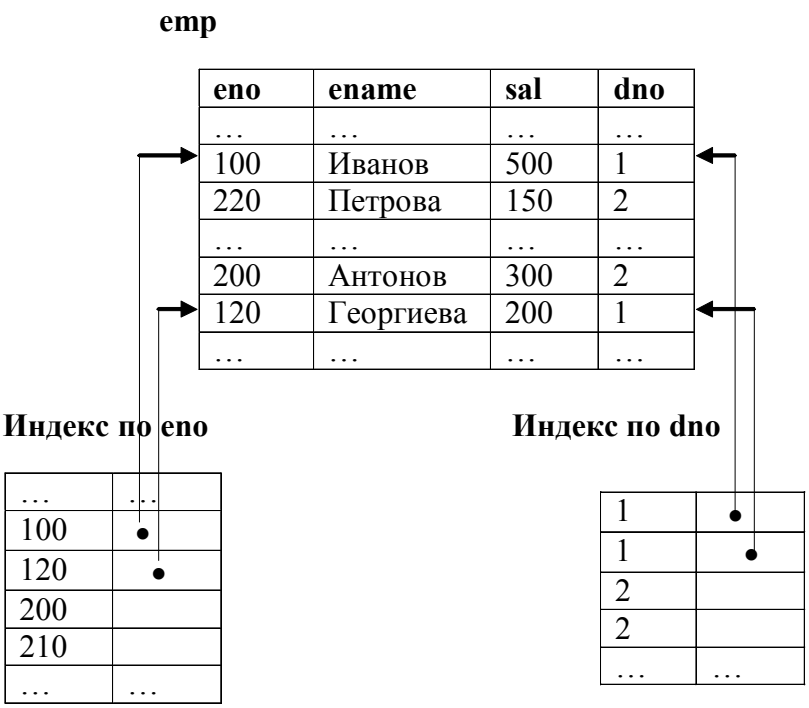
Има и друг вариант при свързване на таблица с област, когато редовете на една таблица могат да бъдат разпределени в няколко области по определени правила. Това е така нареченото фрагментиране на таблица и е полезно при много големи таблици, тъй като дава възможност за паралелен достъп до данните в таблицата. Вместо фразата `IN` е зададена фраза `FRAGMENT`, например:

```
CREATE TABLE emp (  ...  )
FRAGMENT BY EXPRESION
      eno < 1000 IN dbspace1,
      eno >= 1000 AND eno < 2000 IN dbspace2,
      eno >= 2000 IN dbspace3;
```

Фразата `EXTENT SIZE` определя размера на началния екстент, който се разпределя по време на изпълнение на оператора `CREATE TABLE`. Фразата `NEXT SIZE` определя размера на следващите екстенти, които се разпределят по необходимост. Размерите се задават в брой КВ, но трябва да са кратно на размера на страница. По премълчаване и двете значенията са 8 страници.

5.5.3. Индекси

Индекс е обект в БД, осигуряващ бърз достъп до редовете на базова таблица по значенията на определена колона(и) и е свързан с физическото представяне на таблица. На Фиг.7 е изобразена таблицата `emp` и два създадени за нея индекса. Единият осигурява достъп по значенията на първичния ключ `eno`, а другият по значенията на колоната `dno`. Всеки ред в индекса съдържа значение на колоната, по която е създаден и адрес на ред в базовата таблица, съдържащ това значение. Колоната (колоните) на базовата таблица, по която е създаден индекса ще наричаме **ключ на индекса**. Редовете в индекса са разположени по нарастващи или намаляващи значения на ключа.



Фиг. 7. Индекси върху таблица emp

Наличието или отсъствието на индекс е незабележимо за потребителя и приложната програма. В SQL операторите за манипулиране на данни не се указва използването на индекси. При изпълнение на оператор СУБД (оптимизаторът на заявките) решава дали да използва или не индекс когато той съществува. Напр., при изпълнение на `SELECT` оператора:

```
SELECT  eno, ename, sal FROM emp
WHERE dno = 1 and sal < 300;
```

Ако не съществува индекс по `dno`, то последователно ще се обходят всички редове на таблицата, за да се извлече необходимата информация. Ако съществува индекс по `dno`, то системата може първо да търси в индекса значението на номер на отдел 1, и след това чрез адресите в намерените там записи да прочете директно редовете в таблицата `emp`.

Търсенето в индекса е по-бързо, тъй като записите в него са сортирани по значението на ключа и са по-къси от редовете в базовата таблица. Много СУБД, включително Oracle и Informix, използват структурата B+ tree при физическото представяне на индекса, което още повече ускорява търсенето в индекса. Преходът от запис в индекса към ред на базовата таблица също е бърз, тъй като се осъществява по адрес на ред в някаква форма (напр., адрес на блок и отместване в блока). Ускоряването на изпълнението на оператори `SELECT` при наличието на индекси има своята цена. Всеки индекс заема допълнителна дискова памет. Обновяването на данните в базовата таблица изисква изпълнение на съответна операция и върху индекса, тъй като съдържанието на индекса трябва да съответства на съдържанието на таблицата.

За всяка таблица динамично могат да се създават и унищожават индекси чрез оператори на SQL без това да влияе на операторите за манипулиране на данни, като изключим времето за изпълнение. Това е пример за физическа независимост на данните.

```
CREATE [UNIQUE] INDEX име-на-индекс
ON име-на-таблица ( име-на-колона [ASC|DESC] [, …] )
[ параметри-на-съхранение ];
```

Чрез оператора `CREATE INDEX` се създава нов индекс с име *име-на-индекс* за базовата таблица *име-на-таблица*. След името на таблицата се задава ключа на индекса, т.е. списък от колони на базовата таблица и указание за наредбата на значенията в индекса (нарастващи или намаляващи). Всяка СУБД налага ограничения за максималната дължина на ключа на индекса (напр., в Oracle е до 32 колони, в Informix е до 16 колони с максимална обща дължина 255 байта).

Ако е зададена ключовата дума `UNIQUE`, то се създава уникален индекс, т.е. индекс, забраняващ еднакви значения на ключа на индекса. В този случай данните в таблицата трябва да не нарушават това изискване и в момента на създаване на индекса, в противен случай операторът завършва с грешка.

Повечето СУБД автоматично създават индекси, когато се задават някои ограничения за цялостност в оператора `CREATE TABLE` или `ALTER TABLE`:

- уникален индекс за всеки първичен ключ и колони обявени за `UNIQUE`;
- неуникален индекс за всеки външен ключ (в Informix).

Индексите могат динамично да се унищожават чрез оператор `DROP INDEX` без това да повлияе на базовата таблица.

```
DROP INDEX име-на-индекс;
```

Препоръки при използване на индекси

Системата не ограничава броя на индексите, но все пак те трябва да се създават само когато ще е полезно, а именно:

- По колони, по които често се търси и сортира.
- По колони, които се използват при съединение.
- Да не се създава индекс по колони, които имат само няколко различни значения.
- За определен набор от колони на таблица и в определена наредба (`ASC` или `DESC`) може да има само един индекс.
- Неуникалните индекси е по-добре да се създадат след въвеждане на данните в таблицата, тъй като:
  - При обновяване на таблицата ще се поддържат по-малко индекси.
  - Новосъзданеният индекс осигурява по-ефективен достъп.
- Ако обновяванията се правят периодично (напр., един път месечно) по-ефективно ще е да се следва процедурата:
  - `DROP INDEX` за неуникалните индекси.
  - Обновяване на базовите таблици.
  - Отново `CREATE INDEX` за унищожените в началото индекси.

5.5.4. Системни таблици

Системните таблици са специални таблици, които се създават и поддържат автоматично от СУБД. В тях се съхраняват не потребителски данни, а метаданни – данни за всички обекти в БД. При изпълнение на SQL операторите СУБД непрекъснато осъществява достъп до системните таблици. Макар, че те са за вътрешна употреба, потребителят може да получи достъп до тях чрез `SELECT` оператор и така да получи информация за структурата на БД. В различните СУБД се реализират системни таблици, които имат различна структура, наименования и дори броят им е различен.

Всяка системна таблица съхранява информация за определен вид обекти в БД, като основните видове, поддържани от всички системи са за:

- **Таблици**. За всяка таблица се съхранява: име, собственик, тип на таблицата, дата на създаване, брой колони и др.
- **Колони**. Всяка колона се описва с: име, към коя таблица принадлежи, тип на данните, размер, разрешено ли е неопределено значение и др.
- **Виртуални таблици**. Всяка виртуална таблица може да е описана като таблица, но освен това за нея трябва да се съхранява текста на `SELECT` оператора, чрез който се изчислява тя.
- **Потребители**. Всеки потребител регистриран в СУБД се описва с име, парола за достъп и др.
- **Права**. Всяко право дадено на потребител се описва с: кой е дал привилегията, на кого е дадена, самата привилегия и обекта, за когото се отнася. (Правата се разглеждат в раздела сигурност на данните.)

Основни системни таблици в Oracle

Броят на системните таблици в Oracle е доста голям и зависи от версията. Следва списък на няколко от основните системни таблици и имената на част от колоните им. Всички те са собственост на потребител с име SYS, затова са дадени с пълните си имена.

SYS.ALL_TABLES (Owner, Table_name, Tablespace_name, ...)

SYS.ALL_TAB_COLS (Owner, Table_name, Column_name, Data_type, Data_length, Nullable, ...)

SYS.ALL_VIEWS (Owner, View_name, Text, ...)

SYS.ALL_CONSTRAINTS (Owner, Constraint_name, Constraint_type, Table_name, Search_condition, R_owner, R_constraint_name, Delete_rule, ...)

SYS.ALL_INDEXES (Owner, Index_name, Index_type, Table_owner, Table_name, Uniqueness, ...)

SYS.ALL_SYNONYMS

SYS.ALL_SEQUENCE

SYS.USER_UPDATABLE_COLUMNS

6. Защита на данните

6.1. Сигурност на данните

Под понятието сигурност да данните се разбира защита на данните от неправомерен достъп, изменение и унищожаване. Следователно СУБД трябва да разрешава изпълнението на операция над БД само ако потребителят има право. Това означава, че сигурността се реализира от СУБД на базата на изисквания формулирани от АБД на езика SQL.

Принципи при осигуряване на сигурност на данните

Кои са основните компоненти в системата за защита на данните в БД?

Потребители. Когато СУБД изпълнява операция над БД тя винаги е от името на определен потребител. Следователно, потребителите са действащите лица в БД или субектите в системата за защита.

Обекти. Елементите, достъпът до които ще се контролира, са обектите в БД. Преди всичко това са таблици – базови и виртуални, и евентуално други типове, като индекси, съхранени процедури.

Привилегии. Правото на потребител да извършва определено действие над обект се нарича привилегия. Кои са привилегиите според стандартите? Обикновено реализираните типове привилегии са: select, insert, update, delete, което означава правото за изпълнение на съответния SQL оператор над таблицата. Има и една важна привилегия – привилегията собственост. Тя е важна защото дава пълни права на потребителя, включително и правото да унищожи обекта, и е първата привилегия, която се дава за обект.

6.1.1. Идентификация на потребители

За да има система на сигурност е необходимо потребителите да бъдат различавани, т.е. да имат имена или идентификатори. В стандартите се използва термина идентификатор на права на достъп (authorization-id), но повечето СУБД използват понятието идентификатор на потребител (user-id). Някои системи използват собствено пространство от имена на потребители, като Oracle. Други като Informix, Ingres, Postgres използват имената на потребителите, регистрирани в операционната система (ОС).

Идентификацията на потребител за достъп до БД включва обикновено същите стъпки, както във всяка многопотребителска операционна система. Най напред потребителят задава името си, а след това и паролата за да докаже, че е този за когото се представя. Кога се извършва идентификацията на потребител? При интерактивен SQL обикновено това става в началото на сесията.

Например в Oracle стартирането на програмата за интерактивна работа с SQL може да изглежда така:

```
sqlplus scott/tiger
```

Като аргументи се задават името и паролата на потребителя. В Oracle има още два оператора, които могат да се изпълняват след като е стартирана програмата `sqlplus`, а именно:

```
disconnect
connect scott/tiger
```

Чрез `disconnect` се завършва текущата сесия, а с `connect` се започва нова сесия.

В Informix, Ingres и Postgres се разчита на текущия потребителски идентификатор от ОС, т.е. предполага се, че потребителят, който в момента работи в

ОС започва и сесия с БД. Затова там при свързване с БД не е необходимо да се задава име и парола. SQL операторите за това са:

```
DATABASE  име-на-бд;
CLOSE DATABASE;
```

Но е възможно в рамките на една сесия на ОС да се започне сесия с БД от името на друг потребител. Има и оператори:

```
CONNECT TO 'име-на-бд' USER 'име-на-потребител';
DISCONNECT CURRENT;
```

Създаване на потребител

Създаване на потребител, определяне на начина за неговата идентификация и други елементи от профила му, в Oracle се извършва с SQL оператори. Нов потребител, на който да бъде дадено някакво право на достъп до БД, се създава с оператора:

```
CREATE USER име-на-потребител
      IDENTIFIED {BY парола|EXTERNALLY}
      [DEFAULT TABLESPACE име-tablespace];
```

Фразата `IDENTIFIED` определя начина за идентификация и паролата му. Фразата `DEFAULT TABLESPACE` определя областта по премълчаване за таблиците, които той ще създава. Така създадения потребител първоначално няма никакви права. Такива трябва да му бъдат предоставени след това с оператор `GRANT`. Всеки регистриран потребител може да промени паролата си с оператор:

```
ALTER USER име-на-потребител
      [IDENTIFIED {BY парола|EXTERNALLY}]
      [DEFAULT TABLESPACE име-tablespace];
```

Унищожаване на потребител заедно с всички, създадени от него обекти в БД се извършва с оператор:

```
DROP USER име-на-потребител [CASCADE];
```

6.1.2. Категории потребители

Всеки потребител, който има право на достъп до БД, принадлежи към една от трите категории – `CONNECT, RESOURCE` или `DBA`, което определя неговите права на ниво цялата БД. Какви са привилегиите на всяка от категориите? Точният списък от привилегии за всяка категория, може да е различен в различните СУБД, но идеята е следната:

`CONNECT`

- Да се свързва (да създава сесия) с БД.
- Да изпълнява операторите `SELECT, INSERT, UPDATE, DELETE` над таблици, за които са му дадени тези привилегии.

В Informix освен това може:

- Да създава временни таблици, индекси над тях и да ги унищожава.
- Да създава виртуални таблици над таблици, за които има привилегия `SELECT`.
- Да създава синоними.

`RESOURCE`

- Да създава обекти в БД.
- Да създава нови базови таблици, индекси над тях, да ги унищожава и да предоставя привилегии за тях на други потребители.
- Да изменя структурата на таблици, за които има тази привилегия.
- Да създава виртуални таблици, тригери и др.

DBA

- Да създава и унижожава всякакви обекти в БД.
- Да създава потребители.
- Да предоставя и отнема привилегии на потребители.
- Да унищожи БД (ако има SQL оператор за това).

Следващият въпрос е как на потребител се дава право на достъп до БД в определена категория. В Oracle при създаване на БД автоматично се създава първият DBA потребител с име system и парола manager. Първоначално само той има право на достъп до БД. Той може да регистрира други потребители и да им даде привилегия на ниво БД.

В Informix категориите потребители са същите, но е различно правилото, по което се определя първият DBA потребител. Потребителят, който създаде БД става неин собственик и е първият потребител с привилегия DBA. Той след това може да регистрира други потребители от различни категории.

Операторите, с които само потребител DBA предоставя и отнема привилегии на ниво БД са:

```
GRANT {CONNECT|RESOURCE|DBA} [,...]
      TO {PUBLIC|име-на-потребител} [,...]
      [IDENTIFIED BY парола];

REVOKE {CONNECT|RESOURCE|DBA} [,...]
      FROM {PUBLIC|име-на-потребител}[,...];
```

Има една специална ключова дума PUBLIC вместо име на потребител, която означава всички потребители в момента и в бъдеще, т.е. всички потребители, които са регистрирани в системата в момента на проверка на привилегиите.

В Informix организацията на привилегиите е йерархична, т.е. RESOURCE включва всички привилегии на CONNECT, а DBA всички привилегии на RESOURCE. Поради това, ако се предостави право DBA на потребител, не е необходимо да му се дават и права CONNECT и RESOURCE. Ако се отнеме CONNECT или RESOURCE от DBA това няма никакъв ефект. Ако се отнеме RESOURCE или DBA от потребител, той се понижава до CONNECT. За да се отнеме всяко право на достъп до БД от потребител, трябва да му се отнеме по-високата привилегия, ако има такава, след което да му се отнеме и CONNECT. В Oracle това не е така, отнемене на всяко право на достъп до БД от потребител става като му се отнемат всичките права или с оператор DROP USER.

6.1.3. Привилегии на ниво таблици

Потребителят, създал таблица, става неин собственик. Привилегията собственост не може да се отнема или да се предава на друг потребител. Според ANSI стандарта, а и в Oracle, първоначално само той има пълни права над таблицата, включително и да я унищожи. Само собственикът или DBA потребител може да дава на други потребители привилегии. Какви са привилегиите за таблици?

```
SELECT [(списък-имена-на-колони)]
UPDATE[(списък-имена-на-колони)]
INSERT
DELETE
INDEX
ALTER
REFERENCES[(списък-имена-на-колони)]
ALL [PRIVILEGES]
```

В Informix има възможност БД да се създаде не по ANSI стандарта. Тогава при създаване на таблицата системата автоматично предоставя на всички потребители (PUBLIC) всички привилегии без ALTER и REFERENCES.

Предоставяне на привилегии за таблица

Операторът, с който собственикът на таблица или DBA потребител може да предостави на други потребители привилегии е:

```
GRANT списък-от-привилегии ON име-на-таблица
TO {PUBLIC|име-на-потребител} [,...]
[WITH GRANT OPTION];
```

Таблицата *име-на-таблица* може да е базова, виртуална таблица или синоним. Фразата WITH GRANT OPTION означава, че потребителя получава указаните привилегии заедно с правото да ги предава на други потребители.

Привилегиите ALTER и INDEX могат да се дават само на RESOURCE потребител, тъй като предимство имат по-ограничителните привилегии.

Отнемане на привилегии за таблица

Динамично в процеса на работа с БД привилегиите могат както да се предоставят, така и да се отнемат. Собственикът на таблица или DBA потребител може да отнеме всички или част от привилегиите, дадени на потребител с оператора:

```
REVOKE списък-от-привилегии ON име-на-таблица
FROM {PUBLIC|име-на-потребител} [,...];
```

Ако са дадени привилегии на PUBLIC, то се отнемат от PUBLIC. Ако са дадени SELECT(*списък-имена-на-колони*) или UPDATE(*списък-имена-на-колони*) не могат да се отнемат правата за част от колоните. Ако се отнеме привилегия от потребител, предоставена му с WITH GRANT OPTION, то автоматично се отнемат и всички разпространени от него.

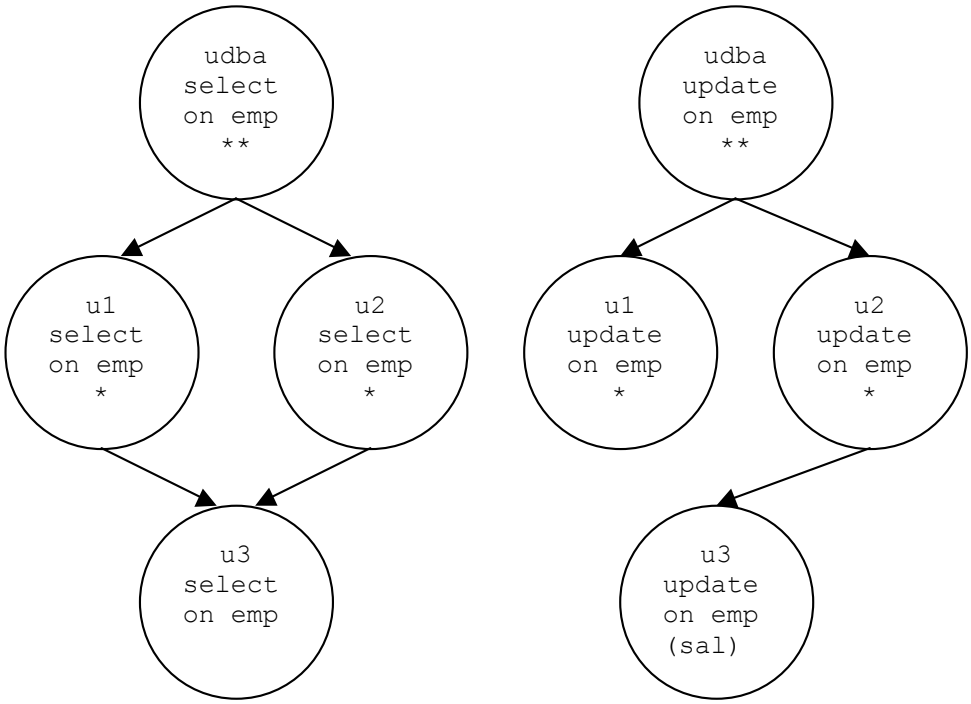
Диаграма на привилегиите

В резултат на изпълнение на операторите GRANT и REVOKE може да се получи сложна система от частично препокриващи се привилегии. За по-доброто разбиране е полезно да се представи процеса на предоставяне и отнемане на привилегии чрез диаграма. Тази диаграма е ориентиран граф, в който всеки връх представя комбинацията "потребител/привилегия". Ребро от връх "U/P" към връх "V/Q" означава, че потребител U е предоставил на потребител V привилегия Q, която се базира на привилегията P. Символите ** представят привилегията собственост. Един символ * означава правото да се предава привилегията.

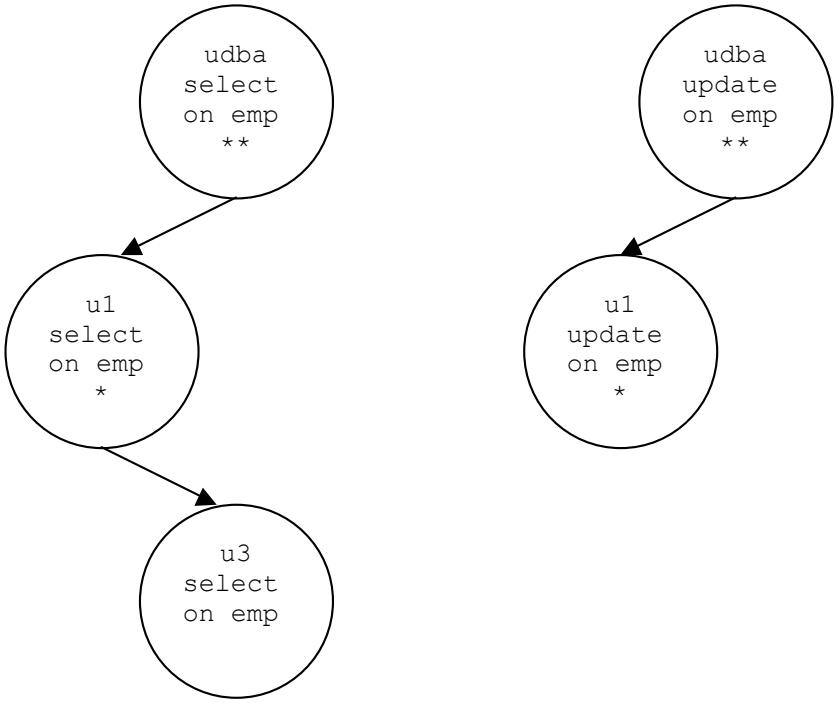
♦ Нека таблицата emp е собственост на udba, който е DBA, а u1, u2 и u3 са потребители, регистрирани в БД като CONNECT. Ако са изпълнени следните оператори в тази последователност.

Стъпка	Потребител	Действие
1	udba	GRANT SELECT, UPDATE ON emp TO u1, u2 WITH GRANT OPTION;
2	u1	GRANT SELECT ON emp TO u3;
3	u2	GRANT SELECT, UPDATE(sal) ON emp TO u3;
4	udba	REVOKE SELECT, UPDATE ON emp FROM u2;

Диаграмата на привилегиите след изпълнение на стъпки 1, 2 и 3 е изобразена на Фиг. 8, а след стъпка 4 на Фиг. 9.



Фиг. 8. Диаграма на привилегиите след стъпки 1-3.



Фиг. 9. Диаграма на привилегиите след стъпка 4.

Виртуални таблици и осигуряване на сигурност

Виртуалните таблици могат да се използват за да се ограничи достъпа на потребител до част от данните в таблица или само до производни данни, но не и до детайлните данни.

♦ На потребител `u1` да се даде достъп само до данните за отдел 1. Потребителят `udba`, който е собственик на таблицата `emp`, създава виртуална таблица, съдържаща

пълните данни за служителите от отдел с номер 1. След това предоставя привилегии на `u1` само към виртуалната таблица.

```
CREATE VIEW vemp1
AS SELECT * FROM emp WHERE dno = 1
WITH CHECK OPTION;
GRANT SELECT, INSERT, UPDATE, DELETE ON vemp1 TO u1;
```

Чрез виртуална таблица потребител не може да получи неконтролирано допълнителни привилегии. Потребител, който създава виртуална таблица трябва да има `SELECT` права за всички колони от всички таблици, участващи в дефиницията. Той става неин собственик и получава право `SELECT`. Ако потребителят има права `INSERT`, `UPDATE`, `DELETE` над базовите таблици, участващи в дефиницията на виртуалната таблица, и ако тя е обновяема, то той автоматично получава съответните права и над нея. Ако привилегиите за базовите таблици са му дадени `WITH GRANT OPTION` или е техен собственик, то може да предава правата над виртуалната таблица.

6.1.4. Роли

В по-новите версии на СУБД се реализира едно ново понятие, свързано със сигурността на данните, понятието роля. Ролята представлява колекция от привилегии. SQL операторите с които се създава, унищожава или се изменя роля са следните.

```
CREATE ROLE име-на-роля
[NOT IDENTIFIED|IDENTIFIED BY парола];

DROP ROLE име-на-роля;

ALTER ROLE име-на-роля
{NOT IDENTIFIED|IDENTIFIED BY парола};
```

След създаването на нова роля, тя е празна, т.е. не включва никакви привилегии. Напълването на ролята с привилегии става с оператор `GRANT`. Така операторите `GRANT` и `REVOKE`, с които се предоставят или отнемат привилегии имат по-общ вид, който в Oracle е следния:

```
GRANT списък-от-привилегии ON име-на-таблица
TO {име-на-потребител|име-на-роля|PUBLIC}[,...]
[WITH GRANT OPTION];

REVOKE списък-от-привилегии ON име-на-таблица
FROM {име-на-потребител|име-на-роля|PUBLIC}[,...];
```

В Oracle се реализирани и така наречените системни привилегии, например:

CREATE TABLE - привилегия да създава таблици,
ALTER ANY TABLE - привилегия да изменя структурата таблици,
DROP ANY TABLE - привилегия да унищожава таблици,
CREATE VIEW - привилегия да се създават виртуални таблици,
DROP ANY VIEW - - привилегия да унищожава виртуални таблици,
CREATE USER - привилегия да се създават потребители,
CREATE ROLE - привилегия да се създават роли
CREATE ANY OBJECT - привилегия да се създават всякакви типове обекти и т.н.

Това се отразява на операторите `GRANT` и `REVOKE`, а именно те имат и следния вид:

```
GRANT {системна-привилегия|име-на-роля}[,...]
TO {име-на-потребител|име-на-роля|PUBLIC}[,...]
[WITH ADMIN OPTION];
```

```
REVOKE {системна-привилегия|име-на-роля} [,...]
FROM {име-на-потребител|име-на-роля|PUBLIC}[,...];
```

Фразата `WITH ADMIN OPTION` означава, че потребителя получава указаните роли или системни привилегии заедно с правото да ги администрира, а именно да ги предава или отнема, да унищожава ролята, да изменя привилегиите, включени в нея.

Информация за потребителите, правата на достъп до обектите в БД и ролите се съхранява в системни таблици. В Oracle това са таблиците:

SYS.ALL_USERS - Всички потребители на БД
SYS.DBA_ROLES - Всички роли в БД
SYS.DBA_SYS_PRIVS - Системните привилегиите дадени на потребители и роли, включително на DBA, RESOURCE, CONNECT
SYS.USER_SYS_PRIVS - Системните привилегиите на потребителя
SYS.DBA_TAB_PRIVS - Всички привилегии, дадени за обекти в БД
SYS.USER_TAB_PRIVS - Привилегии за обекти, дадени на потребителя

6.2. Цялостност на данните

Понятието цялостност на данните означава защита на данните от неправилно изменение и унищожаване. Целта е осигуряване на правдоподобни (ако може и правилни), непротиворечиви данните в БД.

Причини за нарушаване целостността на данните

- Апаратни грешки
- Програмни грешки в ОС, СУБД, приложната програма
- Външни причини – токов удар, земетресение, наводнение
- Грешки на потребителя
- Конкурентен достъп до данните – използване на едни и същи данни по едно и също време от много потребители.

Тъй като причините, които могат да доведат до нарушаване на цялостността, са най-различни, то и механизмите, реализирани в СУБД са различни. Един от механизмите, който се реализира във всяка СУБД, са така наречените ограничения за цялостност. Това са условия, формулирани от АБД на езика SQL, на които трябва да отговарят данните за да се считат за правилни.

6.2.1. Ограничения за цялостност

Ще разгледаме видовете ограничения за цялостност и как те се поддържат в DDL операторите на езика SQL и от СУБД. В повечето случаи, ако ограничението се поддържа, то това е в операторите `CREATE TABLE/ALTER TABLE` или `CREATE INDEX`.

Ограничения за значението на колона

- **Тип на данните**
  Всички значения в една колона са от определен тип на данните.
- **Определеност (задължителност) на значенията** на колона – **`NOT NULL`**
  Забранява редове, в които колоната е със значение `NULL`.
- **Условие за проверка** върху значенията на колона

Изброяват се допустимите значения в колоната или се задава интервал(и) на допустимите значения, или шаблон, на който да отговарят значенията. Този вид ограничения могат да се изразят чрез фразата `CHECK`, която е включена в стандартите и се реализира в повечето СУБД.

♦ Колоната `spec` в таблица `student` може да приема само значенията Math, PM, Inf, Math&Inf.

```
CREATE TABLE student (...,
   spec CHAR(10) CHECK ( spec IN ('Math','PM','Inf','Inf&Math')),
   ...);
```

♦ Заплатата на всеки служител трябва да е по-голяма или равна на 0.

```
CREATE TABLE emp (...,
   sal NUMBER(6,2) CHECK ( sal >= 0 ),...);
```

♦ Колоната `ocenka` в `s_p` може да приема значения в затворения интервал от 2 до 6 или `NULL`.

```
CREATE TABLE s_p (...,
   ocenka NUMBER(4,2) CHECK((ocenka BETWEEN 2 AND 6) OR ocenka IS NULL),
   ...);
```

Ограниченията от тези видове се проверяват при изпълнението всеки оператор `insert` или `update`, когато колоната получава ново значение.

Ограничение за възможен ключ - `UNIQUE`

Това е ограничение за уникалност на значенията на колона(и) в редовете на една таблица, което забранява съществуването на няколко реда в таблицата с еднакви значения в тези колони. Всяка таблица може да има няколко възможни ключа.

Ограничението се проверява при изпълнението всеки оператор `INSERT` или `UPDATE`, изменящ значението на възможния ключ. СУБД трябва да провери дали в таблицата не съществува друг ред със същото (новото) значение на възможния ключ. Тази проверка значително ще се ускори, ако съществува индекс по възможния ключ, тъй като търсенето ще се извърши в индекса, а не в таблицата. Следователно индексът има важно значение при прилагането на това ограничение, затова някои СУБД автоматично създават индекс за всеки възможен ключ.

Този вид ограничение се задава в оператора `CREATE TABLE/ALTER TABLE`, но в някои СУБД може и чрез оператора `CREATE UNIQUE INDEX`.

Ограничение за първичен ключ – `PRIMARY KEY` (Entity integrity)

Всяка таблица може да има само един първичен ключ. Ограниченията за първичен и възможен ключ са идентични (според теорията първичният ключ е възможен ключ, който е избран), но някои СУБД им дават леко различен смисъл. Напр., в Oracle и Informix, за първичния ключ се предполага и ограничение `NOT NULL`, а за възможния не се предполага това ограничение. Но и за двата вида ключа се създава автоматично индекс.

Ограничение за външен ключ – `FOREIGN KEY` (Referential integrity)

Всеки външен ключ в една таблица е свързан с първичен (възможен) ключ на друга таблица, на който съответства. Този вид ограничение засяга данните в две таблици, затова реализацията му е по-сложна. При прилагането на това ограничение от СУБД има три варианта на действие или три правила.

- **ограничително правило** (`RESTRICT`)

Това е правилото по премълчаване. Отхвърля се обновяването на БД, ако то нарушава ограничението. Това е единственото правило, което се прилага при оператор `INSERT` в таблицата с външния ключ и при `UPDATE` на външния ключ. Може да се прилага при `DELETE` за таблицата с първичния ключ или `UPDATE` на първичния ключ.

- **каскадно правило** (`CASCADE`)

Изпълнява се исканото обновяване на БД и ако се наруши ограничението, съответният обновяващ оператор се прилага и върху таблицата, съдържаща външния

ключ. Може да се прилага при DELETE за таблицата с първичния ключ или UPDATE на първичния ключ.

- **правило с присвояване на неопределено значение** (SET NULL)

Изпълнява се исканото обновяване на БД и ако се наруши ограничението, на външния ключ в съответните редове се присвоява NULL. Може да се прилага при DELETE за таблицата с първичния ключ или UPDATE на първичния ключ.

Последните две правила могат да се прилагат независимо по отношение на операторите DELETE и UPDATE.

♦ При изтриване на отдел от dep, за служителите от изтрития отдел в emp да се присвои NULL на dno. При изменение на номера на отдел dno в dep, да се измени по същия начин и външния ключ dno в emp.

```
CREATE TABLE emp (..., dno NUMBER(4),
   FOREIGN KEY (dno) REFERENCES dep(dno)
                     ON DELETE SET NULL
                     ON UPDATE CASCADE );
```

Условия върху значенията на колони в един ред

Това е условие, което може да бъде проверено върху значенията в един ред на таблица, разглеждайки го независимо. Този вид ограничения могат да се изразят чрез фразата CHECK.

♦ Хорариумът на всеки предмет не може да е 0, т.е. сумата от брой лекции и брой упражнения трябва да е по-голяма от 0.

```
CREATE TABLE predmet (...,
   CHECK ( br_l + br_u > 0 ), ...);
```

♦ Ако специалността на студент е PM, то първата цифра от фак. номер да е 3, ако специалността на студент е Math, то първата цифра от фак. номер да е 1, и.т.н.

```
CREATE TABLE student (fn CHAR(6),..., spec CHAR(10), ...,
   CHECK ((spec='PM' AND fn LIKE '3%') OR
          ((spec='Math' OR spec='Math&Inf') AND fn LIKE '1%') OR
          (spec='Inf' AND fn LIKE '4%') OR
          (spec='KN' AND fn LIKE '8%')) , ...);
```

Статистически ограничения

Това са условия върху значенията в цяла таблица или в няколко таблици. В стандарта SQL2 в ограничение CHECK може да има сложни условия с вложен оператор SELECT.

♦ Средната заплата да е по-малка от 1000.

```
CREATE TABLE emp (...,
   CHECK ( 1000 > (SELECT AVG(sal) FROM emp)),...);
```

♦ Сумата от заплатите на служителите във всеки отдел да е по-малка от бюджета на отдела.

```
CREATE TABLE emp (...,
   CHECK (dep.budjet > (SELECT SUM(sal) FROM emp WHERE dno=dep.dno)),
   ...);
```

Динамични ограничения

Това са условия, които сравняват старото и новото съдържание на БД. Този вид ограничения не могат да се изразят директно в оператор CREATE TABLE.

♦ Новата заплата на служител не може да е два пъти по-голяма от старата му заплата.

Ограниченията за цялостност се прилагат при изменение на обектите в БД, към които се отнасят, т.е. проверката им се активизира при настъпване на определени събития в БД. Затова ги наричат **активни елементи** в БД. Напр., ограничението за колона `CHECK` се проверява при изменение на колоната в някой ред (при изпълнение на оператори `UPDATE` или `INSERT`). В релационните БД има и други видове активни елементи - тригери.

6.2.2. Тригери

През 1986 в Sybase е въведено понятието тригер (trigger), което представлява следваща стъпка напред към включването на по-общи ограничения (бизнес правила) в БД. Тригерите са включени и в стандарта SQL3 и се реализират в много СУБД. Тригерите са активни елементи от типа "събитие-условие-действие" (ECA).

1. Тригер се прилага при настъпване на **събитие** (trigger event), определено от програмиста.

2. Тогава се проверява зададено **условие** и ако то не е изпълнено, не се изпълнява никакво действие.

3. Ако условието е истина, СУБД изпълнява **действието** (trigger action), свързано с тригера.

Следващите примери са с използване на тригери в Oracle. Там събитията, с които може да бъде свързан един тригер, т.е които ще го активизират, са изпълнение на един от следните SQL оператори над определена таблица:

```
INSERT ON име-на-таблица
DELETE ON име-на-таблица
UPDATE [OF име-на-колона [,...]] ON име-на-таблица
```

Действието може да се изпълни:

`BEFORE` - преди изпълнение на оператора, активизирал тригера;

`AFTER` - след завършване на оператора, активизирал тригера;

`INSTEAD OF` - вместо оператора (само за виртуална таблица).

В допълнение може да се укаже, действието да се изпълнява за всеки ред на таблицата (row trigger) - `FOR EACH ROW`.

Действието е PL/SQL блок или извикване на съхранена процедура, може да включва условие (фраза `WHEN`) и има вида:

```
[WHEN (условие)] pl/sql-блок
```

Ако действието се изпълнява `FOR EACH ROW`, то в него може да използват както старите, така и новите значения в редовете, изменяни от оператора, активизирал тригера (фраза `REFERENCING`).

Следва общия вид на операторите `CREATE/DROP/ALTER TRIGGER` в Oracle.

```
CREATE [OR REPLACE] TRIGGER име-на-тригер
{BEFORE | AFTER | INSTEAD OF}
{DELETE | INSERT | UPDATE [OF име-на-колона [,...]]}
      [ OR ... ] ON име-на-таблица
[[REFERENCING {OLD AS old|NEW AS new}...] FOR EACH ROW
[WHEN (условие)] ] pl/sql-блок ;

DROP TRIGGER име-на-тригер;

ALTER TRIGGER име-на-тригер {ENABLE | DISABLE|COMPILE};
```

За сравнение в Informix един тригер може да има действие `BEFORE`, `FOR EACH ROW` и `AFTER`, но в този ред. Всяка такава част от действието може да включва условие. Операторът `CREATE TRIGGER` има следния вид:

```
CREATE TRIGGER име-на-тригер
{DELETE | INSERT | UPDATE [OF име-на-колона [,...]]}
      ON име-таблица
[REFERENCING OLD AS pre NEW AS post]
{BEFORE | FOR EACH ROW | AFTER}
[WHEN (условие)] (sql-оператори)
[...];
```

♦ При добавяне на нов служител (въвеждане на ред в таблица `emp`) да се увеличи бюджета на съответния отдел със заплатата му.

Решение с тригер в Oracle:

```
CREATE TRIGGER new_emp
AFTER
INSERT ON emp
FOR EACH ROW WHEN(new.sal IS NOT null)
BEGIN
   UPDATE dep
   SET budget = budget + :new.sal
   WHERE dno = :new.dno;
END;
```

Решение с тригер в Informix:

```
CREATE TRIGGER tr1
INSERT ON emp
REFERENCING NEW AS post
FOR EACH ROW
   (UPDATE dep SET budjet=budjet+post.sal WHERE dno=post.dno);
```

При БД с транзакции проверката на всички ограничения за цялостност (зададени в `CREATE TABLE`) се отлага след след завършване на действието на тригера. Това позволява използването на тригери за поддържане на различните правила за външен ключ (ако те не се поддържат в `CREATE TABLE`) или за поддържане на по-общи бизнес правила.

♦ При изтриване на служител, за външния ключ `emp_pro.eno` да се реализира каскадното правило, а за външния ключ `dep.mgr` да се реализира правило `SET NULL`. Решение с тригер в Oracle:

```
CREATE TRIGGER del_emp
BEFORE DELETE ON emp
FOR EACH ROW
BEGIN
   DELETE FROM emp_pro
      WHERE eno = :old.eno;
   UPDATE dep SET mgr = NULL
      WHERE mgr = :old.eno;
END;
```

♦ За външния ключ `dno` в таблица `emp` да се реализира каскадно правило при `UPDATE` и правило `SET NULL` при `DELETE`. Решение с тригери в Informix:

```
CREATE TABLE emp (...,
   FOREIGN KEY (dno) REFERENCES dep(dno));

CREATE TRIGGER del_dep
DELETE ON dep
REFERENCING OLD AS pre_del
```

```
FOR EACH ROW (UPDATE emp SET dno = NULL WHERE dno = pre_del.dno);

CREATE TRIGGER upd_dep_dno
UPDATE OF dno ON dep
REFERENCING OLD AS pre_upd NEW AS post_upd
FOR EACH ROW
(UPDATE emp SET dno=post_upd.dno WHERE dno = pre_upd.dno);
```

♦ При изменение на заплатата на служител, ако новата заплата е два пъти по-голяма от стара, това да се сигнализира като се добави ред в таблица `warn_tab`, следяща за такива нарушения. Решение с тригер в Informix:

```
CREATE TRIGGER tr2
UPDATE OF sal ON emp
REFERENCING OLD AS pre NEW AS post
FOR EACH ROW WHEN ( post.sal > pre.sal*2 )
(INSERT INTO warn_tab VALUES(pre.eno, pre.sal, post.sal, USER, TODAY));
```

♦ След изменение на заплатата на служители, ако новата средна заплата е по-голяма от 1000, то обновяването да се отмени. Решение с тригерив Informix:

```
CREATE TRIGGER tr3
UPDATE OF sal ON emp
AFTER (EXECUTE PROCEDURE upd_sal());
```

След завършване на оператора `UPDATE` се извиква процедурата `upd_sal`. Тя проверява дали не е нарушено ограничението за средната заплата и ако е отменя всички направени изменения.

```
CREATE PROCEDURE upd_sal()
DEFINE new_avg MONEY;
LET new_sal = (SELECT AVG(sal) FROM emp);
IF new_sal > 1000
      RAISE EXCEPTION -746, 0, 'Too big average salary';
END IF;
END PROCEDURE;
```

6.3. Транзакции и управление на конкурентния достъп

Транзакцията е логическа единица работа над БД, която е:
- **атомарна** (неделима) от гледна точка на предметната област, но
- включва обикновено **няколко операции** над БД,
- които преобразуват БД от едно непротиворечиво състояние в друго **непротиворечиво състояние**.

Операциите над БД могат да са изразени чрез един или няколко SQL оператора. Напр. увеличаване заплатата на всички служители с 10% (един `update`), или изтриване на служител от БД (`delete` от `emp_pro`, `delete` от `emp` и `update` в `dep` на `mgr`), но за предприятието това е едно неделимо действие. Ако състоянието на БД е непротиворечиво преди транзакцията, то ще е непротиворечиво и след нея, но това не се гарантира в междинните точки (между операциите).

Атомарността на транзакцията означава, че СУБД трябва да осигури успешното изпълнение на всички операции в транзакцията или нито една, каквото и да се случи – грешка при изпълнение на оператор, срив на системата. Затова атомарността я наричат принцип “all or nothing”.

6.3.1. Модели на транзакциите

Как програмистът отбелязва къде започва и къде свършва една транзакция? Има няколко SQL оператора за тази цел и различни правила за използването им.

`BEGIN [WORK];`

Отбелязва началото на транзакция.

`COMMIT [WORK];`

Отбелязва успешен край на транзакция. СУБД потвърждава всички изменения в БД от началото на транзакцията, т.е. те стават трайни.

`ROLLBACK [WORK];`

Отбелязва неуспешен край на транзакция. СУБД трябва да отмени всички изменения в БД от началото на транзакцията.

`SAVEPOINT` *име*`;`

Създава маркер в транзакцията, чрез който една голяма транзакция може да се раздели на части и да се прави отмяна на част от измененията в БД. Името трябва да е уникално в транзакцията.

`ROLLBACK [WORK]TO SAVEPOINT` *име*`;`

Отменя част от транзакцията от точка *име* и изтрива всички други маркери (savepoints) след *име*.

- **Модел на ANSI**

Транзакцията започва автоматично с първия SQL оператор или след края на предходната транзакция (след `COMMIT WORK`, `ROLLBACK WORK`). Тук няма оператор `BEGIN WORK`. При изпълнение на DDL оператор се реализира неявен край на транзакция - първо `COMMIT`, след това се изпълнява съответния оператор и отново `COMMIT`. DDL операторите се реализират по модела на еднооператорна транзакция. Моделът на ANSI е реализиран в Oracle.

- **Модел на Informix**

Транзакцията започва с оператор `BEGIN WORK`. Ако не се използват операторите `BEGIN WORK`, `COMMIT WORK`, `ROLLBACK WORK`, то се поддържат еднооператорни транзакции. Всеки SQL оператор е транзакция, т.е. СУБД гарантира неговата атомарност.

По отношение на поддържане на транзакциите в Informix има три възможности, които се избират независимо за всяка БД при създаването ѝ.

- БД без транзакции

```
CREATE DATABASE име-на-бд;
```

- БД с транзакции по модела на ANSI

```
CREATE DATABASE име-на-бд WITH LOG MODE ANSI;
```

- БД с транзакции по модела на Informix

```
CREATE DATABASE име-на-бд WITH [BUFFERED] LOG;
```

6.3.2. Журнал на транзакциите (Transaction Log)

Как СУБД реализира транзакциите? Как осигурява атомарността дори ако преди края ѝ стане срив в системата, т.е. транзакцията остане незавършена? Как осигурява отмяната на измененията при `ROLLBACK`. Историята на всички изменения над БД по време на транзакциите се съхранява в специално място, наречено Transaction Log. В Informix това са така наречените logical log страници в основната област. В Oracle се наричат rollback сегменти, които са в някоя от областите.

При изменение на БД се прилага **WAL протокол** (Write Ahead Logging). Преди да се извърши каквото и да е изменение в БД, съответен log запис, който включва старото и новото значение на реда, се записва на диска. След това се изменя и реда в таблицата на БД. При `COMMIT` се записва log запис за успешен край на транзакцията. При `ROLLBACK` се извличат от log записите оригиналните значения на редовете и се връщат в БД, след което се записва log запис за неуспешен край на транзакцията.

След системен срив ефекта от частично изпълнените транзакции се отменя автоматично когато СУБД отново се стартира (процедура recovery). Тогава СУБД изпълнява автоматично `ROLLBACK` за всички незавършили транзакции. Това е възможно благодарение на WAL протокола (ако няма log запис, то и изменението в БД не е извършено).

Използването на транзакции увеличава значително времето за изменения в БД. С цел по-добра производителност, а и по-висока надеждност, е добре Transaction Log и БД да са на различни дискове. Някои СУБД позволяват да се работи и без транзакции, което е приемливо при учебни и малки БД.

6.3.3. Управление на конкурентния достъп

В общия случай няколко приложни програми (следователно транзакции) работят едновременно с БД. Докато една транзакция изменя БД, други също могат да четат и изменят същите данни. Сега СУБД не само трябва да се грижи за атомарността на транзакциите, но и да гарантира коректно взаимодействие на конкурентните транзакции, т.е. че те няма да си пречат. В противен случай са възможни нежелателни проблеми, като загуба на изменения, достъп до междинни (несъществуващи) или несъгласувани данни.

Загуба на изменения (lost update problem)

Транзакция A	Време	Транзакция B
`select t`	t1	
	t2	`select t`
`update t`	t3	
	t4	`update t`

Изменението на транзакция A се губи.

Достъп до междинни данни (uncommited dependency problem)

Транзакция A	Време	Транзакция B
	t1	`update t`
`select t`	t2	
	t3	`rollback work`

Транзакцията A чете данни от непотвърдената конкурентна транзакция B, която впоследствие е отменена, т.е. вижда несъществуващи данни. Този проблем е известен още като мръсно четене (dirty read). Ако t1 се припокрие частично с t2, то транзакция A ще прочете смесица от променени и непроменени редове.

Достъп до несъгласувани данни

Транзакция A	Време	Транзакция B
`select t`	t1	
	t2	`update t`
	t3	`commit work`
`select t`	t4	

Транзакцията A чете данни от потвърдената конкурентна транзакция, но въпреки това от нейна гледна точка състоянието на БД е несъгласувано. В рамките на транзакцията A едни и същи редове съдържат различни данни (nonrepeatable read).

Подобен проблем се получава и ако вместо `update` конкурентната транзакция изпълни `insert into t`. Тогава при повторния `select` транзакцията A ще види редове-призраци, които преди не са съществували (phantom read).

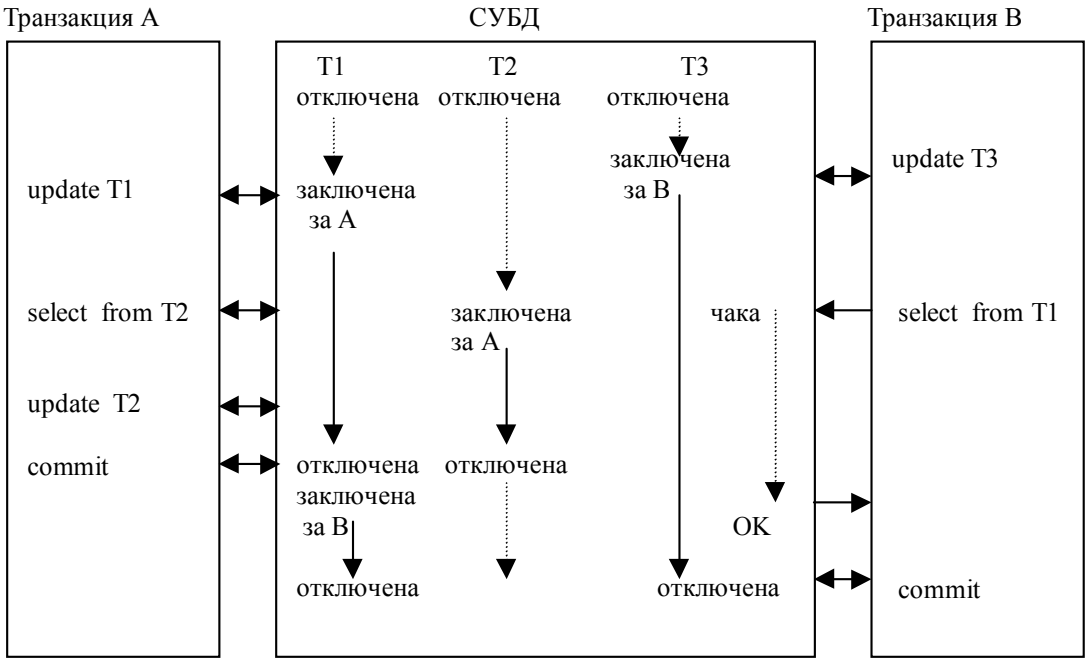
Следователно, за да се осигури цялостност на данните в условията на конкурентно изпълнявани транзакции, е необходим механизъм за управление на конкурентния достъп (Concurrency Control Mechanism). Той трябва да осигури изпълнението на конкурентни транзакции при следното условие:

Когато две транзакции A и B се изпълняват едновременно, СУБД гарантира, че резултатите от тях ще бъдат същите както и ако транзакциите са изпълнени последователно: (а) първо се изпълни A след това B или (б) първо се изпълни B след това A.

Този принцип се нарича **сериализиране** на транзакциите. На практика това означава, че всеки потребител работи с БД така, като че ли няма други потребители, т.е. осигурена е пълна изолация на потребителите един от друг.

Най-популярният механизъм се състои в **заключване/блокиране** на достъпа до част от БД, и е известен като locking. СУБД разполага с набор от ключалки (locks). Преди достъп до обект в БД транзакцията трябва да получи ключалка за него, като може да се наложи да чака докато системата е в състояние да й даде. Когато транзакция T_i получи ключалка за обект (обектът е заключен от T_i), то СУБД гарантира, че никоя друга конкурентна транзакция T_j няма да изменя обекта, докато T_i не го отключи. Освобождаването на ключалките става обикновено в края на транзакцията. Заключване се прилага дори когато в БД не се поддържат транзакции, тогава има конкурентни приложни програми вместо транзакции, но правилата за получаване и освобождаване на ключалка са по-различни.

Следва един прост пример, илюстриращ използването на блокиране при конкурентно изпълнение на две транзакции, които осъществяват достъп до три таблици - T1, T2, T3 (Фиг.10).



Фиг. 10. Конкурентно изпълнение на две транзакции

В промишлените СУБД се реализира по-сложен механизъм на блокиране с цел повишаване на степента на паралелизъм в системата.

Видове заключване

Обикновено се реализират няколко вида ключалки. Основните са:

`SHARE` (`read`)
: Заключва обекта за четене, като СУБД не позволява изменението му от други конкурентни транзакции. Това означава, че няколко транзакции могат да получат ключалка `SHARE` за един обект по едно и също време.

`EXCLUSIVE` (`write`)
: Заключва обекта за монополно ползване. Когато една транзакция получи ключалка `EXCLUSIVE` за обект, то никоя друга транзакция не може да получи каквато и да е ключалка за този обект. Използва се когато транзакцията изменя обекта.

Област на заключване (lock granularity)

Какви са обектите в БД, които се заключват с ключалки или каква е областта на действие на една ключалка? Това може да е:

- цялата БД (в Informix)
- цяла таблица
- ред от таблица.

Цяла БД в Informix се заключва с една ключалка при отваряне. Видът на ключалката зависи от оператора `DATABASE`.

`DATABASE` *`име-на-бд`*`;`

Заключва БД с `SHARE` ключалка. Докато не се освободи ключалката, БД не може да бъде унищожена от друга програма. Отключването става при `CLOSE DATABASE`.

`DATABASE` *`име-на-бд`* `EXCLUSIVE;`

Заключва БД с `EXCLUSIVE` ключалка. Програмата получава монополен достъп до БД. Операторът се изпълнява успешно ако никоя друга програма не е отворила БД в момента. След успешното изпълнение на оператора, никоя друга програма не може да отвори БД, докато тя не се отключи.

Цяла таблица може да се заключи с една ключалка. Това се извършва автоматично от СУБД с `EXCLUSIVE` ключалка, при изпълнение на операторите: `ALTER TABLE, CREATE INDEX, ALTER INDEX, DROP INDEX`. При завършване на оператора таблицата се отключва. Цяла таблица може да се заключи с една ключалка и явно чрез оператор:

```
LOCK TABLE име-на-таблица IN {SHARE|EXCLUSIVE} MODE [NO WAIT];
```

В БД с транзакции операторът трябва да се изпълнява в транзакция.

Отключването на таблица в БД с транзакции се извършва автоматично в края на транзакцията (`COMMIT/ROLLBACK`). В БД без транзакции отключването се извършва при затваряне на БД или явно с оператор, напр. в Informix:

```
UNLOCK TABLE име-на-таблица;
```

Ред от таблица може да се заключи с една ключалка. Ключалка за ред се получава и освобждава само автоматично. Правилата зависят от изпълнявания SQL оператор, дали се поддържат транзакции в БД и от нивото на изолация.

При изпълнение на `INSERT, UPDATE, DELETE` винаги трябва да се получи `EXCLUSIVE` ключалка за всеки изменян ред, освен ако цялата таблица не е заключена вече. Кога ще се освободят така получените ключалки зависи от това дали се поддържат или не транзакции в БД. Ако БД е с транзакции, всички ключалки се освобождават в края на транзакцията. Ако БД е без транзакции (това е възможно в Informix), освобождаването на заключения ред става след като той се запише на диска.

В Informix е възможно да се избира между заключване на ред от таблица или страницата, в която е той. Това се определя при създаването на таблицата или може да бъде променено впоследствие с операторите:

```
CREATE TABLE име-на-таблица ( ... ) LOCK MODE ({PAGE | ROW});
ALTER TABLE име-на-таблица LOCK MODE ({PAGE | ROW});
```

Как се използват ключалки при изпълнение на оператор `SELECT` се определя от нивото на изолация.

Нива на изолация (Isolation levels)

Според строгото определение за транзакция – принципа за сериализиране, нито една конкурентна транзакция не може да повлияе на данните виждани от друга транзакция. Тази абсолютна изолация на една транзакция от другите изисква доста ресурси от СУБД. В някои случаи усилията по блокиране могат да се съкратят, ако СУБД е информирана предварително за действията на транзакцията. Затова е въведено понятието ниво на изолация, което дава възможност на програмиста да постигне компромис между степента на изолация и ефективността на работа. Нивото на изолация определя степента, до която четяща програма е изолирана от конкурентните действия на други програми, изменящи БД. В ANSI стандарта и в СУБД са реализирани няколко нива, при които действат различни правила за използване на ключалки при четене от БД. Нивата по ANSI стандарта са.

	Мръсно четене	Неповторяемо четене	Фантомно четене
`READ UNCOMMITTED`	да	да	да
`READ COMMITTED`	не	да	да
`REPEATABLE READ`	не	не	да
`SERIALIZABLE`	не	не	не

Нивата на изолация, реализирани в Oracle са:

```
READ COMMITTED
```

Всяка заявка в транзакцията вижда само данните от заявки, потвърдени преди нейното започване (на заявката). Използва се по подразбиране в Oracle.

`SERIALIZABLE`

Транзакцията вижда само данните от заявки, които са потвърдени преди започването на транзакцията и данните променени от оператори в самата транзакция.

`READ ONLY`

Транзакцията вижда само данните от заявки, потвърдени преди нейното започване и в нея не са позволени оператори за изменение на данните.

Нивата на изолация, реализирани в Informix са:

`READ UNCOMMITTED`

При изпълнение на `SELECT` оператор редовете се четат, независимо дали са заключени от друг, и не се заключват. Това означава, че може да се прочетат редове от непотвърдени транзакции или несъгласувани данни. Не се осигурява никаква изолация на четящата програма. Предимството е ефективност – четящата програма никога не чака и не кара други да чакат. Това е единственото ниво в БД без транзакции.

`READ COMMITTED`

Гарантира се, че всеки ред извлечен от `SELECT` е от потвърдена транзакция, т.е.извлича ред ако не е заключен с `EXCLUSIVE` ключалка от друга транзакция. Четящата програма не заключва извлечените редове. При това ниво не се решава проблема с несъгласуваните данни. Това е нивото по премълчаване в БД с транзакции по модела на Informix.

`REPEATABLE READ / SERIALIZABLE`

Транзакцията получава `SHARE` ключалка за всеки извлечен ред. Това означава, че друга конкурентна транзакция може да получи след това за същите редове само `SHARE` ключалка. Следователно до края на транзакцията редовете не могат да бъдат изменени от друг. Това ниво използва най-много ключалки и ги държи най-дълго, но решава проблема с несъгласуваните данни. Това е нивото по премълчаване в БД с транзакции по модела на ANSI.

В Informix няма разлика между нивата `REPEATABLE READ` и `SERIALIZABLE`, въпреки че според стандарта има. Най-високата степен на изолация е `SERIALIZABLE`.

Операторът в Oracle, с който се изменя нивото на изолация и се задава типа на достъп до БД, може да се използва само в началото на транзакция и действа само за тази транзакция.

```
SET TRANSACTION [READ ONLY | READ WRITE]
      ISOLATION LEVEL {READ COMMITTED | SERIALIZABLE};
```

Освен него може да се определи ниво на изолация, което да действа във всички транзакции, изпълнявани в една сесия.

```
ALTER SESSION SET ISOLATION_LEVEL {READ COMMITTED | SERIALIZABLE};
```

Параметри на блокиране

Ако транзакция се опитва да получи ключалка за ред или таблица, но СУБД не е в състояние да даде ключалката веднага, каква е реакцията на СУБД. До сега предполагахме, че тогава изпълнението на програмата се подтиска, докато СУБД е в състояние да даде ключалката (програмата чака). В някои СУБД са реализирани няколко възможни реакции при неуспешен опит за блокиране. В Informix се реализира оператор, с който програмата определя една от три възможни реакции:

```
SET LOCK MODE TO { NOT WAIT | WAIT [ секунди ] };
```

`NOT WAIT` - Програмата не чака, а операторът завършва с грешка.

`WAIT` - Изпълнението на програмата се отлага, докато СУБД е в състояние да й даде ключалката.

`WAIT` *`секунди`* – Програмата чака най-много указания брой секунди.

ACID аксиоми за транзакциите

Всичко за транзакциите може кратко да се обобщи в следните четири свойства на транзакциите, които са известни като ACID аксиоми.

Atomicity

Изпълнява се цялата транзакция или нищо.

Consistency

Ако състоянието на БД е непротиворечиво преди транзакцията, то ще е непротиворечиво и след нейното завършване.

Isolation

Изпълнението на една транзакция е изолирано от това на другите конкурентно изпълнявани транзакциии. Всеки потребител трябва да работи с БД без да се безпокои от действията на другите потребители, т.е. като че има монополен достъп до БД.

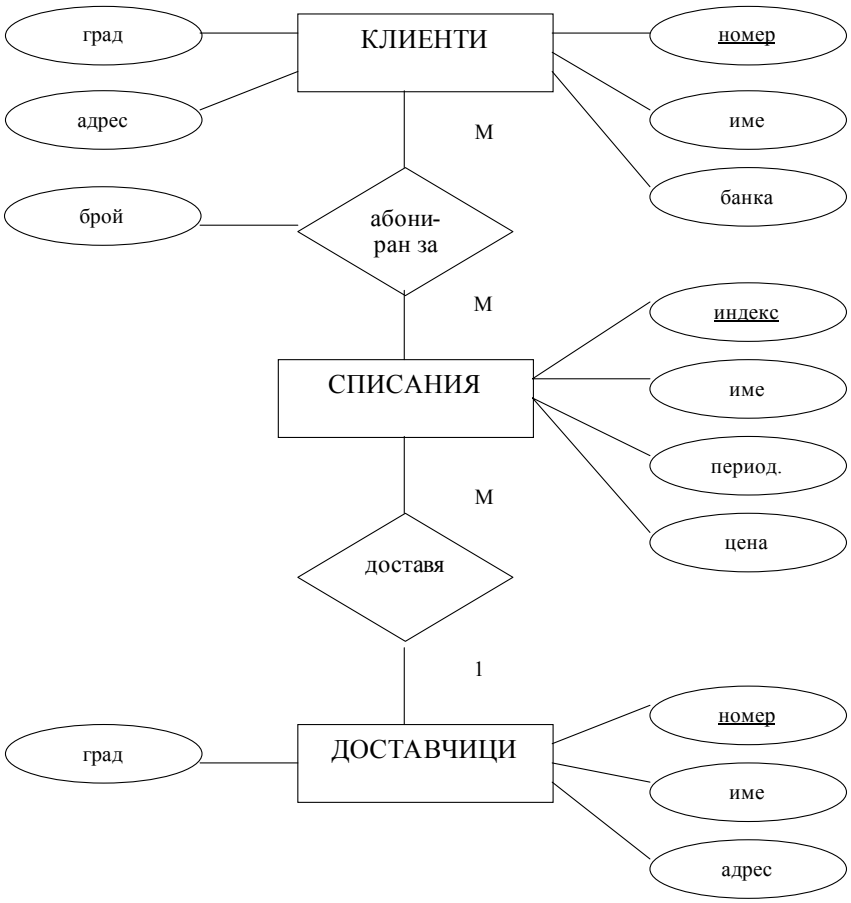
Durability

След потвърждаване на транзакцията ефектът от нея е постоянен.

7. Проектиране на релационна база данни

При проектиране на БД трябва да изберем релациите (таблиците) и техните атрибути, които най-точно и безпроблемно да представят предметната област. За една определена предметна област може да има различни релационни модели. Някои от вариантите са по-добри от други. Защо проекта на БД може да е лош?

Ще разгледаме един лош вариант за релационен модел на БД и проблемите, които той създава. Проектираме БД-абонаменти, която съхранява информация за абонаментите за периодични издания. На Фиг.11 е изобразена ER диаграмата.



Фиг. 11. ER диаграма на БД-абонаменти.

Съответният релационен модел, който е построен е:

`CLIENT (`**`cno`**`, cname, bank, ccity, caddr)`

`AB (`**`cno, jno`**`, jname, jprice, jper, dno, dname, dcity, daddr, cnt)`

Първичен ключ в `CLIENT` е `cno`, а за `AB` първичният ключ е съставен - {`cno,jno`}. В таблицата `AB` колоната `cno` е външен ключ, съответстващ на `CLIENT.cno`. Какви проблеми възникват при такава структура на БД.

- Излишество на данните. Наименованието, цената, периодичността и доставчика на списанието се повтарят за всеки абонамент, т.е. съхраняват се многократно в БД.
- Проблеми при изменение. Вследствие на излишеството съществува опасност за противоречивост на данните. Може да се измени наименованието, цената или периодичността на списание в едни редове, а в други не.
- Проблеми при добавяне. Не може да се добави информация за списание, ако за него няма нито един абонамент.

- Проблеми при изтриване. Ако в някакъв момент се изтрият всички редове за абонаментите на определено списание (всички клиенти са се отказали), то губим и информацията за списанието.

Казано накратко, не можем да съхраняваме информация за списание, ако за него няма поне един абонамент.

Тези проблеми ще изчезнат ако заменим таблицата `AB` с няколко подходящи таблици. Процесът, при който една таблица се разделя на няколко, при което се премахват недостатъците в проекта, се нарича **нормализация** чрез декомпозиция. Код определи три нива на нормализация, които нарече първа, втора и трета нормална форма (1НФ, 2НФ, 3НФ). Целта на теорията на нормализация е да се създават "чисти" проекти на БД, т.е. проекти, в които всеки факт се съхранява веднаж. Такива проекти са добри, защото са:

- лесни за разбиране (това означава простота на модела);
- лесни за разширяване (т.е. точност на представяне на ПО);
- не създават проблеми при използване.

Правилата за проектиране на релационна БД на основата на ER модел, формулирани при разглеждането на релационния модел, имаха същата цел - да се получи "чист" модел.

7.1. Функционални зависимости

Нека `R{A1, A2, …, An}` е релация, а `X` и `Y` са подмножества от атрибути на `R`. Казваме, че `X` функционално определя `Y` (или `Y` функционално зависи от `X`) и го означаваме `X->Y`, ако в `R` не може да има два или повече реда, съдържащи еднакви значения за `X` и различни за `Y`, и това е вярно във всеки момент от съществуването на `R`.

Ако `PK` е първичен ключ на релацията `R`, то са в сила функционалните зависимости `PK->X`, където `X` е произволно подмножество от атрибути на `R`. Това твърдение следва от дефиницията на първичен ключ. Същото твърдение естествено се отнася и за всеки възможен ключ на `R`.

Функционална зависимост (ФЗ) е понятие, което е свързано със семантиката на данните. ФЗ в определена релация се определя чрез анализ на предметната област, тъй като тя е предположение за смисъла на нещата в реалния свят.

Например в БД-абонамент в релациите `AB` и `CLIENT` са в сила ФЗ:

```
jno->jname, jno->jper, jno->jprice, jno->dno,
dno->dname, dno->dcity, dno->daddr, {jno,cno}->cnt
cno->cname, cno->bank, cno->ccity, cno->caddr
```

Тези функционални зависимости означават, че:

- Индексът на списанието го идентифицира.
- Всяко списание има само един доставчик.
- Всеки доставчик се идентифицира с номер.
- Всеки клиент има уникален номер.

Нека F е множество ФЗ за релацията `R`. Една ФЗ `X->Y` се нарича логическо следствие от F, ако всяко значение на `R`, което удоволетворява функционалните зависимости в F удоволетворява и `X->Y`. У. Армстронг формулира правилата, чрез които могат да се изведат всички ФЗ, които са логическо следствие от определено F. Това множество се нарича обвивка на F и се обозначава с F^+. Правилата са известни като аксиоми за извод на Армстронг.

Аксиоми на Армстронг

Нека X, Y, Z и W са атрибути на релацията R (може и съставни).

A1 (рефлексивност). Ако **Y ⊆ X**, то **X -> Y**. (тривиална ФЗ)

A2 (попълнение). Ако **X -> Y**, то **{X, Z} -> {Y, Z}**.

A3 (транзитивност). Ако **X -> Y** и **Y -> Z**, то **X -> Z**.

A4 (обединение). Ако **X -> Y** и **X -> Z**, то **X -> {Y, Z}**.

A5 (псевдотранзитивност). Ако **X -> Y** и **{W, Y} -> Z**, то **{W, X} -> Z**.

A6 (декомпозиция). Ако **X -> Y** и **Z ⊆ Y**, то **X -> Z**.

7.2. Нормални форми

Ако е в сила ФЗ X->Y и Y не зависи от никое подмножество на X, то казваме, че Y е в **пълна ФЗ** от X.

Ако са в сила функционалните зависимости X->Y и Y->Z, но Y->X не е в сила, то казваме че X->Z е **транзитивна ФЗ**.

Атрибут на релация, участващ в първичен или възможен ключ, ще наричаме първичен атрибут, а такъв, който не участва - непървичен атрибут.

Първа нормална форма (1НФ)

Всички релации в релационния модел са в 1НФ. Изискването за 1НФ е значенията на всички атрибути да са атомарни (неделими), т.е. релацията да е нормализирана.

Втора нормална форма (2НФ)

Релацията R се намира в 2НФ, ако е в 1НФ и всеки непървичен атрибут (атрибут, който не участва в първичен или възможен ключ) е в пълна ФЗ от първичния и от всеки възможен ключ на R.

Казано по друг начин в R няма ФЗ на непървичен атрибут от подмножество на първичен или възможен ключ. Следователно ако първичният и всички възможни ключове се състоят от един атрибут, то релацията е и в 2НФ.

В релация AB са в сила ФЗ:

```
{jno,cno}->jname,{jno,cno}->jper,{jno,cno}->jprice,{jno,cno}->dno,
{jno,cno}->dno,{jno,cno}->dname,{jno,cno}->dcity,{jno,cno}->daddr,
jno->jname, jno->jper, jno->jprice, jno->dno,
jno->dname, jno->daddr, jno->dcity,
{jno,cno}->cnt
```

което означава непълна ФЗ на jname, jper, jprice, dno, dname, dcity, daddr от първичния ключ {jno,cno}. Това е релация, която е в 1НФ, но не е в 2НФ. Ще я приведем в 2НФ, като я декомпозираме до две релации:

J (**jno**, jname, jprice, jper, dno, dname, dcity, daddr)

CJ (**cno, jno**, cnt)

Релацията CLIENT е в 2НФ защото първичният ключ е от един атрибут. Така получихме нов модел, включващ само релации в 2НФ - CLIENT, J, CJ.

Трета нормална форма (3НФ)

Релацията R се намира в 3НФ, ако е в 2НФ и всеки непървичен атрибут зависи нетранзитивно от първичния и от всеки възможен ключ на R.

Релацията J е в 2НФ, но не е в 3НФ защото са в сила ФЗ:

jno->dno, dno->dname, dno->dcity, dno->daddr

и не е в сила dno->jno. Следователно, следните ФЗ са транзитивни.

jno->dname, jno->dcity, jno->daddr

Декомпозираме J до JOURNAL и DOST така, че да премахнем транзитивните ФЗ.

JOURNAL (**jno**, jname, jprice, jper, dno)

DOST (**dno**, dname, dcity, daddr)

Така получихме нов модел, включващ само релации в 3НФ - CLIENT, JOURNAL, DOST, CJ.

Не се ли губи информация при това преобразуване на релационния модел? Ще можем ли от CLIENT, JOURNAL, DOST, CJ да получим същата информация както и от началния модел CLIENT, AB.

Дефиниция. Нека R е релация с множество от ФЗ F. Казваме, че декомпозицията на R до R1 и R2 притежава свойството **съединение без загуба** относно F, ако е изпълнено тъждеството:

R ≡ R[R1] join R[R2]

Ако проектираме R по множествата атрибути на R1 и R2 и след това съединим двете проекции, отново ще получим началната релация R. Това е вярно във всеки момент от съществуването на R, т.е. за всяко нейно тяло. Това означава, че при декомпозицията не се губи информация.

Дори при новата структура, в нашите примери на декомпозиция, може да се съхранява повече информация. В общия случай е вярно:

(R1 join R2)[R1] ⊆ R1

(R1 join R2)[R2] ⊆ R2

Ако съединим R1 и R2 и след това проектираме съединението по R1 може да получим множество от редове, което е подмножество на тялото на R1.

Теорема. Декомпозицията на R до R1 и R2 притежава свойството съединение без загуба относно F, тогава и само тогава когато една от следните две ФЗ е следствие от F.

R1 ∩ R2 -> R1 - R2

R1 ∩ R2 -> R2 - R1

Друго важно свойство на една декомпозиция е да **запазва функционалнте зависимости**. Не се ли губят ФЗ при преобразуването на модела?

Дефиниция. Нека R е релация с множество от ФЗ F, която е декомпозирана до R1 и R2. Нека F1 е множеството ФЗ, които са в сила в R1, а F2 – в R2. Казваме, че декомпозицията на R **запазва ФЗ**, ако всяка ФЗ от F е логическо следствие на ФЗ от F1 и F2, т.е в сила е тъждеството:

$F^+ \equiv (F1 \cup F2)^+$

Правила за преобразуване в 2НФ и 3НФ

1. От 1НФ в 2НФ

Нека в R(**K1, K2**, X, Y) първичен ключ ∈ {K1,K2} и са в сила ФЗ

{K1,K2}->X, K1->X, {K1,K2}->Y

Декомпозираме R до R1(**K1, K2**, Y) и R2(**K1**, X).
Според теоремата тази декомпозиция на R до R1 и R2 притежава свойството съединение без загуба, защото K1->X, т.е. R1 ∩ R2 -> R2 - R1.

2. От 2НФ в 3НФ
Нека в R(**K**, X, Y, Z) K е първичен ключ и са в сила ФЗ
K->X, X->Y, K->Y, K->Z, т.е. K->Y е транзитивна.
Декомпозираме R до R1(**K**, X, Z) и R2(**X**, Y)
И тази декомпозиция притежава свойството съединение без загуба, защото X->Y, т.е. R1 ∩ R2 -> R2 - R1.

Нормална форма на Бойс-Код (НФБК)

Релацията R се намира в НФБК, ако е изпълнено условието: винаги когато е в сила пълната ФЗ X->A и A ∉ X , то X е първичен или възможен ключ.

Ако R се намира в НФБК, то тя е и в 3НФ, но обратното не е вярно. Ако в R има два пресичащи се възможни ключа, то R може да е в 3НФ и да не е в НФБК. Примери:

R(**town, addr**, pcode)

В R са в сила функционалните зависимости:

{town,addr}->pcode, {addr,pcode}->town, pcode->town

Първичен ключ е {town,addr}, а {addr,pcode} е възможен ключ. Всички атрибути са първични, следователно R е в 3НФ. Но R не е в НФБК, защото pcode не е възможен ключ.

emp_project(**pno, eno**, pname, ptime)

Това е друг вариант на релация от БД-служители, вместо релациите project и emp_pro, ако всеки проект има уникален номер и наименование, но няма описание. Първичен ключ в релацията е {pno, eno}, а {pname, eno} е възможен ключ. В сила са функционалните зависимости:

{pno, eno}->ptime, {pname, eno}->ptime, pno->pname, pname->pno

Единственият непървичен атрибут ptime е в пълна ФЗ първичния и възможния ключ, следователно emp_project е в 3НФ, но не е в НФБК заради функционалните зависимости pno->pname и pname->pno.