



2005/0134/VPD1/ESF/PIAA/04/APK/3.2.3.2./0046/0122

Vidzemes augstskola

Datu bāzu tehnoloģijas.

2007.

Saturs.

Saturs. 2

1.	Ievads.....	12
1.1.	Literatūra	12
1.2.	Kas ir kas (definīcijas):	12
	• Datu bāzes.....	12
	• Dati	12
	• Relācija	12
	• Matemātiskie relāciju algebras termini:	12
	• Relāciju DB.....	13
	• Datu bāzu vadības sistēma	13
	• DB sistēma.....	13
	• SQL.....	13
1.3.	DB uzbūves pamati.....	13
	• Tabulas:.....	13
	• Indeksi.....	14
	• Skati	14
	• Triggeri	14
	• Saglabātās procedūras.....	14
	• Pieejas tiesību informācija,	14
2.	Datu bāzu vadības sistēmas	15
2.1.	ALTERNATĪVAS DBVS IZMANTOŠANAI:.....	15
2.2.	DBVS izmantošanas priekšrocības un galvenās funkcijas:.....	15
	• Ātrāka un vieglāka DB sistēmu (lietotāja aplikāciju) izstrāde !!!	15
	• Datu relāciju uzturēšana.....	15
	• Integritātes nosacījumu uzturēšana	15
	• Dublēšanās kontrole. Visas datu vienības DB tiek glabāti 1 eksemplārā.....	15
	• Problēmas, kas rodas no datu dublēšanās:	15
	• vieglāka pielāgojamība (<i>flexibility</i>), jo tiek realizēta programmu-datu neatkarība	15
	• Dažādu lietotāja interfeisu uzturēšana:	15
	• transakciju uzturēšana.....	16
	• datu vienlaicīgas pieejas kontrole	16
	• lietotāju pieejas tiesību kontrole	16
	• rezerves kopēšanas (<i>backup</i>) un atjaunošanās (<i>recovery</i>) uzturēšana	16
2.3.	Programmu - datu neatkarību (<i>program-data independence</i>).....	16
2.4.	Metadati.....	16
2.5.	Lietotāja aplikāciju sadarbība ar DB	17
	• Pa tiešo ar saviem datu failiem (šāda risinājuma trūkumi tika iepriekš apskatīti).....	17
	• Izmantojot DBVS piedāvāto API.....	17
	• Izmantojot DBVS ODBC draiverus.....	17
2.6.	API.....	17
2.7.	ODBC	18

2.8.	Servera process	18
2.9.	DBVS arhitektūra.....	19
	• Sintakses pārbaude (<i>parser</i>).....	19
	• Optimizācija.....	20
	• Izpilde (<i>execution engine</i>).....	20
	• Transakciju vadība	20
	• Vienlaicīgas pieejas vadība.....	20
	• Failu un indeksu vadība	20
	• Žurnāli un atjaunošanās	20
	• Bufera vadība.....	20
	• Diska vadība (<i>storage management</i>).....	20
2.10.	Mūsdienu DBVS:	21
	• Nelielām sistēmām:.....	21
	• Vidējām sistēmām.....	21
	• Lielām sistēmām	22
3.	DB sistēmas	23
3.1.	DB sistēmas	23
	• Tehniskais nodrošinājums.....	23
	• Programmnodrošinājums	23
	• Dati	23
	• Personāls	23
3.2.	DB sistēmu arhitektūra	24
	• Centralizēta arhitektūra.....	25
	• Klienta – servera arhitektūra.....	26
	• 3 līmeņu arhitektūra (<i>three tiers architecture</i>).....	27
	• Sadalītās (<i>distributed</i>) DB.....	28
	• Decentralizētas DB	29
3.3.	DB Sistēmu iedalījums pēc mērķiem un lietotāju skaita:	29
	• Personālās (<i>personal</i>) DB sistēmas.....	29
	• Darba grupu (<i>workgroup</i>) DB sistēmas	29
	• Nodaļu (<i>department</i>) DB sistēmas	29
	• Uzņēmumu (<i>Enterprise</i>) sistēmas	29
3.4.	DB var iedalīt vairākos veidos:.....	29
	• Tradicionālās datu bāzes ar teksta un ciparu informāciju	29
	• Datu noliktavas (<i>data warehouses</i>) un OLAP (<i>Online Analytical Processing</i>)	29
	• Multimediju datu bāzes.....	30
	• Ģeogrāfiskās IS.....	30
3.5.	PAPILDUS LIERATŪRA	30
3.6.	Paškontroles jautājumi.	30
4.	DB veidošana	31
4.1.	MySQL	31
4.2.	PostgreSQL	31
4.3.	Tabulu veidošana.....	31

4.4.	Datu tipi	32
•	Skaitliskie datu tipi	32
•	Datuma/laika datu tipi.....	32
•	Teksta datu tipi.....	33
•	Citi datu tipi	33
•	Datu tipu konvertācija.....	34
4.5.	PAPILDUS LIERATŪRA	34
4.6.	Paškontroles jautājumi.	34
5.	Sistēmas izstrāde un DB projektēšana	35
5.1.	Aplikāciju izstrādes posmi:	35
•	Lietotāja prasību identificēšana,	35
•	Sistēmas analīze,	35
•	Loģiskā projektēšana,	35
•	Fiziskā projektēšana.....	35
•	Ieviešana	35
•	Uzturēšana	35
5.2.	DB izveides posmi:	35
•	Loģiskā projektēšanas posmā:	35
○	ER diagrammu veidošana,	35
○	ER diagrammu pārveidošana par relācijām,	35
○	Relāciju normalizācija,	35
○	Ierobežojumu noteikšana.	35
•	Fiziskā projektēšanas posmā:.....	35
○	Tabulu veidošana izmantojot SQL vai GUI.....	35
5.3.	ER diagrammu veidošana	36
•	Entītijas	36
•	Saites.....	36
•	Atribūti.....	36
5.4.	Unified Modeling Language (UML)	37
5.5.	ER diagrammu pārveidošana par relācijām	37
•	Atslēgas.....	37
•	No entītijām izveido relācijas	38
•	Izveido binārās relāciju saites	39
•	Izveido unārās relāciju saites	40
•	Izveido daudzkārtīgas relāciju saites.....	41
•	Izveido virstipu/apakštipu relācijas.....	42
5.6.	Normalizācija	43
•	Anomālijas	43
5.7.	Pirmā normālforma	44
•	Relācija ir pirmajā normālformā, ja:	44
•	Ja ir šāda situācija, to iespējams novērst 3 dažādos veidos.....	44
5.8.	Otrā normālforma	45
•	Relācijas ir otrajā normālformā, ja:.....	45
5.9.	Trešā normālforma	46

•	Relācija ir trešajā NF, ja:	46
5.10.	Boyce-Codd normālforma.....	47
•	Relācijas ir BCNF , ja	47
5.11.	Denormalizācija (denormalization).....	48
•	Relācijas ar saiti 1:1	48
•	Relācijas ar saiti N:M, kura realizēta kā atsevišķa relācija.	48
•	Norādes saitē 1:N.....	49
•	Vienkāršota atslēga	49
•	Izskaitļotie atribūti	50
5.12.	Ierobežojumu noteikšana un tabulu veidošana izmantojot SQL vai GUI.....	50
5.13.	ER modelēšanas rīki.....	50
5.14.	PAPILDUS LIERATŪRA	50
5.15.	PAŠKONTROLES JAUTĀJUMI.....	50
6.	Datu pievienošana DB.....	52
6.1.	INSERT	52
7.	Datu iegūšana no DB.....	52
•	Atlases nosacījumi	53
•	Agregātfunkcijas	54
•	Grupēšana	54
•	Rezultātu sakārtošana.....	54
7.1.	PAPILDUS LIERATŪRA	55
7.2.	Paškontroles jautājumi.	55
8.	SELECT paplašinātās iespējas	56
8.1.	HAVING.....	56
8.2.	Datu kolonnas no dažādām tabulām.....	56
•	INNER JOIN.....	57
•	WHERE	58
•	LEFT JOIN	58
•	RIGHT JOIN.....	58
•	FULL JOIN.....	58
•	Dati no 3 vai vairāk tabulām	59
8.3.	Dati vienā kolonnā no dažādām tabulām	60
•	UNION	60
8.4.	INTERSECT	61
8.5.	EXCEPT	61
8.6.	Apakšpieprasījumi (subqueries)	61
•	Tiešie datu avoti:.....	61
•	Datu avoti nosacījumiem:	61

8.7.	Pieprasījumu veidošana	62
8.8.	PAPILDUS LIERATŪRA	63
8.9.	Paškontroles jautājumi.	63
9.	Ierakstu glabāšana un failu struktūra.....	64
9.1.	Disku apakšsistēma (cietie diski un I/O kontrolieri):	64
9.2.	Buferešana.....	64
9.3.	RAID.....	64
	• RAID datu dalīšanas veidi:	64
	• RAID līmeņi	66
	• JBOD (<i>Just A Bunch Of Disks</i>).....	67
9.4.	Tīkla glabātuves (<i>Network Storage</i>).....	68
	• Tieši pievienotās datu glabātavas (<i>Direct Attached Storage</i>)	68
	• Tīkla datu glabātavas (<i>Network Attached Storage</i>).....	68
	• Glabātavu tīkls (<i>Storage Area Network</i>)	68
9.5.	DB organizācijas modeļi:	68
	• Teorija.....	68
	• MySQL	70
	• PostgreSQL.....	70
9.6.	Faila organizācija:	71
	• Teorija.....	71
	• PostgreSQL.....	72
	• MySQL	72
9.7.	Indeksi	72
	• B-tree	73
	• Bitmapu indeksi	75
	• Indeksu izmantošana.....	75
	• Praktiskā realizācija	76
9.8.	PAPILDUS LIERATŪRA	77
9.9.	Paškontroles jautājumi.	77
10.	Citi DB objekti	78
10.1.	Secības (<i>sequences</i>) PostgreSQL.....	78
10.2.	Skati (<i>views</i>).....	78
11.	Datu integritāte	79
11.1.	NULL vērtības.	79
	• NULL un JOIN	79
	• NULL un referenciālā integritāte.....	79
	• NULL ietekme uz aprēķiniem.....	79
11.2.	Datu integritāte.....	79

11.3. Ierobežojumus sadalījums:	80
• Tabulas integritātes ierobežojumi (<i>entity integrity</i>):	80
• Domēna nosacījumi (<i>domain constraints</i>):	80
• Saišu integritāte (<i>referential integrity</i>):	80
11.4. Kolonnas ierobežojumi :	81
11.5. Tabulas ierobežojumi:	81
11.6. Darbības, ko nosaka saišu integritāte:	82
• No action	82
• Kaskādes izmaiņas	82
• Dzēšana uzstādot NULL	82
• Dzēšana uzstādot vērtību pēc noklusēšanas	83
11.7. Nosacījumu pārbaudes brīdis	83
• Parametri	83
• SET CONSTRAINTS	84
11.8. MySQL	84
11.9. PAPILDUS LIERATŪRA	84
11.10. Paškontroles jautājumi.	85
12. Datu manipulācijas komandas	86
12.1. INSERT	86
12.2. UPDATE	86
12.3. DELETE	87
12.4. PAPILDUS LIERATŪRA	87
12.5. Paškontroles jautājumi.	87
13. Saglabātās procedūras	88
13.1. Ievads	88
13.2. Saglabāto procedūru izmantošanas priekšrocības:	88
• Uzlabojas ātrdarbība,	88
• Mazāks atmiņas izlietojums	88
• Ar to palīdzību iespējams detalizētāk uzstādīt datu drošības nosacījumus	88
• Ļauj pārdalīt skaitļošanas noslodzi starp serveri un klientu.	88
• Samazina datortīkla noslodzi	88
13.3. PostgreSQL	88
• Sintakse	89
13.4. MySQL	90
13.5. PAPILDUS LIERATŪRA	90
13.6. Paškontroles jautājumi.	90

14.	PL/pgSQL.....	91
14.1.	Programmas struktūra.....	91
14.2.	Argumentu saņemšana un atgriešana.....	92
	• Argumentu saņemšana	92
	• Argumentu atgriešana	92
14.3.	Komentāri:	93
14.4.	Deklarācijas.....	93
	• Ierakstu mainīgie.....	93
	• Mainīgā tips	94
14.5.	Konstrukcijas	94
	• Piešķiršana	94
	• SELECT INTO	94
	• Dinamiskie pieprasījumi	94
14.6.	Kontroles struktūras	95
	• IF-THEN-ELSE	95
	• LOOP cikla operators	95
	• Cikla beigu komanda	95
	• WHILE cikla operators	96
	• FOR cikla operators	96
	• FOR cikla operators pieprasījumu rezultātu caurskatīšanai	96
14.7.	Kursori.....	97
	• Kursora mainīgais	97
	• Kursora atvēršana	97
	• Kursora aizvēršana.....	98
	• Kursora izmantošana.....	98
14.8.	Paziņojumu izvadišana.....	98
14.9.	PAPILDUS LIERATŪRA	98
14.10.	Paškontroles jautājumi.	99
15.	Trigeri.....	100
15.1.	Trigeri var tikt lietoti, lai :	100
	• veidotu sarežģītus drošības uzstādījumus,	100
	• veidotu sarežģītus biznesa likumus datu integritātei,	100
	• veidotu sarežģītu atsauksmju integritāti,	100
	• nodrošinātu specifisku audita datu vākšanu	100
	• veidotu kopiju tabulas (<i>replicas</i>).....	100
	• kā arī citiem mērķiem pēc sistēmu veidotāja prasmes un ieskatiem.	100
15.2.	Trigeri sastāv no četrām daļām:	101
	• Notikums.....	101
	• Trigera izpildes brīdis	101
	• Izpildes līmenis:	101
	• Darbības	101
15.3.	PostgreSQL	101

15.4.	MySQL	102
15.5.	PAPILDUS LIERATŪRA	102
15.6.	Paškontroles jautājumi.	103
16.	Transakciju mehānismi.....	104
16.1.	Nekontrolēta vienlaicīga pieeja datiem un to izmaiņšana var radīt vairākas problēmas:	104
	• Zaudēto izmaiņu problēma.	104
	• Īslaicīga labojuma (<i>temporary update</i>) problēma	104
	• Nekorekto statistisko aprēķinu problēmas	104
16.2.	Atjaunošanas (<i>recovery</i>) nepieciešamība	104
16.3.	Iespējamās kļūdu rašanās iemesli ir:	105
	• Tehniskā nodrošinājuma kļūda:	105
	• Transakcijas vai DB sistēmas kļūda:.....	105
	• Vienlaicīgas pieejas izmantošana:	105
16.4.	Transakciju īpašības.....	105
	• Atomārums (<i>atomicity</i>)	105
	• Nepretrunīguma saglabāšana (<i>Consistency preservation</i>)	105
	• Izolācija (<i>isolation</i>)	105
	• Pastāvība (<i>permanency</i>).....	105
16.5.	Transakcijas stāvokļi.....	105
	• BEGIN_TRANSACTION	106
	• READ vai WRITE	106
	• END_TRANSACTION	106
	• COMMIT_TRANSACTION	106
	• ROLLBACK.....	106
16.6.	Sistēmas žurnāls.....	106
16.7.	Transakcijas PostgreSQL	106
16.8.	Transakcijas MySQL	107
16.9.	PAPILDUS LIERATŪRA	107
16.10.	Paškontroles jautājumi.	108
17.	Laiksakritības kontrole metodes.....	109
17.1.	Bloķēšanas līmeņi	109
	• Datu bāze līmenis.....	109
	• Tabulas līmenis	109
	• Bloka vai lapas līmenis	109
	• Ieraksta līmenis	109
	• Lauka līmenis.....	109
17.2.	Bloķēšanas (<i>locking</i>) metodes	109
	• Binārais bloķēšanas mehānisms.....	109
	• Lasīšanas/rakstīšanas bloķēšanas mehānisms.....	110

•	Divu fāzu bloķēšanas mehānisms (2PL)	110
•	Dedloki (<i>deadlocks</i>)	110
•	Badošanās (<i>starvation</i>).....	112
17.3.	PosetgreSQL	112
•	Iespējamie bloķēšanas līmeņi un darbības, kas tos izsauc:	112
•	Bloķēšanas mehānisma vadības komandas	114
17.4.	PAPILDUS LIERATŪRA	114
17.5.	Paškontroles jautājumi.	115
18.	Datu bāzu drošība	116
18.1.	Vispārējs atskats	116
•	D klase - nekādas aizsardzības	116
•	C klase – sadalās 2 apakšklasēs	116
•	B klase sadalās 3 apakšklasēs	116
•	A klase	116
18.2.	Drošības mērķi veidojot DB sistēmu:.....	117
•	Novērst neautorizētu personu pieeju pašai sistēmai (<i>access control</i>)	117
•	Nodrošināt tiesību kontroles mehānismus:	117
•	Novērst iespēju iegūt datus no DB apejot DBVS	118
•	Nodrošināt attiecīgo likumdošanas aktu ievērošanu.	118
18.3.	Drošības prasību noskaidrošana:	118
•	Kādiem lietotājiem jānodrošina pieejas tiesībām DB	118
•	Kādai DB daļai katram lietotājam ir tiesības piekļūt	118
•	Kādas operācijas lietotājam ir tiesības veikt un ar kādiem datiem.....	118
18.4.	DB sistēmas pieejas tiesību administrēšanu var iedalīt vairākās daļās:	118
•	Kontu izveidošana (<i>account creation</i>)	118
•	Tiesību piešķiršana (<i>privilege granting</i>)	118
•	Tiesību atcelšana (<i>privilege revocation</i>)	118
•	Drošības līmeņa piešķiršana (<i>security level assignment</i>)	118
18.5.	DBVS Drošības mehānismus iedala 2 daļās:	119
•	Atsevišķie (<i>discretionary</i>) drošības mehānismi	119
•	Mandātu (<i>mandatory</i>) drošības mehānismi.....	119
18.6.	Atsevišķā pieejas kontrole ar tiesību piešķiršanu / atcelšanu.	119
•	Tiesību tipi:	119
•	Tiesību noteikšana izmantojot skatus (<i>views</i>)	119
•	Tiesību vadība izmantojot saglabātās procedūras (<i>Stored Procedures</i>).....	120
•	Ierakstu līmeņa tiesības (<i>row level security</i>)	120
•	Tiesības piešķirt tiesības GRANT OPTION	120
•	Grupas (<i>groups</i>) un lomas (<i>roles</i>)	120
18.7.	PostgreSQL	121
•	Lietotāju autentifikācija	121
•	Lietotāja tiesību vadība	121
18.8.	MySQL	123
18.9.	Datu šifrēšana	124

18.10. Audits.....	125
18.11. PAPILDUS LIERATŪRA	125
18.12. Paškontroles jautājumi.	125
19. Datu glabātavas un OLAP.....	126
19.1. Datu glabātavu un tradicionālo datu bāzu salīdzinājums:	126
19.2. Datu glabātavas sistēmas arhitektūra.....	127
19.3. Datu glabātavas	128
1) Datu iegūšana.....	128
2) Datu pārveide.....	128
3) Datu ielāde datu glabātavā.....	128
19.4. OLAP.....	129
19.5. Hierarhiskās dimensijas.....	130
19.6. Lietotāja interfeiss.	130
19.7. PAPILDUS LIERATŪRA	130
19.8. Paškontroles jautājumi.	130
20. Citi DB veidi.	131
20.1. Objektu un objektu-relāciju datu bāzes.	131

1. Ievads

1.1. LITERATŪRA

- ⇒ R. Ramakrishnan. "Database Management Systems"
- ⇒ Ramez Elmasri "Fundamentals of Database Systems"
- ⇒ Jeffery A. Hoffer "Modern Database Management"
- ⇒ C. J. Date "Introduction to Database Systems"

Pieejama arī krievu valodā. Diemžēl ļoti teorētiska.

- ⇒ <http://www.oracle.com/technology/documentation/database10g.html>

(pieejamas 138 grāmatas HTML un PDF formātā).

- ⇒ <http://www.postgresql.org/docs/>
- ⇒ <http://dev.mysql.com/doc/>
- ⇒ Dan Tow "SQL Tuning"
- ⇒ Schneider Robert D. "MySQL Database Design and Tuning"

1.2. KAS IR KAS (DEFINĪCIJAS):

- Datu bāzes

Datu bāze (DB) ir loģiski saistītu datu apkopojums. (nejaušu datu sakopojumu nevar uzskatīt par datu bāzi). Datu bāze var būt jebkura izmēra un sarežģītības. Datu bāze var tikt veidota un uzturēta kā uz papīra tā arī datorā.

- Dati

Dati teksts, skaitļi, attēli, skaņa, video un cita veida faktu atspoguļojums.

- Relācija

Relācija ir n ierakstu (*tuples*) kopa, kur katrs ieraksts ir sakārtots m vērtību saraksts, kura katra vērtība pieder atribūta noteiktam domēnam.

- Matemātiskie relāciju algebras termini:

- ⇒ relācija (*relation*) = tabula,
- ⇒ relāciju saite (*relationship*) = saite,
- ⇒ ieraksts (*tuple*) = ieraksts,
- ⇒ atribūts (*attribute*) = kolonna,
- ⇒ domēns (*domain*) = pieļaujamo vērtību kopums.

Ar ko atšķiras tabula no relācijas? (un attiecīgi visi šie termini viens no otra). Matemātiskos terminus lieto, veicot relāciju teorētisko modeļu izveidi. Praktiski realizētu modeļu aprakstīšanai tiek lietoti datortermini.

- Relāciju DB

Relāciju DB, kura sastāv no atsevišķām savās starpā loģiski saistītām tabulām.

- Datu bāzu vadības sistēma

Datu bāzu vadības sistēma (DBVS) (*Data Base Management System*) ir programmu kopums, kas atļauj veidot un uzturēt datu bāzes.

- DB sistēma

DB sistēma (DBS) (*Data base system*) – sistēma, kuras pamatmērķis ir datu apstrāde. Sistēma apvieno sevī datus (datu bāzi) un aplikācijas šo datu apstrādei. Datu bāzu sistēma ir sinonīms terminam "datu apstrādes sistēma".

- SQL

Structured Query Language ir īpaša valoda, kas radīta dažādu darbību veikšanai ar datu bāzēm. SQL ir standartizēta valoda, ko lielā mērā atbalsta visas DBVS.

1.3. DB UZBŪVES PAMATI.

Kā jau definīcija („Datu bāze ir loģiski saistītu datu apkopojums”) un nosaukums liecina datu bāzes galvenā sastāvdaļa ir dati. Datu bāzes galvenais uzdevums – uzglabāt un sniegt iespējas ērtāk darboties ar datiem. Tomēr datu bāzē glabājas ne tikai paši dati, bet arī dažāda papildus informācija, kas nepieciešama, lai varētu ātri un droši šos datus apstrādāt. Fiziski uz diska DB glabājas vienā vai vairākos operētājsistēmas failos (tas atkarīgs no katras konkrētās DBVS). No loģiskā viedokļa DB sastāv no daudziem objektiem (tabulām, indeksiem un citiem).

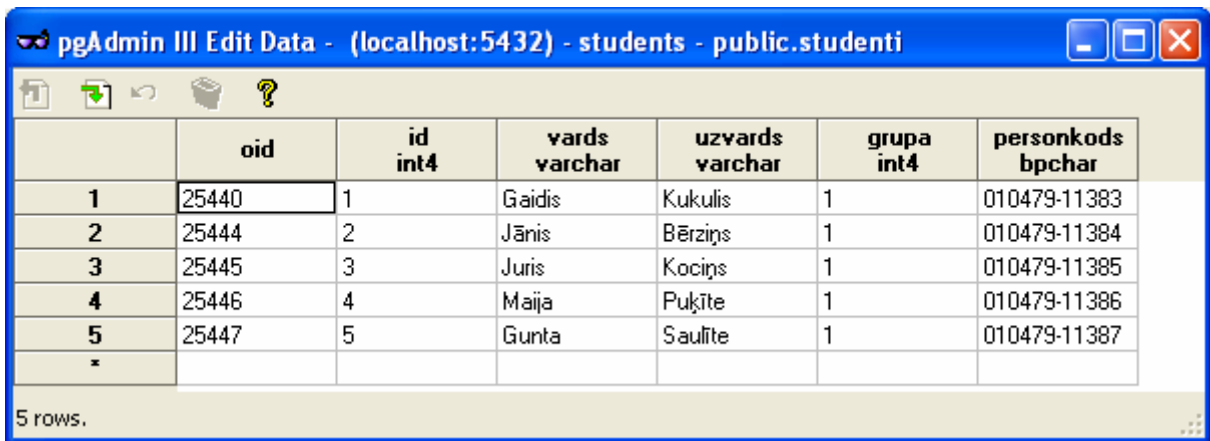
DB sastāvdaļas:

- Tabulas:

Lai lielā datu apjomā būtu iespējams orientēties, dati, kas glabājas datu bāzē, tiek sadalīti vairākās loģiski cieši saistītās kopās. Parasti šīs kopas tiek veidotas tā, lai tās varētu attēlot ar 2 dimensiju tabulu palīdzību.

Katrai tabulai ir savs unikāls nosaukums DB ietvaros un tā sastāv no vienas vai vairākām kolonnām. Kolonna satur informāciju ar vienu nozīmi, piemēram kolonnas, kas satur stundeta vārdu un uzvārdu). Katri kolonnai ir unikāls nosaukums tabulas ietvaros, kā arī noteikts datu tips. Rindīņas kolonnās tiek sauktas par ierakstiem.

Piemēram:



	oid	id int4	vards varchar	uzvards varchar	grupa int4	personkods bpchar
1	25440	1	Gaidis	Kukulis	1	010479-11383
2	25444	2	Jānis	Bērziņš	1	010479-11384
3	25445	3	Juris	Kociņš	1	010479-11385
4	25446	4	Maija	Puķīte	1	010479-11386
5	25447	5	Gunta	Saulīte	1	010479-11387
*						

5 rows.

Tabulu skaits datu bāzē var svārstīties no dažiem desmitiem līdz pat vairākiem tūkstošiem tabulu, viss atkarīgs no datiem, kurus nepieciešams DB jāuzglabāt.

- Indeksi

Indeksi ir datu struktūras, ar kuru palīdzību DBVS var ātrāk atrast kādu konkrētu ierakstu tabulā. Gadījumos, ja jāveic sarežģīti pieprasījumi un ierakstu skaits tabulās sniedzas desmitos vai simtos tūkstošu, tas ir ļoti svarīgi, lai nodrošinātu lietotājam vēlamu sistēmas ātrdarbību.

- Skati

Skati (*view*) ir īpaši virtuālas tabulas, kurās apvienoti (un iespējams atlasīti) dati no reālām tabulām. Skati paši par sevi datus nesatur, bet ļauj ērti darboties ar citu tabulu datiem.

- Trigeri

Par trigeri (*trigger*) sauc SQL komandu kopu, kura tiek glabāta datu bāzē un kura tiek automātiski izpildīta, ja tiek veikta attiecīgo trigeri izsaucoša darbība ar šo datu bāzi.

- Saglabātās procedūras.

Saglabātās procedūras (*stored procedure*) ir programmu koda moduļi (procedūras), kas tiek glabāti datu bāzē uz servera un arī to izpilde pēc izsaukšanas notiek uz servera.

- Pieejas tiesību informācija,

Lai dati, kas glabājas DB būtu nodrošināti pret nesankcionētu pieeju, DB tiek glabāta arī informācija par to kādas pieejas tiesības ir katram lietotājam vai lietotāju grupai.

Sīkāk ar šīm un citām DB sastāvdaļām iepazīsimies kursa gaitā.

2. Datu bāzu vadības sistēmas

Datu bāzu vadības sistēma (DBVS) ir programmatūras sistēma (aplikāciju kopums), kas paredzēta DB veidošanai un uzturēšanai.

2.1. ALTERNATĪVAS DBVS IZMANTOŠANAI:

Lai izveidotu DB sistēmu uz datora obligāti nav jālieto DBVS, var izmantot arī jebkuru programmēšanas valodu (C++, Java vai citas), kas uztur darbu ar failiem un veidot savus datu failu struktūras un to apstrādes procedūras kā daļu no lietotāja aplikācijas. Tomēr DBVS izmantošana krietni atvieglo DB veidošanas un uzturēšanas procesu, jo liela daļa funkcionalitātes ir jau tajās iebūvēta. Tāpēc mūsdienās tikai gadījumos ar ļoti specifiskām prasībām DB izveidei netiek lietotas DBVS.

2.2. DBVS IZMANTOŠANAS PRIEKŠROCĪBAS UN GALVENĀS FUNKCIJAS:

- Ātrāka un vieglāka DB sistēmu (lietotāja aplikāciju) izstrāde !!!

Lietotāja aplikācijas darbs ar DB izmantojot DBVS parasti notiek augstākā līmenī (mazāk un vienkāršāks kods) nekā tas būtu pa taisno griežoties pie datu failiem. Līdz ar to no aplikāciju izstrādes viedokļa tiek krietni atviegkota šo aplikāciju datu failu apstrādes programmēšana.

- Datu relāciju uzturēšana
- Integritātes nosacījumu uzturēšana
- Dublēšanās kontrole. Visas datu vienības DB tiek glabāti 1 eksemplārā.
- Problēmas, kas rodas no datu dublēšanās:
 - ⇒ Tie paši dati jāievada vairākas reizes, arī veicot datu labošanu tas jādara vairākas reizes.
 - ⇒ Nelietderīgi tiek izmantots disks
 - ⇒ Var rasties nesakritības starp datiem, kam vajadzētu būt vienādiem

Ir pieļaujama kontrolēta dublēšanās, lai uzlabotu dažādu darbību ātrdarbību. Bet šāda dublēšanās ir jāizstrādā ar lielāku vērību apsverot visus iespējamus nesakritību gadījumus.

- vieglāka pielāgojamība (*flexibility*), jo tiek realizēta programmu-datu neatkarība
- Dažādu lietotāja interfeisu uzturēšana:
 - ⇒ programmēšanas interfeiss,
 - ⇒ grafiskais klienta interfeiss (GUI)
 - ⇒ komandrindas interfeiss,

- transakciju uzturēšana
- datu vienlaicīgas pieejas kontrole
- lietotāju pieejas tiesību kontrole
- rezerves kopēšanas (*backup*) un atjaunošanās (*recovery*) uzturēšana.

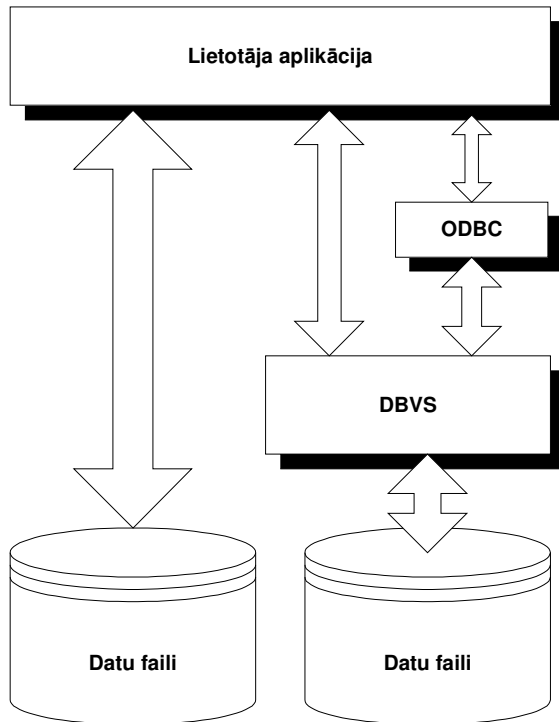
2.3. PROGRAMMU - DATU NEATKARĪBU (*PROGRAM-DATA INDEPENDENCE*)

Ja programma izmanto savu failu apstrādi, tad parasti datu apraksti ir daļa no šīs programmas. (piem. struct vai class deklarācijas). Gadījumā, ja ir nepieciešams mainīt datu failu struktūru, tad ir jāmaina visas programmas daļas, kas griežas pie šī failu. DBVS turpretim darbojas kā starpnieks starp datu apstrādes programmu un DB. DBVS (parasti) nodrošina programmu - datu neatkarību. Tas nozīmē, ka ir iespējams veikt izmaiņas DB struktūrā gan loģiskā (piem. pievienot tabulai jaunu lauku) gan fiziskā (piem. izveidot klāsterus) līmenī un tas nerada vajadzību veikt izmaiņas lietotāja aplikācijā, lai tā arī turpmāk spētu darboties ar šo DB.

2.4. METADATI

Par metadatiem sauc informāciju, kas apraksta pašus datus. Tā ir nepieciešama informācija, lai DBVS spētu sekmīgi strādāt ar datiem, ko lietotājs vēlas glabāt DB. Lielākoties DBVS metadatus glabā pašā DB, bet daļa informācijas tiek glabāta arī sistēmas DB, bet tie nekādā veidā nav saistīti ar konkrēto lietotāju aplikāciju. Pateicoties tam tiek nodrošināta pilnīga programmu - datu neatkarība.

2.5. LIETOTĀJA APLIKĀCIJU SADARBĪBA AR DB



Ir vairāki veidi, kā veikt lietotāja aplikāciju sadarbību ar DB:

- Pa tiešo ar saviem datu failiem (šāda risinājuma trūkumi tika iepriekš apskatīti)
- Izmantojot DBVS piedāvāto API
- Izmantojot DBVS ODBC draiverus.

Pēc šāda modeļa darbojas arī vairums no DB administrēšanas rīkiem (tai skaitā pgAdmin, MySQL Query Browser un SQLYog). Tie nav DBVS sastāvdaļa, bet neatkarīgas programmās, kas izmantojot tīklu pieslēdzas pie DBVS un veic ar to nepieciešamās darbības.

2.6. API

API ir saīsinājums no Aplikāciju Programmēšanas Interfeisus (*Application Programming Interface*). Tās ir funkciju bibliotēkas, kuras ir iespējams izsaukt no C++, Java, PHP un citām valodām un izmantot, lai veiktu ar datiem nepieciešamās darbības. Tomēr API ir katrai DBVS specifisks un līdz ar to netiek nodrošināta aplikācijas neatkarība no DBVS. Ja rodas vēlme nomainīt DBVS, ir jāveic arī daudz izmaiņu lietotāja aplikācijā. Tāpat ir apgrūtināta dažādu aplikāciju izstrādes rīku un DBVS savstarpēja sadarbība, jo jānodrošina saite – katras ar katru. Lai šo situāciju risinātu, tika radīts ODBC. Sīkāku informāciju par katras DBVS API var saņemt no tās izstrādātāja.

2.7. ODBC

Lai vienkāršotu DBVS un klienta aplikāciju sadarbības izstrādi, tika radīts ODBC (*Open DataBase Connectivity*) standarts. Šo standartu izstrādāja Microsoft un šobrīd tas ir ļoti populārs risinājums, kā nodrošināt universālu saiti starp DBVS un klientu aplikācijām. ODBC arī ir API, tikai tā ir vienota visām DBVS un līdz ar to klienta aplikācija, kura pieslēdzas DBVS caur ODBC, kļūst pilnīgi neatkarīga no šīs DBVS. Neko nemainot pašā aplikācijā ir iespējams vienu DBVS un tās ODBC draiveri aizvietot ar citu. Līdzīgi kā ODBC darbojas arī JDBC (*Java Database Connectivity*). Tas ir Java API, kas nodrošina Java aplikāciju piekļūšanu DB. ODBC turpretim ir universāls, no valodas neatkarīgs API.

2.8. SERVERA PROCESS

DBVS uz datora darbojas kā viens vai vairāki procesi daudzuzdevumu operētājsistēmā. Ja DBVS servera process nedarbojas, tad attiecīgi arī DBVS pieslēgties nav iespējams.

Operētājsistēma attiecīgajam procesam nodod visus pieprasījumus, kas griežas pie noteikta porta. PostgreSQL pēc noklusēšanas izmanto 5432 portu, MySQL – 3306, Oracle – 1521, MS SQL server – 1433. Instalējot iespējams norādīt arī citu porta numuru, bet bez īpašas iemesla tā nevajadzētu rīkoties.

Instalējot DBVS PostgreSQL un MySQL operētājsistēmās Windows ir iespējams uzstādīt, vai šie procesi palaižas automātiski pie operētājsistēmas ielādes, vai arī tie lietotājam jāstartē manuāli. Ja DBVS tiek izmantota pietiekami reti, tad ieteicams izvēlēties manuālo procesa startēšanu, jo katru reizi pie sistēmas ielādes netiek ielādēts DBVS process, kas aizņem atmiņu un procesora resursus.


Manuāli procesus iespējams startēt:

1) Postgres

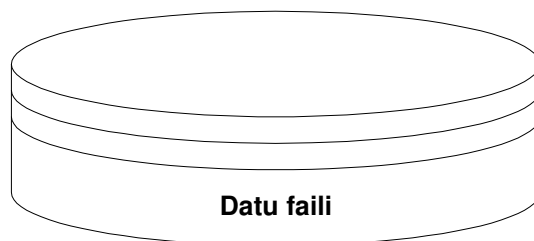
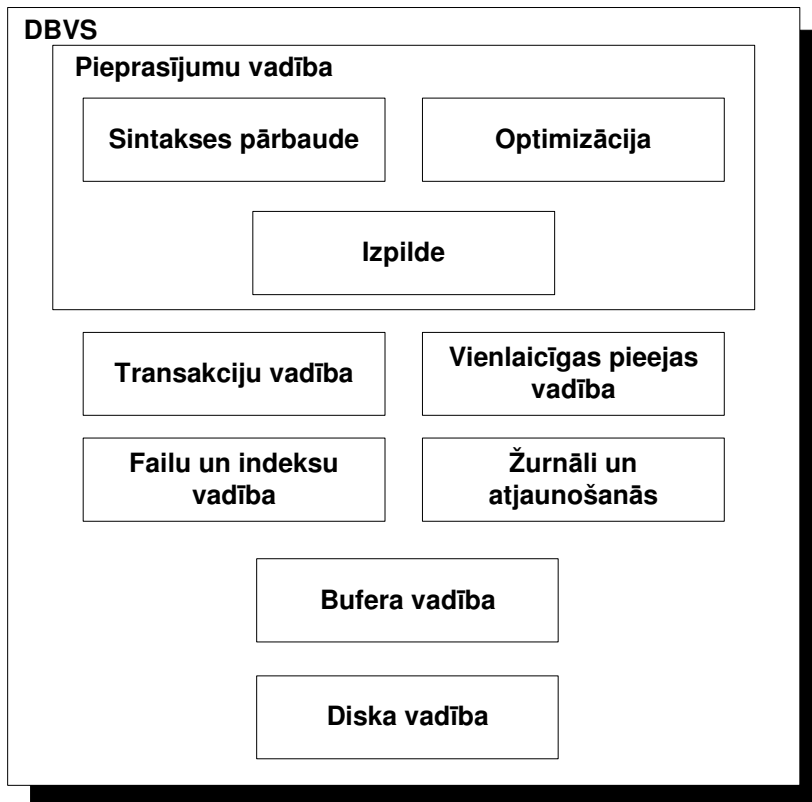
Start/Programs/PostgreSQL/Start service

2) MySQL

Start/Programs/MySQL/MySQL System Tray Monitor

Tiek palaista aplikācija, kas uzdevumjoslās novieto MySQL servera vadības ikonu  . Uzklīkšķinot uz tās ar peles labo pogu atveras izvēlne ar pieejamām izvēlnēm – Startēt instanci (Start instance), apturēt (Stop instance), kā arī citas darbības, kuras sīkāk neapskatīsim.

2.9. DBVS ARHITEKTŪRA



Šis ir mazliet teorētisks skats uz DBVS arhitektūru. Katrā konkrētā sistēmā šī arhitektūra ir savādāka, bet kopīgās pamatiezīmes saglabājas.

DBVS sastāv no vairākiem moduļiem, kam katram ir savas specifiskas funkcijas. Šie moduļi savā starpā sadarbojas, lai nodrošinātu nepieciešamo funkciju veikšanu.

- Sintakses pārbaude (*parser*)

Modulis kā ieejas datus saņem SQL pieprasījumu. Tiek veikta pieprasījuma sintakses pārbaude un pieprasījums tiek pārveidots sintakses pārbaudes koka (*parser tree*) formātā. Tā ir koka datu struktūra, kurā sākotnējā pieprasījuma operatori, operācijas un argumenti tiek izvietoti pa koka lapām. Šis koks tālāk tiek nodots optimizācijas modulim.

- Optimizācija

Tiek veikta šī koka optimizācija. Pēc optimizācijas tiek izveidots cits koks, kuru sauc par pieprasījuma izpildes plānu (*query execution plan*). Šis koks tiek nodots tālāk izpildes modulim.

- Izpilde (*execution engine*)

Modulis, kurš veic pieprasījuma plāna izpildi, sadarbojoties ar citiem moduļiem zemākstāvošiem moduļiem, kam uzticētas katras specifiskas funkcijas izpilde.

- Transakciju vadība

Veic darbības, kas saistītas ar transakciju izpildes vadību.

- Vienlaicīgas pieejas vadība

Veic darbības un kontroles, kas saistītas ar vienlaicīgas pieejas vadīšanu.

- Failu un indeksu vadība

Notiek failu un indeksu vadība ierakstu un lauku līmenī. Tiek veidota attiecīgā faila struktūra un veikta datu apstrāde atbilstoši izvēlētajam faila modelim, kā arī indeksu atjaunināšana veicot izmaiņas datos.

- Žurnāli un atjaunošanās

Šis modulis veic darbību reģistrēšanu žurnālā (*log actions*) un atgriež nepieciešamo informāciju ja nepieciešams veikt atjaunošanos.

- Bufera vadība

Daļa no diska blokiem tiek glabāta buferī, kur tos sauc par lapām. Buferis ir DBVS iekšējā kešatmiņa. Bufera vadība nosaka, kuras lapas ir tikušas izmainītas un jāieraksta diskā, kuras tiek izmantotas un kuras vairs nav vajadzīgas un var tikt atbrīvotas citiem datiem.

- Diska vadība (*storage management*)

Veic ievades/izvades operācijas sadarbojoties ar disku kontrolieri. Veic datu apmaiņu starp buferi un disku datu bloka līmenī.

Sīkāk par šo tēmu un attiecīgo DBVS var iepazīties ar materiāliem:

<http://www.postgresql.org/docs/8.1/interactive/internals.html>

<http://dev.mysql.com/doc/internals/en/index.html>

2.10. MŪSDIENU DBVS:

- Nelielām sistēmām:

⇒ Access,

MS izstrādāta DBVS nelielām vienlietotāja sistēmām. Ļoti populāra, jo tiek ietverta MS Office programmu paketes sastāvā. Tajā ir ietverti arī aplikācijas izstrādes rīki.

<http://office.microsoft.com/en-us/FX010857911033.aspx>

⇒ Interbase un FireBird,

Interbase ir Borland izstrādāta DBVS. Tā sevī neietver aplikāciju izstrādes rīkus. Aplikāciju izstrādei Borland piedāvā Delphi/Builder saimes aplikācijas. Vienā brīdī Borland izplatīja Interbase zem "GPL līdzīgas" licences, bet tad atkal atgriezās pie komerciālas licencēšanas. Tomēr "GPL" variants turpina pastāvēt un attīstīties izmantojot nosaukumu Firebird (nejaukt ar Mozilla Firebird pārlūkprogrammu).

<http://www.borland.com/us/products/interbase/index.html>

<http://www.firebirdsql.org/>

⇒ MySQL.

Ļoti populāra DBVS, tai ir pieejamas dažādas versijas, kas tiek izplatītas izmantojot gan komerciālus gan GPL (General Public License), tātad bezmaksas licencēšanas nosacījumus. Sākotnēji tā tika izstrādāta darbam Unix/Linux OS, bet šobrīd ir pieejama arī Windows vidē. It īpaši populāra MySQL ir veidojot DB, kuras tiek izmantotas Web lapu dinamiskai ģenerēšanai.

<http://www.mysql.com/>

- Vidējām sistēmām

⇒ PostgreSQL,

Vēl viena DBVS, kura tiek izplatīta atbilstoši atvērtā koda licences nosacījumiem (BSD licence).

<http://www.postgresql.org/>

⇒ Sybase.

<http://www.sybase.com/>

⇒ Informix.

<http://www-3.ibm.com/software/data/informix/>

Populārākajiem DBVS izstrādātājiem ir izstrādātās DBVS versijas dažāda lieluma sistēmām. Piemēram Oracle DBVS ir 4 versijas: "Express Edition", "Standard Edition One", "Standard Edition" un "Enterprise Edition", IBM DB2 ir "Personal", "Workgroup", "Enterprise" un vēl citas versijas.

- ⇒ Oracle Standard Edition
- ⇒ MS SQL server
- ⇒ IBM DB2 Workgroup Edition

- Lielām sistēmām

- ⇒ Oracle,
"Oracle Enterprise Edition"

<http://www.oracle.com/ip/deploy/database/oracle10i/>

- ⇒ IBM DB2,
"DB2 Enterprise Server Edition"

<http://www-3.ibm.com/software/data/db2/>

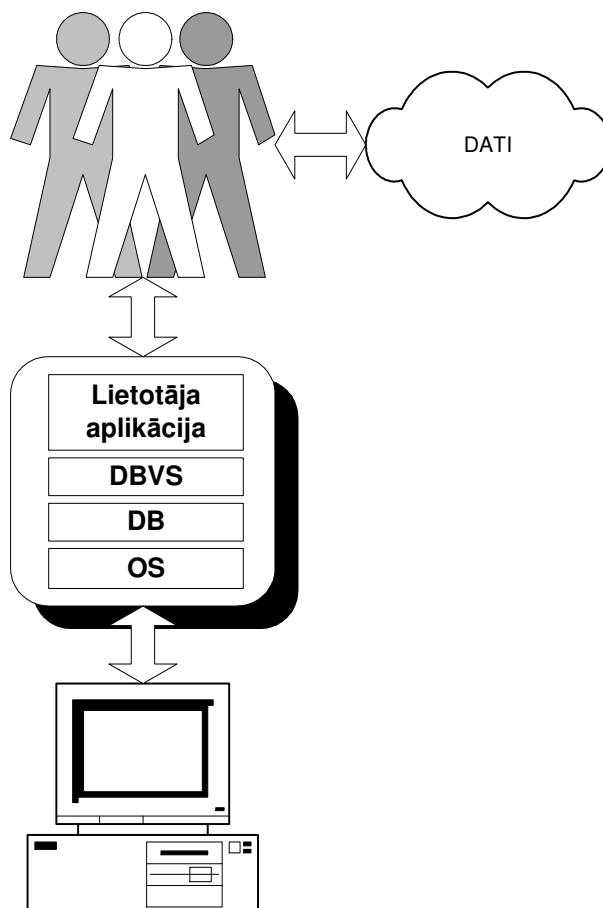
- ⇒ MS SQL server.
"Microsoft SQL Server Enterprise Edition"

<http://www.microsoft.com/sql/default.asp>

3. DB sistēmas

3.1. DB SISTĒMAS

Lai DB sistēma varētu veiksmīgi darboties ir nepieciešamas 4 sastāvdaļas:



- Tehniskais nodrošinājums
- Programmnodrošinājums
 - ⇒ OS - Operāciju sistēma
 - ⇒ DBVS - Datu bāzu vadības sistēma.
 - ⇒ DB – Datu bāze
 - ⇒ Lietotāju aplikācija – programma, kura nodrošina lietotājam vajadzīgās funkcionalitātes veikšanu.
- Dati
- Personāls
 - ⇒ Lietotāji

- ⇒ Administratori
- ⇒ Izstrādātāji.

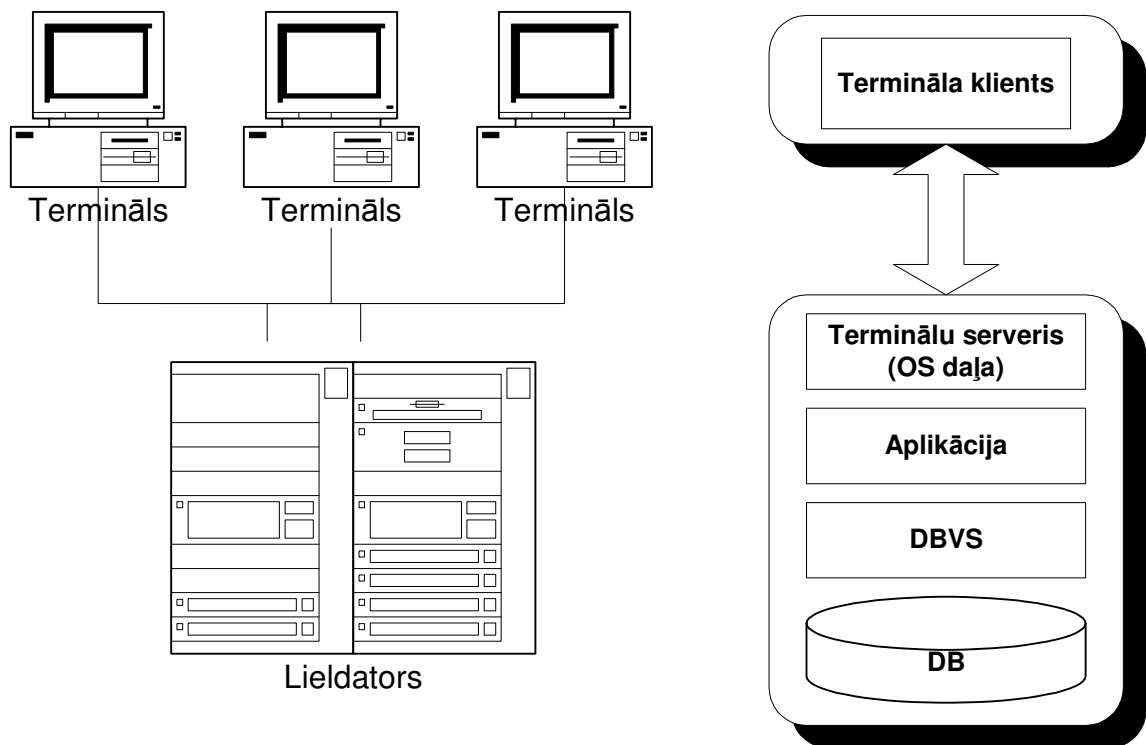
3.2. DB SISTĒMU ARHITEKTŪRA

Iespējamās vairāku veidu arhitektūras:

1. Centralizēta arhitektūra,
2. Klienta – servera arhitektūra,
3. 3 līmeņu arhitektūra,
4. Sadalītās (distributed) DB

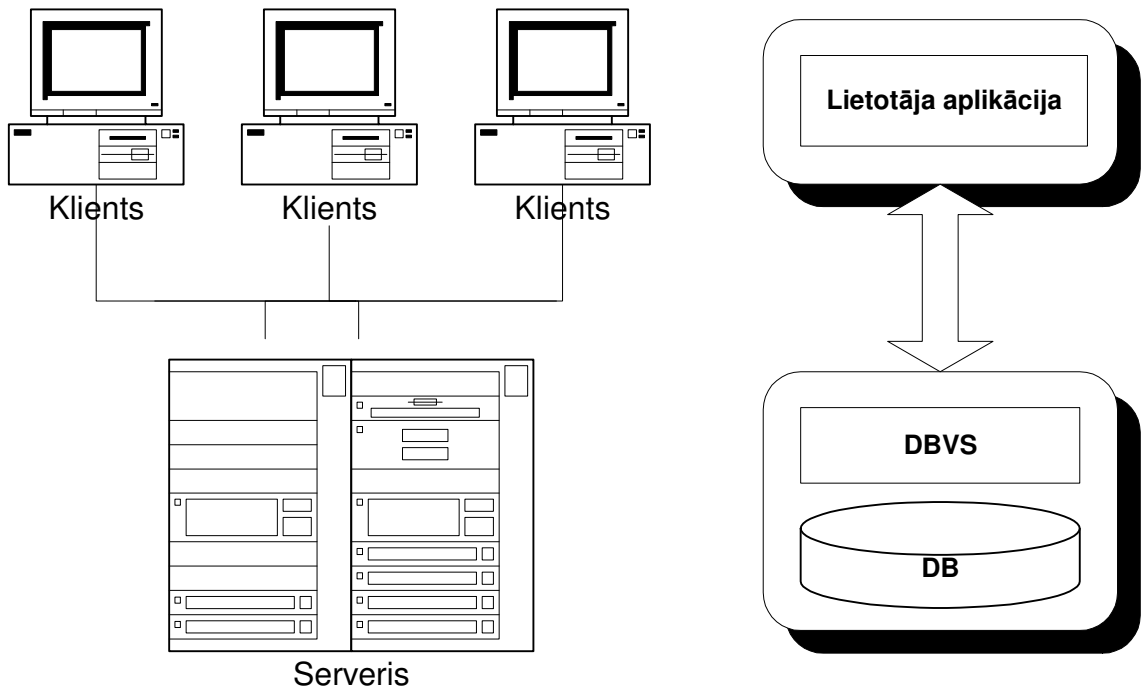
Apskatīsim sīkāk katru no tām:

- Centralizēta arhitektūra.



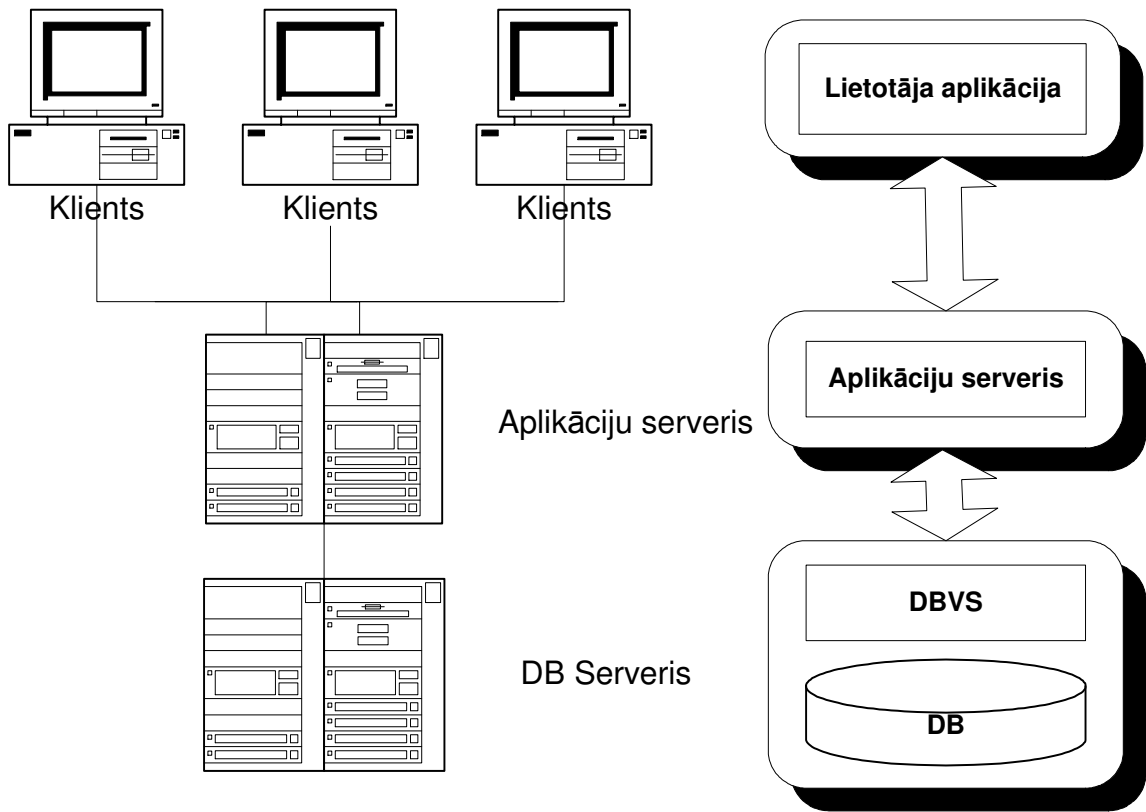
Šāda sistēma tika izmantota sistēmās, kas tika izstrādātas agrākos laikos. To pamatā ir visu sistēmas funkciju (DBVS, lietotāja aplikācija) apvienošana vienā datorā. Šāda arhitektūra veidojās balstoties uz lieldatoru (*mainframe*) arhitektūru, kurā bija viens centrālais dators un lietotāji izmantoja terminālus, lai pieslēgtos šim lieldatoram. Parasti šādi lieldatori darbojas ar Unix operētājsistēmu. Lietotāji attālināti pieslēdzās šim datoram un darināja uz tā aplikācijas, viss apstrādes process notika uz lieldatora. Dators, no kura lietotājs veica pieslēgšanos veica tikai monitora un klaviatūras funkcijas. Ir maldīgs priekšstats, ka šādas sistēmas ir jau izmirušas. Šādu lielu sistēmu izstrādē ir tikuši ieguldīti lieli līdzekļi un sistēmas ir veiksmīgi apliecinājušas sevi darbībā. Mūsdienu datoru arhitektūras mainījušās un jaudas pieaugušas nesalīdzināmi, bet sistēmu savietojamība ir saglabājusies. Līdz ar to uz mūsdienu Unix serverim bieži vien tiek darbinātas programmas, kas izstrādātas pirms 10 gadiem. Tas gan lielākoties attiecināms uz Rietumu pasauli, kur arī pirms 10 gadiem dators nebija nekas eksotisks. Tomēr šobrīd arī vairākās Latvijas bankās izmanto šādas arhitektūras sistēmas, kas ir pirktas no Rietumu izstrādātājiem. Tāpat nesen vienā no lielākajām Austrijas bankām redzēju šādu sistēmu.

- Klienta – servera arhitektūra.



Tā ir datorsistēmas arhitektūra, kur tīkla vidē datu apstrādes process tiek sadalīts starp klientu un serveri. Šī arhitektūra veidojās 90-to gadu sākumā līdz ar personālo datoru (PC) bumu. Personālie datori ieguva arvien lielākas un lielākas skaitļošanas jaudas un to izplatība strauji auga, it sevišķi mazā un vidējā biznesa vidē, kuriem pirms tam datori nebija pieejami. Veidojās arī pirmās DBVS priekš personālajiem datoriem, tādas kā Dbase, Clipper, FoxPro. Tās gan vēl nedarbojās pēc klienta servera arhitektūras. Drīzāk to varētu saukt par centralizētu arhitektūru, kur mazas DB atradās uz katra personālā datora. Attīstoties lokālajiem datortīkliem radās iespēja vairākiem lietotājiem lietot vienu DB. DB tika pārnesta uz tīkla failu serveri, tomēr šis serveris neveica DBVS funkcijas. Tomēr šādi risinājumi drīz vien sastapās ar daudziem ierobežojumiem it sevišķi lielākās sistēmās ar daudziem lietotājiem. Tad kā nākošais solis radās klienta – servera arhitektūra, kur DB sistēmas funkcijas tika sadalītas starp serveri un klientu. Klients ir lietotāja dators (kopā ar programmām), kas veic lietotāja interfeisa izvadi un daļu no datu apstrādes. Serveris ir dators, kas nodrošina klientus (parasti pie servera pieslēdzas daudzi klienti) ar tiem nepieciešamo universālām funkcijām, dotajā gadījumā pieeju DB.

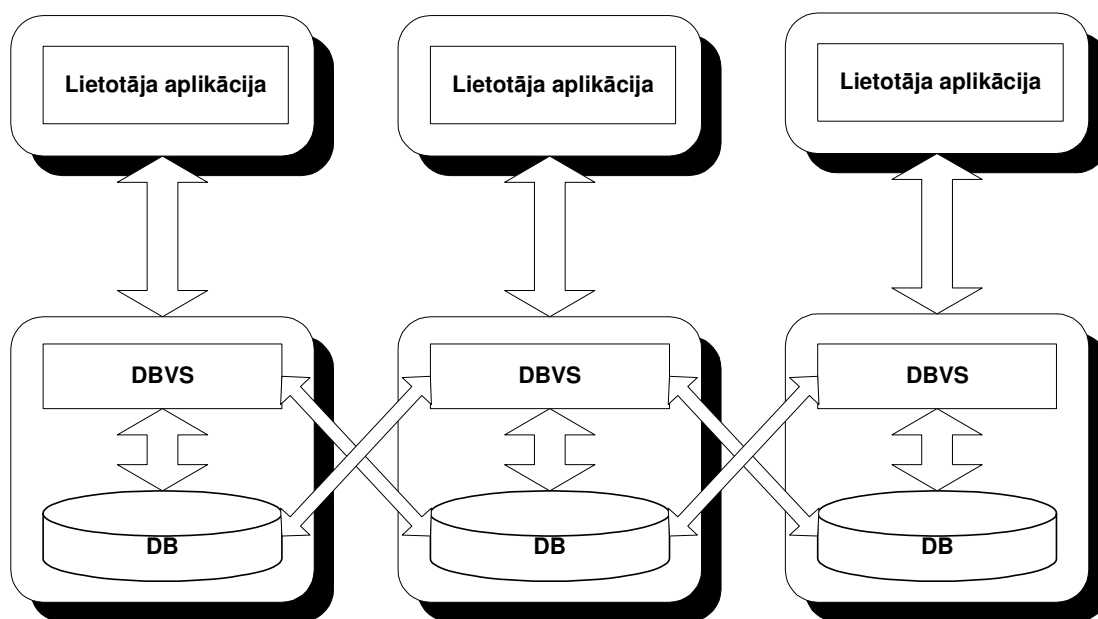
- 3 līmeņu arhitektūra (*three tiers architecture*)



Mūsdienās bieži tiek izmantota arī 3 līmeņu arhitektūra. Šajā gadījumā daļa vai pat visa biznesa loģika no lietotāja aplikācijas tiek pārnesta uz atsevišķu aplikāciju serveri. Klienta aplikācijas funkcijas tiek ierobežotas ar lietotāja interfeisa izvadi un datu attēlošanu. Šādu arhitektūru atbalsta tādas populāras izstrādes platformas kā Java2 Platform Enterprise Edition (J2EE) un Microsoft .NET.

Pie sistēmām ar šādu arhitektūru var pieskaitīt arī visus web saitus, kuri lapu veidošanā izmanto datu bāzes. Šajā gadījumā lietotāja aplikācija ir pārlūkprogramma, bet aplikāciju serveris ir web serveris, kas izpilda PHP (vai kādā citā valodā) rakstītu programmu un šīs izpildes rezultātu nosūta pārlūkprogrammai.

- Sadalītās (*distributed*) DB



Sadalītās DB ir viena loģiska DB, kas fiziski ir sadalīta pa serveriem, kas atrodas dažādās vietās un ar ir savstarpēji saistīti caur datortīklu. Visu šo DB vadību veic DBVS, kas lietotājam padara caurspīdīgu DB sadalījumu pa dažādiem serveriem. Nepieciešamība pēc šādām DB radās, jo daudzām kompānijām ofisi bija izvietoti pa visu valsti/pasauli. Radās nepieciešamība nodrošināt tos ar vienotu DB. Būtu iespējams veidot vienu DB, bet un pieslēgties tai caur datortīklu, tomēr tam bija vairāki trūkumi: lielas komunikāciju izmaksas, liela atkarība no sakaru kanālu kvalitātes. Tā kā lielākā daļa datu ko apstrādāja lietotāji, ir lokālas izcelsmes, tad izdevīgāk tos arī ir glabāt lokāli. Tas dod iespēju samazināt nepieciešamo sakaru kanālu caurlaidību un ļauj lietotājiem strādāt ar daļu no sistēmas funkcijām, arī, ja sakaru kanāli nedarbojas.

Ir vairākas pamatmetodes, kā tiek veidotas sadalītas DB.

⇒ Datu replikācijas

Datu kopijas tiek glabātas vietās, kur ir nepieciešama pieeja šiem datiem. Priekšrocības ir ātra darbība (jo dati glabājas lokāli), trūkums – sarežģīti veikt izmaiņas. Tāpēc šādi tiek dublēti relatīvi statiski dati, kurus izmainīt vajag ļoti reti un ko vajag tikai lasīšanas operācijām.

⇒ Horizontāla sadalīšana

Kad vienas un tās pašas tabulas ieraksti var glabāties vairākās vietās. Piemēram dati par katru klientu tiek glabāti ofisā, kurā viņš tiek apkalpots.

⇒ Vertikāla sadalīšana

Raksturīga kompānijām, kur dažādi ofisi veic dažādas funkcijas. Piemēram dati par marketingu tiek glabāti vienā ofisā, dati par noliktavu citā un par ražošanu veļ citā.

Var tika izmantotas arī dažādas kombinācijas no šīm metodēm.

- Decentralizētas DB

Decentralizēta DB ir vairākas DB ar vienu loģisko struktūru, kas atrodas uz dažādiem datoriem un nav savā starpā saistītas caur datortīklu. Šādu DB par nevar uzskatīt par vienu DB, bet gan par neatkarīgu DB apvienojumu.

3.3. DB SISTĒMU IEDALĪJUMS PĒC MĒRĶIEM UN LIETOTĀJU SKAITA:

- Personālās (*personal*) DB sistēmas

Veidotas, lai apkalpotu tikai vienu lietotāju. Tiek plaši lietotas, jo palīdz uzlabot darbinieka konkrēto uzdevumu veikšanu. Pie tam šie uzdevumi parasti nav raksturīgi daudziem darbiniekiem uzņēmumā. Piemēram sekretārei BD ar informāciju par svarīgāko klientu jubilejām, lai varētu tos apsveikt. Parasti tās veido pats lietotājs vai arī kompānijas IT speciālisti. Šādām DB ir vienkārša struktūra un tikai ļoti ierobežota funkcionalitāte. Toties arī izstrāde un uzturēšana ir vienkārša.

- Darba grupu (*workgroup*) DB sistēmas

Darba grupu DB ir neliels lietotāju skaits, kas parasti nepārsniedz 25. Tās tiek veidotas ar mērķi veicināt lietotāju sadarbību kādā konkrētā projektā.

- Nodaļu (*department*) DB sistēmas

Šīs DB sistēmas parasti apkalpo 10-100 lietotājus. Tās mērķis ir nodrošināt kādas nodaļas uzdevumu efektīvu veikšanu. Piemēram personāla daļā izmanto DB ar visu darbinieku datiem, algu aprēķiniem utt., tirdzniecības daļa uztur klientu DB.

- Uzņēmumu (*Enterprise*) sistēmas

Lielākā daļa no sistēmām ir tieši šajā līmenī. Šīs sistēmas apvieno sevī daudzas apakšsistēmas, kas savstarpēji mijiedarbojas. Šīs apakšsistēmas var būt gan nodaļu gan darba grupu līmenī. Uzņēmumu sistēmās parasti netiek ietvertas personālās DB sistēmas.

3.4. DB VAR IEDALĪT VAIRĀKOS VEIDOS:

- Tradicionālās datu bāzes ar teksta un ciparu informāciju

Tādu ir liels vairums no veidojamajām DB, tomēr to dominējošā loma lēnām samazinās,

- Datu noliktavas (*data warehouses*) un OLAP (*Online Analytical Processing*)

Datu noliktava ir subjektorientēta, integrēta, ar laika dimensiju (time-variant) datu kolekcija, kas izveidota vadības lēmumu pieņemšanas atbalstīšanai. Šie risinājumi kļūst ļoti populāri pēdējos gados. OLAP ir rīku kopa, kas ļauj grafiskā vidē veikt datu analīzi dažādos griezumos. Sīkāk mēs šos risinājumus apskatīsim vienā no pēdējām lekcijām.

- Multimēdiu datu bāzes

Tās ir DB, kas atļauj lietotājiem glabāt dažādu tipu multimēdiu datus, kā attēlus, skaņu, video, kā arī ļauj veidot pieprasījumus un iegūt rezultātus atkarībā no multimēdiu objektu satura.

- Ģeogrāfiskās IS

Šādas DB tiek izmantotas, lai apkopotu, glabātu, modelētu un analizētu informāciju, kas apraksta ģeogrāfiskus objektus. Atšķirībā no citām sistēmām ir ļoti svarīgs datu grafiska attēlošana un arī rediģēšana.

3.5. PĀPILDUS LIERATŪRA

Iepazīties ar dokumentācijas sadaļām:

Database Management Systems 3-21.lpp

PostgreSQL 8.1.0 Documentation 1-20 lpp.;

3.6. PAŠKONTROLES JAUTĀJUMI.

1. Kas ir datu bāze.
2. No kā sastāv datu bāze.
3. Kas ir Datu bāzu vadības sistēma (DBVS).
4. Kādas ir DBVS galvenās funkcijas.
5. Kādas ir populārākās mūsdienu DBVS.
6. Kas ir datu bāzu sistēmas.
7. Kādas ir datu bāzu sistēmu arhitektūras, ar to tās atšķiras.

4. DB veidošana

4.1. MYSQL

```
CREATE DATABASE [IF NOT EXISTS] db_nosaukums
```

Izveido katalogu datu direktoriņā, kurš tiek uzstādīts konfigurācijas failā my.cnf. Veidojot jaunu datu bāzi tiek izveidots tikai tukšs katalogs, kurā pēc to izveidošanas tiks glabāti faili ar tabulu datiem .

```
DROP DATABASE [IF EXISTS] db_nosaukums
```

Dzēš datu bāzi. Ja direktoriņā atrodas tikai DB faili, tad tiek izdzēsts arī direktorijs, bet ja tajā ir vēl arī citi faili, tad tiek izdzēsti tikai DB faili.

4.2. POSTGRESQL

```
CREATE DATABASE db_nosaukums  
[WITH [LOCATION = 'db_celš']  
[TEMPLATE = template]  
[ENCODING = encoding]]
```

```
DROP DATABASE db_nosaukums
```

4.3. TABULU VEIDOŠANA

Tabulu veidošanai tiek izmantota SQL komanda CREATE TABLE. MySQL un PostgreSQL komandas formāts mazliet atšķiras, tomēr pamatiezīmes tiek saglabātas. Tāpēc pirms tabulu izveides ieteicams dokumentācijā iepazīties ar sintaksi. It sevišķi tas attiecas uz gadījumiem kad tiek izmantotas kādas paplašinātās sistēmas iespējas, (specifiski datu tipi, salikti ierobežojumi utt.)

Komandas CREATE TABLE vispārīgā forma:

```
CREATE TABLE tabulas_nosaukums (  
kolonnas_nosaukums datu_tips [DEFAULT vērtība] [kolonnas_ierobežojumi],  
...  
[tabulas_ierobežojumi]  
)
```

kolonnas_ierobežojumi un *tabulas_ierobežojumi* satur NOT NULL, UNIQUE, PRIMARY KEY, CHECK un citus ierobežojumus, kas nosaka kādas vērtības var un kādas nevar pieņemt dotās kolonnas lauki. Sīkā ierobežojumi tiks apskatīti vienā no nākamajām lekcijām.

MySQL un PostgreSQL šīs komandas darbojas līdzīgi, tomēr jāatceras, ka MySQL neuztur datu integritātes pārbaudes (sīkāk par to skatīsimies aiznākamajā lekcijā).

Piemēram:

```
CREATE TABLE atzimes (
  atzime int,
  nosaukums CHAR(15) NOT NULL,
  apraksts varchar(255),
  CONSTRAINT atzimes_atzime_key1 UNIQUE (atzime),
  CONSTRAINT atzimes_atzime1 CHECK ((atzime >= 0) AND (atzime <= 10))
) ;
```

Lai izmainītu tabulas struktūru, var izmantot komandu ALTER TABLE. Šai komandai ir daudzas variācijas atkarībā no tā, kādas izmaiņas vēlams panākt.

```
ALTER TABLE tabulas_nosaukums ADD COLUMN
kolonnas_nosaukums datu_tips [DEFAULT vērtība], [kolonnas_ierobežojumi]
```

Komanda DROP TABLE *tabulas_nosaukums* dzēš tabulu no DB.

4.4. DATU TIPI

Datu tipi var tikt iedalīti 3 daļās:

- Skaitliskie datu tipi

ANSI SQL	MySQL	PostgreSQL	pieļaujamās vērtības	izmērs
-	tinyint	-	-127 līdz +128	1 baiti
+	smallint	smallint	-32768 līdz +32767	2 baiti
-	mediumint	-	-8388608 līdz +8388607.	3 baiti
+	int;integer	integer	-2147483648 līdz +2147483647	4 baiti
-	bigint	bigint	nu ļoti lielas	8 baiti
+	decimal;numeric	decimal;numeric	Mainīgas, bez ierobežojumiem	mainīgs
+	float	real	-3.402823466E+38 līdz -1.175494351E-38,	4 baiti
+	double;rael	double precision	-1.7976931348623157E+308 līdz -2.2250738585072014E-308,	8 baiti
-	-	serial	1 līdz 2147483647	4 baiti
-	-	bigserial	1līdz 9223372036854775807	8 baiti

Serial un bigserial nav parasti datu tipi, bet gan pieraksta veids mehānismam, kas ļauj iešķirt ierakstam unikālu identifikatoru. Sīkāk tas tiks apskatīts pie DB objektiem – secībām (*sequences*)

- Datuma/laika datu tipi

ANSI SQL	MySQL	PostgreSQL	pieļaujamās vērtības	izmērs
-	datetime	-	1000-01-01 00:00:00 līdz 9999-12-31 23:59:59	8 baiti
?	date		1000-01-01 līdz 9999-12-31(solis-diena)	3 baiti
?		Date	4713 p.m.e. līdz 32767 (diena)	4 baiti
?	time		00:00:00 - 23:59:59 (sekunde)	3 baiti
?		Time	00:00:00.00-23:59:59.99	8 baiti

-	year		0000-9999	1 baits
?	timestamp		1970 līdz 2037 (sekunde)	4 baiti
?		timestamp	4713 p.m.ē līdz 1465001 (1 mikrosekunde)	8 baiti
+	-	interval	-178000000 līdz 178000000 gadiem (solis - 1 mikrosenkunde)	12 baiti

- Teksta datu tipi

ANSI SQL	MySQL	PostgreSQL	pieļaujamās vērtības	izmērs
+	char (n)	char (n)	n simbolu garš teksts	n
+	varchar (n)	varchar (n)	līdz n simbolu garš teksts	n+4
-		text	neierobežota garuma teksts	
-	tinytext		līdz 255 simbolus garš teksts	garums+1
-	text		līdz 65535 simbolus garš teksts	garums+2
-	mediumtext		līdz 16777215 simbolus garš teksts	garums+3
-	longtext		līdz 4294967295 simbolus garš teksts	garums+4

Daudzās DBVS no ātrdarbības viedokļa ir labāk izmantot char nevis varchar, bet Postgres šāda priekšrocība nav, līdz ar to jebkurā gadījumā ieteicams lietot varchar.

- Citi datu tipi

⇒ PostgreSQL

PostgreSQL	pieļaujamās vērtības	izmērs
money	-21474836.48 līdz +21474836.47 (novecojis)	4 baiti
bytea	bināro vērtību rinda	garums +4
boolean	true/false, 1/0, t/f, y/n	1 baits
point	ģeometrisks objekts	16 baiti
line	ģeometrisks objekts	32 baiti
box	ģeometrisks objekts	32 baiti
lseg	ģeometrisks objekts	
path	ģeometrisks objekts	
polygon	ģeometrisks objekts	
circle	ģeometrisks objekts	24 baiti
inet	tīkla vai datora IP adrese /maska	12 baiti
cider	tīkla IP adrese /maska	12 baiti
macaddr	MAC adrese	6 baiti
bit	binārās vērtības	
bit varying	mainīga garuma binārās vērtības	

⇒ MySQL

MySQL	pieļaujamās vērtības	izmērs
enum	Veidojot kononnu ar šo tipu tiek nodefinētas pieļaujamās vērtības	

set	Rindas tips, kuŗš var pieņemt vienu vai vairākas iepriekš definētas vērtības	
tinyblob	binārais objekts	
blob,	binārais objekts	
mediumblob	binārais objekts	
longblob	binārais objekts	

- Datu tipu konvertācija

Lielu daļu no nepieciešamajām datu tipu konvertācijām PostgreSQL veic automātiski. Tomēr dažreiz sistēmas izstrādātājs vēlas datu tipu konvertāciju veikt savādāk, nekā PostgreSQL to veic automātiski.

Lai veiktu datu tipu konvertāciju var tikt izmantoti vairāki paņēmieni:

⇒ Konstrukcija ::

Šī konstrukcija var tikt lietota lai norādītu konvertācijas operāciju, kas jāveic ar datu tiem.

Piemēram:

```
Select skaits::integer, skaits::varchar from tabula
```

⇒ Funkcija cast()

Funkcija cast(*dati as tips*) darbojas līdzīgi kā ::.

Piemēram:

```
Select Cast(skaits as integer) from tabula
```

4.5. PAPILDUS LIERATŪRA

Iepazīties ar dokumentācijas sadaļām:

PostgreSQL 8.1.0 Documentation 77-98.lpp.

MySQL 5.0 reference manual 11. Data Types.

4.6. PAŠKONTROLES JAUTĀJUMI.

1. Ar kādu SQL komandu veido datu bāzi.

2. Ar kādu SQL komandu veido tabulas.

5. Sistēmas izstrāde un DB projektēšana.

5.1. APLIKĀCIJU IZSTRĀDES POSMI:

- Lietotāja prasību identificēšana,

Process, kurā sistēmas izstrādātāji mēģina noskaidrot, kādas funkcijas klienti vēlas redzēt savā jaunajā sistēmā.

- Sistēmas analīze,

Tiek veikta klienta patreizējās biznesa sistēmas analīze, lai noskaidrotu esošo sistēmas stāvokli un iespējas to integrēt izstrādājamā sistēmā, noskaidrot iespējamās pretrunas un citus faktorus, kas ietekmē sistēmas izstrādi.

- Loģiskā projektēšana,

Tiek izstrādāti biznesa modeļi, datu plūsmu diagrammas, ER modeļi, lietotāja ekrānu un atskaišu formas.

- Fiziskā projektēšana

Balstoties uz loģiskās projektēšanas posmā radītiem dokumentiem, tiek veikta DB fiziskā projektēšana, programmēšana, testēšana un dokumentācijas rakstīšana

- Ieviešana

Sistēmas ieviešana klienta uzņēmumā un akcepttestēšana.

- Uzturēšana

Kļūdu novēršana un modifikāciju veikšana sistēmā pēc klienta pieprasījuma.

Šobrīd DB kursā sīkāk pievērsīsimies loģiskās projektēšanas daļai, kurā tiek veikta ER modeļa izstrāde, normalizācija un fiziskās projektēšanas daļai, kurā notiek reālās DB veidošana.

5.2. DB IZVEIDES POSMI:

- Loģiskā projektēšanas posmā:

- ER diagrammu veidošana,
- ER diagrammu pārveidošana par relācijām,
- Relāciju normalizācija,
- Ierobežojumu noteikšana.

- Fiziskā projektēšanas posmā:

- Tabulu veidošana izmantojot SQL vai GUI

5.3. ER DIAGRAMMU VEIDOŠANA

ER (*Entity – relationship*) modelis ir konceptuāls datu modelis, kas attēlo datus un to saistības.

ER diagramma ir ER modeļa grafisks attēlojums.

ER diagrammas tiek ļoti bieži izmantotas veicot DB konceptuālo izstrādi. Tas ir ērts veids, kā aprakstīt un saprast analizējamo sistēmu. Tāpat šādas diagrammas atvieglo sistēmas izpratni citiem izstrādātājiem, kas nav piedalījušies tieši pašā sistēmas analīzes procesā un ir ērts veids kā komunicēties ar sistēmas pasūtītājiem un aprakstīt tiem plānotās datu struktūras.

Daudzi mūsdienu programmatūras rīki uztur ER diagrammu veidošanu un tālāk automātisku DB struktūras ģenerēšanu no šīm diagrammām. Tas krietni paātrina DB izstrādes procesu. Kā populārākos no šādiem rīkiem var minēt Oracle Designer, Erwin. Arī Latvijā (LU) ir izstrādāts vizuālās modelēšanas rīks GRADE.

ER diagramma sastāv no entītijām (jeb būtībām), saitēm un atribūtiem.

- Entītijas

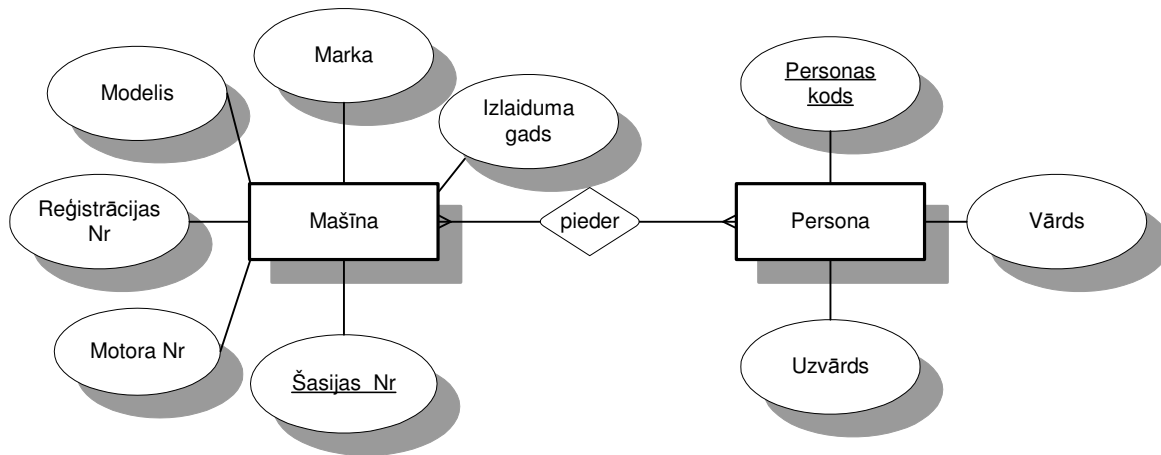
Entītija (*entity*) ir datu objekts, kas reprezentē kādu analizējamās sistēmas objektu, par kuru tiek apkopta informācija. Parasti šāds analizējamās sistēmas objekts ir persona, vieta vai lieta. ER diagrammās entītijas tiek attēlotas kā četrstūri ar tajos ierakstītiem nosaukumiem. Entītijas instance ir dotā objekta konkrēts viensums (gabals, eksemplārs). Piemēram, veidojot bibliotēkas sistēmu būs nepieciešama entītija "Grāmatas". Savukārt šīs entītijas "Grāmatas" instance būs viena grāmata, piemēram "Ievads datu bāzu sistēmās".

- Saites

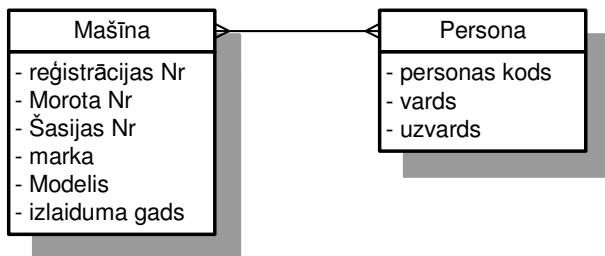
Saite reprezentē reālās pasaules asociācijas, kas saista divas entītijas. Tām nav atbilstoša objekta reālajā pasaulē. Saites tiek attēlotas kā līnijas (iespējams ar rombu vidū), kas savieno 2 (vai vairākas) entītijas. Rombā ierakstīta asociācija, kura saista šīs entītijas.

- Atribūti

Atribūts ir entītijas īpašība, kura glabā kādu informāciju par entītiju, kurai piesaistīts šis atribūts. Izšķir 2 veidu atribūtus – identificējošos un aprakstošos. Identificējošie atribūti tiek izmantoti lai viennozīmīgi identificētu kādu entītijas instanci. Piemēram personas kods studentiem, kas izmanto grāmatas bibliotēkā. Aprakstošais atribūts glabā informāciju, kas apraksta doto entītijas instanci. Piemēram, grāmatas nosaukums. Ir iespējama situācija, kad atribūtam ir gan identificējošs gan aprakstošs raksturs. Atribūti tiek attēloti kā ovāli, kas saistīti ar entītiju ar līniju bez romba vidū. Šīs līnijas nav saites, saites savā starpā saista tikai entītijas.



Mūsdienās ir vērojama tendence, ka ER diagrammu klasiskā attēlošanas metodoloģija tiek aizvietota ar UML metodoloģiju, jo tā ir kompaktāka un labāk pārskatāma.



5.4. UNIFIED MODELING LANGUAGE (UML)

ULM līdzīgi kā ER modeļi ir domāti grafiskai sistēmas apraksta izveidošanai. Atšķirībā no ER modeļiem, kas orientēti uz DB izveidi, UML ir vairāk domāta objektorientētu sistēmu izstrādes konceptuālo modeļu veidošanai. Tā ietver sevī ne vien klašu diagrammas (kas pēc savas būtības ir līdzīgas ER diagrammām, bet arī no citu veidu diagrammām, kuras attēlo sistēmu citos līmeņos. (darbību diagrammas, komponentu diagrammas, stāvokļu diagrammas).

5.5. ER DIAGRAMMU PĀRVEIDOŠANA PAR RELĀCIJĀM

Šis solis nav obligāts. Ja sistēmas veidotājs labi orientējas datu modeļos un spēj veikt normalizāciju tieši ER modelī. Tāpat UML stila ER modeļu pieraksts jau ir tuvu relāciju attēlojumam un tad nekāda papildus pārveidošana nav nepieciešama.

- Atslēgas

Kandidātatslēga (*candidate key*) – atribūts vai atribūtu kombinācija, kas unikāli identificē rindiņu relācijā, neatkaroties, un no kā nav iespējams iegūt citu kandidātatslēgu izņemot no šīs atslēgas kādu atribūtu.

Primārā atslēga (*primary key*) – viena no kandidātatslēgām, kas reāli tiek izmantota rindiņas unikālai identificēšanai.

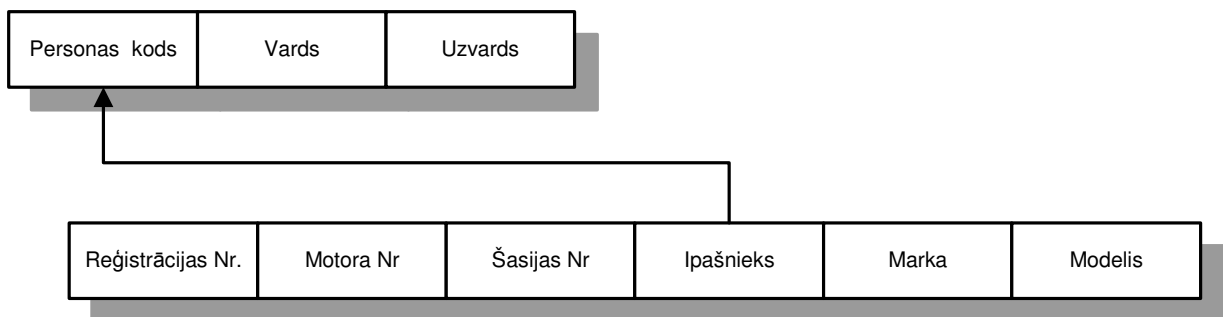
Automašīnas

Reģistrācijas Nr	Šasijas Nr	Motora Nr	Marka	Modelis	Gads
AA-2222	QWQSAS	13413242	AUDI	A4	2000
BB-3333	QW98089	54353532	BMW	740	2001
CC-9999	OIU67776	345345345	MAZDA	6	2002

Šajā gadījumā gan Reģistrācijas Nr, gan Šasijas Nr, gan Motora Nr ir kandidātatslēgas un var tikt lietotas kā primārās atslēgas.

Visu šo trīs atribūtu apvienojums arī unikāli identificē ierakstus, bet izņemot no šīs atslēgas jebkuru atribūtu, mēs atkal iegūstam atslēgu, kas unikāli identificē ierakstus. Līdz ar to šo atribūtu apvienojumu nevar uzskatīt par kandidātatslēgu.

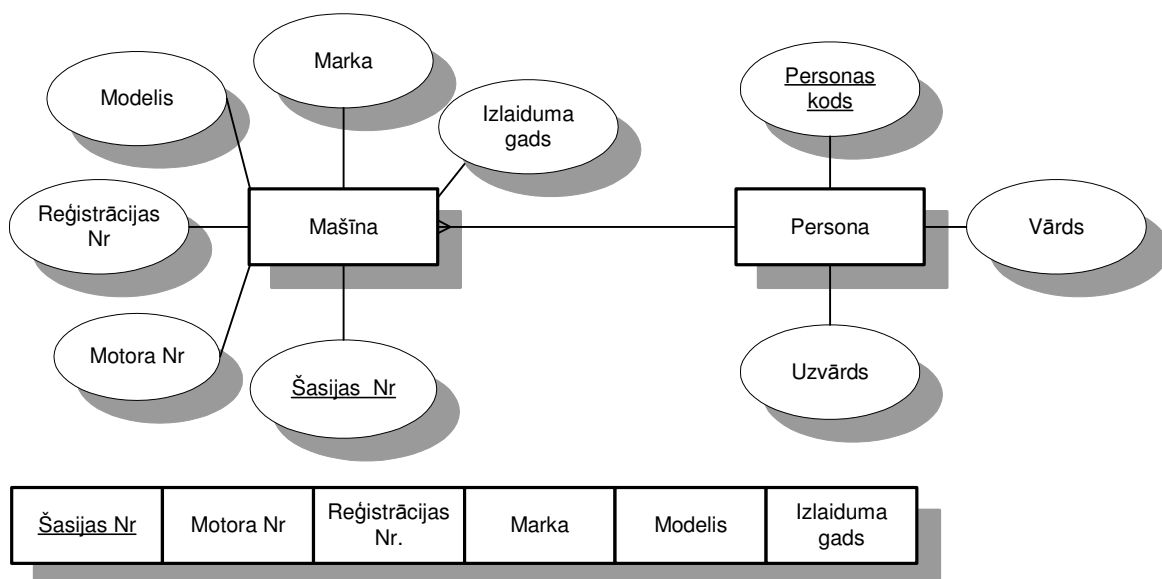
Ārējā atslēga (foreign key) – atribūts(-ti), kura vērtības ir kādas citas relācijas atribūts, un kas tiek izmantota relāciju saišu veidošanai starp šīm relācijām.



ER diagrammu pārveidošanas uz relācijām process var iedalīt vairākos soļos:

- No entītijām izveido relācijas

Katra ER diagrammas entīcija tiek pārveidota par relāciju. Katrs entīcijas atribūts kļūst par relācijas atribūtu. Entīcijas identifikators kļūst par relācijas primāro atslēgu.



<u>Personas kods</u>	Vards	Uzvards
----------------------	-------	---------

- Izveido binārās relāciju saites

Relāciju saišu izveidošana atkarīga gan no saišu kardinalitātes, gan arī no to pakāpes.

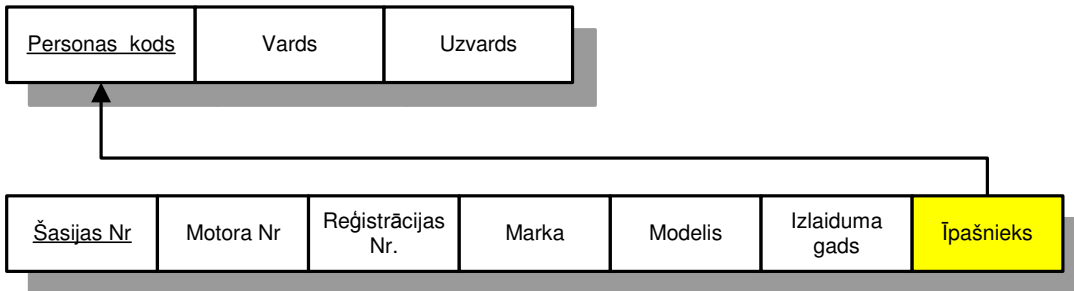
Kardinalitāte ir instanču skaits, kas var tikt piekārtota vienai saitei.

Saišu pakāpe ir cik entītijas ir saistītas ar šo saiti. Binārās saites saista 2 entītijas.

Atkarībā no kardinalitātes binārās saites var iedalīt:

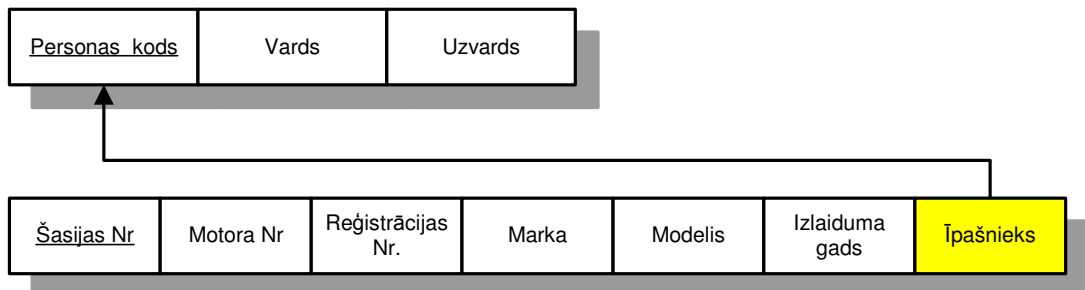
⇒ 1:N

Kad izveidotas relācijas katrai no entītijām, kas saistītas ar relāciju saiti, paņem primāro atslēgu no relācijas, kas atrodas saites 1: pusē un pievieno šo atslēgu relācijai, kas atrodas :N pusē kā ārējo atslēgu.

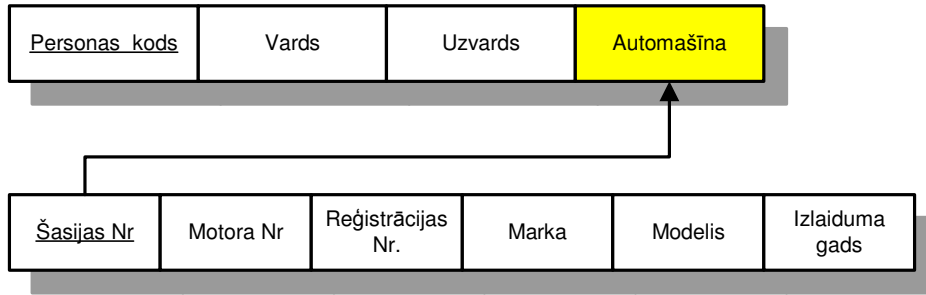


⇒ 1:1

Šī saite var tik uzskatīta par 1:N saites speciālu gadījumu. Arī šeit jāņem primārā atslēga no vienas relācijas un tā kā ārējā atslēga jāpievieno otrai relācijai. Bet tā kā abās saitēs kardinalitāte ir 1, tad saites virziens nav būtisks un līdz ar to arī ārējo atslēgu iespējams pievienot jebkurai no šīm relācijām. Šajā gadījumā jāskatās, kura no relācijām ir obligāta relāciju saitē un no tās vajag ņemt primāro atslēgu, tādā veidā izvairoties no iespējamās Null vērtības glabāšanas ārējā atslēgā.

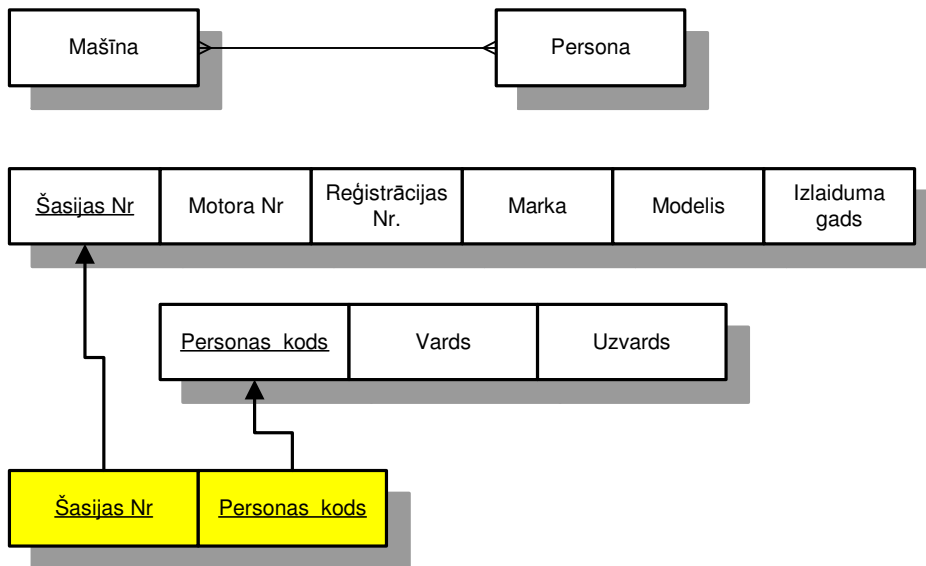


vai



⇒ M:N

Lai realizētu šādas saites, ir nepieciešams veidot jaunu relāciju. Tās vienīgie atribūti ir abu relāciju, kuras saistītas ar šo relāciju saiti, primārās atslēgas.

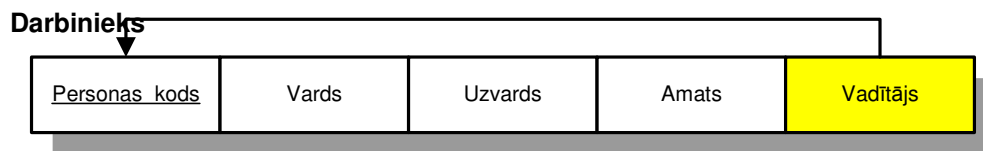
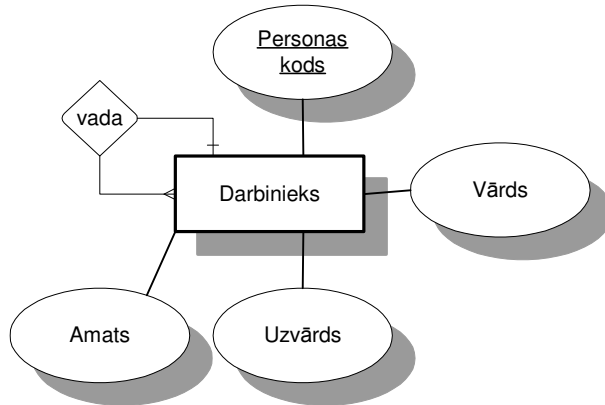


- Izveido unārās relāciju saites

Unārās relāciju saites ir saites, kas savieno vienas entītijas instances.

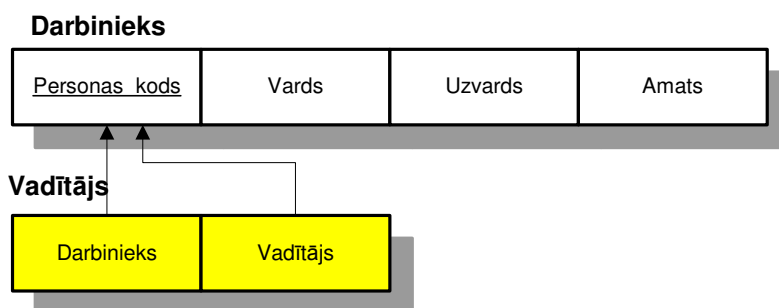
⇒ 1:N; 1:1

Relācijai pievieno ārējo atslēgu, kas ir šīs pašas relācijas primārā atslēgu.



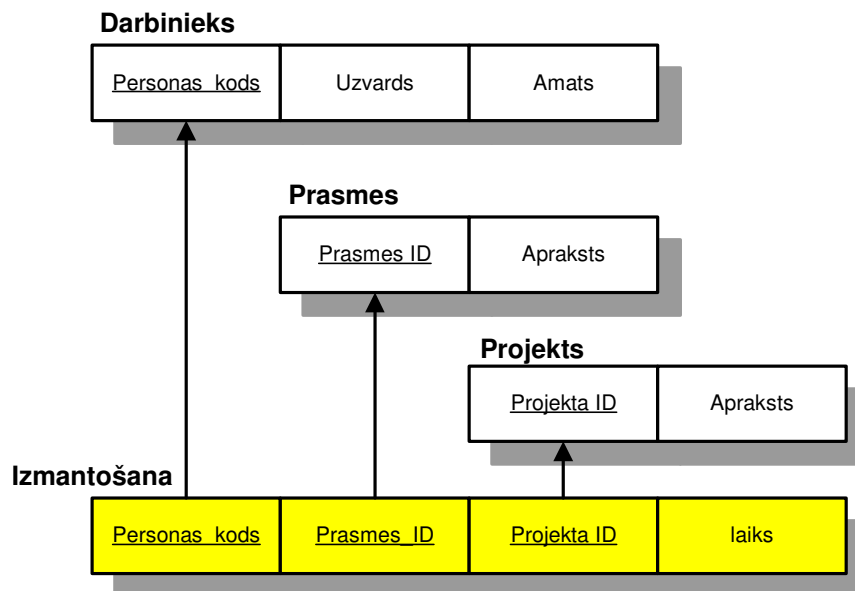
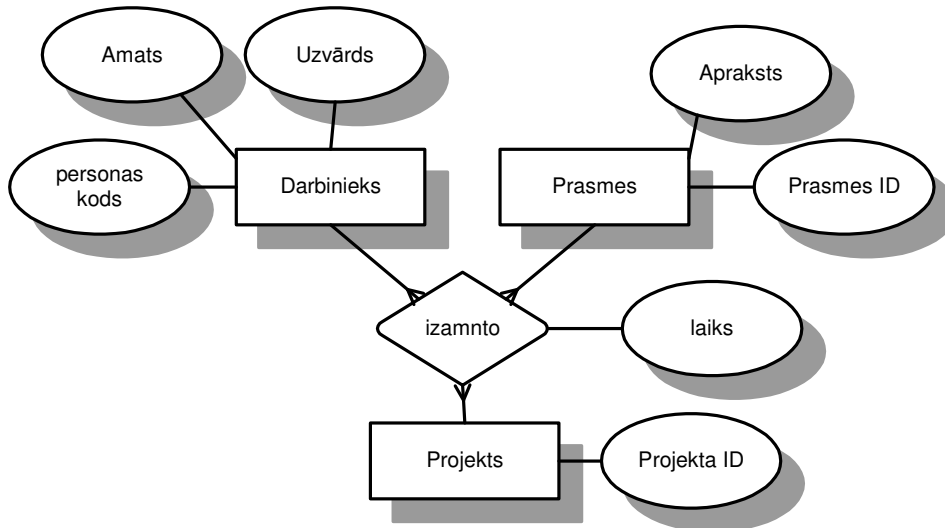
⇒ M:N

Jāveido papildus relācija, kurā ir divas ārējās atslēgas, kuras norāda uz vienu un to pašu primāro atslēgu (bet katra uz citu instanci) un viena no tām reprezentē saites vienu pusi – otra – otru.



- Izveido daudzkārtīgas relāciju saites

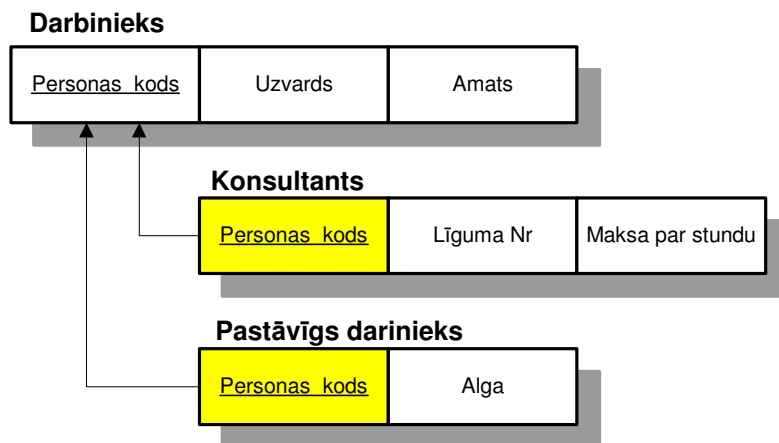
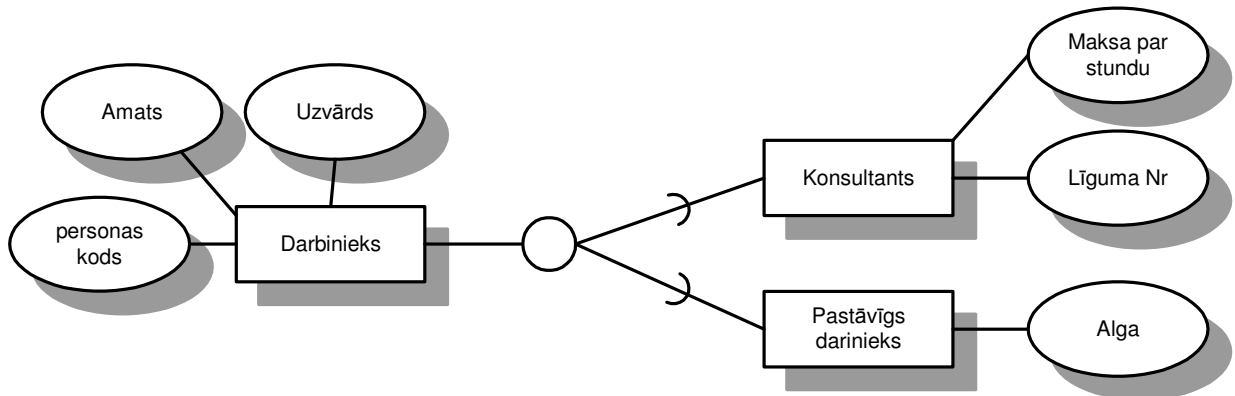
Lai realizētu šādu ER diagrammu, arī jāveido jauna relācija, kurā kā ārējās atslēgas ietilpst visu ar saiti saistīto relāciju primārās atslēgas. Ja šo atslēgu apvienojums unikāli neidentificē ierakstu, tad jāpievieno vēl kāds atribūts, kurš to nodrošina.



- Izveido virstipu/apakštīpu relācijas

Relāciju datu modelis šobrīd tiešā veidā neuztur virstipu/apakštīpu relāciju saites. Bet ir vairāki veidi, kā relāciju datu modelī to iespējams izdarīt. Visbiežāk izmantotais no tiem ir šāds:

Izveido atsevišķas relācijas virstīpam un apakštīpiem. Visus atribūtus, kas ir kopīgi visiem virstīpā ietilpstošajiem apakštīpiem pievieno virstīpa relācijai, bet katra apakštīpa relācijai pievieno tos atribūtus, kas ir unikāli katram apakštīpam pievieno attiecīgā apakštīpa relācijai. Katrai apakštīpa relācijai pievieno ārējo atslēgu, kas norāda uz virstīpa ārējo atslēgu.



5.6. NORMALIZĀCIJA

Normalizācija ir formāls process, kurā atribūti (kolonas) sadalītu pa relācijām, (tabulām) tā, lai uzlabotu datu kvalitāti. Sākumā veidojot ER modeļus tas tiek darīts intuitīvi, un normalizācija palīdz pārbaudīt, vai tas ir izdarīts pareizi un ja ne, tad dod risinājumus, ko mainīt, lai uzlabotu datu shēmu. (DB loģisko modeli). Varētu teikt, ka normalizācija ir relācijas ar anomālijām dekompozīcijas (sadalīšanas) process, kā rezultātā tiek radītas vairākas mazākas, labāk strukturētas relācijas. Normalizācija ir vajadzīga, lai uzlabotu datu kvalitāti un izvairītos no datu anomālijām.

- Anomālijas

Anomālija – kļūda vai datu pretrunīgums, kas rodas, kad lietotājs mēģina izmainīt (update) tabulu, kura satur dublētus (redundant) datus. Izdala trīs veidu anomālijas:

⇒ Iestarpināšanas (insertation) anomālijas

Piemēram: tabula stundeti – studiju programmas

Studenti-studiju programmas

Vards	Uzvārds	Pers.kods.	Studiju progr.
Juris	Bērziņš	134132-42221	Informācijas tehnoloģijas
Jāns	Ozoliņš	543535-32342	Tūrisms

Maija	Kociņa	345345-345320	Tūrisms
-------	--------	---------------	---------

Rodas problēma, ja piemēra grib izveidot jaunu studiju programmas ierakstu, jo to nevar izdarīt tam nepiesaistot nevienu studentu.

Pievienojot jaunu studentu, tam ir jānorāda arī studiju programma. Problēmas, ja ievada nekorektu (nesakrītošu teksta informāciju)

⇒ dzēšanas (deletion) anomālijas

Ja izdzēš visus datus par kādas studiju programmas studentiem, datu bāzē nepaliek arī dati par šo studiju programmu.

⇒ labošanas (modification) anomālijas

Ja maina studiju programmas nosaukumu, tas ir jāizmaina visu studentu ierakstos, kuri ir dotajā programmā. Ja tas neizdodas, datu bāze kļūst nekonsistentā (inconsistent).

Normalizācijas tiek sadalīta pa soļiem, katru no kuriem raksturo attiecīgā normālforma. Normālforma ir relācijas stāvoklis, kurā tā atbilst noteiktiem noteikumiem.

5.7. PIRMĀ NORMĀLFORMA

- Relācija ir pirmajā normālformā, ja:
 - ⇒ relācijā nav daudzvērtību (multivalued) atribūti un
 - ⇒ relācijā nav salikti (composite) atribūti un to kombinācijas.

Daudzvērtību atribūti ir atribūti, kuri vienai entītijai (kolonnai) ir vienlaicīgi vairākas vērtības.

Nosaukums	Izdevniecība	Kods	Autors
Zaļā zeme	Avots	12123	A.Upītis
Zemes balss	Zvaigzne	34243	F.Kalniņš
			A.Ozoliņš
Informātika	Jumava	22435	K.Cipariņš

- Ja ir šāda situācija, to iespējams novērst 3 dažādos veidos.
 - ⇒ Izveidot atsevišķu tabulu, kurā glabājas autori un norādes uz grāmatu

Nosaukums	Izdevniecība	Kods
Zaļā zeme	Avots	12123
Zemes balss	Zvaigzne	34243
Informātika	Jumava	22435

Kods	Autors
12123	A.Upītis
34243	F.Kalniņš
34243	A.Ozoliņš
22435	K.Cipariņš

⇒ Dublēt ierakstus katrai vērtībai

Nosaukums	Izdevniecība	Kods	Autors
Zaļā zeme	Avots	12123	A.Upītis
Zemes balss	Zvaigzne	34243	F.Kalniņš
Zemes balss	Zvaigzne	34243	A.Ozoliņš
Informātika	Jumava	22435	K.Cipariņš

⇒ Ieviest vairākas kolonnas, ja ir zināms maksimālais vērtību skaits.

Nosaukums	Izdevniecība	Kods	Autors_1	Autors_2
Zaļā zeme	Avots	12123	A.Upītis	
Zemes balss	Zvaigzne	34243	F.Kalniņš	A.Ozoliņš
Informātika	Jumava	22435	K.Cipariņš	

Salikti atribūti ir atribūti, kas var tikt sadalīti sīkākās daļās, kurai katrai ir sava neatkarīga jēga sistēmas darbības kontekstā.

Adrese
Rīga, Koku iela 10 -12, LV1010

Pilsēta	iela	māja	dzīvoklis	Indekss
Rīga	Koku	10	12	LV1010

5.8. OTRĀ NORMĀLFORMA

- Relācijas ir otrajā normālformā, ja:

⇒ tā ir 1NF

⇒ atribūti, kas neietilpst kādā no kandidātatslēgām, ir pilnīgi funkcionāli atkarīgi (tieši vai pārejoši) no visas kandidātatslēgas, nevis tikai kāds tā sastāvdaļas.

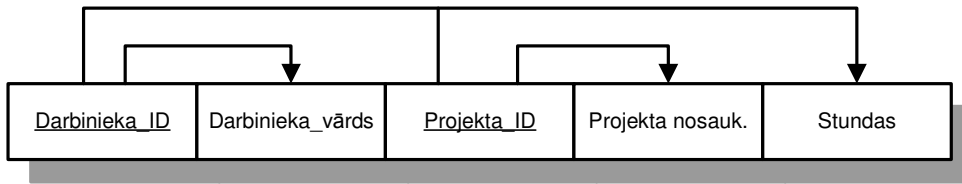
Funkcionālā atkarība – relācijā R atribūts B ir funkcionāli atkarīgs no atribūta A, ja katra eksistējoša A vērtība viennozīmīgi nosaka B vērtību. Atribūts var būt funkcionāli atkarīgs ne tikai no viena, bet arī no vairāku atribūtu kombinācijas.

Kandidātatslēga – atribūts vai atribūtu kombinācija, kas var unikāli identificēt rindiņu relācijā, neatkarīgi, un no kā nav iespējams iegūt citu kandidātatslēgu izņemot no šīs atslēgas kādu atribūtu.

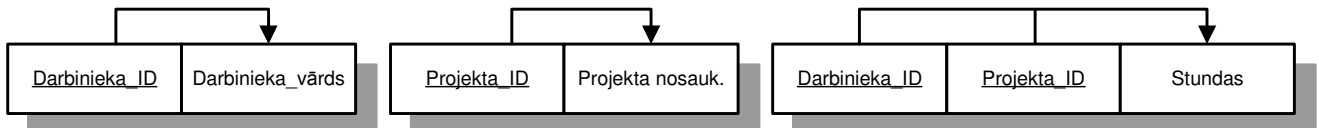
Primārā atslēga – viena no kandidātatslēgām, kas reāli tiek izmantota rindiņas unikālai identificēšanai.

Lai pārveidotu relāciju uz 2NF tā jāsadala vairākās mazākās relācijās, kuras atbilst 2NF.

Nav 2 NF



Pārveidota uz 2NF



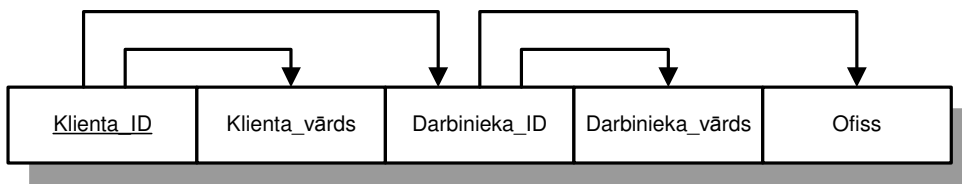
5.9. TREŠĀ NORMĀLFORMA

- Relācija ir trešajā NF, ja:
 - ⇒ Tā ir 2 NF,
 - ⇒ tajā neeksistē pārejošas atkarības

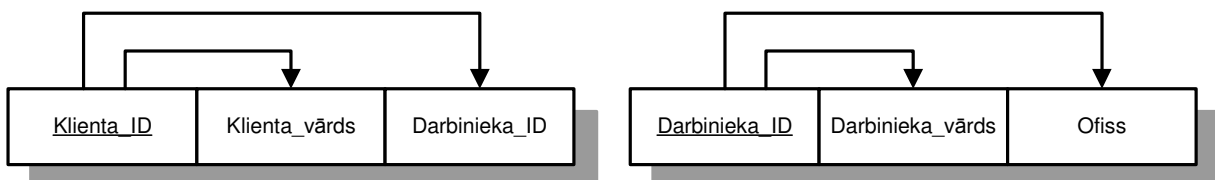
Pārejoša atkarība (transitive dependency) – funkcionāla atkarība starp atribūtiem, kas nav kandidātslēgas daļa.

Lai pārveidotu relāciju uz 3NF tā jāsadala vairākās mazākās relācijās, kuras atbilst 3NF.

Nav 3 NF



Pārveidota uz 3 NF

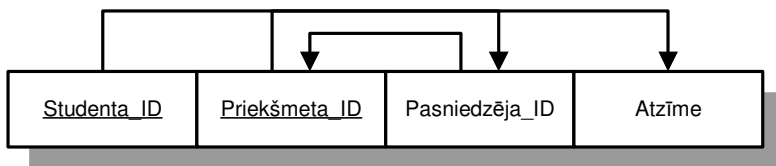


5.10. BOYCE-CODD NORMĀLFORMA

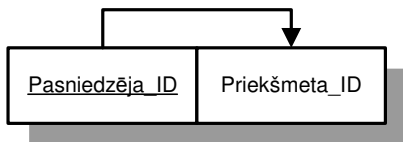
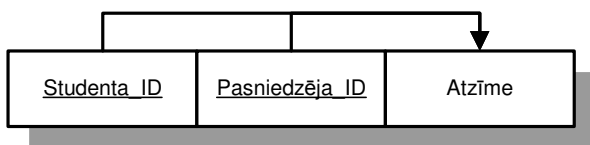
- Relācijas ir BCNF, ja
 - ⇒ Tā ir 3NF
 - ⇒ Determinanti ir tikai kandidātslēgas

Determinants (*determinant*) ir atribūts, kas atrodas funkcionālās atkarības kreisajā pusē. No šī atribūta ir atkarīgi citi atribūti. X ir determinants relācijā $X \rightarrow Y$.

Nav BCNF (ir 3NF, bet pasniedzēja_id ir atkarīgs no priekšmeta_ID, kurš nav kandidātslēga.



Pārveidota par BCNF



Varētu pārveidot arī par

{Studenta_ID, Pasniedzēja_ID} un {Studenta_ID, Priekšmeta_ID} vai

{Pasniedzēja_ID, Priekšmeta_ID} un {Studenta_ID, Priekšmeta_ID}

Eksistē arī vēl ceturta (multivalued) normālforma, piektā (project-join) normālforma un domēnatslēgas (domain-key) normālforma. Tomēr tās ir vairāk teorētiskas un tiek uzskatīts, ka, ja ir sasniegta BCNF, tad relācijas ir labi strukturētas un tālāka normalizācijas nav nepieciešama.

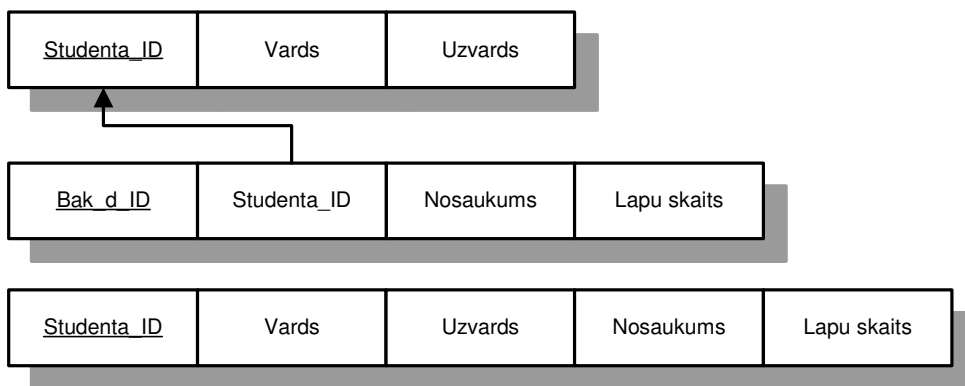
5.11. DENORMALIZĀCIJA (DENORMALIZATION)

Normalizētas tabulas nodrošina ļoti labu datu kvalitāti, bet nebūt ne labu DBVS ātrdarbību. Tāpēc praktiski veidojot tabulas pēc normalizācijas īpašos gadījumos tiek veikts normalizācijai pretējs process – denormalizācija. Tā mērķis ir uzlabot DB ātrdarbību. Ātrdarbība ir tieši atkarīga no tā cik tuvu uz diska viena no otras atrodas ieraksti, kas tiek izmantotas vienā SQL pieprasījumā. Citiem vārdiem sakot vai dati no diska varēs tikt nolasīti secīgā kārtībā vai arī tie būs jāmeklē. Denormalizācija palielina iespēju, ka var rasties datu anomālijas. Tāpēc procedūrām, kas veic izmaiņas denormalizētos datos testējot jāpiegriež sevišķa vērība.

Var tikt izdalīti vairāki gadījumi, kad tiek pielietota denormalizācija

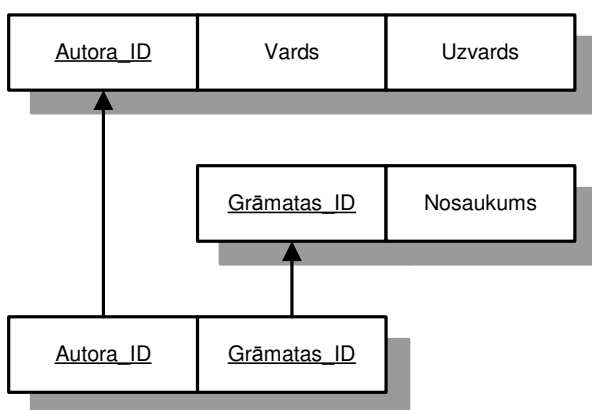
- Relācijas ar saiti 1:1

Ir iespējams šīs divas relācijas apvienot vienā, tādā veidā paātrinot darbības ar šo relāciju.



- Relācijas ar saiti N:M, kura realizēta kā atsevišķa relācija.

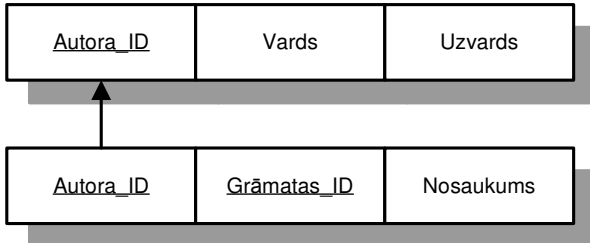
Šī atsevišķā relācija sastāv no divu relāciju primārajām atslēgām. Lai iegūtu rezultātu, nepieciešams lasīt datus no 3 tabulām. Ja pie vienas no saites relācijas primārās atslēgas pievienotu arī attiecīgos atribūtus no vecāku relācijas, tad būtu iespējams iegūt rezultātu izmantojot 2 tabulas.



Autora_ID	uzvards	vards
1	Kociņš	Juris
2	Kalniņš	Jānis
3	Ozoliņš	Pēteris

Grāmatas_ID	Nosaukums
1	Zemes balss
2	Informātika

Autora_ID	Grāmatas_ID
1	1
2	1
3	2



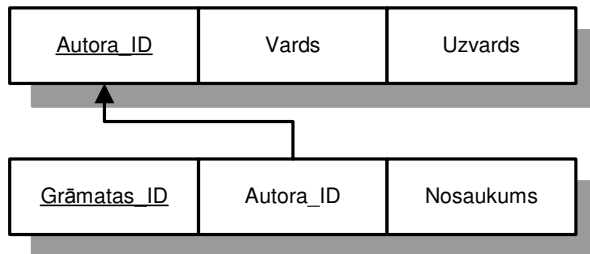
Autora_ID	uzvards	vards
1	Kociņš	Juris
2	Kalniņš	Jānis
3	Ozoliņš	Pēteris

Autora_ID	Nosaukums
1	Zemes balss
2	Zemes balss
3	Informātika

- Norādes saitē 1:N

Ja relācijai saites 1: pusē nav citu saišu, kā vien šī, tad iespējams relācijas apvienot dublējot nepieciešamos atribūtus.

(pieņemam, ka grāmatai var būt tikai viens autors)



Grāmatas_ID	Nosaukums	Vards	Uzvards
-------------	-----------	-------	---------

- Vienkāršota atslēga

Ja nepieciešama primārā atslēga un tā veidojas no daudzām kolonnām, tad var ieviest atsevišķu jaunu kolonnu, kas kalpo kā atslēga.

- Izskaitļotie atribūti

Ja kāds atribūts tiek izskaitļots no cita atribūta pieprasījuma izpildes laikā, tad dažos gadījumos var būt lietderīgāk glabāt abus – sākotnējo un izskaitļoto vērtību. Tādā veidā tiek pievienota viena papildus kolonna, bet vairs nav nepieciešamas katru reizi veikt operācijas, lai iegūtu šo izskaitļoto vērtību. It sevišķi tas ir svarīgi, ja izskaitļotā vērtība ir iegūstama izpildot apakšpieprasījumu.

5.12. IEROBEŽOJUMU NOTEIKŠANA UN TABULU VEIDOŠANA IZMANTOJOT SQL VAI GUI

Tie ir pēdējie soļi pēc datu normalizācijas, kas veicami, lai izveidotu DB. Šīs darbības sīkāk apskatīsim nākamajās lekcijās.

5.13. ER MODELĒŠANAS RĪKI

Šobrīd pasaulē ir pieejami daudz un dažādi rīki, kas ļauj veikt dažādas operācijas automatizētai sistēmu izstrādei. Vienkāršākie no tiem ļauj tikai zīmēt ER diagrammas, bet sarežģītākie veic kompleksu sistēmu biznesa modelēšanu un automātiski ļauj ģenerēt datu bāzes. Populārākie no tiem ir:

- Oracle's Designer
- IBM Rational Rose
- ERwin
- Data Architect
- Sybase PowerDesigner
- GRADE Modeler
- MS Visio
- DBDesigner
- MySQL Workbench

5.14. PAPILDUS LIERATŪRA

Database Management Systems 417-456.lpp

5.15. PAŠKONTROLES JAUTĀJUMI.

1. Kādi ir aplikāciju izstrādes posmi, ko dara katrā no tiem.
2. Kas ir ER diagrammas un kāpēc tās nepieciešamas.

3. Kas ir normālformas un kāpēc tās nepieciešamas.
4. Nosauciet jums zināmās normālformas un īsi aprasktie katru no tām.
4. Kas ir denormalizācija, kādos gadījumos to izmanto.

6. Datu pievienošana DB

6.1. INSERT

SQL komanda "INSERT" veic jaunu ierakstu pievienošanu tabulai.

Sintakse:

```
INSERT INTO tabula [ ( kolonna [, ...] ) ]  
    DEFAULT VALUES | VALUES ( izteiksme_1 [, ...] )
```

Ja kolonnas netiek norādītas, tad vērtība no "izteiksme_1" tiks piešķirta pirmajai kolonnai, otrās – otrajai u.t.t. Gadījumā, ja norādītas vairāk vērtības kā kolonnas, tad tiks izvadīts kļūdas paziņojums un datu pievienošana netiks veikta. Savukārt, ja kolonnas norādītas vairāk nekā pievienojamās vērtības, tad kolonnām, kurām nepietiek vērtības tiks piešķirta NULL vērtība.

Piemēram:

```
INSERT INTO masinas VALUES (1, 'BMW', '320', 13128, 'sedans', 2003, 1);  
INSERT INTO masinas VALUES (2, 'BMW');  
INSERT INTO masinas (pk_kods, marka, gads) VALUES (3, 'BMW');
```

7. Datu iegūšana no DB

Lai no DB iegūtu datus tiek izmantota komanda SELECT. Tā ļauj atlasīt vajadzīgos datus no vienas vai vairākām tabulām un sakārtot tos vēlāmā kārtībā.

Komandas SELECT vispārīgā (vienkāršotā) forma:

```
SELECT izteiksme_1  
    FROM tabulas_nosaukums  
    [ WHERE nosacijums ]  
    [ GROUP BY izteiksme_2 ]  
    [ ORDER BY izteiksme_3 [ASC|DESC]]  
izveiksme_1,2,3 - kolonnu saraksts vai izteiksme.  
Ja izteiksmes_1 ir *, tad tiks atgrieztas visas kolonnas, kas ir tabulā  
nosacijums - nosacījums, kas nosaka kuri ieraksti tiks atgriezti
```

Piemēram:

Tabula "masinas":

kods	nosaukums	modelis	cena	tips	gads
1	Audi	A6	20199	Sedans	2001
2	Audi	A4	14999	Kombi	2000
3	BMW	M3	23001	Coupe	2002
4	MAZDA	M6	12101	Sedans	2003

SELECT * FROM masinas; – tiek atgriezta visa tabula:

kods	nosaukums	modelis	cena	tips	gads
1	Audi	A6	20199	Sedans	2001
2	Audi	A4	14999	Kombi	2000
3	BMW	M3	23001	Coupe	2002
4	MAZDA	M6	12101	Sedans	2003

SELECT nosaukums, modelis FROM masinas; - atgriež tikai norādītās kolonnas;

nosaukums	modelis
Audi	A6
Audi	A4
BMW	M3
MAZDA	M6

- Atlases nosacījumi

Lai uzstādītu nosacījumus, pēc kuriem notiks datu atase, uzņemto WHERE. Izvadīti tiks tikai tie dati, kam šis nosacījums atgriež vērtību "paties". Vienkāršākajā gadījumā nosacījumā izmanto salīdzināšanu ar kādu konstantu vērtību.

Piemēram:

SELECT * FROM masinas WHERE cena<20000;

kods	nosaukums	modelis	cena	tips	gads
2	Audi	A4	14999	Kombi	2000
4	MAZDA	M6	12101	Sedans	2003

Ir iespējams arī veidot sarežģītākus nosacījumu izmantojot loģiskos operatorus AND, OR un NOT.

Piemēram:

SELECT * FROM masinas WHERE cena<20000 AND tips="Sedans";

Operators LIKE tiek izmantots gadījumos, kad nosacījumā nepieciešams izmantot šablonu (*pattern*).

Piemēram:

SELECT * FROM masinas WHERE marka LIKE "a%";

Atgriež visas automašīnas, kuru markas sākas ar burtu "a".

SELECT * FROM masinas WHERE marka LIKE "_a%";

Atgriež visas automašīnas, kuru markas otrais burts ir "a".

SELECT * FROM masinas WHERE marka LIKE "%a%";

Atgriež visas automašīnas, kuru markas satur burtu "a".

Salīdzināšanu var veikt ne tikai ar konstantām vērtībām, bet arī ar citu pieprasījumu rezultātiem. Sīkāk to apskatīsim nākamajā lekcijā.

- Agregātfunkcijas

Agregātfunkcijas ir funkcijas, kas no vairākām ieejas vērtībām izrēķina vienu atgriežamo vērtību.¹

Biežāk lietotās agregātfunkcijas ir:

- count() – atgriež ieejas vērtību skaitu,
- sum() – atgriež ieejas vērtību summu,
- min() – atgriež mazāko no ieejas vērtībām,
- max() – atgriež lielāko no ieejas vērtībām,
- avg() – izrēķina ieejas vērtību vidējo lielumu.

Piemēram :

```
SELECT count(*) FROM masinas;
```

- Grupēšana

Lai grupētu ierakstus, tiek lietota komanda GROUP BY.



Ja lieto grupēšanu, tad arī norādot, kurus laukus pieprasījums atgriež, drīkst norādīt tikai laukus, kas izmantoti grupēšanā, kā arī agregātfunkcijas.

Piemēram:

```
SELECT tips, count(*) FROM masinas GROUP BY tips;
```

tips	count
Sedans	2
Kombi	1
Coupe	1

- Rezultātu sakārtošana

Lai rezultātus sakārtotu tiek lietota konstrukcija ORDER BY aiz kā norāda kolonnu pēc kura tiks veikta kārtošana.

Piemēram:

```
SELECT * FROM masinas ORDER BY cena;  
(tiek atgriezta visa tabula sakārtota pēc cenas augošā kārtībā)
```

kods	nosaukums	modelis	cena	tips	gads
4	MAZDA	M6	12101	Sedans	2003
2	Audi	A4	14999	Kombi	2000
1	Audi	A6	20199	Sedans	2001
3	BMW	M3	23001	Coupe	2002

Pēc noklusēšanas vērtības tiek kārtotas augošā secībā, tomēr norādot komandu DESC.

Ir iespējams veikt kārtošānu arī pēc vairākām kolonnām, tās atdalot ar komatu. Šajā gadījumā ja pirmās kolonnas vērtības vairākiem ierakstiem sakrītīs, ieraksti tiks sakārtoti vadoties no otrās kolonnas vērtībām.

Piemēram:

```
SELECT * FROM masinas ORDER BY nosaukums, modelis DESC;  
(tiek atgriezta visa tabula sakārtota pēc nosaukuma augošā un markas dilstošā secībā)
```

kods	nosaukums	modelis	cena	tips	gads
1	Audi	A6	20199	Sedans	2001
2	Audi	A4	14999	Kombi	2000
3	BMW	M3	23001	Coupe	2002
4	MAZDA	M6	12101	Sedans	2003

7.1. PAPILDUS LIERATŪRA

MySQL dokumentācija sadaļa 13.2.7. SELECT Syntax

PostgreSQL dokumentācija sadaļa 7. Queries

PostgreSQL dokumentācija sadaļas SELECT un INSERT komandu ietojums

7.2. PAŠKONTROLES JAUTĀJUMI.

1. Kāda ir INSERT komandas pamatsintakse.
2. Kāda ir SELECT komandas pamatsintakse.
3. Kam komandā SELECT lieto konstrukcijas GROUP BY un HAVING.

8. SELECT paplašinātās iespējas

8.1. HAVING

HAVING tiek lietotas kopā ar GROUP BY un darbojas līdzīgi kā WHERE. Atšķirība ir tāda, ka nosacījums WHERE tiek pielietots datiem pirms tie tiek grupēti, bet HAVING atlasa rezultātus jau no sargrupētiem datiem.

Piemēram

masinas

pk_kods	marka	modelis	cena	tips	gads
1	Audi	A6	20199	Sedans	2004
2	Audi	A6	14999	Kombi	2000
3	BMW	M3	23001	Coupe	2001
4	MAZDA	M6	12101	Sedans	2005
5	MAZDA	M6	12101	Sedans	2002

```
SELECT marka,modelis,count(*)
FROM masinas
WHERE gads > 2002
GROUP BY gads;
```

Marka	modelis	count
Audi	A6	1
BMW	M3	1
MAZDA	M6	2

```
SELECT marka,modelis,count(*)
FROM masinas
WHERE gads > 2000
GROUP BY gads
HAVING count(*)>1;
```

marka	modelis	count
MAZDA	M6	2

8.2. DATU KOLONNAS NO DAŽĀDĀM TABULĀM

Ar komandas SELECT palīdzību iespējams iegūt datus ne vien no vienas, bet arī no vairākām tabulām. Tādā gadījumā pie katras konkrētās kolonnas var griezties izmantojot paplašinātu sintaksi *tabula.kolonna*

Piemēram:

masinas

pk_kods	marka	modelis	cena	tips	gads	fk_ipasnieks
1	Audi	A6	20199	Sedans	2004	1
2	Audi	A4	14999	Kombi	2000	2
3	BMW	M3	23001	Coupe	2002	
4	MAZDA	M6	12101	Sedans	2003	1

personas

pk_kods	varda	uzvards
1	Juris	Kociņš
2	Jānis	Bērziņš
3	Ilz	Rozīte

SELECT personas.vards, personas.uzvards, masinas.marka, masinas.modelis, masinas.gads,
FROM personas,masinas.

Šādā gadījumā tiek iegūts rezultāts, kur katrs tabulas "personas" ieraksts tiek savienots ar katru tabulas "masinas" ierakstu. Šāds risinājums nekur praktiski nav izmantojams.

Vards	uzvards	nosaukums	modelis	Gads
Juris	Kociņš	Audi	A6	2004
Juris	Kociņš	Audi	A4	2000
Juris	Kociņš	BMW	M3	2002
Juris	Kociņš	MAZDA	M6	2003
Jānis	Bērziņš	Audi	A6	2004
Jānis	Bērziņš	Audi	A4	2000
Jānis	Bērziņš	BMW	M3	2002
Jānis	Bērziņš	MAZDA	M6	2003
Ilz	Rozīte	Audi	A6	2004
Ilz	Rozīte	Audi	A4	2000
Ilz	Rozīte	BMW	M3	2002
Ilz	Rozīte	MAZDA	M6	2003

Ja nepieciešams apvienot datus no vairākām tabulām, ir kāds kritērijs pēc kā notiek šo datu atbilstības noteikšana. Parastā ir ārējā atslēga, kas norāda uz kādas citas tabulas ierakstu. Lai pieprasījumā veidotu atbilstības starp tabulām tiek izmantota komandas SELECT opcija JOIN tabula ON nosacījums. Join ir vairākas formas un tās atgriež dažādus rezultātus. Jāatzīmē, ka ar NULL vērtībām ierastu atbilstība neveidojas. (NULL vienā tabulā netiek uzskatīts par vienādu ar NULL otrā tabulā)

- INNER JOIN

Šī ir forma, kas tiek veikta pēc noklusēšanas, ja norāda tikai JOIN un nenorāda tā tipu. Izpildot šo apvienošanas komandu tiks izvadīti visi ieraksti, kur nosacījumā norādītās vērtības būs identiskas.

SELECT * FROM masinas

INNER JOIN personas ON masinas.fk_ipasnieks = personas.pk_kods

pk_kods	nosaukums	modelis	cena	tips	gads	fk_ipasnieks	pk_kods	varda	uzvards
1	Audi	A6	20199	Sedans	2004	1	1	Juris	Kociņš
4	MAZDA	M6	12101	Sedans	2003	1	1	Juris	Kociņš
2	Audi	A4	14999	Kombi	2000	2	2	Jānis	Bērziņš

- WHERE

WHERE darbojas līdzīgi kā INNER JOIN

SELECT * FROM masinas, personas

WHERE masinas.fk_ipasnieks = personas.pk_kods

- LEFT JOIN

Šajā join tiks iekļauti visi ieraksti no tabulas "masinas" un tiem atbilstošie ieraksti no tabulas "personas". Ieraksti no tabulas "personas", kuriem nav atbilstošu ierastu tabulā "masinas" netiks parādīti.

SELECT * FROM masinas

LEFT JOIN personas ON masinas.fk_ipasnieks = personas.pk_kods

pk_kods	nosaukums	modelis	cena	tips	gads	fk_ipasnieks	pk_kods	vars	uzvards
1	Audi	A6	20199	Sedans	2004	1	1	Juris	Kociņš
2	Audi	A4	14999	Kombi	2000	2	2	Jānis	Bērziņš
3	BMW	M3	23001	Coupe	2002				
4	MAZDA	M6	12101	Sedans	2003	1	1	Juris	Kociņš

- RIGHT JOIN

Šajā join tiks iekļauti visi ieraksti no tabulas "personas" un tiem atbilstošie ieraksti no tabulas "masinas". Ieraksti no tabulas "masinas", kuriem nav atbilstošu ierastu tabulā "personas" netiks parādīti.

SELECT * FROM masinas

RIGHT JOIN personas ON masinas.fk_ipasnieks = personas.pk_kods

pk_kods	nosaukums	modelis	cena	tips	gads	fk_ipasnieks	pk_kods	vars	uzvards
1	Audi	A6	20199	Sedans	2004	1	1	Juris	Kociņš
4	MAZDA	M6	12101	Sedans	2003	1	1	Juris	Kociņš
2	Audi	A4	1	Kombi	2000	2	2	Jānis	Bērziņš
							3	Ilz	Rozīte

- FULL JOIN

Šajā join ir LEFT JOIN un RIGHT JOIN apvienojums. No abām tabulām tiks iekļauti gan ierasti, kur vērtības atbilst, gan arī tie, kam vērtības neatbilst.

SELECT * FROM masinas

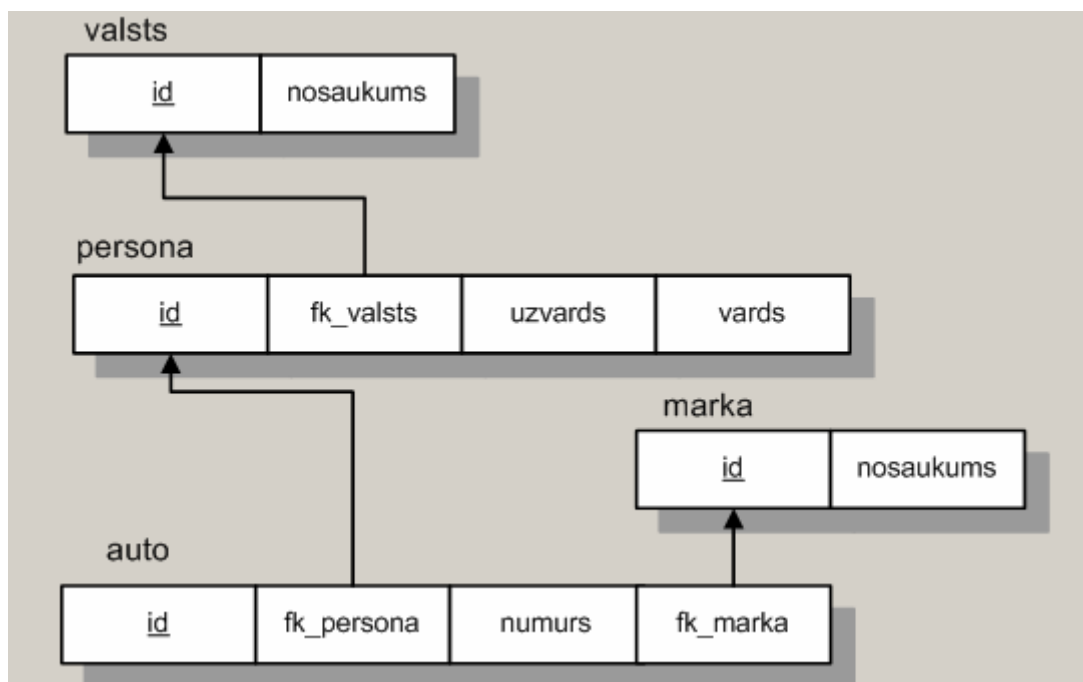
FULL JOIN personas ON masinas.fk_ipasnieks = personas.pk_kods

pk_kods	nosaukums	modelis	cena	tips	gads	fk_ipasnieks	pk_kods	vars	uzvars
1	Audi	A6	20199	Sedans	2004	1	1	Juris	Kociņš
4	MAZDA	M6	12101	Sedans	2003	1	1	Juris	Kociņš
2	Audi	A4	14999	Kombi	2000	2	2	Jānis	Bērziņš
							3	Ilz	Rozīte
3	BMW	M3	23001	Coupe	2002				

- Dati no 3 vai vairāk tabulām

Līdzīgi kā iepriekš apskatītajos gadījumos ar 2 tabulām, ir iespējams iegūt datus arī no 3 vai vairāk tabulām.

Piemēram dota DB ar šādu shēmu:



Uzdevums - no DB jāatgriež dati par visām automašīnām, to īpašniekiem, valstu nosaukumi, kurā šīs personas dzīvo un visbeidzot automašīnas markas.

SELECT auto.numurs, persona.uzvars, persona.vars, valsts.nosaukums, marka.nosaukums
FROM auto

INNER JOIN persona ON persona.id = auto.fk_persona

INNER JOIN valsts ON valsts.id = persona.fk_valsts

INNER JOIN marka ON marka.id = auto.fk marka;

Var izmantot arī WHERE nosacījumus:

```
SELECT auto.numurs, persona.uzvards, persona.vards, valsts.nosaukums, marka.nosaukums  
FROM auto, personas, valsts, marka  
WHERE persona.id = auto.fk_persona AND  
       valsts.id = persona.fk_valsts AND  
       marka.id = auto.fk_marka;
```

8.3. DATI VIENĀ KOLONNĀ NO DAŽĀDĀM TABULĀM

Lai apvienotu vienā kolonnā datus no vairākām tabulām (vai apakšpieprasījumiem), var lietot komandas UNION, INTERSECT un EXCEPT.

- UNION

Šī komanda atgriež ierakstu, ja tas atrodas kaut vienā no pieprasījumiem.

select1 UNION select2. Šajā gadījumā jāievēro, ka abiem "select" jāatgriež vienāds kolonnu skaits un datu tipi.

Piemēram:

masinas

pk_kods	nosaukums	modelis	cena	tips	gads	fk_ipasnieks
1	Audi	A6	20199	Sedans	2004	1
2	Audi	A4	14999	Kombi	2000	2
3	BMW	M3	23001	Coupe	2002	
4	MAZDA	M6	12101	Sedans	2003	1

personas

pk_kods	vards	uzvards
1	Juris	Kociņš
2	Jānis	Bērziņš
3	Ilz	Rozīte

```
SELECT nosaukums,modelis FROM masinas
```

UNION

```
SELECT vards,uzvards FROM personas
```

nosaukums	modelis
Audi	A6
Audi	A4
BMW	M3
MAZDA	M6
Juris	Kociņš
Jānis	Bērziņš
Ilz	Rozīte

UNION bieži sanāk lietot, lai datus kopā ar datiem iegūtu arī kopsummas.

Piemēram:

Pieprasījums, kas atgriež visas mašīnas jaunākas par 2000. gadu un to kopējo vērtību.

```
SELECT 1, nosaukums, modelis, gads, cena
      FROM masinas
      WHERE gads>2000
UNION
SELECT 2, "Kopā", " -", "-", sum(cena)
      FROM masinas
      WHERE gads>2000
```

8.4. INTERSECT

```
select1 INTERSECT select2;
```

Atšķirībā no UNION, INTERSECT atgriež ierakstus, kas atkārtojas gan select1 gan select2.

8.5. EXCEPT

```
select1 EXCEPT select2;
```

EXCEPT atgriež ierakstus, kas ir select1 bet nav select2.

8.6. APAKŠPIEPRASĪJUMI (*SUBQUERIES*)

Apakšpieprasījums ir pieprasījums, kurš tiek izmantots kā datu avots kādam citam pieprasījumam. Apakšpieprasījumus var izmantot gan kā tiešos datu avotus, gan arī veidojot datu atlasēs nosacījumus izmantojot "WHERE" un "HAVING".

- Tiešie datu avoti:

Ja apakšpieprasījums tiek izmantots kā datu avots, tad tas tiek rakstīts FROM sadaļā tabulas nosaukuma vietā. Apakšpieprasījumu raksta iekavās aiz kurām ar „as” jānorāda tā nosaukums. Apakšpieprasījums var tikt izmantots kā vienīgais datu avots, bet var tikt izmantots kopā ar citu tabulu vai apakšpieprasījumu.

```
SELECT m1.modelis FROM (SELECT * FROM masinas) AS m1;
```

```
SELECT g_tek.apgr, g_iepr.apgr, (g_tek.apgr-g_iepr.apgr) FROM
  (SELECT sum(cena) as apgr FROM masinas WHERE gads = 2003) AS g_tek,
  (SELECT sum(cena) as apgr FROM masinas WHERE gads = 2002) AS g_iepr;
```

- Datu avoti nosacījumiem:

Bieži ir nepieciešams veidot datu atlasēs nosacījumus, kuros tiek izmantotas vērtības, ko iegūst atlasot tos no tās pašas vai kādas citas tabulas.

⇒ IN / NOT IN

Tiek izmantots, lai izvadītu tikai tos ierakstus, kuriem atbilstošās vērtības atrastas (vai attiecīgi nav atrastas NOT IN gadījumā) apakšpieprasījumā

```
SELECT * FROM masinas WHERE fk_ipasnieks IN  
(SELECT pk_kods FROM personas WHERE vards = „Jānis”)
```

⇒ EXIST

Praktiskais pielietojums līdzīgs kā IN. Darbības principi tomēr atšķiras. Tiek parādīti visi ieraksti, kuriem apakšpieprasījums atgriež kaut vienu rindiņu. EXISTS pakļaujas plašākām variācijās iespējām.

```
SELECT * FROM masinas WHERE EXISTS (SELECT 1 FROM personas WHERE  
masinas.fk_ipasnieks=personas.pk_kods)
```

⇒ Operatori =,<,>

Gadījumā, ja apakšpieprasījums atgriež vienu vērtību IN vietā iespējams izmantot arī operatorus =, <, > .

```
SELECT * FROM masinas where cena >  
(SELECT max(cena) FROM masinas WHERE marka = 'Ford')
```

⇒ ANY

ANY var tikt izmantots kopā ar kādu no salīdzināšanas operatoriem. Ja ar kaut vienu no apakšpieprasījuma atgrieztajām vērtībām iegūst patiesu nosacījumu, tad tekošais ieraksts galvenajā pieprasījumā tiks parādīts.

```
SELECT * FROM masinas WHERE cena > ANY
```

```
(SELECT cena FROM masinas WHERE marka = 'Ford')
```

⇒ ALL

Līdzīgi kā ANY, arī ALL var tikt izmantots kopā ar kādu no salīdzināšanas. Tekošais ieraksts galvenajā pieprasījumā tiks parādīts tikai gadījumā, ja nosacījums būs patiess ar visām no apakšpieprasījuma iegūtajām vērtībām.

```
SELECT * FROM masinas WHERE cena > ALL
```

```
(SELECT cena FROM masinas WHERE marka = 'Ford')
```

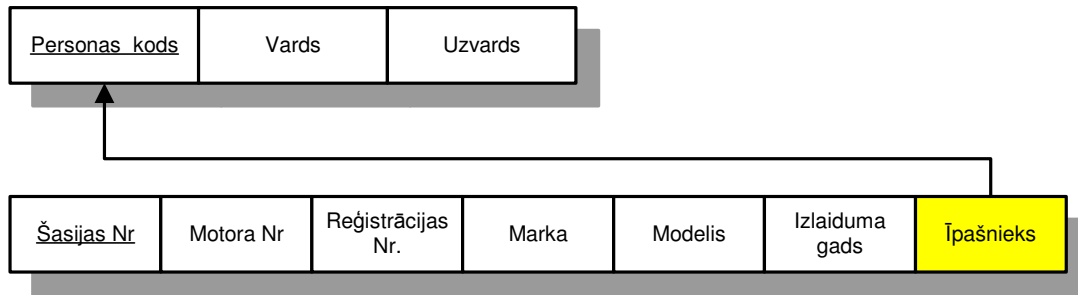
8.7. PIEPRASĪJUMU VEIDOŠANA

Sarežģītus pieprasījumu veidošanu ieteicams iedalīt vairākos posmos. Katrā no posmiem tiek veidots un pārbaudīts darbībā kāds no apakšpieprasījumiem un beigās tie visi tiek sakombinēti kopā vienā pieprasījumā.

Piemēram:

Uzdevums – atgriezt visu personu sarakstu, kuriem pieder mašīnas jaunākas par 2000. gadu.

Dati tiek glabāti tabulās ar šādu shēmu:



Šo uzdevumu var sadalīt vairākos posmos:

1) atrast visas mašīnas, kas jaunās par 2000. gadu

```
SELECT * FROM masinas WHERE gads>2000
```

2) atrast visu īpašnieku personas kodus, kam pieder mašīnas, kas atrastas 1) posmā

```
SELECT ipasnieks FROM masinas WHERE gads>2000
```

3) atrast datus par visām personām, kam pieder mašīnas, kuru saraksts iegūts 2) posmā

```
SELECT * FROM personas  
WHERE personas_kods IN  
(SELECT ipasnieks FROM masinas WHERE gads>2000)
```

8.8. PAPILDUS LIERATŪRA

MySQL dokumentācija sadaļa 13.2.7.1. JOIN Syntax

PostgreSQL dokumentācija sadaļas 7.2.1.1. Joined Tables

8.9. PAŠKONTROLES JAUTĀJUMI.

1. Kā atšķiras WHERE un HAVING lietojums pieprasījumos.
2. Kā vienā pieprasījumā apvienot datus no vairākām tabulām?
3. Kā izmanto apakšpieprasījumus.

9. Ierakstu glabāšana un failu struktūra

Ierakstu kopumu, kas viedo DB, kaut kur ir jāglabā. Uz datu nesējiem dati tiek organizēti failā(-os). DB failu organizācijas principi ir specifiski katrai DBVS, tomēr ir arī kopējas iezīmes. Kā datu nesējs datu glabāšanai parasti tiek izmantots cietais disks vai disku masīvs.

9.1. DISKU APAKŠSISTĒMA (CIETIE DISKI UN I/O KONTROLIERI):

Diski parasti strādā par kārtu lēnāk kā datora operatīvā atmiņa, tāpēc nepieciešamas pievērst īpašu uzmanību to ātrdarbībai, jo bieži vien tieši disku ātr(lēn)darbība ir šaurā vieta sistēmā, kas bremzē visas sistēmas optimālu darbu. Diska ātrdarbību raksturo dažādi lielumi kā piemēram griešanās ātrums un meklēšanas laiks. Tā kā fiziski diska ātrdarbību palielināt ir grūti, tad lai uzlabotu piekļūšanas ātrumu datiem, tiek izmantotas dažādas buferēšanas (jeb kešošanas) tehnikas. Par buferēšanu sauc procesu, kad neliels daudzums datu no lēnākā datu nesēja (diska) tiek saglabāti ātrākajā (kešatmiņā) pirms vēl pēc tiem ir radies konkrēts pieprasījums. Ir pat sistēmas, kurās visa DB glabājas operatīvajā atmiņā, tas nodrošina daudz augstāku ātrdarbību. Diemžēl operatīvās atmiņas izmaksas ir krietni lielākas kā diska izmaksas, tāpēc tikai retos gadījumos tas ir ekonomiski izdevīgi.

9.2. BUFERĒŠANA

Parasti sistēmās ir 2 līmeņu buferēšana. Zemākais līmenis ir cietās diska iebūvētā kešatmiņa. Mūsdienu diskos tā svārstās no 1 līdz 16 Mb. Parastiem diskos ko lieto datoros ir 1-2Mb, bet ir pieejami speciāli diskos, kuri domāti serveriem un tiem šis atmiņas apjoms var sasniegt 16Mb. Augstāka līmeņa kešatmiņa tiek veidota datora operatīvajā atmiņā, kuras maza daļa tiek izdalīta, tieši diska datu pagaidu glabāšanai. Dažreiz bez šiem vēl ir 3 līmenis, kurš ir novietots uz SCSI kontroliera un atrodas starp 1 un 2 līmeņa keš atmiņām.

9.3. RAID

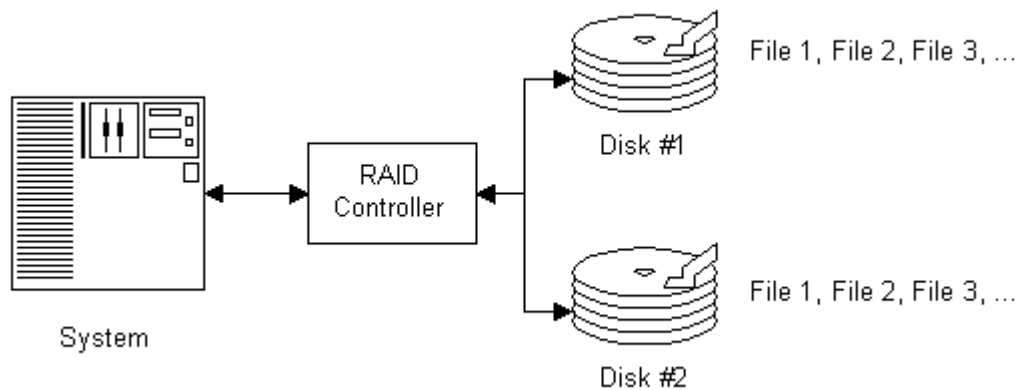
Lai vēl vairāk uzlabotu datu lasīšanas un rakstīšanas ātrumu (un arī datu drošību dažos risinājumos) var tikt izmantoti RAID (Redundant Arrays of Inexpensive Disks) risinājumi.

RAID sastāv no vairākiem cietiem diskos, kuru kopēju saskaņotu darbību vada RAID kontrolieris, kurš var būt gan aparatūras (hardware) gan programmatūras (software). Tiek izmantoti vairāki veidi, kādā datus sadalīt pa diskos, lai panāktu vēlamo efektu (ātrdarbība, drošība).

- RAID datu dalīšanas veidi:

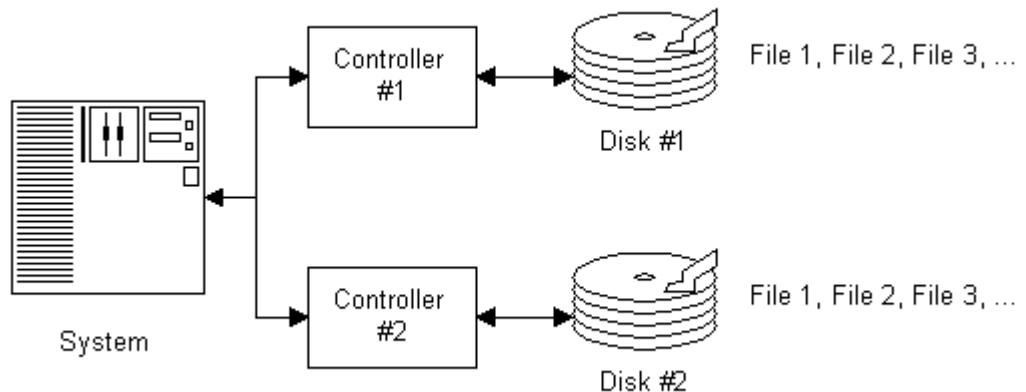
⇒ Spoguļošana (mirroring)

2 diski, kas pieslēgti pie viena RAID kontroliera, glabā pilnīgi identisku informāciju (ieguvums: palielinās lasīšanas ātrdarbība un drošība; trūkums – neefektīga diska vietas izmantošana,)



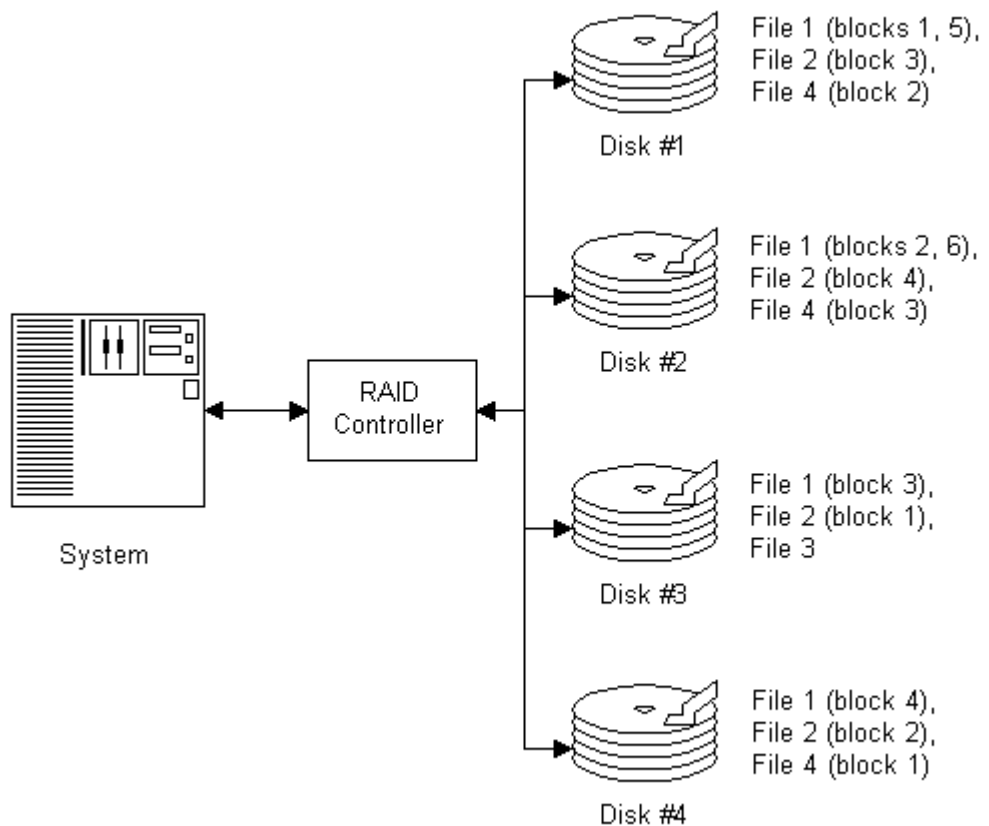
⇒ Dupleksošana (Duplexing)

2 diski, kas pieslēgti katrs pie sava RAID kontroliera glabā pilnīgi identisku informāciju. (ieguvums salīdzinot ar spoguļošanu – pat ja sabojājas 1 RAID kontrolieris, sistēma darbojas)



⇒ Virknēšana (Striping)

Informācija starp diskiem tiek sadalīta secīgi pa blokiem. 1 bloks no faila nonāk uz 1 diska, 2 uz otrā utt. (ieguvums – uzlabota lasīšanas un rakstīšanas ātrdarbība, trūkums – samazinās datu drošība, jo ja sabojājas kaut viens no diskiem visi dati ir zaudēti)



⇒ Paritāte (parity)

Kontrollielums, kas pēc noteikta algoritma tiek iegūts veicot darbības ar datu N vienībām. Ja viena no šīm vienībām tiek sabojāta, pateicoties kontrollielumam izmantojot speciālus algoritmus, to ir iespējams atjaunot. (iegumums: efektīvāka diska vietas izmantošana kā spoguļošanā)

- RAID līmeņi

Dažādi kombinējot šos risinājumus tiek iegūti 7 RAID līmeņi (0-6). Katram līmenim ir savas priekšrocības un trūkumi

⇒ RAID0

Vienkārša virknēšana (striping) bez paritātes. RAID0 piemīt ļoti laba ātrdarbība un tas ir salīdzinoši lēts. Būtisks trūkums ir tas, ka samazinās datu fiziskā drošība.

⇒ RAID1

Spoguļošana vai dupleksēšana. RAID1 uzlabo drošību un mazliet - datu lasīšanas ātrums. Trūkums - neefektīvi izmantota vieta uz diska.

⇒ RAID2

Izmanto tehnoloģiju līdzīgu virknēšanai ar paritāti. Mūsdienās reāli netiek izmantots.

⇒ RAID3

Dati tiek sadalīti uz vairākiem diskem virknējot pa baitiem (vai blokiem, kas parasti mazāki par 1024 baitiem) un paritātes informācija tiek glabāta uz atsevišķa diska. Masīvā jābūt vismaz 3 diskem. Sabojājoties 1 no diskem informāciju ir iespējams atjaunot. Šādi risinājumi parasti tiek izmantoti, ja jāveic datu lasīšana/rakstīšana lieliem gabaliem (multimēdiiju objekti)

⇒ RAID 4

Līdzīgi kā RAID 3, dati tiek sadalīti uz vairākiem diskem virknējot pa blokiem (1-32kb) un paritātes informācija tiek glabāta uz atsevišķa diska. Masīvā jābūt vismaz 3 diskem. Sabojājoties 1 no diskem informāciju ir iespējams atjaunot. Šādi risinājumi tiek izmantoti diezgan reti, jo tas ir kaut kas vidējs starp 3 un 5 un tam nav īsta mērķauditorija.

⇒ RAID5

Dati un paritātes informācija tiek sadalīta vienmērīgi pa visiem diskem. Masīvā jābūt vismaz 3 diskem. Sabojājoties 1 no diskem informāciju ir iespējams atjaunot. Šis ir vispopulārākais no patreiz pieejamiem serveru RAID risinājumiem.

⇒ RAID6

Līdzīgi kā RAID5, tikai paritātes informācija tiek glabāta dubulti. Dati un paritātes informācija tiek sadalīta vienmērīgi pa visiem diskem. Masīvā jābūt vismaz 4 diskem. Sabojājoties 2 no diskem informāciju ir iespējams atjaunot. Reti lietots, jo tikai retos gadījumos nepieciešama tik liela drošība.

⇒ RAID7

Storage Computer Corporation licenzēts produkts, ko nebūtu korekti saukt par RAID standartu, kā tas ir ar visiem citiem iepriekšminētajiem, bet šī kompānija to tirgo ar šādu tirdzniecības zīmi.

⇒ Daudzlīmeņu RAID (1+0;0+1;0+3;u.c.)

Pastāv risinājumi, kas ļauj veidot vairāku RAID līmeņu kombinācijas. (tomēr ne visas kombinācijas eksistē) Šādi risinājumi tiek izmantoti, lai veidotu vairāku RAID līmeņu labāko īpašību apvienojumu, rezultātā iegūstot vēl ātrāku un drošāku, bet diemžēl arī dārgāku sistēmu.

- JBOD (*Just A Bunch Of Disks*)

JBOD ir zināmas līdzības ar RAID, bet to nevar uzskatīt par RAID. JBOD nepalielina ne datu lasīšanas/rakstīšanas ātrumu ne arī dod papildus datu drošību. JBOD vienīgā funkcija ir no vairākiem maziem diskem OS līmenī izveidot 1 lielu disku. Serveros nav ieteicams.

9.4. TĪKLA GLABĀTUVES (*NETWORK STORAGE*)

Šī tēma varbūt nav tieši saistīta ar DB, bet šīs zināšanas varētu būt noderīgas veidojot DB sistēmu, un konkrēti izvēloties tās aparatūru.

Datu glabāšanas ierīču attīstības gaitā ir radušies dažādi datu glabāšanas veidi.

- Tieši pievienotās datu glabātavas (*Direct Attached Storage*)

Šajā kategorijā ietilpst lielākā daļa no sistēmām un to pamatiezīme ir, ka datu glabāšanas ierīces ir tieši pieslēgti pie servera. Kā piemērs tam ir parastie cietie diski, ko izmanto datoros. Šiem diskam nav obligāti jābūt iekšējiem. Piemēram arī ārējais disks vai disku masīvs, kuru kontrolieris ir serverī pieder pie šīs kategorijas.

- Tīkla datu glabātavas (*Network Attached Storage*)

Šajos risinājumos tiek izmantotas ierīces, kas tieši tiek pieslēgtas pie datortīkla. Parasti tie ir speciāli datori, kuriem ir pašiem savas adreses un kuru datiem klienti var piekļūt caur serveri vai atsevišķos gadījumos arī pa tiešo. Lielākā priekšrocība ir tā, ka datorsistēmās, kur darbojas daudzi serveri, kuri pie tam izmanto dažādas platformas, ir iespējams veikt centralizētu datu glabāšanu un rezerves kopiju veidošanu. Šādas sistēmas ir viegli paplašināmas un veidotas nepārtrauktam darbam, disku nomainīšanu ir iespējams veikt neizslēdzot sistēmu (*hot swappable*). Lai nodrošinātu datu drošību tiek pielietotas dažādas RAID tehnoloģijas. Trūkums ir tāds, ka tiek palielinātā kopējā tīkla iekāru noslodze.

- Glabātavu tīkls (*Storage Area Network*)

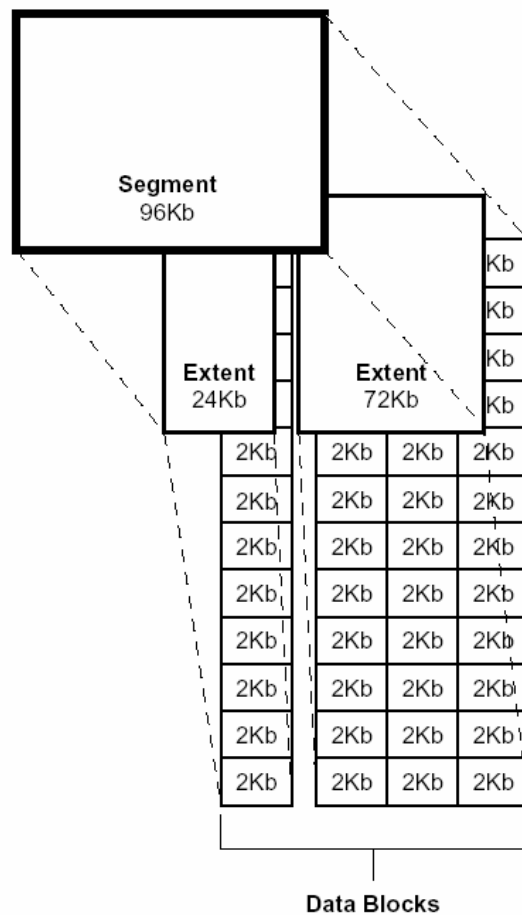
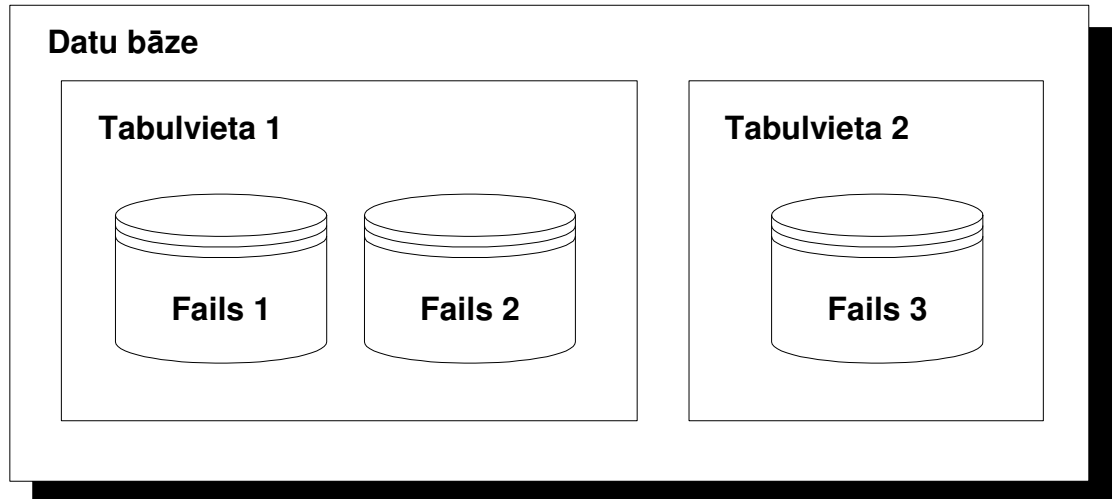
Līdzīgi kā tīkla datu glabātavas arī glabātavu tīklā pamatā ir iekārtas, kas darbojas neatkarīgi no servera. Glabātavu tīklā datu glabātavas ir savienotas savā starpā un ar serveriem izmantojot slēdzi (*switch*) tā veidojot mazu tīklu. Slēdzis parasti ir fibre chanel un nodrošina 2 Gbps ātru datu apmaiņu. (atsevišķi modeļi līdz pat 8Gbps). Tādā veidā šī tīkla noslodze nekādā veidā neietekmē to tīkla daļu, ko izmanto lietotāji pieslēdzoties serverim.

9.5. DB ORGANIZĀCIJAS MODEĻI:

Dažādās DBVS tiek piedāvāti dažādi risinājumi, kā DB tiek izvietota uz diska. Pat vienas DBVS ietvaros iespējamas dažādi DB organizācijas modeļi un tabulu fiziskā organizācija. Apskatīsim vienu no tiem. Šāds modelis ir diezgan populārs un to var sastapt vairākās DBVS (Oracle, MySQL InnoDB).

- Teorija
⇒ Tabulvieta

Tabulvieta (*Tablespace*) – loģiska vienība, kurā tiek glabāti DB dati. DB var būt viena vai vairākas tabulvietas. Attiecīgi katra tabulvieta var fiziski glabāties vienā vai vairākos OS failos, kuri var atrasties uz dažādiem datu nesējiem. Pie tam vienas tabulas ieraksti var tikt izvietoti pa vairākos failos, lietotājam pat to nezinot.



⇒ Segments

Katrā segmentā glabājas kāda atsevišķa DB tabula vai indekss. Segments sastāv no ekstentiem.

⇒ Ekstens

Ekstentu varētu analogijā uztvert kā vietu, kas nepieciešama viena ieraksta glabāšanai. (bet ne vienmēr tā ir).

⇒ Datu bloks

Par datu bloku (*data block*) tiek saukta mazākā vienība, ko DBVS spēj nolasīt no diska. Šis DB datu bloks nav tas pats kas datu nesēja (cietā diska) datu bloks. Parasti šis DVBS datu bloks ir tikpat liels kā cietā diska bloks, bet DB datu bloka lielumu ir iespējams mainīt norādot to DB (vai DBVS) parametros. Ieteicams lietot DB bloka lielumus, kuri sastāv no vesela skaita cietā diska bloku. Datu blokus sauc arī par lapām (*pages*).

Datu bloks sastāv no šādām daļām:

1. iesākums (*header*), kurš satur informāciju par bloka adresi
2. tabulas direktorijs (*table directory*), kurš satur informāciju par to, kuras tabulas dati atrodas šajā blokā
3. ieraksta direktorijs (*row directory*), kurš satur informāciju par to, kura ieraksta kādi dati atrodas šajā blokā ieskaitot konkrētas adrese ierakstu datos
4. ierakstu dati (*row data*), kur glabājas paši dati (viens ieraksts var atrasties arī vairākos blokos)
5. Ierakstu organizācija failā

- MySQL

MySQL ir iespējams izmantot vairākus tabulu tipus, kuri izmanto dažādus DB organizācijas modeļus.

⇒ MyISAM

Šajā gadījumā katrai tabulai tiek veidoti 3 atsevišķi OS faili. Vienā no tiem ar paplašinājumu glabājas dati par faila struktūru, otrā indeksi un trešajā paši dati.

⇒ InnoDB

DB tiek glabāta izmantojot iepriekš apskatīto tabulvietas organizācijas modeli. Atšķirībā no vispārīgā modeļa, DB var būt tikai viena tabulvieta, kura var būt izvietota vairākos failos.

- PostgreSQL

PostgreSQL sākot ar versiju 5.0 uztur tabulvietas. Ar komandas "CREATE TABLESPACE" ir iespējams izveidot vienu vai vairākas tabulvietas un tad veidojot tabulas ir iespējams norādīt, kurā no šīm tabulvietām atradīsies jaunveidojamā tabula.

```
CREATE TABLE studenti(i int, vards varchar)
TABLESPACE tabulvieta_1;
```

Ar PostgreSQL datu bloku uzbūvi iespējams sīkāk iepazīties dokumentācijā:

(<http://www.postgresql.org/docs/8.0/interactive/storage-page-layout.html>)

9.6. FAILA ORGANIZĀCIJA:

Dati parasti tiek glabāti pa ierakstiem. Ieraksts sastāv no vairākiem laukiem, kuriem ir nodefinēti datu tipi. Izdalās divu veidu lauki. Vieni ir ar fiksētu garumu, kad lauka garums jau iepriekš ir stingri noteikts un otri ir mainīga garuma lauki. Mainīga garuma lauki tehniski ir sarežģītāk realizējami, un darbības ar tiem ir lēnākas, turpretim ja datu objekta lielums var būtiski mainīties no ieraksta uz ierakstu, tad šādi lauki ļauj būtiski samazināt DB lielumu. Šādā gadījumā ieraksti tiek atdalīti viens no otra ar speciālu simbolu vai simbolu virkni.

Failu organizācija (*file organisation*) ir veids, kā ierasti tiek fiziski sakāroti failā. Izmantojot mūsdienīgas DBVS nav jāuztraucas par to, kā veidot šo failu organizāciju, jo tās veidošana uzticēta DBVS, tomēr lietotājam ir dota iespēja izvēlēties kādu no konkrētās DBVS uzturētajiem failu organizācijas modeļiem. Katrai no šiem failu organizācijas modeļiem ir savas priekšrocības un trūkumi.

- Teorija

- ⇒ Virknes faila organizācija

Virknes faila organizācija (*sequential file organization*) – ieraksti failā tiek glabāti pēc kārtas sakārtoti pēc primārās atslēgas vērtības (netiek izmantota).

- ⇒ Parastā faila organizācija

Parastā faila organizācija (*ordinary/heap file organization*) – ieraksti failā tiek glabāti viens aiz otra to pievienošanas kārtībā. Pēc pievienošanas ieraksti savu vietu tabulā nemaina. Tabula var būt indeksēta, bet šis indekss nav tabulas daļa. (Oracle šādu tabulas organizāciju sauc par heap organization)

- ⇒ Indeksēta faila organizācija

Indeksēta faila organizācija (*indexed file organization*) – indeksi failā tiek glabāti B-Tree struktūras veidā kā indeksu lietojot primāro atslēgu. Jaunu ierakstu pievienošana vai dzēšana (un sekojoša pārindeksācija) var radīt ierakstu pārvietošanos uz citu vietu tabulā. Šādas tabulas organizācijas priekšrocības ir ātrāka meklēšana pēc primārā indeksa. Ja primārā atslēga ir liela (izmēra ziņā), tad tiek ietaupīta vieta, jo parastā faila organizācijā primārā indeksa dati tiek glabāti gan tabulā gan to dublikāti indeksā.

- ⇒ Hašēta faila organizācija

Hašētā faila organizācijā (*hashed file organization*) katra ieraksta adrese tiek noteikta no primārās atslēgas izmantojot hašēšanas algoritmu. Tipisks hašēšanas algoritms ir primārās atslēgas dalīšana ar atbilstošu pirmskaitli un atlikuma izmantošana par adresi. Pirmskaitli parasti izvēlas tuvu plānotajam ierakstu skaitam. Piemēram mums ir 1000 studenti un studentu apliecības numurs sastāv no 10 cipariem. Tādā gadījumā par pirmskaitli izvēlamies 997 un katra studenta apliecības numuru izdalām ar 997 un datus noglabājam ierastā, kuru iegūstam kā šīs operācijas atlikumu. Problēmas var radīt ja diviem numuriem iegūst vienu un to pašu adresi. Lai risinātu šo problēmu tiek izmantotas speciālas metodes, ko sīkāk neapskatīsim.

Ietekme	Virknes	Parastā	Indeksēta	Hašēta
Vieta uz diska	Optimāli	Optimāli	Optimāli, bet nepieciešama papildus vieta indekam	nepieciešama papildus vieta
Virknes lasīšana pēc primārās atslēgas	ļoti ātra	vidēja	ātra	ātra
Nejauša lasīšana pēc primārās atslēgas	ļoti lēna	vidēja	ātra	ļoti ātra
Ierakstu dzēšana	paliek neizmantošana vieta	paliek neizmantošana vieta	jāveic pārindeksācija	ļoti viegli
Ierakstu pievienošana	jāpārraksta pus fails	ļoti viegla	jāveic pārindeksācija	ļoti viegli
Ierakstu labošana	jāpārraksta pus fails	vidēja	jāveic pārindeksācija	ļoti viegli

- PostgreSQL

PostgreSQL neparedz nekādas iespējas mainīt tabulu tipus. No parastās faila organizācijas ir iespējams iegūt indeksētu faila organizāciju izmantojot komandu

```
CLUSTER indekss ON tabula
```

Šajā gadījumā indeksam dotajā tabulā ir jau jābūt definētam jau iepriekš. Izpildot šo komandu notiks fiziska ierakstu pārkārtošana atbilstoši indeksā uzstādītajai kārtībai.

- MySQL

MySQL ir tabulas tips HEAP. Šī tipa tabulas tiek veidotas izmantojot hašēto faila organizāciju. Šis tabulu tips var tikt izmantots pagaidu tabulu veidošanai atmiņā. (Jāpiezīmē, ka Oracle un citā literatūra ar HEAP apzīmē parasto failu organizāciju).

Pārējie tabulu tipi izmanto parasto failu organizāciju.

```
CREATE TABLE test TYPE=HEAP
SELECT ip,SUM(downloads) as down
FROM log_table GROUP BY ip;
```

9.7. INDEKSI

Lielākā daļa no DB operācijām ir saistīta ar kāda ierakstu atrašanu, kas atbilst uzstādītajiem nosacījumiem. Katras rindiņas pārbaude tabulā, vai tā atbilst vēlamajiem nosacījumiem var būt nepieņemami lēna, it sevišķi ja tabula satur daudzus ierakstus. Lai paātrinātu piekļuvi datiem šādos gadījumos tiek lietoti indeksi.

Indekss ir datu struktūra, izmantojot kuru ir iespējams paātrināt vajadzīgās atribūta vērtības atrašanu relācijā.

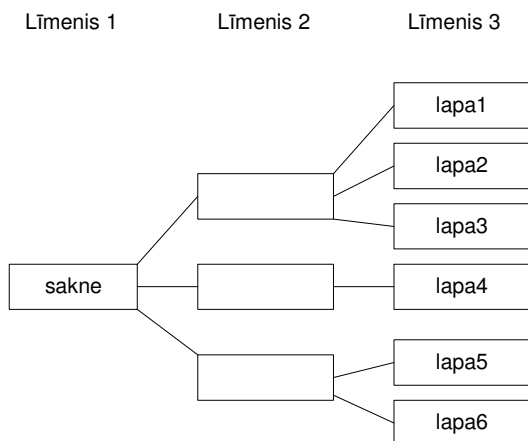
Mūsdienās DBVS pārsvarā tiek lietoti daudzlīmeņu (multilevel) indeksi, kuri balstās uz B+-tree datu struktūrām.

Tas ļauj optimāli samazināt nepieciešamo diska lasīšanas operāciju skaitu līdz ieraksta atrašanai.

Popularitāti gūst arī bitmapu indeksi, kas ir ļoti piemēroti dažādiem datu noliktavu risinājumiem.

- B-tree

B+-tree ir balansēta koka struktūra, tas nozīmē, ka lai nonāktu no koka saknes (root) līdz jebkurai lapai ir jāveic vienāds skaits soļu.



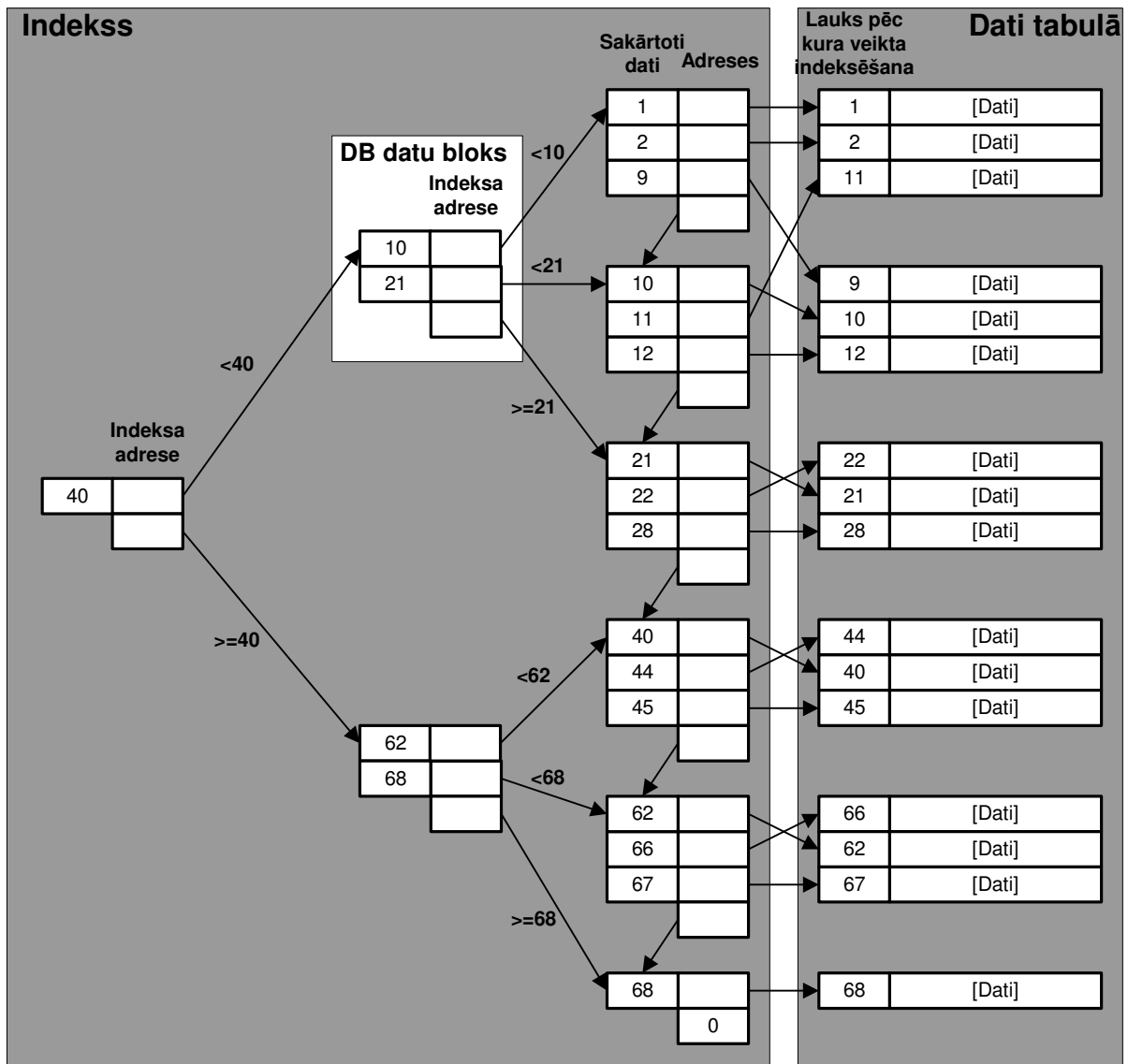
Attēlā redzams B+-tree koks, kurā no elementi izkārtoti trīs līmeņos un visas lapas atrodas vienādā attālumā no saknes.

No saknes vai zarošanās punkta iziet viena vai vairākas saites. To skaits katrā punktā var atšķirties, bet nevar būt tā, ka no dalīšanās punkta neiziet neviena saite.

Šādu struktūru sauc par daudzlīmeņu indeksāciju, kur augstākie līmeņi indeksē nākamā zemākā līmeņa datus.

B+-tree dalīšanās punktos glabājas dalošā informācija un nākamā līmeņa indeksa adreses un tikai lapās glabājas datu adreses.

Šādi vienkāršoti izskatās indeksu koks:



To, cik saites iziet no katra dalīšanās punkta nosaka vairāki parametri:

indeksējamā lauka lieluma,

DB bloka lieluma

adreses lieluma.

Parasti vienā dalīšanās punktā ietver tik daudz saišu, par cik informāciju ir iespējams saglabāt vienā DB blokā, kas ir minimālā lasīšanas vienība no diska.

Pieņemsim, ka indeksējamais lauks ir integer – 4 baiti, DB bloka lielums ir 1024 baiti un adreses lielums ir 5 baiti. Tādā gadījumā vienā blokā, ietilpst p dalīšanās ieraksti.

$$p \cdot 4 + (p+1) \cdot 5 = 1024 ;$$

$$9p = 1024 - 5$$

$$p = 1019 / 9 = 113.$$

Lai veiktu jaunu ierastu pievienošanu šādā kokā ir izveidots speciāls algoritms, kas pārkārto šo datu struktūru atbilstoši jaunajam stāvoklim. Līdzīgi ir arī ar datu dzēšanu no koka. Papildus tam šajos algoritmos ir ietverti mehānismi, kas koku pārkārto, ja tā viena zarošanās punkta izejošo saišu apjoms pēc jaunu datu pievienošanas pārsniedz pieļaujamo. Taču sīkāk šos algoritmu neapskatīsim.

- Bitmapu indeksi

Šie indeksi ir ļoti piemēroti tabulām, kuru kolonnas satur bieži atkārtojošos (zemas kardinalitātes) informāciju. Piemēram studentu tabulā tās varētu būt nodaļas (vai grupas) kolonnas.

Indekss tiek veidots kā bitu tabula (katra vērtība var būt 1 vai 0), kur rindiņas satur ieraskta identifikatoru un attiecīgo indekss bitu vērtības. Kolonnas ir visas iespējamās datu vērtības. Piemēram:

Datu tabula:

ID	Vārds	Uzvārds	Grupa
1	Jānis	Kociņš	IT
2	Juris	Bērziņš	BI
3	Kārlis	Priedītis	TU
4	Andris	Krūmiņš	KO

Bitmapu indekss:

ID	IT	BI	TU	KO
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

Bitmapu indeksi ir ļoti ātri lasīšanā, bet to veidošana ir ļoti sarežģīta, tāpēc tos nav efektīvi izmantot attiecībā uz tabulu kolonnām, kuras bieži tiek mainītas.

- Indeksu izmantošana

Veidojot DB, ir jāizvēlas kādām kolonnām vai to kombinācijām indeksus veidot. Ir jāizvēlas starp meklēšanas ātrdarbību, ko dod šie indeksi un lēnāku ierakstu pievienošanu, dzēšanu un mainīšanu. Tāpēc, ja tabula tiek mainīta salīdzinoši reti, tad tai var tikt veidoti daudz un dažādi indeksi, kas uzlabo ierakstu meklēšanas ātrumu, bet ja dati tabulā tiek regulāri mainīti, tad jāmēģina iztikt tikai ar primārās atslēgas indeksu.

Kā izvēlēties kādus indeksus izmantot:

- ⇒ Indeksi vislielāko efektivitāti dod lielās tabulās,
- ⇒ katrai tabulai jābūt primārai atslēgai, kas vienlaicīgi ir arī indekss.
- ⇒ indeksi ir ļoti lietderīgi kolonnām, kuras bieži tiek izmantotas atlasot datus ar WHERE
- ⇒ indeksi ir ļoti lietderīgi kolonnām, kuras bieži tiek kārtotas ar ORDER BY un GROUP BY
- ⇒ indekss ir jāizmanto tikai, ja kolonnā ir daudz dažādu vērtību. Nav ieteicams veidot indeksu kolonnām, kuras satur nedaudzās bieži atkārtojošās vērtības. Piemēram nodaļu, kurās studē students.
- ⇒ Pirms veidot indeksus ieteicams iepazīties ar konkrētās DVBS ierobežojumiem. Daudzām sistēmām ir ierobežots indeksu skaits, ko var veidot vienai tabulai un kā arī ir ierobežots indeksa atslēgas garums.
- ⇒ Jāuzmanās veidojot indeksus kolonnām, kuru vērtības var būt NULL. Bieži vien kolonnas ar NULL vērtībām netiek iekļautas indeksā. Līdz ar to nedarbojas meklēšana, kur nosacījums ir ATRIBŪTS = NULL.

- Praktiskā realizācija

Parasti indeksi tiek veidoti definējot tabulas, tomēr tos iespējams izveidot arī ar atsevišķas komandu pēc tabulas izveides.

- ⇒ PostgreSQL

```
CREATE [ UNIQUE ] INDEX indeksa_nosauk ON tabulas_nos  
[ USING veids ] ( kolonna [, ... ] )  
[ WHERE nosacijums ]
```

USING – norāda kāda veida indekss tiks veidots:

BTREE - Lehman-Yao high-concurrency B-trees (pēc noklusēšanas)

RTREE - R-trees using Guttman's quadratic split algorithm.

HASH - Litwin's linear hashing.

GIST - Generalized Index Search Trees.

WHERE nosacījums – indekss tiks veidots tikai daļai no tabulas, tiem ierakstiem, kuriem nosacījums ir patiess.

Piemēram :

```
CREATE INDEX i_m_mod ON masinas (marka,modelis);
```

- ⇒ MySQL

```
CREATE [UNIQUE|FULLTEXT] INDEX indeksa_nos ON tabulas_nos  
(kol_nos[(garums)],... )
```

UNIQUE - katra indeksējamā vērtība ir unikāla un nevar atkārtoties

FULLTEXT – speciāls text un varchar tipa lauku indeksēšanas veids, izmantojot kuru pēc tam var veikt meklēšana ar `MATCH(kolonna) AGAINST(mekl_teksts)`

indeksa_nos - indeksa nosaukums

tabulas_nos - tabulas nosaukums

kol_nos – kolonnas nosaukums pēc kuras notiek indeksēšana

garums – var un varchar datu tiem var uzstādīt cik garu daļu no vērtības izmantot indeksā.

9.8. PĀRMAKŠĀ LĪERATŪRA

Database Management Systems - III Data storage & indeksing 195-298.lpp;

MySQL dokumentācija sadaļa 13.1.4. CREATE INDEX Syntax;

PostgreSQL dokumentācija sadaļa 11. Indexes;

PostgreSQL dokumentācija sadaļas CREATE INDEX komandu ietojums;

9.9. PĀRMAKŠĀS JĀUTĀJUMI.

1. Kas ir RAID, kāpēc to lieto.
2. Kādi ir RAID veidi, kā tie atšķiras.
3. DB organizācijas modeļi – no kā sastāv DB.
4. Kas ir faila organizācija.
5. Kādi ir failu organizācijas modeļi un kas ir katra modeļa pamatiezīmes.
6. Kas ir indeksi, kam tos lieto.
7. Kā darbojas indeksi.
8. Kādi ir indeksu veidi
9. Ar kādām komandām veido indeksus.

10. Citi DB objekti

10.1. SECĪBAS (*SEQUENCES*) POSTGRESQL

Secības tiek lietotas, lai automātiski veidotu numurus unikālai ierakstu identificēšanai. Secības veido izmantojot komandu `CREATE SEQUENCE sec_nosauk [INCREMENT increments]`.

Lai ‘insert’ komandā piešķirtu šo vērtību, jālieto atslēgas vārds ‘default’.

Piemēram:

```
CREATE SEQUENCE studenti_id;  
insert into studenti values (DEFAULT, 'Jānis', 'Kociņš', 1, '101010-  
12121');
```

Bieži vien rodas nepieciešamība jau pirms ierasta pievienošanas iegūt šo vērtību, lai to piemērām izmantotu datu izdrukā. Tādos gadījumos var izmantot funkciju `nextval(secība)`;

Tabulā veidojot kolonnu ar tipu 'serial' tiek automātiski izveidota sekvenca ar nosaukumu `tabulasnosaukums_laukanosaukums_seq`.

10.2. SKATI (*VIEWS*)

Skati pieder pie tām lietām, kas pašas par sevi ir vienkāršas, bet grūti definējamas vienkāršā un saprotamā veidā.

Definīcija - Skats ir tabula, kas iegūta no citām tabulām. Šīs tabulas var būt parastas tabulas vai arī citi skati.

Skats savā būtībā ir virtuāla tabula, tā pati par sevi datu ieraktus nesatur, bet attēlo datus no citām tabulām. No skatiem datus parasti var tikai lasīt, bet tos nevar pievienot vai mainīt.

Skatus veido izmantojot komandu:

```
CREATE VIEW nosaukums AS pieprasijums
```

Piemēram:

```
CREATE VIEW skats_1 AS SELECT vards, uzvards, pers_kods from studenti
```

11. Datu integritāte

11.1. NULL VĒRTĪBAS.

NULL vērtība tiek lietota, kad patiesā atribūta vērtība nav zināma. NULL nav tas pats, kas tukša rinda vai skaitlis 0. Tie sniedz kādu informāciju par šo vērtību, NULL pasaka, ka informācijas par šo vērtību nav.

Šobrīd relāciju teorijā nav risinājuma, kas ļautu izmantot NULL vērtības dažādās operācijās un iegūt vēlamos rezultātus, tāpēc pirms pieļaut kolonnai NULL vērtību vajag rūpīgi apsvērt kādas problēmas tas varētu radīt.

Problēmas, ko var radīt NULL vērtības:

- NULL un JOIN

Problēmas rodas, ja mēģina ar JOIN apvienot relācijas, kur kāda no saites atribūtu vērtībām ir NULL. Tādā gadījumā izmantojot dažādus JOIN veidus (INNER JOIN/RIGHT JOIN/LEFT JOIN), iespējams iegūt dažādus rezultātus. INNER JOIN neparādās neviens no ierakstiem, kur saite ir NULL, pat ja abās tabulās ir ieraksti ar NULL vērtībām. RIGHT JOIN/LEFT JOIN attiecīgi parāda vienu no abām tabulām pilnībā, bet otrā tikai tās vērtības, kurām ir saite un tā nav NULL. Sīkāk tas tiks apskatīts pie komandas JOIN.

- NULL un referenciālā integritāte

Ja ārējā atslēga satur NULL, tad šādu atslēgu ļauj pievienot, lai arī vēlāk nav iespējams iegūt primāro atslēgu uz kuru tā atsaucas.

- NULL ietekme uz aprēķiniem

Tāpat NULL ietekme uz aprēķiniem (SUM/AVERAGE/COUNT) ir uzmanīgi jāpārbauda.

Tāpēc ieteicams NULL atļaut lietot tikai tajos atribūtos, kuri netiek izmantoti, lai veiktu sekojošas darbības:

veidotu JOIN saites,

kā ārējā atslēga,

aprēķinos.

11.2. DATU INTEGRITĀTE

Par datu integritāti sauc datu stāvokli, kad tie ir pilnīgi, precīzi un nepretrunīgi. Lai nodrošinātu šādu datu stāvokli un pēc iespējas novērstu dažādu darbību veikšanu, kuru rezultātā tiks pārkāpta datu integritāte, tiek pielietoti dažādi formāli ierobežojumi (constraints).

Jau 1. lekcijā mēs apskatījām tabulas izveidošanas komandu .

```
CREATE TABLE tabulas_nosaukums (  
kolonnas_nosaukums1 datu_tips [DEFAULT vērtība], [kolonnas_ierobežojumi1]  
kolonnas_nosaukums2 datu_tips [DEFAULT vērtība], [kolonnas_ierobežojumi2]  
.....  
[tabulas_ierobežojumi]  
)
```

Tagad sīkāk pievērsīsimies daļām, kurās tiek definēti kolonnas un tabulas ierobežojumus. Tās ir divas dažādas ierobežojumu realizācijas formas. Lielā mērā tās dublē viena otru un, lai iegūtu vēlamu rezultātu, ir iespējams izmantot jebkuru no šīm ierobežojumu formām. Tomēr ir nosacījumi, kurus iespējams izveidot tikai ar tabulu ierobežojumu palīdzību.

Ierobežojumus pēc to būtības var iedalīt sīkāk 3 veidos. Katru no tiem var realizēt gan ar kolonnas gan tabulas ierobežojumiem.

11.3. IEROBEŽOJUMUS SADALĪJUMS:

- Tabulas integritātes ierobežojumi (*entity integrity*):

Šie nosacījumi nosaka, ka katrai tabulai ir jābūt derīgai (valid) primārajai atslēgai. Tas nozīmē, ka visām primārās atslēgas vērtībām jābūt unikālām nenulles vērtībām.

NOT NULL – kolonnas vērtība nevar saturēt NULL vērtību

NULL – tiek atļautas NULL vērtības kolonnā (pēc noklusēšanas)

UNIQUE – kolonnas vērtībām ir jābūt unikālām

PRIMARY KEY – šī kolonna tiek uzstādīta par primāro atslēgu. Tai ir spēkā NOT NULL un UNIQUE. Katrai tabulai var būt tikai viena primārā atslēga.

Der iegaumēt, ka NULL un NOT NULL var izmantot tikai kā kolonnu ierobežojumus.

- Domēna nosacījumi (*domain constraints*):

Visi viena lauka ieraksti var saturēt datus tikai no viena domēna. Par domēnu sauc atomāru datu vērtību kopu. Bieži vien ir nepieciešams ļoti precīzi nodefinēt šo kopu. Vispārīgākā gadījumā domēns var būt datu tips (katram datu tipam ir galīgs skaits iespējamo vērtību). Tomēr var būt arī nepieciešamība arī stingrāk noteikt šo pieļaujamo datu vērtību kopu. Piemēram, nosakot, ka kādas kolonnas vērtība var būt tikai skaitlis starp 0 un 99.

CHECK (nosacījums) - lai varētu veikt datu izmaiņas DB, nosacījumam ir jāatgriež true.

CHECK nosacījumā var izmantot tikai tekošā ieraksta kolonnas un dažādas operācijas ar tām. Nevar tikt izmantoti SELECT operatori.

- Saišu integritāte (*referential integrity*):

Šie nosacījumi nosaka, ka katrai ārējai atslēgas vērtībai ir jābūt atbilstošai primārās atslēgas vērtībai uz kuru šī ārējā atslēga atsaucas. Var būt īpaši gadījumi kad tiek pieļauta NULL vērtība ārējai atslēgai, bet no tā pēc iespējas vajag izvairīties.

```
FOREIGN KEY ar_atslēga REFERENCES ref_tabula [ ( ref_kolonna ) ] [ MATCH FULL |  
MATCH PARTIAL ]
```


Norāda, ka dotā kolonna var saturēt tikai vērtības, kas ir jau atrodas tabulas *ref_tabula* kolonnā *ref_kolonna*.

NULL vērtības iespējams pievienot, pat ja tabulā uz kuru notiek atsaukšanās šādu vērtību nemaz nav. Opcijas [MATCH FULL | MATCH PARTIAL] tikai norāda kādā veidā notiks vērtību salīdzināšana, ja dažu atribūtu vērtības ir NULL un dažu nav.

(default) – pieļauj, ka dažas vērtības ārējās atslēgas atribūtiem ir NULL un dažas nav NULL

MATCH FULL - nepieļauj NULL vērtības kādā no atslēgas atribūtiem, ja citā atribūtā vērtība nav NULL.

MATCH PARTIAL – nav implementēts.

11.4. KOLONNAS IEROBEŽOJUMI :

Šie ierobežojumi tiek uzstādīti definējot katru kolonnu.

```
kolonnas_nosaukums1 datu_tips [DEFAULT vērtība], [kolonnas_ierobežojumi1]

[ CONSTRAINT ierobezojuma_nosaukums ]
[NOT NULL | NULL | UNIQUE | PRIMARY KEY]
[CHECK (izteiksme)]
[ REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL ] ]
```

ierobezojuma_nosaukums nav obligāts. Nosaukums parādās kļūdas ziņojumā, ja ir uzstādīts, tāpēc ieteicams loģiski saprotams nosaukums, kurš apraksta šī ierobežojuma jēgu.

Piemēram:

```
CREATE TABLE studenti (
    vards      CHAR(20) NOT NULL,
    uzvards    CHAR(30) NOT NULL,
    pers_kods  INT PRIMARY KEY,
    st_apl_nr  INT NOT NULL UNIQUE
);

CREATE TABLE eksamens (
    st_apl_nr  INT NOT NULL REFERENCES studenti (st_apl_nr),
    datums     DATE NOT NULL,
    prieksmeta_id INT NOT NULL,
    atzime     INT CHECK (atzime >=0 AND <= 10)
);
```

11.5. TABULAS IEROBEŽOJUMI:

Šie ierobežojumi tiek definēti pēc tam, kad visas kolonnas un to ierobežojumi ir nodefinēti. Pamatā viss darbojas līdzīgi kā kolonnu ierobežojumu gadījumā tikai ir iespējams izmantot arī kolonnu kombinācijas, lai veidotu vajadzīgos ierobežojumus.

```
[ CONSTRAINT ierobezojuma_nosaukums ]
UNIQUE ( kolonnas_vards [, ... ] ) |
PRIMARY KEY (kolonnas_vards [, ... ] ) |
CHECK (nosacījums) |
```

```
FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ] [ MATCH FULL | MATCH PARTIAL ]
```

Piemēram:

```
CREATE TABLE studenti (  
    vards      CHAR(20) NOT NULL,  
    uzvards    CHAR(30) NOT NULL,  
    pers_kods  INT ,  
    st_apl_nr  INT NOT NULL,  
    CONSTRAINT st_apl_nr_uniq UNIQUE (st_apl_nr)  
    CONSTRAINT pers_kods_pkey PRIMARY KEY (pers_kods)  
);
```

Ja nepieciešams noteikt, ka katram studentam katrā priekšmetā var būt tikai viena atzīme, tad to nevar izdarīt tikai ar kolonnu nosacījumiem. Tādā gadījumā jāizmanto tabulu nosacījumi, kas ļauj izmantot vairākas kolonnas šajos nosacījumos.

```
CREATE TABLE eksamens (  
    st_apl_nr      INT NOT NULL,  
    datums         DATE NOT NULL,  
    prieksmeta_id INT NOT NULL,  
    atzime         INT CONSTRAINT st_apl_prieksm_uniq UNIQUE  
        (st_apl_nr, prieksmeta_id),  
    CONSTRAINT atzimes_check CHECK (atzime >=0 AND <= 10),  
    CONSTRAINT st_apl_f_key FOREIGN KEY (st_apl_nr) REFERENCES studenti  
        (st_apl_nr)  
);
```

11.6. DARBĪBAS, KO NOSAKA SAIŠU INTEGRITĀTE:

Ja tiek mainīti vai dzēsti ieraksti, rodas jautājums – ko darīt ar ierakstiem citās tabulās, kurās atrodas norādes uz maināmajiem/dzēšamajiem ierakstiem.

Piemēram: Ir viena tabula ar datiem par studentiem un otra ar eksāmenu atzīmēm. Ko DBVS darīt, ja mēģina izdzēst studenta ierakstu, uz kuru atsaucas vairāki ieraksti eksāmenu tabulā.

Ir vairākas iespējas, kuru no tām izmantot ir jāizvēlas veidojot DB

- No action

Nekādas darbības (dzēšana vai primārās atslēgas maiņa) nevar tikt veiktas, ja pastāv ieraksti, kuri atsaucas uz šo ierakstu.

- Kaskādes izmaiņas

Ja tiek veiktas darbības ar primāro atslēgu, tiek dzēsti/mainīti arī ārējās atslēgas ieraksti.

- Dzēšana uzstādot NULL

Ja tiek dzēsta primārā atslēga, ierakstiem, kuri uz to atsaucas, ārējās atslēgas vērtība tiek nomainīta uz NULL

- Dzēšana uzstādot vērtību pēc noklusēšanas

Ja tiek dzēsta primārā atslēga, ierakstiem, kuri uz to atsaucas, ārējās atslēgas vērtība tiek nomainīta uz vērtību pēc noklusēšanas .

Lai šīs darbības varētu vadīt izmanto komandu kombināciju. Pirmā daļa nosaka uz kādu darbību būs sekojošā reakcija, otrā daļa nosaka, kāda būs šī reakcija.

ON DELETE ON UPDATE	NO ACTION CASCADE SET NULL SET DEFAULT
------------------------	---

Šīs komandas tiek rakstītas tieši pēc ārējo atslēgu vai saišu definēšanas.

Ja tiek lietots SET NULL un šai kolonnai null vērtības nav atļautas, tad tiek izvadīts kļūdas paziņojums un dzēšana vai izmaiņas nevienā tabulā netiek veiktas. Tas pats attiecas uz gadījumiem, kad SET DEFAULT vērtība neatrodas galvenajā tabulā un līdz ar to tā nevar tikt uzstādīta, jo tad tiktu pārkāpta saišu integritāte.

Piemēram:

```
CREATE TABLE eksamens (  
  st_apl_nr      INT      NOT NULL REFERENCES studenti (st_apl_nr) ON  
    DELETE SET NULL ON UPDATE NO ACTION,  
  datums        DATE     NOT NULL,  
  prieksmeta_id INT     NOT NULL,  
  atzime        INT     CHECK (atzime >=0 AND <= 10)  
);
```

```
CREATE TABLE eksamens (  
  st_apl_nr      INT      NOT NULL,  
  datums        DATE     NOT NULL,  
  prieksmeta_id INT     NOT NULL,  
  atzime        INT     CONSTRAINT st_apl_prieksm_uniq UNIQUE  
    (st_apl_nr, prieksmeta_id),  
  CONSTRAINT atzimes_check CHECK (atzime >=0 AND <= 10),  
  CONSTRAINT st_apl_f_key FOREIGN KEY (st_apl_nr) REFERENCES studenti  
    (st_apl_nr) ON DELETE CASCADE ON UPDATE CASCADE  
);
```

11.7. NOSACĪJUMU PĀRBAUDES BRĪDIS

Transakcijās ir iespējams nosacījumus pārbaudīt tūlīt vai arī atlikt to pārbaudi līdz transakcijas beigām. Lai vadītu šo procesu var tikt lietoti vairāki parametri. (PostgreSQL šobrīd atļauj mainīt pārbaudes brīdi tikai ārējās atslēgas ierobežojumiem.)

- Parametri

⇒ Not deferrable

Nav iespējams pārbīdīt pārbaudes brīdi, pārbaude notiek nekavējoties pēc datu ievades (pēc noklusēšanas)

⇒ Deferrable

Ir iespējams ar pārbaudes brīdi pārbīdīt uz transakcijas beigām

⇒ Initially deferred

Pārbaudes brīdis tiek pārbīdīts uz transakcijas beigām

⇒ Initially immediate

Pārbaude notiek nekavējoties (pēc noklusēšanas)

Visus šos parametrus var uzstādīt veidojot jaunu tabulu. Parametrus `Not deferrable` | `Deferrable` pēc tabulas izveides mainīt nav iespējams. Lai tos izmainītu ir jāveido jauna tabula.

```
CREATE TABLE games
(scores NUMBER,
 CONSTRAINT unq_num UNIQUE (scores)
 DEFERRABLE INITIALLY DEFERRED);
```

- SET CONSTRAINTS

Pārbaudes atribūts `Initially deferred` | `Initially immediate`, kas uzstādīti pēc noklusēšanas, ir iespējams izmainīt ar komandas `SET CONSTRAINTS` palīdzību.

```
SET CONSTRAINTS { ALL | constraint [, ...] } { DEFERRED | IMMEDIATE }
```

Piemēram:

```
SET CONSTRAINT divno_uniq DEFERRED
```

11.8. MYSQL

MySQL nepilnīgi uztur datu integritātes nosacījumus. Netiek uzturēti domēna nosacījumi un saišu integritāte tiek pārbaudīta tikai tabulu tipam InnoDB. Veidojot citu tipu tabulas, komandas, kas nosaka šo integritāti, tiek izpildītas, bet tālāk nekāda reāla integritātes pārbaude netiek veikta.

11.9. PAPILDUS LIERATŪRA

Database Management Systems - INTEGRITY CONSTRAINTS OVER RELATIONS 57-64 lpp

MySQL dokumentācija sadaļa 14.2.6.4. FOREIGN KEY Constraints

PostgreSQL dokumentācija sadaļa 5.3. Constraints

11.10. PAŠKONTROLES JAUTĀJUMI.

1. Kas ir datu integritāte.
2. Kādi ir ierobežojumu veidi, ko katrs no tiem nodrošina.
3. Kādas komandas nodrošina datu integritāti.
4. Kādas darbības iespējams veikt izmantojot saišu integritātes nosacījumus.

12. Datu manipulācijas komandas

12.1. INSERT

SQL komanda "INSERT" veic jaunu ierakstu pievienošanu tabulai. Ar "INSERT" iespējams pievienot gan vienu ierakstu gan arī veselu kopu, ko atgriež kāds pieprasījums. Tāpat iespējams pievienot vērtības tikai vienā vai arī vairākās kolonnās.

Sintakse:

```
INSERT INTO tabula [ ( kolonna [, ...] ) ]  
{ DEFAULT VALUES | VALUES ( izteiksme_1 [, ...] ) | SELECT pieprasijums }
```

Ja kolonnas netiek norādītas, tad vērtība no "izteiksme_1" tiks piešķirta pirmajai kolonnai, otrās – otrajai u.t.t. Gadījumā, ja norādītas vairāk vērtības kā kolonnas, tad tiks izvadīts kļūdas paziņojums un datu pievienošana netiks veikta. Savukārt, ja kolonnas norādītas vairāk nekā pievienojamās vērtības, tad kolonnām, kurām nepietiek vērtības tiks piešķirta NULL vērtība.

Piemēram:

```
INSERT INTO masinas VALUES (1,'BMW','320',13128,'sedans',2003,1);
```

```
INSERT INTO masinas VALUES (2,'BMW');
```

```
INSERT INTO masinas (pk_kods, marka, gads) VALUES (3,'BMW');
```

Lai tabulai pievienotu vairākus ierakstus, jāizmanto opcija "SELECT". Tā dod iespējas pievienot tabulai visas vērtības, ko atgriež dotais pieprasījums. Līdzīgi kā pievienojot vienu ierakstu, arī šajā gadījumā "SELECT" nedrīkst atgriezt vairāk kolonnas kā ir tabulā, kurai vēlas datus pievienot.

Piemēram:

```
INSERT INTO masinas SELECT * FROM masinas1;
```

```
INSERT INTO masinas (marka,modelis) SELECT marka, modelis FROM masinas1;
```

12.2. UPDATE

SQL komanda "UPDATE" veic ierakstu vērtību izmaiņas tabulā.

Sintakse:

```
UPDATE [ONLY] tabula SET kolonna = izteiksme [, ...]  
[ FROM from_saraksts ]  
[ WHERE nosacijums ]
```

Opcija "ONLY" tiek lietota tikai veidojot objektu DB, lai vadītu komandas darbību attiecībā uz šīs tabulas bērnu tabulām.

Opcijas "FROM from_saraksts" pielietojums nav īsti izprotams. Tiek teikt, ka tas tiek izmantots, lai kolonnas no citām tabulām var izmantot "WHERE" nosacījumā, bet to var darīt arī bez tā.

WHERE nosacijums – ļauj uzstādīt nosacījumu, un vērtības tiks mainītas tikai ierakstiem, kur nosacījums ir patiess. Tāpat ar nosacījuma palīdzību ir iespējams veidot saites starp tabulām, lai vienas tabulas ierakstiem piešķirtu atbilstošās vērtības no otras tabulas.

Piemēram:

```
UPDATE masinas SET cena=cena+10;  
UPDATE masinas SET cena=cena+10 WHERE marka="Audi"  
UPDATE masinas SET cena=masinas2.cena  
WHERE pk_kods=masinas2.pk_kods;
```

12.3. DELETE

SQL komanda "DELETE" veic ierakstu dzēšanu no tabulas.

Sintakse:

```
DELETE [ONLY] FROM tabula [WHERE nosacījums]
```

Piemēram:

```
DELETE FROM masinas;  
DELETE FROM masinas WHERE cena>20000;
```

Opcija "ONLY" tiek lietota tikai veidojot objektu DB, lai vadītu komandas darbību attiecībā uz šīs tabulas bērnu tabulām.

Nosacījums, kas seko aiz "WHERE" ir veidojams tāpat kā "SELECT" pieprasījumos. Tiks izdzēsti visi ieraksti, kuriem nosacījums būs patiess.

Ja nosacījums "WHERE" netiek izmantots, tad no tabulas tiek izdzēsti visi ieraksti. Tomēr gadījumos, ja nepieciešams izdzēst visus ierakstus, ieteicams izmantot komandu

TRUNCATE *tabula*

12.4. PAPILDUS LIERATŪRA

PostgreSQL dokumentācija – komandu INSERT, UPDATE un DELETE lietojums.

12.5. PAŠKONTROLES JAUTĀJUMI.

1. Kāpēc nepieciešamas datu manipulācijas komandas.
2. Kā var noteikt, kurs ierakstus izdzēsīs komanda DELETE.
3. Kā izmantojot komandu INSERT tabulai pievienot vairākus ierakstus no kādas citas tabulas.

13. Saglabātās procedūras

13.1. IEVADS

Saglabātās procedūras (*stored procedures*) ir programmu koda moduļi (procedūras), kas tiek glabāti datu bāzē uz servera, un arī to izpilde pēc izsaukšanas notiek uz servera. Saglabātās procedūras darbojas līdzīgi kā "parastās" procedūras, kas rakstītas jebkurā citā programmēšanas valodā, tām ir savs nosaukums, tās var saņemt argumentus un atgriezt vērtības. Lai rakstītu saglabātās procedūras iespējams izmantot gan SQL, gan uz SQL balstītas procedurālas programmēšanas valodas kā PL/pgSQL un PL/SQL, gan arī citas programmēšanas valodas (piem. C). Šo valodu izmantošana atkarīga no katras konkrētās DBVS. Šajā nodaļā apskatīsim tikai saglabāto procedūru veidošanu izmantojot SQL, bet nākamajā nodaļā tiks apskatīta arī PL/pgSQL izmantošana veidojot saglabātās procedūras DBVS PostgreSQL.

13.2. SAGLABĀTO PROCEDŪRU IZMANTOŠANAS PRIEKŠROCĪBAS:

- Uzlabojas ātrdarbība,

Procedūras DB tiek glabātas kompilētā veidā un līdz ar to nav nepieciešams papildus laiks, lai tās pārvērtu mašīnkodā.

- Mazāks atmiņas izlietojums

Katra procedūra atmiņā tiek glabāta tikai vienā eksemplārā arī daudzlietotāju vidē.

- Ar to palīdzību iespējams detalizētāk uzstādīt datu drošības nosacījumus

Piemēram, atļaujot lasīt vai rakstīt datus tikai caur šo procedūru un nevis pa tiešo. Kad procedūra tiek palaista, tā darbojas ar tās īpašnieka (owner) tiesībām un nevis tā, kurš šo procedūru ir palaidis. Līdz ar to lietotājam, kuram piemēram vajadzīga pieeja tikai statistiskiem datiem (kopsūmma/skaits) var liegt tiešu pieeju datiem, bet šo statistikas datu ieguvu veikt izmantojot saglabāto procedūru.

- Ļauj pārdaļīt skaitļošanas noslodzi starp serveri un klientu.

- Samazina datortīkla noslodzi

Ja nepieciešams veikt sarežģītu datu apstrādi, kur SQL iespējas ir nepietiekamas un jāizmanto procedurāla programmēšanas valoda, tad šo apstrādi veicot uz klienta datora tam ir jāpiegādā arī visi nepieciešamie dati, kas visticamāk būs ļoti apjomīgi. Turpretim, ja šīs apstrādes veikšanai tiek izmantotas PL/pgSQL un saglabātās procedūras, tad visa apstrāde notiks uz servera un datortīkls netiks noslogots.

13.3. POSTGRESQL

PostgreSQL saglabātās procedūras var veidot ar komandas `SET FUNCTION` palīdzību. Jāatzīmē, ka dokumentācijā, lai apzīmētu saglabātās procedūras, pārmaiņus tiek lietoti apzīmējumi procedūra un funkcija.

- Sintakse

```
CREATE FUNCTION funkc_nos ( [ arg_tips [, ...] ] )  
  RETURNS [SETOF] atgr_tips  
  AS 'komandas'  
  LANGUAGE prog_valoda  
  [ WITH ( atributs [, ...] ) ]
```

Piemēram:

```
CREATE FUNCTION viens ()  
  RETURNS integer  
  AS  
  'SELECT 1;'  
  LANGUAGE SQL;  
  
SELECT viens();
```

funkc_nos – funkcijas nosaukums

agr_tips – argumenta, kas tiks nodots funkcijai, datu tips.

Pie argumentiem, ko funkcija saņem var griezties izmantojot \$n, kur n ir attiecīgā argumenta kārtas numurs.

atgr_tips – argumenta, kas tiks saņemts no funkcijas, datu tips.

Funkcijai var noteikti, vai tā atgriež vienu vērtību vai vērtību kopu. Vienkāršākajā gadījumā funkcija atgriež vienu vērtību - pēdējā SELECT operatora rezultāta pirmo rindu. Ja šis operators atgriež vairāk kā vienu rindu, tad jāizmanto ORDER BY, lai precīzi noteiktu, kura rinda ir pirmā. Ja SELECT atgriež tukšu kopu, tad funkcija atgriezīs NULL vērtību.

Gadījumā, ja nepieciešams, lai funkcija varētu atgriezt vairāk kā vienu vērtību, lieto opciju SETOF. Tomēr šādu opciju var lietot tikai, ja funkcija tiek izsaukta no SELECT operatora, kur tā kalpo kā datu avots.

Piemēram:

```
CREATE FUNCTION filtrs(decimal, integer)  
  RETURNS SETOF text AS '  
  SELECT nosaukums FROM preces  
    WHERE cena > $1 and preces_veids = $2'  
  LANGUAGE SQL;  
  
SELECT filtrs(10,1);
```

komandas – funkcijas programmas kods

Programmas kodam jābūt ieslēgtam starp vienkāršajām pēdiņām ('). Ja programmas kodā nepieciešamas izmantot šīs pēdiņas, tad tās jālieto divas reizes (") (nejaukt ar dubultajām pēdiņām). Programmas kods var sastāvēt no vienas vai vairākām SQL komandām, kas savā starpā atdalītas ar (;) . Programmas kodā var tikt izmantotas gan SELECT gan arī datu manipulācijas komandas INSERT, UPDATE, DELETE, tomēr pēdējai ir jābūt komandai SELECT, kura nosaka funkcijas atgriežamo vērtību.

Izveidojot funkciju tiek pārbaudīts, vai ir pieejami dati, ar ko tai vajag manipulēt. Piemēram, ja funkcijā ir `SELECT` komanda, tad tiek pārbaudīts, vai dati, ko izmanto `SELECT` eksistē. Ja tie neeksistē, tiek izvadīts kļūdas apziņojums un saglabātā procedūra netiek izveidota.

prog_valoda – Programmēšanas valoda, kurā rakstīta funkcija. Dotajā gadījumā SQL. Lai veidotu sarežģītākas procedūras, var tikt izmantotas arī citas valodas.

atributs – Neobligāts atribūts, ko var norādīt, lai veiktu funkcijas izpildes optimizāciju.

Atribūts var būt :

⇒ `iscachable`

Šī vērtība norāda, ka vieniem un tiem pašiem ieejas datiem, funkcija atgriezīs tās pašas vērtības.

⇒ `isstrict`

Šī vērtība norāda, ka ja kāda no ieejas argumentu vērtībām ir `NULL`, tad funkcija atgriezīs `NULL` vērtību.

Piemēram:

```
CREATE FUNCTION tp1 (integer, numeric)
  RETURNS integer
  AS
  'UPDATE bank SET balance = balance - $2 WHERE accountno = $1;
  SELECT 1; '
  LANGUAGE SQL;

SELECT tp1 (1,1)
```

13.4. MYSQL

MySQL saglabātās procedūras uztur sākot ar 5.0 versiju.

13.5. PAPILDUS LIERATŪRA

MySQL dokumentācija sadaļa 17. Stored Procedures and Functions

PostgreSQL dokumentācija – komandas `CREATE FUNCTION` lietojums

13.6. PAŠKONTROLES JAUTĀJUMI.

1. Kas ir saglabātās procedūras.
2. Kam lieto saglabātās procedūras.
3. Ar kādām komandām veido saglabātās procedūras.

14. PL/pgSQL

SQL valoda ir datu manipulācijas valoda, kas speciāli veidota šādiem mērķiem. Tai pietrūkst universālo programmēšanas valodu iespējas. Tomēr bieži rodas vajadzība pēc plašākām datu apstrādes iespējām, kā to spēja piedāvāt SQL. Tāpēc Oracle izstrādāja procedurālu programmēšanas valodu PL/SQL (Procedural Language/SQL). Tā piedāvāja izstrādātājiem daudz plašākas iespējas datu apstrādē apvienojot sevī SQL darbu ar datiem un universālo valodu plašajās iespējas izmantojot mainīgos, ciklus, nosacījumus un procedūras.

Arī PostgreSQL piedāvā sistēmu izstrādātājiem valodu PL/pgSQL, kas ir ļoti līdzīga Oracle radītajai PL/SQL. Lai izmantotu šo valodu, DBVS vispirms ir jāpievieno šai datu bāzei iespēja izmantot PL/pgSQL. To izdara no komandrindas izpildot komandu :

```
createlang plpgsql db_nosaukums
```

Ja pirms tam PL/pgSQL ir bijusi pievienota DB nokuras tikai izveidota šī datu bāze ar opciju WITH TEPLATE = *db_nosaukums*, tad atkārtota valodas apstrādes pievienošana nav nepieciešama. Ja nekas netiek norādīts, tad jauna datu bāze tiek veidota par pamatu izmantojot datu bāzi „template1”, kurai sākotnēji nav pievienota iespēja izmantot PL/pgSQL.

Lai no SQL koda pievienotu datu bāzei PL/pgSQL, ir jāizpilda sekojošas komandas:

```
CREATE OR REPLACE FUNCTION plpgsql_call_handler()  
  RETURNS language_handler AS  
'C:/Program Files/PostgreSQL/8.0/lib/plpgsql.dll',  
  'plpgsql_call_handler'  
  LANGUAGE 'c' VOLATILE;  
  
CREATE TRUSTED PROCEDURAL LANGUAGE 'plpgsql'  
  HANDLER plpgsql_call_handler;
```

Attiecīgais ceļš var būt jāizmaina atkarībā no *plpgsql.dll* bibliotēkas atrašanās vietas.

14.1. PROGRAMMAS STRUKTŪRA

PL/pgSQL ir strukturēta programmēšanas valoda, kura sastāv no blokiem.

```
[ DECLARE  
  declarācijas ]  
BEGIN  
  Programmas_kods  
END;
```

Piemēram:

```
DECLARE  
  x integer :=0;  
BEGIN  
  WHILE x<10 LOOP  
    INSERT INTO atzimes VALUES (x);  
    x:=x+1;  
  END LOOP;  
RETURN 1;
```

END;

PL/pgSQL var tikt izmantota tikai saglabātajās procedūrās, tāpēc pilna sintakse būtu šāda:

```
CREATE FUNCTION f_1 ()
  RETURNS integer
  AS '
  DECLARE
    x integer :=0;
  BEGIN
    WHILE x<10 LOOP
      INSERT INTO atzimes VALUES (x);
      x:=x+1;
    END LOOP;
    RETURN 1;
  END; '
LANGUAGE PLPGSQL;
```

14.2. ARGUMENTU SAŅEMŠANA UN ATGRIEŠANA

- Argumentu saņemšana

Līdzīgi kā saglabātajās procedūrās izmantojot SQL valodu, arī PL/pgSQL argumentus saņem izmantojot \$1,\$2 utt. Tomēr šeit DECLARE sadaļā ar komandas

```
mainiga_nosaukums ALIAS FOR $n
```

palīdzību šiem argumentiem var piešķirt nosaukumus un padarīt tos līdzīgus pārējiem mainīgajiem. (n – saņemtā argumenta kārtas numurs)

Piemēram:

```
CREATE FUNCTION kvadr (integer)
  RETURNS integer
  AS '
  DECLARE
    x ALIAS FOR $1;
  BEGIN
    x := x * x;
    RETURN x;
  END; '
LANGUAGE PLPGSQL;
```

- Argumentu atgriešana

Lai atgrieztu no funkcijas vērtību, tiek lietota komanda:

```
RETURN izteiksme
```

Katrai funkcijai ir jābūt RETURN komandai. Ja izpildes gaitā tiek sasniegtas funkcijas beigas un nav RETURN komandas, tiek izvadīts kļūdas paziņojums un programmas izpilde tiek pārtraukta.

14.3. KOMENTĀRI:

Komentāri iezīmēšanai lieto:

-- komentārs – šādi iezīmē komentārus, kuri aizņem vienu rindiņu un beidzas līdz ar rindiņas beigām.

/* komentārs */ - šādi var iezīmēt vienu vai vairākas rindiņas saturošus komentārus. Šajā komentārā var tikt ietverti arī ar – veidoti komentāri.

14.4. DEKLARĀCIJAS

Deklarāciju daļā tiek deklarēti mainīgie, kas tiks izmantoti programmas kodā. Visi mainīgie PL/pgSQL ir jādeklarē, izņēmums ir tikai mainīgais, kas tiek izmantots kā skaitītājs „FOR .. LOOP” ciklā.

Sintakse:

```
nosaukums [CONSTANT] tips [NOT NULL] [ { DEFAULT | := } izteiksme ];
```

nosaukums - mainīgā nosaukums

tips - mainīgā datu tips (tie paši datu tipi, ko uztur PostgreSQL)

izteiksme - izteiksme, kuras vērtība kļūs par mainīgā sākotnējo vērtību. Ja sākotnējā vērtība nav noteikta, mainīgais iegūst NULL vērtību.

Piemēram :

```
X intger ;  
Y integer := 10;
```

- Ierakstu mainīgie

Ieraksta mainīgais ir mainīgais, kurš var saglabāt vienu tabulas ierakstu ar vairākām kolonnām. Pie katras atsevišķās lauka vērtības var griezties izmantojot sintaksi mainīgais.kolonna

Sintakse:

```
nosaukums tabbular_nos%ROWTYPE;
```

Piemēram:

```
DECLARE  
    v_atzimes atzimes%ROWTYPE;  
BEGIN  
    SELECT INTO v_atzimes * FROM atzimes;  
    ...  
END;
```

Līdzīgi darbojas

```
nosaukums RECORD
```

Tikai šis mainīgais sākumā netiek piesaistīts nevienai tabulai. Tā struktūra tiek izveidota tikai, kad tam tiek piešķirta vērtība. Pie tam šim mainīgam struktūra var mainīties katru reizi kad tam tiek piešķirta vērtība.

Piemēram:

```
DECLARE
    v_tabula RECORD;
BEGIN
    SELECT INTO v_tabula * FROM atzimes;
    ...
    SELECT INTO v_tabula * FROM auto;
    ...
END;
```

Jāatzīmē, ka izpildot iepriekšējos 2 piemēros minētās operācijas, gadījumos, kad SELECT atgriezīs vairāk kā vienu ierakstu, mainīgais iegūs tikai pirmā ieraksta vērtības.

Vispiemērotākais šo mainīgo pielietojums ir kopā ar komandām FOR un FETCH pieprasījumu rezultātu caurskatīšanai. Sīkāk šīs komandas tiks apskatītas nākamajos punktos.

- Mainīgā tips

Šī komanda ļauj noteikt mainīgā tipu, vai veidot mainīgos izmantojot citu mainīgo vai ierastu datu tipus.

Sintakse:

```
nosaukums tipa_avots%TYPE;
```

Piemēram: (mainīgais v_atzime iegūst tabulas atzime kolonnas atzime tipu.)

```
v_atzime atzime.atzime%TYPE
```

14.5. KONSTRUKCIJAS

- Piešķiršana

Piešķiršanas operators ir :=

Piemēram

```
X := 100;
```

- SELECT INTO

Šī komanda var tikt izmantota, lai piešķirtu mainīgajam kādu tabulas ierakstu vai lauku. Ja SELECT atgriež vairāk kā vienu ierakstu, tad mainīgajam tiks piešķirts pirmais ieraksts. Pie tam, ja nav izmantots ORDER BY, tad nav skaidri noteikts, kurš ieraksts būs pirmais.

Piemēram:

```
SELECT INTO myrec * FROM EMP WHERE empname = myname;
```

- Dinamiskie pieprasījumi

PL/pgSQL piedāvā iespēju veidot pieprasījumus programmas koda izpildes gaitā un pēc tam tos izpildīt. To var veikt ar komandas EXECUTE palīdzību. Teksta mainīgo savienošana notiek ar operatora „||” palīdzību

Piemēram

```
DECLARE
  m_vert ALIAS FOR $1;
  v_piepras varchar (100);
BEGIN
  v_piepras := 'INSERT INTO atzimes VALUES ( '
    || m_vert || ' )';
  EXECUTE v_piepras;
RETURN 1;
END;
```

14.6. KONTROLES STRUKTŪRAS

- IF-THEN-ELSE

Dalīšanās operators. Ir arī citādas sintakses formas ar ELSEIF. Sīkāk skatīt dokumentāciju.

Sintakse:

```
IF nosacījums THEN
  programmas_kods
[ELSE
  programmas_kods]
END IF;
```

Piemēram:

```
IF number = 0 THEN
  result := 'zero';
ELSE
  result := 'positive';
END IF;
```

- LOOP cikla operators

Tiek izpildīts bezgalīgi, līdz to pārtrauc EXIT vai RETURN

Sintakse:

LOOP

programmas_kods

END LOOP;

- Cikla beigu komanda

Šī komanda tiek izmantota, lai beigtu LOOP ciklu izpildi.

Sintakse:

```
EXIT [WHEN nosacījums ];
```

Piemēram:

```
DECLARE
  x integer :=0;
BEGIN
LOOP
  INSERT INTO atzimes VALUES (x);
  x:=x+1;
  EXIT WHEN x=10;
END LOOP;
END;
```

- **WHILE cikla operators**

Darbojas līdzīgi kā LOOP cikls kopā ar EXIT, tikai nosacījums tiek uzstādīts jau pašā WHILE komandā. Bez tam izpilde tiek beigta, kad nosacījums kļūst FALSE (pretēji LOOP)

Sintakse:

```
WHILE nosacijums LOOP
  programmas_kods
END LOOP;
```

Piemēram:

```
DECLARE
  x integer :=0;
BEIGN
WHILE x<10 LOOP
  INSERT INTO atzimes VALUES (x);
  x:=x+1;
END LOOP;
END;
```

- **FOR cikla operators**

Vēl viena komanda, kas darbojas ļoti līdzīgi WHILE un LOOP/EXIT. Šis operators paredzēts cikliskai skaitlisku vērtību pārlasīšanai ar soli +1 vai -1 (ja izmantota opcija REVERSE). Jāpievērš uzmanība tam, ka mainīgais nav jādeklarē, tā palielināšana/samazināšana notiek automātiski un tas darbojas tikai cikla ietvaros.

Sintakse:

```
FOR mainigais IN [ REVERSE ] sakuma_ver .. beigu_vert LOOP
  programmas_kods
END LOOP;
```

Piemēram:

```
FOR x IN 0 .. 10 LOOP
  INSERT INTO atzimes VALUES (x);
END LOOP;
```

- **FOR cikla operators pieprasījumu rezultātu caurskatīšanai**

Svarīgākā FOR iespēja ir veidot ciklus pieprasījuma rezultātu caurskatīšanai.

Sintakse:

```
FOR ier_mainigais IN select_pieprasijums LOOP
  programmas_kods
END LOOP;
```


Piemēram:

```
DECLARE
  x RECORD;
  sum integer :=0;
BEGIN
  FOR x IN SELECT * FROM atzimes LOOP
    sum:=sum+x.atzime;
  END LOOP;
  RETURN sum;
END;
```

14.7. KURSORI

Kursors (*Cursors*) ir datu struktūra, kurā programmas darbības laikā uz laiku var tikt glabāti pieprasījumu rezultāti. Kursoros ir iespēja griezties pie katras rindiņas atsevišķi izmantojot komandu FETCH. Tas var tikt izmantots, lai samazinātu atmiņas noslodzi, ja pieprasījums atgriež daudz ierakstu. Papildus tam kursora mainīgais var tikt atgriezts no saglabātajām procedūrām, tā dodot iespēju atgriezt vairākus ierastus.

- Kursora mainīgais

Griešanās pie kursora notiek izmantojot kursora mainīgo, kura tips ir refcursor. Šis mainīgais var tikt izveidots vai nu to deklarējot ar šo tipu vai arī

```
nosaukums CURSOR [ ( argumenti ) ] FOR select_pieprasijums ;
```

nosaukums - kursora mainīgā nosaukums

argumenti – pāris mainīgais un datu tips, ko kursors var saņemt kā parametrus

select_pieprasijums – select pieprasījuma teksts.

Piemēram:

```
DECLARE
  v_kurs1 refcursor;
  v_kurs2 CURSOR FOR SELECT * from eksamens;
  v_kurs3 CURSOR (v_atzime int) FOR SELECT * from eksamens where
    atzime = v_atzime;
```

- Kursora atvēršana

Lai varētu izmantot kursoru, tas vispirms ir jāatver. To var izdarīt vairākos veidos:

⇒ OPEN FOR SELECT

Kursors tiek atvērts un tam tiek piešķirts norādītais pieprasījums. Šādi var tikt atvērts kursors, kurš nodefinēts ar refcursor.

Piemēram:

```
OPEN v_kurs1 CURSOR SELECT * FROM eksamens;
```

⇒ OPEN FOR EXECUTE

Atver kursoru veidojot pieprasījumu dinamiski.

Piemēram:

```
v_piepras := 'SELECT * FROM eksamens';  
OPEN curs1 FOR EXECUTE v_piepras;
```

⇒ OPEN (ar parametriem)

Atver iepriekš definētu kursoru.

Piemēram:

```
OPEN v_kurs2;  
OPEN v_kurs3(10);
```

Ja, kursors ir jau atvērts, tas vispirms ir jāaizver un tikai tad to var atkārtoti atvērt.

- Kursora aizvēršana

Kursoru aizverot tiek atbrīvoti resursi, ko tas izmantoja.

Piemēram :

```
CLOSE v_kurs1;
```

- Kursora izmantošana

Kursors var tikt izmantots, lai grieztos pie kādas atsevišķa šī kursora ieraksta.

⇒ FETCH

Šī komanda atgriež kursora nākamo ierakstu.

```
FETCH kursors INTO mainigais;
```

Mainīgais šajā gadījumā var būt ierasta tipa (RECORD vai % ROWTYPE) vai arī vairāku mainīgo saraksts, kur katrs mainīgais iegūst viena lauka vērtību.

Piemēram:

```
FETCH v_kursors INTO v_main;
```

14.8. PAZIŅOJUMU IZVADĪŠANA

PostgreSQL piedāvā iespēju log failā vai servera konsoles logā izvadīt paziņojumus. (Protams gadījumos, ja log fails vispār tiek veidots). To var izmantot ne tikai kļūdu gadījumā, bet arī izstrādājot sistēmu debuging procesā, lai kontrolētu mainīgo vērtības.

```
RAISE NOTICE | DEBUG 'Paziņojums % % ', argumants_1, argumants_2;
```

Šajā gadījumā % tiek lietots vietā, kur jāizvada arguments. Pirmais % izvadot paziņojumu tiks aizvietots ar pirmo argumants_1, otrais % ar argumants_2 utt.

Piemēram :

```
RAISE DEBUG 'Kļūda ' ' ;
```

14.9. PAPILDUS LIERATŪRA

PostgreSQL dokumentācija sadaļa 36. PL/pgSQL

14.10. PAŠKONTROLES JAUTĀJUMI.

1. Kas ir PL/pgSQL valoda.
2. Kāpēc nepieciešama PL/pgSQL valoda.
3. Kādas ir pamata atšķirības starp SQL un PL/pgSQL.

15. Triggeri

Par triggeri (*trigger*) sauc SQL komandu kopu, kura tiek glabāta datu bāzē un kura tiek automātiski izpildīta, ja tiek veikta attiecīgo triggeri izsaucoša darbība. Darbības, kas var izsaukt triggera izpildi var būt datu manipulācijas komandu INSERT, UPDATE un DELETE izpilde.

Veicot darbības ar datiem viens triggeris var izsaukt arī otra triggera izpildīšanos. Tādā veidā viens klienta pieprasījums var izsaukt virkni integritātes vai loģikas pārbaudes tieši uz servera, neradot papildus tīkla noslodzi.

Jāpiebilst, ka SQL92 standartā triggeri netiek definēti, tie parādīsies tikai jaunajā SQL2003 standartā.

PostgreSQL uztur triggerus un realizē tos ar saglabāto procedūru palīdzību. Citās DBVS sistēmās triggeru kods ir pašu triggeru sastāvdaļa un saglabāto procedūru izmantošana nav nepieciešama. MySQL pagaidām neuztur triggerus, bet sola ka tie parādīsies versijā 4.1

15.1. TRIGERI VAR TIKT LIETOTI, LAI :

- veidotu sarežģītus drošības uzstādījumus,

Piemēram noteikt, ka kāda konkrētam klientam datus var mainīt tikai klientu menedžeris, kas ir tam piesaistīts.

- veidotu sarežģītus biznesa likumus datu integritātei,

CHECK nevar izmantot SELECT operatoru, lai iegūtu no DB nepieciešamo informāciju un izmantotu to pārbaudot nepieciešamo biznesa uzstādījumu ievērošanu. Ar triggeriem šo ierobežojumu var apiet.

- veidotu sarežģītu atsauksmju integritāti,

Triggerus ir iespējams lietot arī vienkāršāku integritātes nosacījumu pārbaudei (NOT NULL, UNIQUE, CHECK), tomēr tiek rekomendēts izmantot uzstādītos integritātes nosacījumus, kad vien tie spēj nodrošināt nepieciešamo funkcionalitāti.

- nodrošinātu specifisku audita datu vākšanu

Piemēram kad kāds lietotājs ir veicis izmaiņas kādā konkrētā tabulā.

- veidotu kopiju tabulas (*replicas*)

Šādas iespējas diez vai jums būs vajadzīgas pārāk bieži. Liela daļa mūsdienu DBVS nodrošina replikāciju veidošanas utilītas, tomēr tādu nav PostgreSQL un MySQL.

- kā arī citiem mērķiem pēc sistēmu veidotāja prasmes un ieskatiem.

15.2. TRIGERI SASTĀV NO ČETRĀM DAĻĀM:

- Notikums

Notikums ir darbība, kas izraisa trigeru izpildi. Parasti notikumi ir operācijas, kas veic datu izmaiņas datu bāzē, lai arī vispārīgā gadījumā šie notikumi varētu būt laika notikumi (trigeris tiktu izpildīts noteiktā laikā), vai arī kādi citi notikumi. Oracle atļauj veidot trigerus, kur notikums var būt DB struktūras maiņa vai DB servera palaišana un apstādināšana. PostgreSQL atļauj tikai trigerus, ko izsauc operācijas INSERT, UPDATE un DELETE. Arī jaunajā SQL2003 standartā ir tikai šie notikumi. Trigeri var izsaukt arī jebkurš no vairākiem iepriekšminētajiem notikumiem, ja trigerī tas ir nodefinēts.

- Trigeru izpildes brīdis

Trigeris var tikt izpildīts pirms vai pēc notikumu izraisījušo operāciju izpildīšanas (BEFORE | AFTER). Piemēram, ja trigeri izsauc notikums INSERT, tad vienā gadījumā trigeru darbības izpildīsies pirms dati tiks ierakstīti datu bāzē, bet otrā – pēc šo datu ierakstīšanas DB.

BEFORE – trigeris tiks izpildīts pirms izraisošā notikuma

AFTER - trigeris tiks izpildīts pēc izraisošā notikuma.

- Izpildes līmenis:

FOR EACH ROW – trigeris tiek izpildīts katrai rindai, kas tiek mainīta.

FOR EACH STATEMENT - trigeris tiek izpildīts katrai komandai tikai 1 reizi, pat ja tiek izmainīti dati vairākos ierakstos. (PostgreSQL šobrīd nedarbojas)

- Darbības

Šajā daļā tiek definētas darbības, kas jāveic, ja ir veikta trigeru izsaucošā darbība. Vispārējā gadījumā te seko SQL komandu virkne. PostgreSQL darbības ir jā saglabā DB kā atsevišķa saglabātā procedūra un tad tā tiek izsauc no trigeru.

15.3. POSTGRESQL

Katrā DBVS sintakse var mazliet atšķirties. Parasti komandas tiek rakstītas pašā trigerī, bet PostgreSQL vajag izveidot atsevišķu saglabāto procedūru, kura veic nepieciešamās darbības. Šī funkcijas atgriež mainīgo ar datu tipu OPAQUE. Tas ir datu tips, kas konkrēti neraksturo atgriežamā mainīgā datu tipu un ļauj atgriezt jebkura tipa mainīgo. Definējot funkciju, kuru izmanto trigerī, tai nevar būt argumenti. Lai saņemtu ieejas datus, tā izmanto mainīgo masīvu TG_ARGV[]. Kā indeksu lieto argumenta kārtas numuru sakot ar 0. Ja funkcijā nepieciešams darboties ar datiem, kuru izmaiņa izsaukusi trigeri, tad tiem var piekļūt izmantojot mainīgos NEW un OLD, kuri attiecīgi glabā datus pirms un pēc izmaiņām. Ja datu nav (OLD operācijas INSERT un NEW operācijas DELTE gadījumā, tad tiek atgriezta NULL vērtība.

PostgreSQL trigeru definēšanas sintakse ir šāda:

```
CREATE TRIGGER trigera_vards { BEFORE | AFTER } {INSERT|DELETE|UPDATE}  
ON tabulas_vards FOR EACH { ROW | STATEMENT }  
EXECUTE PROCEDURE funkcija (argumenti)
```

Piemēram:

```
CREATE TRIGGER "tr_1"  
BEFORE INSERT OR UPDATE OR DELETE ON "atzimes"  
FOR EACH ROW EXECUTE PROCEDURE f_tr1("Params-1");  
  
CREATE or replace FUNCTION "f_tr1"() RETURNS "opaque" AS '  
DECLARE  
v_laiks timestamp;  
v_liet varchar;  
BEGIN  
v_laiks := now();  
v_liet := current_user;  
RAISE NOTICE '=== Sakums ===';  
RAISE NOTICE 'Laiks - %', v_laiks;  
RAISE NOTICE 'Lietotajs - %', v_liet;  
RAISE NOTICE 'Operacija -%', TG_OP;  
RAISE NOTICE ' Vec_vert - %', old.atzime;  
RAISE NOTICE ' Jaun_vert - %', new.atzime;  
RAISE NOTICE 'Params - %', TG_ARGV[0];  
RAISE NOTICE '=== Beigas ===';  
RETURN NEW;  
END;  
' LANGUAGE 'plpgsql';
```

15.4. MYSQL

MySQL sākot ar 5.0 versiju uztur trigerus (4.x versijās trigeri nedarbojas). Tie darbojas ļoti līdzīgi kā trigeri PostgreSQL.

15.5. PAPILDUS LIERATŪRA

Database Management Systems – Triggers and Active Databases 164-168. lpp

MySQL dokumentācija sadaļa 18. Triggers

PostgreSQL dokumentācija sadaļa 33. Triggers

15.6. PAŠKONTROLES JAUTĀJUMI.

1. Kas ir trigeri.
2. Kam galvenokārt trigeri tiek lietoti.
3. No kādām daļām sastāv trigeri.
4. Ar kādām komandām veido triggerus.

16. Transakciju mehānismi.

Transakcija ir DB apstrādes loģiska vienība, kurā ietilpst viena vai vairākas operācijas ar DB. Šīs operācijas tiek uzvertas kā viens veselums un tās jāveic vai nu visas vai kļūdas gadījumā neviena.

Vienlietotāja (*single-user*) DB sistēma ir sistēma, kuru vienlaicīgi var izmantot tikai 1 lietotājs. Mūsdienās šādas sistēmas vairs neviens neveido. (Izņēmums varētu būt personālās DB)

Daudzlietotāju sistēmās daudzi lietotāji var izmantot sistēmu un pieslēgties DB vienlaicīgi.

Sevišķi svarīgs transakciju mehānisms ir tieši daudzlietotāju sistēmās, kur vienlaicīgi vienus un tos pašus datus izmanto un apstrādā daudzi lietotāji un DBVS jānodrošina tas, lai vairākiem lietotājiem darbojoties ar vieniem un tiem pašiem datiem nerastos problēmas.

16.1. NEKONTROLĒTA VIENLAICĪGA PIEEJA DATIEM UN TO IZMAINĪŠANA VAR RADĪT VAIRĀKAS PROBLĒMAS:

- Zaudēto izmaiņu problēma.

Piemērs:

- ⇒ Lietotājs_1 nolasa datus no DB un sāk tos rediģēt,
- ⇒ Lietotājs_2 arī nolasa tos pašus datus un arī sāk rediģēt,
- ⇒ Lietotājs_1 saglabā izdarītās izmaiņas,
- ⇒ Lietotājs_2 arī saglabā izdarītās izmaiņas un Lietotāja_1 izdarītās izmaiņas tiek pārrakstītas Lietotājam_2 to pat nezinot.

- Īslaicīga labojuma (*temporary update*) problēma

Ja viena transakcija mēģina labot DB, bet rodas kļūda un tajā brīdī pirms vēl izmaiņas ir atceltas, otrs lietotājs nolasa šos datus un nemaz nezina, ka dati tūlīt tiks atgriezti sākotnējā stāvoklī. Šādu problēmu sauc arī par netīras lasīšanas (*dirty read*) problēmu.

- Nekorekto statistisko aprēķinu problēmas

Veicot dažādus statistiskus aprēķinus (summēšanu, vidējo rēķināšanu utt), tai laikā kad notiek šis aprēķins, liet izdarītas izmaiņas DB, kā rezultātā iegūtais rezultāts vairs neatbilst patiesībai.

16.2. ATJAUNOŠANAS (*RECOVERY*) NEPIECIEŠAMĪBA

Ja transakcija tiek nosūtīta DBVS izpildei, tad sistēma rūpējas par to, lai vai nu visas transakcijā ietilpstošās operācijas tiktu veiksmīgi izpildītas un ierakstītas DB vai arī neviena no tām neietekmē DB vai citu transakciju. DBVS nedrīkst pieļaut, ka dažas transakcijas operācijas tiek izpildītas, bet dažas ne. Tas varētu notikt, ja transakcijas izpildēs laikā, kad dažas no tajā ietilpstošajām operācijām ir jau izpildītas, rodas kļūda un transakcija nevar tikt tālāk izpildīta.

16.3. IESPĒJAMĀS KĻŪDU RAŠANĀS IEMESLI IR:

- Tehniskā nodrošinājuma kļūda:

Rodas problēmas ar tehnisko nodrošinājumu (procesors, atmiņa utt). Ja būs nopietna problēma ar diskiem, tad transakcijas tur nespēs līdzēt. Bet piemēram gadījumā, ja dators tiek netīšām izslēgts, transakciju mehānismi pēc atkārotas servera palaišanas prot datu bāzi atjaunot sākotnējā stāvoklī.

- Transakcijas vai DB sistēmas kļūda:

Var rasties, ja tiek izmantoti nepareizi datu tipi vai nepieļaujamas vērtības, pārpildīšanās, dalīšana ar 0 vai citām loģiskām programmēšanas kļūdām.

- Vienlaicīgas pieejas izmantošana:

Vairākas transakcijas ir „deadlock” stāvoklī vai arī kāda transakcija tiek atcelta (*rollback*).

16.4. TRANSAKCIJU ĪPAŠĪBAS

- Atomārums (*atomicity*)

Transakcija ir atomāra apstrādes vienība, un vai nu tiek izpildītas visas operācijas, kas tajā ietilpst, jeb arī netiek izpildīta neviena.

- Nepretrunīguma saglabāšana (*Consistency preservation*)

Pēc transakcijas izpildes rezultātā DB no viena nepretrunīga stāvokļa pāriet uz citu nepretrunīgu stāvokli. Transakcijas izpildes gaitā stāvoklis DB var būt pretrunīgs. Piemēram DB tiek glabāta informācija par bankas konta sākuma atlikumu, operācijām ar to un beigu atlikumu. Veicot jaunu operāciju ir gan jāpievieno šī operācija gan arī jāizmaina konta beigu atlikums, tās ir 2 operācijas ar DB un nekādā gadījumā nedrīkst pieļaut gadījumu, ka tiek izpildīta un DB saglabāta tikai viena no tām, jo tad dati DB būs pretrunīgi, pieskaitot pie sākuma atlikuma visas dienas operācijas nesanāks norādītais beigu atlikums. Šādā gadījumā abas operācijas tiek apvienotas vienā transakcijā un DBVS rūpējas par to, lai nodrošinātu nepretrunīguma saglabāšanu.

- Izolācija (*isolation*)

Transakcijas izpildē un rezultātu korektumā nevar iejaukties cita transakcija, kas izpildās paralēli.

- Pastāvība (*permanency*)

Izmaiņas ko datu bāzē veikusi apstiprināta transakcija (transakcija, kura ir apstiprinātā stāvoklī), ir jābūt ierakstītām datu bāzē. Šīs izmaiņas nevar tika zaudēts kļūdas gadījumā (izņemot protams datu nesēju kļūdas)

16.5. TRANSAKCIJAS STĀVOKĻI

Lai varētu veikt atjaunošanu, sistēmai ir jāreģistrē kad transakcija sākas, beidzas, apstiprinās vai atceļas. Tās attiecīgi ir sekojošas operācijas:

- BEGIN_TRANSACTION

Iezīmē transakcijas sākumu, transakcija nonāk aktīvā stāvoklī (active state)

- READ vai WRITE

Lasīšanas vai rakstīšanas operācijas, kas tiek izpildītas kā daļa no transakcijas.

- END_TRANSACTION

Iezīmē transakcijas beigas, tā nonāk daļēji apstiprinātā stāvoklī

- COMMIT_TRANSACTION

Komanda, ka transakcija ir veiksmīgi pabeigta un izmaiņas var tikt apstiprinātas DB un tās nebūs atceļamas. Transakcija nonāk apstiprinātā stāvoklī.

- ROLLBACK

Komanda, ka transakcija ir beigusies neveiksmīgi un izdarītās izmaiņas DB ir jāatceļ. Transakcija nonāk atceltā stāvoklī.

Pabeigtā stāvoklī transakcija nonāk, kad attiecīgi pēc apstiprinātā vai atceltā stāvokļa no sistēmas tiek izdzēsti visi dati, kas saistīti ar šo transakciju.

16.6. SISTĒMAS ŽURNĀLS

Lai būtu iespējams atjaunoties pēc kļūdām, kas veiksmīgu datu izmaiņu veikšanu, tiek uzturēts sistēmas žurnāls (*system log*), kurā tiek ierakstītas visas operācijas, kas izmaina datus datu bāzē. Sistēmas žurnāls tiek glabāts uz diska un to nevar ietekmēt dažādas kļūdas. Žurnālā tiek glabāta informācija par transakciju sākumu un beigām un datu vērtībām, kuras ir tikušas mainītas saglabājot gan veco, gan jauno vērtību.

Kad transakcija nonāk daļēji apstiprinātā stāvoklī, visas izmaiņa, kas ir jāizdara datu bāzē, ir ierakstītas žurnālā (log failā). Kad tiek saņemta komanda COMMIT_TRANSACTION, šīs izmaiņas no log faila tiek ierakstītas datu bāzē. Kad tas ir izdarīts, log failā tiek ierakstīts apstiprinājums (COMMIT), ka transakcija ir veiksmīgi ierakstīta DB. Ja rodas kļūdas, tiek meklētas visas transakcijas, kuras nav apstiprinātā stāvoklī un notiek to atgriešanās sākotnējā stāvoklī (ROLLBACK).

16.7. TRANSAKCIJAS POSTGRESQL

PostgreSQL (un vairums citu DBVS) katra atsevišķa datu manipulācijas komanda darbojas kā transakcija. Tas nozīmē, ka sistēma pati automātiski nodrošina to, lai veiktajām izmaiņām būtu transakcijas raksturiezīmes – vai nu ir veiktas visas pieprasītās izmaiņas, vai arī kļūdas gadījumā izmaiņas nav izdarītas vispār. Piemēram gadījumā, ja ar izpildot komandu `INSERT INTO tabula1 SELECT * FROM tabula_2` kādu no SELECT atgrieztajām vērtībām tabulā nevar ierakstīt (specifiski datu integritātes nosacījumi), tad netiks pievienots neviens no ierakstiem.

Līdzīgi darbojas arī saglabātās procedūras. Visas izmaiņas, kas tiek veiktas procedūras ietvaros tiek uztvertas kā viena transakcija. Gadījumā, ja kādu no tām nav iespējams veikt, tad izmaiņas netiks veiktas vispār.

⇒ Transakcijas sākums

Transakcijas sākumu norāda komanda "BEGIN [WORK | TRANSACTION]". To nevajadzētu jaukt ar "BEGIN " ko lieto saglabāto procedūru struktūras definēšanai. Tāpēc, lai arī WORK un TRANSACTION nav obligāti jālieto, lai norādītu transakcijas sākumu, tomēr ieteicams tos lietot, lai viennozīmīgi būtu skaidra "BEGIN " nozīme. PostgreSQL neuztur iespēju iekļaut transakciju kādas citas transakcijas sastāvā, tāpēc gadījumā, ja pirms transakcijas beigām tiek atkārtoti lietota komanda "BEGIN TRANSACTION ", tiek izvadīts brīdinājuma paziņojums.

⇒ Transakcijas apstiprināšana

Komandas "COMMIT [WORK | TRANSACTION]" vai "END [WORK | TRANSACTION]" var tik lietotas lai apstiprinātu visas veiktās izmaiņas un ierakstītu tās datu bāzē.

⇒ Transakcijas atcelšana

Lai atceltu izmaiņas, kas veiktas kopš transakcijas sākuma, var tikt lietotas komandas ROLLBACK [WORK | TRANSACTION] " un " ABORT [WORK | TRANSACTION] ". Gadījumā, ka transakcijas izpildes gaitā ir radusies kāda kļūda un visas operācijas nav veiktas, DBVS automātiski veic transakcija atcelšanu.

Piemēram :

```
BEGIN TRANSACTION;  
    UPDATE accounts SET bal = my_bal - debit WHERE acctno = 7715;  
    UPDATE accounts SET bal = my_bal + credit WHERE acctno = 7720;  
COMMIT TRANSACTION;
```

16.8. TRANSAKCIJAS MYSQL

MySQL transakcijas uztur tikai tabulu tipiem InnoDB un BDB. MySQL visbiežāk izmantotais tabulu tips MyISAM transakcijas neuztur. Tabulu tipiem, kur transakcijas darbojas tās līdzīgi kā PostgreSQL var izmantot ar komandām BEGIN/COMMIT/ROLLBACK. Jāpiezīmē, ka komandas END un ABORT , kuras PostgreSQL dublēja iepriekšminētās komandas, MySQL nedarbojas.

16.9. PAPILDUS LIERATŪRA

Database Management Systems – Transaction management 523-539. lpp

PostgreSQL dokumentācija sadaļa 3.4. Transactions

16.10. PAŠKONTROLES JAUTĀJUMI.

1. Kas ir transakcijas.
2. Kāpēc nepieciešams lietot transakcijas.
3. Kādi ir transakcijas stāvokļi.
4. Ar kādām komandām vada transakciju darbību.

17. Laiksakritības kontrole metodes

Šajā nodaļā sīkāk apskatīsim laiksakritības kontroles (*concurrency control*) metožu izmantošanu. Tās ir nepieciešamas, lai nodrošinātu transakciju, kuras tiek izpildītas vienlaicīgi, savstarpēju izolāciju.

17.1. BLOKĒŠANAS LĪMENI

- Datu bāze līmenis

Tiek nobloķēta visa DB. Tā nav pieejama citiem lietotājiem. Šis līmenis tiek lietots dažādiem speciāliem uzdevumiem, kā arī, piemēram, rezerves kopiju veidošanai.

- Tabulas līmenis

Šis līmenis tiek pielietots operācijās, kur tiek mainīta tabulas struktūra.

- Bloka vai lapas līmenis

Tiek bloķēts viens bloks.

- Ieraksta līmenis

Tiek bloķēts tikai viens ieraksts.

- Lauka līmenis

Tikai viens konkrēts lauks ierakstā tiek nobloķēts. Šādu metodi var lietot, kad regulāri tiek labots tikai viens lauks. Piemēram: preces daudzums noliktavā. Tomēr šāda realizācija ir diezgan sarežģīta un DBVS tiek lietota samērā reti.

17.2. BLOKĒŠANAS (*LOCKING*) METODES

Dažādas bloķēšanas metodes ir galvenais veids, kas tiek izmantota komerciālās DBVS, lai nodrošinātu transakciju izolētību. Bloķēšanas mainīgais (*lock*) ir mainīgais, kas saistīts ar datu vienību (*data item*) un apraksta šīs datu vienības stāvokli attiecībā pret dažādām operācijām. Šie bloķēšanas mainīgie mēdz būt vairāku tipu un atkarībā no tā tiek iedalīti arī bloķēšanas mehānismi:

- Binārais bloķēšanas mehānisms.

Tam var būt tikai 2 stāvokļi: bloķēts un nebloķēts (*locked* un *unlocked*). Ja datu vienība ir bloķētā stāvoklī, tad citas DB operācijas, kā vien tās, ko veic bloķējošā transakcija, ar šo datu vienību nevar tikt veiktas. Pirms transakcija sāk darbu ar datu vienību, tā pārbauda šīs datu vienības stāvokli un ja tas nav bloķēta tad nobloķē to, veic vajadzīgās darbības un tad atkal atbloķē. Ja datu vienība ir bloķēta transakcija gaida līdz tā tiks atbloķēta. Lai gaidīšana noritētu veiksmīgi, tiek veidota rinda, kurā transakcijas "gaida". Tādā veidā tiem panākts, ka vienlaicīgi ar datu vienību darbojas tika 1 transakcija un nekādas neatbilstības nevar rasties.

- Lasīšanas/rakstīšanas bloķēšanas mehānisms.

Binārais mehānisms ir pārāk ierobežojošs, jo tikai viena transakcija vienlaicīgi var izmantot datu vienību. Datu vienībai ir iespējami 3 stāvokļi – bloķēts rakstīšanai, bloķēts lasīšanai un nebloķēts. Ja datu vienība ir bloķēta lasīšanai, tad arī citas transakcijas var šo datu vienību izmantot lasīšanai, rakstīšanai to izmantot nevar. Bet bloķētas rakstīšanai gadījumā citas transakcijas to nevar izmantot ne lasīšanai ne rakstīšanai. Tomēr šāda bloķēšana negarantē vienmēr pareizus rezultātus, jo nav garantēta transakciju nepretrunīgums. Lai risinātu šo problēmu tiek izmantota papildus protokoli. Pazīstamākais no tiek ir divu fāzu bloķēšanas mehānisms.

- Divu fāzu bloķēšanas mehānisms (2PL)

Divu fāzu bloķēšanas mehānisms (*Two-phase locking*) raksturo tas, ka visas bloķēšanas operācijas (read lock, write lock) atrodas pirms pirmās atbloķēšanas operācijas. Līdz ar to transakcija sadalās 2 fāzēs:

- ⇒ augošajā fāzē, kurā transakcija var iegūt operācijām jaunas datu vienības, bet nevar tās atbrīvot,
- ⇒ dilstošajā fāzē, kuras laikā datu vienības var tikt atbrīvotas (*unlocked*), bet nevar tikt pievienotas.
- ⇒ Ja izmanto šādu 2 fāzu bloķēšanas mehānismu var apgalvot, ka sistēmas žurnāls (*schedule*) ir virknējams un līdz ar to nekādas problēmas nevar rasties.

Divu fāzu bloķēšanas mehānismu var iedalīt sīkāk:

- ⇒ Pamata (*basic*) – tika apskatīts vispārējā gadījumā
- ⇒ Konservatīvos (*conservative*) – visas datu vienības tiek nobloķētas pirms transakcijas sākuma. Ja kāda nevar tikt nobloķēta, transakcija nesākas un gaida kamēr visas tai nepieciešamās datu vienības tiks atbrīvotas. Šāds mehānisms ir brīvs no dedlokiem.
- ⇒ Stingros (*strict*) – šis ir vispopulārākais praksē izmantojamais. Transakcijai bloķē rakstīšanai datu vienības līdz tam laikam, kad tā tiek apstiprināta (*commit*) vai atcelta (*rollback*)
- ⇒ Ļoti stingros (*rigorous*) - Transakcijai bloķē rakstīšanai un lasīšanai datu vienības līdz tam laikam, kad tā tiek apstiprināta (*commit*) vai atcelta (*rollback*).

- Dedloki (*deadlocks*)

Dedlocks ir stāvoklis, kas iestājas, kad pirmā transakcija ir bloķējusi datu elementu ko vajag otrajai un otrā – ko vajag pirmajai. Līdz ar to viena transakcija gaida uz otru un nekas nenotiek. Atkarībā no tā, kādas ir transakcijas un kāda ir sistēmas noslodze var izvēlēties dažādus dedloku novēršanas veidus. Ja transakcijas ir lielas, tajās iesaistīti daudzi datu elementi, tad jāizmanto kāda no dedloku novēršanas protokoliem. Gadījumā, ja varbūtība, ka radīsies dedloki ir maza, var izmantot dedloku noteikšanu vai taimoutus. Lai no šādām situācijām izvairītos tiek lietoti:

- ⇒ dedloku novēršanas protokoli (*deadlock prevention protocol*).

Šie protokoli ir dažādas matemātiski izstrādātas metodes, kā to teorētiski varētu skaisti risināt deadloku problēmas.

1. Divu fāzu konservatīvā bloķēšana

Šajā gadījumā transakcija jau pirms sākumā mēģina nobloķēt visus tai nepieciešamos datu objektus un ja kāds no tiem nav pieejams, tad netiek bloķēti neviens no tiem. Šāda pieeja nav pārāk praktiska, jo resursi tiek bloķēti uz ilgāku laiku nekā tas būtu optimāli.

2. Nepretrunīgas datu objektu pieejas kārtības izveide

Šī metode nodrošina to, ka visas transakcijas bloķēs datu objektus vienā kārtībā un līdz ar to deadloki nevar rasties.

3. Laikspiedola (*timestamp*) metodes

Par laikspiedolu tiek saukts transakcijas unikāls identifikators, ko izveidojot transakciju tai piešķir DBVS. To varētu uzskatīt par transakcijas sākuma laiku. Laikspiedolu var veidot dažādos veidos. Viena no iespējām ir izmantot skaitītāju, kura vērtību katru reizi kad tiek izveidota transakcija palielina par 1 un tad piešķir transakcijai. Šajā gadījumā tikai jāpārliedzinās, ka nenotiek skaitītāja pārpildīšanās, jo tā lielākā vērtība nav bezgalība. Cita iespēja laikspiedola izveidei ir izmantot tekošā sistēmas datuma/laika kombināciju, bet arī šajā gadījumā jāpārliedzinās, ka vienā laika vienībā netiek izveidotas 2 transakcijas.

Šī pasākuma doma ir kārtot transakcijas pēc to laikspiedoliem. Metodei ir jānodrošina, lai katra konfliktējoša pieejas operācija tiktu izpildīta pareizā virknētā kārtībā. Lai to panāktu, ar katru datu vienību tiek saistītas divas laikspiedola vērtības – lasīšanas un rakstīšanas laikspiedoli. Tie saglabā attiecīgi visjaunākās (ar vislielāko laikspiedolu) transakcijas laikspiedolu. Tādā gadījumā pirms transakcija veic attiecīgi rakstīšanas vai lasīšanas operācijas ar datu objektu, tā pārbauda vai šī objekta attiecīgais laikspiedols ir mazāks par dotās transakcijas laikspiedolu. Citiem vārdiem vai neviena jaunāka transakcija nav izmantojusi objektu jau pirms tam. Ja tas ir spēkā, tad transakcija veic attiecīgo darbību un izmaina laikspiedolu. Ja tomēr kāda jaunāka transakcija jau ir veikusi darbības ar šo objektu, tad vecākā transakcija, kura mēģināja operēt ar objektu tiek pārstartēta (roll back and strat no jauna) ar jaunu laikspiedolu. Problēmas rodas gadījumos, ja kāda cita transakcija jau ir izmantojusi šos rezultātus, kuri tagad tiek atcelti. Tad arī šī transakcija ir jāatceļ.

⇒ **Dedloku noteikšana**

Viens no vienkāršākajiem deadloku noteikšanas veidiem ir veidojot resursu gaidīšanas shēmu, kur ar bultām attēlo kura transakcija uz kuras transakcijas bloķētiem resursiem gaida. Ja šādā shēmā atrod cikliskas gaidīšanas saites (elot pa bultām var nonākt pie sākotnējās transakcijas) un tad jāizvēlas viena transakcija – upuris (*victim*), un tai jāveic "ROLLBACK" operācija.

⇒ Taimouti (*timeouts*)

Ļoti vienkārša un praktiska metode. Gadījumos, kad transakcija gaida kādu resursu vairāk kā noteiktu laika sprīdi, tiek veikta "ROLLBACK" operācija neatkarīgi no tā, vai deadlok stāvoklis patiešām bija vai ne.

- Badošanās (*starvation*)

Jāseko, lai gaidīšana rindās būtu godīga. Gadījumā, ja viena veida transakcijām ir prioritāte pār citām, var rasties situācija, ka otrā tipa transakcijas visu laiku tikai gaida un gaida un netiek izpildīta, jo resursus bloķē kāda transakcija ar augstāku prioritāti. Tādu situāciju sauc par badošanos. Tādos gadījumos gaidīšanu var veidot tā, ka laiks ko transakcija pavada gaidot palielina tās prioritāti. Līdz ar to transakcijai, kura ilgi gaida, būs visaugstākā prioritāte un tā varēs izpildīties.

17.3. POSETGRESQL

Pārsvarā DBVS bloķēšanu veic automātiski. Uzstādāmais bloķēšanas līmenis ir atkarīgs no darbības ko lietotājs vēlas veikt. Ja tā ir operācija, kuras bloķēšanas līmenis konfliktē ar jau uzstādīto bloķēšanas līmeni, tad lietotāja operācija gaida, kad doto resursu atbrīvos. Resursa atbrīvošana notiek pēc transakcijas, kura doto resursu ir nobloķējusi, izpildes beigām,

- Iespējamie bloķēšanas līmeņi un darbības, kas tos izsauc:

⇒ Pieejas dalīšana (ACCESS SHARE)

ACCESS SHARE ir līmenis, kas vismazāk ierobežo iespējamās darbības ar tabulu, kam uzstādīts dotais bloķēšanas līmenis. Šo bloķēšanas līmeni izmanto gadījumos, kad tiek veikta pieprasījuma (*query*) izpilde, kurā tiek izmantota šī tabula. Ja tabula ir bloķēta ar ACCESS SHARE, tad šai tabulai cits lietotājs nevar piekļūt izmantojot operācijas, kam nepieciešams ACCESS EXCLUSIVE bloķēšanas līmenis.

⇒ Ierakstu dalīšana (ROW SHARE)

Šis ir pieejas bloķēšanas līmenis, kurš tiek izmantots izpildot komandu "SELECT ..FOR UPDATE". Šis pieprasījums darbojas līdzīgi parastam pieprasījumam, bet opcija "FOR UPDATE" norāda, ka vēlāk plānots datiem, ko iegūst šajā pieprasījumā, veikt "UPDATE" operāciju. Ja tabula ir bloķēta ROW SHARE līmenī, tad citi lietotāji tai nevar piekļūt operācijām, kam nepieciešami bloķēšanas veidi EXCLUSIVE vai ACCESS EXCLUSIVE.

⇒ Ierakstu vienlietotāja pieeja (ROW EXCLUSIVE)

Šis bloķēšanas līmenis tiek izmantots veicot operācijas "INSERT, DELETE, UPDATE". Ja tabula ir bloķēta ROW EXCLUSIVE veidā, tad citi lietotāji tai nevar piekļūt operācijām, kam nepieciešami bloķēšanas līmeņi SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE un ACCESS EXCLUSIVE.

⇒ Dalīšana, vienlietotāja izmaiņas (SHARE UPDATE EXCLUSIVE)

Šo bloķēšanas līmeni izmanto komanda "VACUM". Tas nepieļauj ar doto tabulu veikt operācijas, kam nepieciešami SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE vai ACCESS EXCLUSIVE bloķēšanas līmeņi.

⇒ Dalīšana (SHARE)

To automātiski izsauc komanda "CREATE INDEX". Ja tabula ir bloķēta SHARE līmenī, tad citi lietotāji tai nevar piekļūt operācijām, kam nepieciešami bloķēšanas līmeņi ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE un ACCESS EXCLUSIVE.

⇒ Dalīšana, ierakstu vienlietotāja pieeja (SHARE ROW EXCLUSIVE)

Ja tabula ir bloķēta SHARE ROW EXCLUSIVE līmenī, tad citi lietotāji tai nevar piekļūt operācijām, kam nepieciešami bloķēšanas līmeņi ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE vai ACCESS EXCLUSIVE.

⇒ Vienlietotāja rakstīšana (EXCLUSIVE)

Tā tabulu bloķē tā, ka citi lietotāji var tikai izmantot tikai operācijas, kuras šo tabulu bloķē ACCESS SHARE veidā (dati no šīs tabulas tiek tikai lasīti).

⇒ Vienlietotāja pieeja (ACCESS EXCLUSIVE)

Tā nepieciešama veicot tabulas struktūras maiņu, tabulas dzēšanu vai datu bāzes pārkārtošanu izmantojot komandu "VACUM FULL". Citi lietotāji nevar veikt nekādas darbības ar šo tabulu.

Sekojošajā attēlā redzams, pēc kuru bloķēšanas līmeņu uzstādīšanas, kuri bloķēšanas līmeņi uz doto tabulu ir pieejami citiem lietotājiem vai citām tā paša lietotāja operācijām (zaļi) un kuri nav (sarkani).

Pieejamie bloķēšanas līmeņi pēc bloķēšanas

	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
Bloķēšanas veids								
ACCESS SHARE								
ROW SHARE								
ROW EXCLUSIVE								
SHARE UPDATE EXCLUSIVE								
SHARE								
SHARE ROW EXCLUSIVE								
EXCLUSIVE								
ACCESS EXCLUSIVE								

- Bloķēšanas mehānisma vadības komandas

Tomēr veidojot sistēmu iespējams transakcijas ietvaros vadīt šo bloķēšanas mehānismu. To veic komanda "LOCK". Atbloķēšana notiek beidzoties transakcijai, gan ja tā beidzas veiksmīgi, gan arī gadījumos, kad tiek veikts "ROLLBACK".

```
LOCK [ TABLE ] nosaukums [, ...] IN bloķēšanas_veids MODE
```

Piemēram :

```
LOCK TABLE masinas IN EXCLUSIVE MODE;
```

17.4. PAPILDUS LIERATŪRA

Database Management Systems – Concurrency Control 540-593. lpp

PostgreSQL dokumentācija sadaļa 12. Concurrency Control

17.5. PAŠKONTROLES JAUTĀJUMI.

1. Kas ir laiksakritības kontrole.
2. Kas ir bloķēšana un kam to lieto.
3. Kādi ir bloķēšanas līmeņi, ar ko tie atšķiras.
4. Kādas ir bloķēšanas metodes, ar ko tās atšķiras.
5. Kādos līmeņos PostgreSQL veic bloķēšanu.
6. Ar kādām komandām PostgreSQL veic bloķēšanas mehānismu vadību.

18. Datu bāzu drošība

Liekas, ka pēdējos gados ir tik daudz ticis runāts par datoru sistēmu drošību, ka neko jaunu vairs pateikt nav iespējams. Tomēr lielākoties tiek uzsvērti ārējie draudi – vīrusi, hakeri, servisu bloķēšana. Tomēr kā liecina PriceWaterhouseCoopers veiktais pētījums "Information Security Breaches 2002" 30-50% no draudiem, atkarībā no uzņēmuma lieluma, rada darbības ko veic kompānijas darbinieki. Tāpēc veidojot DB sistēmu jāatceras, ka nopietna uzmanība jāvelta drošības mehānismu izstrādei.

18.1. VISPĀRĒJS ATSKATS

Mandātu drošības sistēmas (kas tās ir tiks apskatīts mazliet vēlāk) sāka savu uzplaukumu, kad 90-to gadu sākumā ASV aizsardzības ministrija noteica, ka visām tās sistēmām ir jāizmanto mandātu tiesību sistēma. Centrālās izlūkošanas pārvaldē (National Security Agency) tika izstrādātas vadlīnijas sistēmu drošības klasificēšanai. Tagad tās pasaulē plašāk pazīstamas kā Varavīksnes sērijas (Rainbow Series) dokumenti (grāmatas). Grāmatām piešķirti dažādu krāsu nosaukumi, lai padarītu tos vieglāk atpazīstamus un lietojamus. Piemēram dokuments NCSC-TG-003 tiek saukts par oranžo grāmatu (*Orange Book*). Tieši oranžā grāmata liek vispārējus pamatus datorsistēmu drošības klasifikācijai un pārējās grāmatas apraksta kādu konkrētu jomu. Uz datu bāzēm tieši attiecas NCSC-TG-021 (*Lavender/Purple Book*). Sīkāka informācija:

<http://www.fas.org/irp/nsa/rainbow.htm>

Šīs grāmatas sistēmas pēc to drošības parametriem iedala 4 klasēs A,B,C un D. D klase ir visvājāk aizsargātā, A visvairāk aizsargātā.

- D klase - nekādas aizsardzības
- C klase – sadalās 2 apakšklasēs
 - ⇒ C1 klase – ir lietotāju autorizācija, un iespēja lietotājam aizsargāt savus datus no citiem
 - ⇒ C2 klase – C1+ tiek veikta lietotāju darbību sistēmā reģistrācija
- B klase sadalās 3 apakšklasēs
 - ⇒ B1 klase – katram sistēmas objektam tiek piešķirts drošības pakāpe un pieejas tiesības lietotājiem attiecībā ar to
 - ⇒ B2 klase – papildus formāls drošības prasību apraksts. Jāatrod un jānovērš iespējamie informācijas noplūdes kanāli, lai netieši nevarētu iegūt informāciju uz kuru nav tiesības.
 - ⇒ B3 klase – papildus vēl jānodrošina sekošana un datu atjaunošana vajadzības gadījumā, kā arī nozīmēt sistēmas drošības administratoru
- A klase

Matemātiski ir jāpierāda, izvēlētais drošības mehānisms nodrošina tam uzliktās funkcijas

Sistēmu drošības klasificēšanai attīstoties tālāk sadarbojoties dažādu valstu par datorsistēmu drošību atbildīgajām organizācijām (CSE (Kanāda), SCSSI (Francija), BSI (Vācija), NLNCSA (Nīderlande), CESH (Lielbritānija), NIST (ASV), NSA (ASV)) tika izstrādāti jauni dokumenti, kas šobrīd plašāk pazīstami ar kopēju nosaukumu "Vispārīgie kritēriji" (*Common Criteria*). Nosaukums *Common Criteria* (bieži tiek lietot abreviatūra CC) ir saīsinājums no *Common Criteria for Information Technology Security Evaluation*.

"Vispārējie kritēriji" domāti gan lietotājiem, kas plāno iegādāties kādu sistēmu, kas novērtēta izmantojot šo metodoloģiju, gan sistēmas izstrādātājiem, kas vēlas savu sistēmu sertificēt, vai vismaz to izstrādājot vadīties pēc vispārpieņemtiem standartiem.

Sistēmas tiek novērtētas atbilstībai kādam no 7 līmeņiem - EAL1 līdz EAL7 (Evaluation Assurance Levels). Kā zināms datorprogrammas mēdz būt ļoti dažādas un to lietojuma mērķi ir ļoti dažādi. Lai varētu konkrētāk izstrādāt kritērijus dažādām produktu kategorijām, tiek lietots "Aizsardzības profila" (Protection Profile) jēdziens. Aizsardzības profilā tiek apvienoti drošības vērtēšanas kritēriji, kas ir būtiski konkrētai programmaproduktu kategorijai (operētājsistēmas, ugunsdzēsības, DBVS utt).

"Vispārīgie kritēriji" versija 2.1 ir apstiprināta kā starptautisks standarts ISO 15408, kas reglamentē attiecīgo jomu.

Sīkāku informāciju var meklēt internetā:

<http://niap.bahialab.com/cc-scheme/>

18.2. DROŠĪBAS MĒRĶI VEIDOJOT DB SISTĒMU:

Sistēmu drošība ir pietiekami plašs jēdziens un DB sistēmu drošība ir tikai viens posms no nodrošinot vispārēju sistēmas drošību. Vajag atcerēties, ka sistēma ir tik droša, cik droša ir tās vājākā vieta. Tāpēc visai mazus rezultātus dod drošības nodrošināšana tikai vienā vai dažās jomās. Sīkāk apskatīsim DB sistēmu drošību pieņemot, ka arī pārējām jomām tiek veltīta pienācīga uzmanība.

DB sistēmu drošībai ir 4 pamatmērķi:

- Novērst neautorizētu personu pieeju pašai sistēmai (*access control*)

Vienkāršākais kontroles līmenis, kad lietotājam ir jābūt sistēmā pierēģistrētam, lai viņš varētu ar to darboties. Procesu kurā tiek noskaidrota lietotāja identitāte sauc par autentifikāciju (authentication). Parasti autentifikācija notiek ievadot lietotāja vārdu un paroli, tomēr pastāv iespējas veikt autentifikāciju arī izmantojot dažādus citus mehānismus, piemēram speciālas atslēgas kartes lietošana vai cilvēka biometrijas parametru noteikšana (pirkstu nospiedumi, acs tīklene, sejas atpazīšana).

- Nodrošināt tiesību kontroles mehānismus:

Šis līmenis papildina iepriekšējo. Katram lietotājam vai lietotāju grupai tiek piešķirtas tiesības veikt darbības ar datiem un viņš var rīkoties tikai šo tiesību ietvaros, tai pašā laikā pieeja citām šīs DB daļām ir liegta.

- Novērst iespēju iegūt datus no DB apejot DBVS

Sevišķi slepenu datu aizsardzībai var tik lietoti dažādi šifrēšanas veidi, lai novērstu datu lasīšanu no DB faila binārā veidā.

- Nodrošināt attiecīgo likumdošanas aktu ievērošanu.

Jānoskaidro vai uz attiecīgo sistēmu attiecas vai neattiecas kādi likumdošanas ierobežojumi un attiecīgi jānodrošina sistēmas atbilstību tiem. Latvijā ar 2000.gada 6 aprīli spēkā stājās "Fizisko personu datu aizsardzības likums", kas nosaka kādām jābūt sistēmām, kurās tiek ievadīti dati par fiziskām personām.

Ar likumu var iepazīties <http://www.likumi.lv/doc.php?id=4042>

Par mērķi var tikt izvirzīts kā viens tā arī vairāki vai visi no šiem pamatmērķiem. Tas atkarīgs no konkrētās sistēmas.

18.3. DROŠĪBAS PRASĪBU NOSKAIDROŠANA:

Sākas viss pavisam netehniski ar atbildēšanu uz jautājumiem:

- Kādiem lietotājiem jānodrošina pieejas tiesībām DB
- Kādai DB daļai katram lietotājam ir tiesības piekļūt
- Kādas operācijas lietotājam ir tiesības veikt un ar kādiem datiem

Kad šie jautājumi ir noskaidroti, var ķerties pie tehniskās realizācijas. Parasti persona, kura atbild par tiesību piešķiršanu lietotājiem atbilstoši organizācijas politikai, ir DB administrators. Dažreiz (lielās kompānijās) ir atsevišķs darbinieks - drošības administrators, kurš nodarbojas ar kompānijas datorsistēmu drošības administrēšanu.

18.4. DB SISTĒMAS PIEEJAS TIESĪBU ADMINISTRĒŠANU VAR IEDALĪT VAIRĀKĀS DAĻĀS:

- Kontu izveidošana (*account creation*)

Izveido jaunu lietotāja kontu un paroli, kas ļauj lietotājam vai lietotāju grupai pieslēgties DB

- Tiesību piešķiršana (*privilege granting*)

Piešķir noteiktiem lietotāja kontam noteiktas tiesības

- Tiesību atcelšana (*privilege revocation*)

Atceļ lietotāja kontam tiesības, kas tam iepriekš bija piešķirtas

- Drošības līmeņa piešķiršana (*security level assignment*)

Piesaista lietotāja kontu pie noteikta drošības klasifikācijas līmeņa, ja sistēma tos uztur.

18.5. DBVS DROŠĪBAS MEHĀNISMUS IEDALA 2 DAĻĀS:

- Atsevišķie (*discretionary*) drošības mehānismi

Tiesību piešķiršana kādam konkrētam lietotājam vai lietotāju grupai. Šādas tiesības uztur lielākā daļa DBVS, tā ir ļoti elastīga, bet lielās sistēmās var kļūt grūti vadāma. Šāda sistēma lielākoties visām DBVS ir iebūvēta, sistēmu izstrādātājiem tikai jāizmanto tās piedāvātās iespējas.

- Mandātu (*mandatory*) drošības mehānismi

Lietotāju un datu sadalīšana pa līmeņiem (drošības klasēm) un tiesību piešķiršana bāzējoties uz šīm klasēm. Lietotājs var piekļūt tikai datiem, kuru drošības līmenis ir vienāds vai zemāks par viņam piešķirto drošības līmeni. Šis ir papildus koncepts, ko lieto tikai lielās sistēmās ar sarežģītu drošības struktūru. DBVS parasti nav izveidota šādi drošības mehānismi un sistēmas veidotājiem tie jārada pašiem.

18.6. ATSEVIŠĶĀ PIEEJAS KONTROLE AR TIESĪBU PIEŠĶIRŠANU / ATCELŠANU.

- Tiesību tipi:

⇒ lietotāja konta līmeņa (jeb sistēmas) tiesības

Lietotāja kontam tiek piešķirtas tiesības veikt kādu operāciju nenorādot konkrētu objektu ar ko to var veikt. Šādas tiesības lielākoties piešķir tikai ŠĪ tipa tiesības ir:

1. CREATE TABLE
2. CREATE SHEMA
3. CREATE VIEW
4. ALTER
5. DROP
6. MODIFY

⇒ shēmas objektu līmeņa tiesības

Šajā līmenī tiek noteikts katrai tabulai atsevišķi, kādas tiesības ir katram lietotājam. ŠĪ tipa tiesības ir:

1. SELECT – datu lasīšana
2. INSERT, DELETE, UPDATE – datu izmaiņas (var nedefinēt, ka ir tiesības mainīt tikai noteiktus laukus tabulā)
3. REFERENCE – dod tiesības atsaukties uz tabulu definējot integritātes nosacījumus

ŠĪ tipa tiesības var piešķirt tabulas īpašnieks. Par tabulas īpašnieku kļūst tās veidotājs, kuram izveidojot tabulu sākumā ir visas tiesības uz šo tabulu. Tomēr īpašnieku var arī nomainīt.

- Tiesību noteikšana izmantojot skatus (*views*)

Ja ir vajadzīgs piešķirt lietotājam tiesības lasīt tikai dažus laukus, tad tam ļoti ērti var izmantot skatus. Skatā iekļauj tikai vajadzīgos laukus un piešķir lietotājam tiesības to lietot, bet pamattabulā šādas tiesības nedod.

- Tiesību vadība izmantojot saglabātās procedūras (*Stored Procedures*)

Šīs procedūras var tik izmantotas, lai ļautu lietotājam veikt darbības, kas viņam nav atļautas veikt tieši DB. Piem. lietotājam var būt tiesības izpildīt kādu saglabāto procedūru, kura veic datu izmaiņas tabulā, kurā šim lietotājam nav tiesību veikt izmaiņas. Tas notiek tāpēc, ka procedūra tiek izpildīta ar tās īpašnieka (owner) tiesībām un nevis tā, kurš to palaiž. Savukārt procedūrā jau krietni detalizētāk ir iespējams iestrādāt kontroles mehānismus, kas neļautu veikt neparedzētas datu izmaiņas.

- Ierakstu līmeņa tiesības (*row level security*)

Bieži vien ir nepieciešams noteikt, ka lietotājam ir tiesības veikt manipulācijas tikai ar noteiktiem ierakstiem tabulā, bet liegta pieeja citiem. To ir iespējams realizēt ar

- ⇒ skatiem, izmantojot WHERE
- ⇒ Virtual Private Database (Oracle)
- ⇒ Label based access control (Oracle)

- Tiesības piešķirt tiesības GRANT OPTION

Jebkuras tiesības var tika piešķirtas ar vai bez tiesībām šīs tiesības piešķirt tālāk. To norāda ar GRANT OPTION. Ja tas ir norādīts, lietotājs savas tiesības var piešķirt arī citiem lietotājiem. Šādā veidā tiesības var tikt pavairotas bez īpašnieka ziņas.

- Grupas (*groups*) un lomas (*roles*)

Īpaši vajag atzīmēt iespēju veidot lietotāju grupas un tiesības piešķirt jau grupai. Grupa var ietvert sevī arī citas grupas. Tas atvieglo lielu sistēmu pieejas tiesību vadību un ļauj veidot sarežģītas grupu tiesību hierarhiskas struktūras.

Lomas ne ar ko būtisku neatšķiras no grupām. Lomai parasti piešķir visas nepieciešamās tiesības, lai veiktu kādu konkrētu uzdevumu. Un tad jau lietotājam piešķir (vai nepiešķir) visas tiesības, kas saistītas ar šā uzdevuma izpildi. Tādā veidā tiek panākta vieglāka un pārskatāmāka pieejas tiesību administrēšana.

Izmantošanas priekšrocības:

- ⇒ Ja nepieciešams jauns lietotājs, kas var veikt uzdevumu, kuram jau ir nodefinēta loma, tad atliek tikai viņam piešķirt doto lomu un viņam būs visas nepieciešamās tiesības.
- ⇒ Ja darba gaitā tiek mainīta DB vai darba uzdevumi, loma ir jāmaina tikai vienreiz, un visi lietotāji, kam būs dotā loma, iegūs nepieciešamās tiesības. Ja lomu nebūtu, tad katram lietotājam tiesības nāktos piešķirt atsevišķi.
- ⇒ Lietotājam viegli var piešķirt lomu tikai uz brīdi kamēr viņš veic kādu uzdevumu un pēc tam to atkal atcelt.

18.7. POSTGRESQL

- Lietotāju autentifikācija

Lietotāja autentifikācija ir process, kurā tiek noskaidrota lietotāja identitāte. Tālāk šim lietotājam var tik dotas dažādas tiesības rīkoties ar datu bāzes objektiem. Autentifikāciju un sekojošo tiesību piešķiršanu sauc arī par autorizāciju.

PostgreSQL lietotāju autentifikācija notiek atdalīti no operāciju sistēmas lietotāju autentifikācijas. Protams ir iespējams izmantot tos pašus lietotāju vārdus un paroles, taču tie tiek glabāti pilnīgi neatkarīgi no OS lietotāju datiem. PostgreSQL ļauj veidot arī lietotājus, kuri nav reģistrēti OS.

Lai vadītu lietotāju autentifikāciju, PostgreSQL izmanto konfigurācijas failu „*pg_hba.conf*”. Šis fails glabājas DB klāsterā direktoriņā (šo ceļu norāda veidojot BD klāsteri ar komandu „*initdb -D cels*”, kā arī startējot DB serveri.

Šis fails sastāv no rindiņām ar sekojošu struktūru:

```
host database IP-address IP-mask authentication-method
```

kur,

database – DB nosaukums, var lietot arī *all*, tad šie uzstādījumi attieksies uz visām DB.

IP-address IP-mask – Norāda pieslēdzoties no kurām adresēm tiks lietota šī autentifikācijas metode.

authentication-method – Pielietojamā autentifikācijas metode.

Autentifikācijas metodes:

trust – bez paroles iespējams pieslēgties ar jebkuru lietotāja vārdu. Šādu uzstādījumu ieteicams lietot tikai vienlietotāja sistēmās, kur neautorizētas personas nevar piekļūt.

password, md5, crypt – lietotāja autentifikācija tiek veikta ar lietotāja vārda un paroles palīdzību. "Password" neizmanto paroles kodēšanu, tāpēc, ja pieslēgšanos veic caur tīklu, to labāk nelietot. Vislabāk izmantot "md5", jo tas nodrošina paroles kodēšanu.

Kerberos – lietotāja autentifikācija tiek veikta speciālas programmatūras palīdzību, kuru lieto arī citu programmu lietotāju autentifikācijai. Sīkāku informāciju varat atrast <http://www.pdc.kth.se/kth-krb/>

Piemēram:

```
host all 127.0.0.1 255.255.255.0 md5
```

Šāda rindiņa nosaka to, ka lietotājiem, kas pieslēdzas no lokālā datora, tiks pielietota paroles autentifikācija izmantojot paroles kodēšanu ar „md5”

- Lietotāja tiesību vadība

Pēc tam kad veikta lietotāja autentifikācija, nepieciešams mehānisms, kas konkrētajam lietotājam atļauj vai neatļauj veikt dažādas darbības ar dažādām DB sastāvdaļām. Lai standartizētu šo jomu, tā iekļauta SQL valodas sastāvā un līdz ar to visas DBVS kuras uztur SQL, uztur arī lietotāja tiesību vadības komandas.

⇒ Lietotāju pievienošana, labošana un dzēšana

Lai varētu veikt lietotāja autentifikāciju un tiesību vadību, vispirms lietotājs ir jāizveido. To dara ar komandas "CREATE USER" palīdzību.

Sintakse:

```
CREATE USER lietotaja_vards [ [ WITH ] SYSID uid  
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'parole'  
| CREATEDB | NOCREATEDB  
| CREATEUSER | NOCREATEUSER  
| IN GROUP grupa [, ...]  
| VALID UNTIL 'abstime' ]
```

Ja lietotāju izveido nenorādot paroli, tad parole būs NULL, bet jebkura ievadītā vērtība nebūs NULL un līdz ar to ar šo lietotāju pieslēgties sistēmai nebūs iespējams. Lai mainītu paroli, var izmantot komandu "ALTER USER"

Lietotāja parametrus var mainīt izmantojot komandu "ALTER USER *lietotaja_vards*"

Lietotāju var izdzēst izmantojot komandu "DROP USER *lietotaja_vards*".

Piemēram:

```
CREATE USER ivars;  
CREATE USER juris PASSWORD 'parole';  
CREATE USER janis PASSWORD 'parole' IN GROUP administrators;
```

⇒ Grupu veidošana un dzēšana, lietotāju pievienošana un dzēšana no grupām,

Grupas var veidot ar komandu

```
CREATE GROUP gr_nosaukums [[ WITH ] SYSID gid | USER lietotaja_vards  
[, ...] ]
```

Lai labotu un dzēstu grupas attiecīgi tiek lietotas komandas komandu "ALTER USER *gr_nosaukums*" un "DROP GROUP *gr_nosaukums*".

Lai grupai pievienotu vai dzēstu no tās lietotājus, lieto komandas :

```
ALTER GROUP gr_nosaukums ADD USER lietotaja_vards [, ... ]  
ALTER GROUP gr_nosaukums DROP USER lietotaja_vards [, ... ]
```

Piemēram:

```
CREATE GROUP db_admin;  
CREATE GROUP db_admin WITH USER ivars,juris;  
ALTER GROUP db_admin ADD USER ivars,juris;
```

⇒ Tiesību piešķiršana un dzēšana darbībai ar dažādiem DB objektiem

Tiesību piešķiršanu un dzēšanu veic attiecīgi ar komandām "GRANT" un "REVOKE".

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES  
| TRIGGER } [, ...] | ALL }  
ON [ TABLE ] objectname [, ...]  
TO { username | GROUP groupname | PUBLIC } [, ...]
```

```
REVOKE { { SELECT | INSERT | UPDATE | DELETE | RULE |  
REFERENCES | TRIGGER } [, ...] | ALL }  
ON [ TABLE ] object [, ...]  
FROM { username | GROUP groupname | PUBLIC } [, ...]
```

Tiesību veidi :

SELECT – lasīšanas tiesības,

INSERT - jaunu ierakstu pievienošanas tiesības,

UPDATE – esošos ierakstu rediģēšanas tiesības,

DELETE – ierakstu dzēšanas tiesības,

RULE – noteikumu veidošanas tiesības izmantojot komandu „CREATE RULE”

REFERENCES – tiesības veidot citās tabulās atsauci uz doto tabulu izmantojot „FOREIGN KEY
ar_atslega REFERENCES „,

TRIGGER – trigeru veidošanas tiesības.

Piemēram:

```
GRANT SELECT ON TABLE masinas TO ivars;  
GRANT SELECT, INSERT ON TABLE masinas TO ivars, juris;  
GRANT SELECT ON TABLE masinas TO GROUP db_admin;  
GRANT ALL ON TABLE masinas TO GROUP db_admin;
```

18.8. MYSQL

Lietotāju autentifikācija notiek ar lietotāja vārda, lietotāja datora vārda vai IP adreses un paroles palīdzību. MySQL atšķirībā no PostgreSQL visu informāciju par pieejas tiesībām glabā sistēmas datu bāzē mysql.

Tam tiek izmantotas 5 tabulas.

user – glabā informāciju par lietotāju (vārds,parole, datums no kura pieslēdzas) un tā kopējām tiesībām. Šīs tiesības attiecas uz visām datu bāzēm, tāpēc ar šīs tabulas palīdzību labāk lietotājam nekādas tiesības nepiešķirt, bet izdarīt to caur citām tabulām, kuras ļauj detalizētāk vadīt šīs tiesības.

db – nosaka kuriem lietotājiem kādas tiesības katrā datu bāzē.

hosts - papildina tabulu db, ja nepieciešams norādīt vairāk kā vienu datoru, no kā pieslēdzas lietotājs.

tables_priv – ļauj noteikt tiesības tabulām

columns_priv - ļauj noteikt tiesības kolonnām

Līdzīgi ka PostgreSQL tiesības var piešķirt un dzēst ar komandu "GRANT" un "REVOKE" palīdzību. Ar šīm pašām komandām tiek veidoti jauni lietotāji. Tas izveidojas pirmo reizi piešķirot lietotājam kaut kādas tiesības. Jāatzīmē, ka MySQL ļauj vadīt ne tikai visas tabulas, bet arī atsevišķu kolonnu tiesības. Kā trūkumu var minēt to, ka MySQL neuztur lietotāju grupu veidošanu, līdz ar to ir apgrūtināta pieejas tiesību vadīšana izmantojot DBVS līdzekļus.

Sintakse:

```
GRANT tiesibas [(kolonnas)] [,...]
ON {tabula | * | *.* | db_nosaukums.*}
TO lietotajs [IDENTIFIED BY 'parole']
```

Tiesības var būt:

FILE, RELOAD, ALTER, INDEX, SELECT, CREATE, INSERT, SHUTDOWN,
DELETE, PROCESS, UPDATE, DROP, USAGE

Kolonnas: kolonnu uzskaitījums, atdalīts ar komatiem.

Tabula – tabula tekošajā datu bāzē, uz kuru attieksies izdarītās izmaiņas. Norādot „*” šīs tiesības tiks attiecinātas uz visām tabulām tekošajā datu bāzē.

Lietotājs- norāda kuram lietotājam būs attiecīgās tiesības. Lietotājs var tik identificēts tikai ar lietotāja vārda vai arī ar lietotāja vārda un datora, no kura viņš pieslēdzas, palīdzību. Lai atdalītu lietotāja un lietotāja datora nosaukumu (vai IP adresi) tiek lietots simbols „@”

Piemēram:

```
GRANT SELECT ON masinas TO ivars;
GRANT ALL ON masinas TO ivars@192.168.1.1;
GRANT SELECT, INSERT, UPDATE, DELETE ON auto.* TO juris;
```

Lai izmainītās tiesības stātos spēkā, ir jāizpilda komanda „FLUSH PRIVILEGES”

Ja rediģē tiesības izmantojot kādu grafisko interfeisu, tad „%” simbols apzīmē visas adreses/datu bāzes/lietotājus, atkarībā no konteksta kādā tas lietots.

Piemēram:

root@% - root lietotājs, kas var pieslēgties no jebkuras mašīnas.

%@localhost – jebkurš lietotājs, kas pieslēdzas no localhost mašīnas.

18.9. DATU ŠIFRĒŠANA

Reāli dzīvē datu bāzēm tiek izmantota ļoti reti. Tomēr, ja dati ir ļoti svarīgi un slepeni, papildus drošības iegūšanai var veikt šifrēšanu. Iespējams šifrēt kā visu DB failu, tā arī atsevišķus ierakstus. Tas piemēram nepieciešams, ja datus vēlas pasargāt no DB administratora. Cik man zināms neviena DBVS neuztur iebūvētas lauku šifrēšanas funkcijas, bet tās ir iespējams ieprogrammēt lietotāja aplikācijā izmantojot kādu no populāriem šifrēšanas algoritmiem.

18.10. AUDITS

Ir ļoti svarīgi veikt lietotāju darbību monitorēšanu un auditu. Parast ir nepieciešams auditēt ne tikai lietotājus, bet arī DB veikspēju, kļūdas un daudzas citas lietas, bet šajā nodarbībā apskatīsim tikai to, kas saistīts ar lietotāju darbību auditu.

Audits nepieciešams, lai pārliecinātos, ka lietotājs veic tikai tam atļautās darbības. Bieži vien jau paša audita esamība attur lietotājus no vēlmes veikt dažādas nesankcionētas darbības.

18.11. PAPILDUS LIERATŪRA

1. Database Management Systems – 17. Security 497-520.lpp
2. PostgreSQL dokumentācija sadaļa 17. Database Users and Privileges

18.12. PAŠKONTROLES JAUTĀJUMI.

1. Tiesības kādu darbību veikšanai var lietotājam piešķirt vai nepiešķirt.
2. Ar kādām SQL komandām veic lietotāja tiesību vadību.

19. Datu glabātavas un OLAP.

Parastas datu bāzes tiek veidotas ar primāro mērķi veikt datu apstrādi (ievadi, labošanu dzēšanu) kalpojot kā biznesa sistēmu uzskaites rīki. Šādas sistēmas tiek optimizētas darbam ar atsevišķiem ierakstiem. (On-Line Transaction Processing). Tiek piedāvātas arī datu pieprasīšanas iespējas, bet gadījumos, ja šie pieprasījumi ir sarežģīti un datu bāze ir pietiekami liela, tad pieprasījumu izpilde ir nepieņemami lēna. Pie tam lai veidotu pieprasījumus, ir jāpārzina datu bāzes struktūra visos sīkumos kā rezultātā nestandarta pieprasījumu veidošana ir sarežģīts un laikietilpīgs process.

Biznesa vide kļūst arvien dinamiskāka un aug arī prasības pret informācijas pieejamību lēmumu izdarīšanai. Sistēmas, kas izstrādātas ikdienas biznesa operāciju uzskaitē, vairs nespēj apmierinošā kvalitātē pildīt informācijas piegādes funkcijas lēmumu pieņemšanas procesā.

Lai uzlabotu šīs specifiskās informācijas pieejas ātrumu un padarītu šo procesu viegli vadāmu un saprotamu pat cilvēkiem, bez specifiskām zināšanām programmēšanas jomā (vidējā un augstākā līmeņa uzņēmuma vadība), tika izstrādāta datu glabātavu koncepcija.

Datu glabātava (data warehouse) ir subjektorientēta, integrēta datu kolekcija ar laika dimensiju, kura pieejama tikai datu lasīšanai un kas domāta lēmumu pieņemšanas procesa atbalstīšanai.

Subjektorientēta – datu glabātava ir organizēta ap nedaudziem subjektiem (entītēm), kas tiek reģistrēti uzņēmuma sistēmās. Šie subjekti var būt piemēram klienti, produkti vai darbinieki.

Integrēta – Datu glabātavā var tikt apvienoti dati no dažādām uzņēmuma sistēmām, ja uzņēmumā to ir daudz un tie tiek glabāti savstarpēji savietojamā stāvoklī.

Laika dimensija – Dati, kas glabājas datu glabātuvē, var tikt analizēti pētot to izmaiņas laikā. Piemēram kā mainījies firmas apgrozījums pa gadiem.

Tikai datu lasīšanai – Gala lietotāji datus no datu glabātavas var tikai lasīt, tie nevar veikt nekādas manipulācijas ar šiem datiem. Dati, kas vienreiz ievadīti datu glabātavā vairs nav maināmi.

Datu glabātava nav viens konkrēts produkts. Līdzīgi kā jēdzies "DBVS" tā tikai vispārīgi sniedz priekšstatu par to, kas ir šis produkts un kāda ir tā funkcionalitāte.

Komerčiālo DBVS veidotāji piedāvā rīkus, kuri ļauj no to datu bāzēm ērti veidot datu glabātavas. Diemžēl ne PostgreSQL ne MySQL pagaidām šādu iespēju nav.

19.1. DATU GLABĀTAVU UN TRADICIONĀLO DATU BĀZU SALĪDZINĀJUMS:

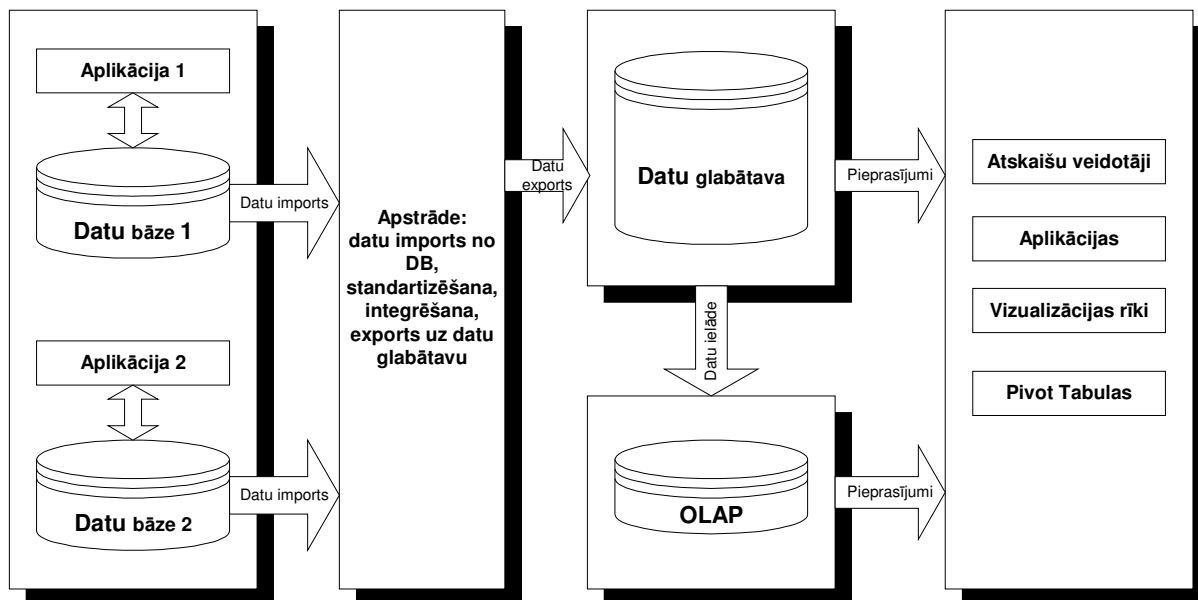
Tradicionālo datu bāzes	Datu glabātavas
Transakciju orientētas	Subjektorientētas
Ļoti daudz lietotāji	Nedaudzi lietotāji
Ne pārāk lielas	Ļoti lielas
Tekošie dati	Visi iepriekšējie dati
Normalizēti dati (daudz tabulu, katrā tabulā dažas kolonnas)	Denormalizēti dati (dažas kolonnas ar daudz kolonnām)
Nepārtrauktas izmaiņas	Parasti izmaiņas notiek 1reiz dienā.
Vienkārši pieprasījumi	Ļoti sarežģīti pieprasījumi

Maz indeksu

Daudz indeksu

19.2. DATU GLABĀTAVAS SISTĒMAS ARHITEKTŪRA

Datu glabātavas sistēmu arhitektūru var sadalīt 4 daļās:



1) DB sistēmas, kas kalpo kā datu avoti.

Apstrādes procedūras

Tās veic datu importu no datu avotiem, apstrādā tos, veic nepieciešamās datu korekcijas, lai nodrošinātu datu integritāti, datu kārtošana un apstrāde, lai iegūtu datu glabātavai nepieciešamos datus. Pēc tam seko datu eksports uz datu glabātavu.

Datu glabātava, OLAP

Datu glabātavā glabājas nepieciešamie dati, kas ir saformēti atbilstoši nosacījumiem, kas uzstādīti veidojot šo datu glabātavu. Dati tiek glabāti izmantojot zvaigznes relāciju modeli.

OLAP (On-Line Analytical Processing)

Speciāls middle tier servera programma, kas darbojas kā starpnieks starp datu glabātavu un klientu aplikācijām, kas pieprasa datus no tās. Atšķirībā no datu glabātavas, OLAP uzglabā konsolidētus datus daudzdimensiju matricas formā.

Gala lietotāju pieprasījumu rīki.

Dažādas programmas, kas māk griezties pie datu glabātavas un OLAP un iegūt no tām datus, ko pieprasa lietotājs.

19.3. DATU GLABĀTAVAS

Lai datu glabātavas varētu veikt savas funkcijas un nodrošināt nepieciešamo informāciju lēmumu pieņemšanas procesā, tās regulāri ir jāpapildina ar jaunākajiem datiem no uzņēmuma uzskaites sistēmām. Parasti šis process tiek veikts katras dienas beigās, kad uzskaites sistēmās vairs netiek veidotas jaunas transakcijas. (tomēr to iespējams darīt arī pa nedēļām – katras nedēļas beigās) Biežums atkarīgs no tā, cik svarīgas ir bieža informācijas aktualizēšana. Datu pievienošanas procesu anglicki sauc par ETL (Extraction, Transformation, Loading). Šo procesu var sadalīt 3 daļās:

1) Datu iegūšana

Datu iegūšanas procesā no uzņēmuma datu bāzēm tiek iegūti nepieciešamie izejas dati, kuri tālāk tiek apstrādāti un saglabāti datu glabātavā. Bieži uzņēmumos ir vairāk kā viena uzskaites sistēma un datu iegūšana jāveic no visām šīm sistēmām, kuras var darboties izmantojot dažādas platformas un dažādas DBVS. Datu iegūšanas veidus var sīkāk iedalīt 2 apakšveidos:

⇒ Pilnā datu ieguve (Full Extraction),

No uzskaites sistēmas tiek iegūti visi datu glabātavai nepieciešami dati, kas tajā ir. Šādas metodes priekšrocība ir tā, ka nav jāveido sarežģīti mehānismi kā kontrolēt, vai dati ir tikuši jau iepriekš ievietoti datu glabātavā vai arī ne. Trūkums ir tas, ka tiek nelietderīgi apstrādāts milzīgs daudzums ar datiem, jo vismaz 90% no tiem jau atrodas datu glabātavā. Tomēr tā kā parasti ETL process tiek veikts i automātiskā režīmā pa nakti, kad uzņēmuma datorsistēma nav noslogota, tad ļoti bieži tiek izmantots tieši šāds risinājums.

⇒ Papildinošā datu ieguve (Incremental Extraction)

No uzskaites sistēmas tiek iegūti tikai tie dati, kuri jau nav datu glabātavā. (dati, kuri pievienoti, mainīti vai dzēsti kopš pēdējās datu iegūšanas reizes. Parasti uzskaites sistēmas nevar tikt modificētas, lai atspoguļotu, kuri dati ir mainīti un kuri ne, tāpēc šīs salīdzināšanas funkcijas jāuzņemas datu iegūšanas procedūrām.

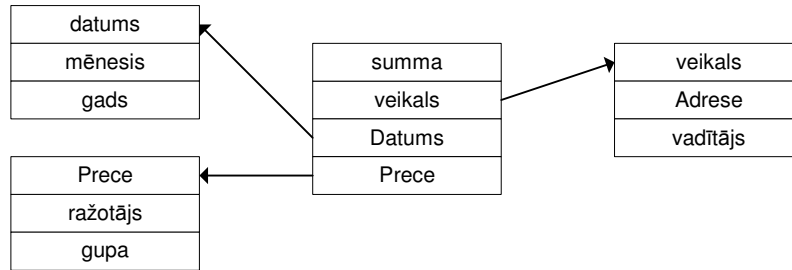
2) Datu pārveide

Datu pārveides procesā no uzskaites sistēmas iegūtie dati tiek pārveidoti atbilstoši struktūrai kāda ir datu glabātavai. Kā jau tika minēts iepriekš, datu glabātava sastāv no nedaudzām nenormalizētām tabulām, un to struktūra ļoti atšķiras no tabulu struktūras kas tiek izmantotas uzskaites sistēmās. Datu pārveide tiek veikta pa soļiem, kur katra soļa beigās tiek ģenerēta pagaidu tabula, kura tālāk tiek izmantota kā datu avots nākamajā solī.

3) Datu ielāde datu glabātavā

Vienkāršākais no soļiem – dati parasti tiek saglabāti teksta failā ar tabulas struktūru (flat file), no kura tie pēc tam tiek ielādēti datu glabātavā. Eksistē arī citi veidi (Transportable Tablespaces, Distributed Operations)

Datu glabātavā dati parasti tiek glabāti tā saucamajā zvaigznes shēmās. Centrā atrodas faktu tabula, kurā glabājas biznesa sistēmu rezultāti un ap to izvietotas dimensiju tabulas, kurās glabājas dati par dimensijām.



19.4. OLAP

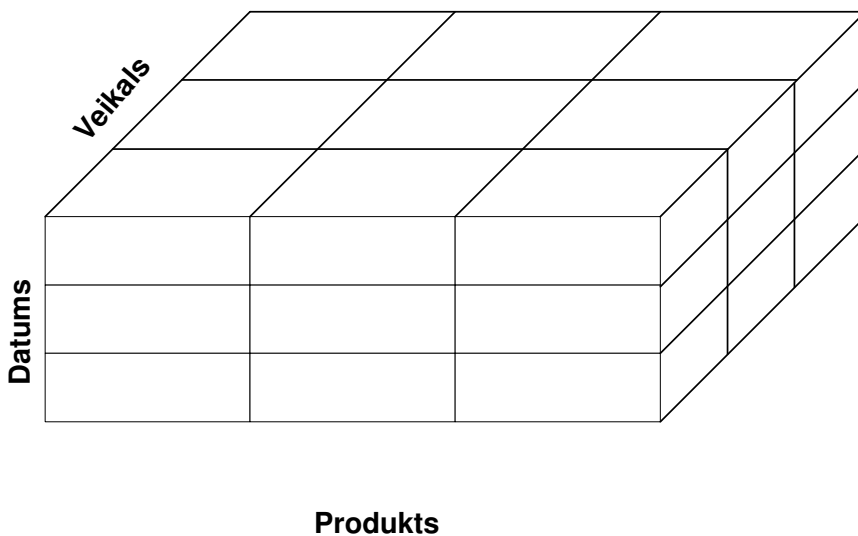
OLAP dati tiek glabāti multidimensionālās matricās, kas tiek sauktas par datu kubiem (data cubes). Šīm matricām var būt gan 3 gan arī vairāk dimensijas. Datu pieejas ātrums datiem, kas izvietoti datu kubos, ir daudz lielāks, kā tabulās, kas veidotas pēc relāciju datu modeļa. Šādos kubos parasti neglabā visu informāciju par karu transakciju, bet tikai summāro informāciju atbilstoši datu dimensijām.

Piemēram :

Dati relāciju datu bāzē (2 dimensiju tabula)

Produkts	Datums	Veikals
Produkts_1	01.01.2003	Veikls_1
Produkts_2	01.01.2003.	Veikals_2
Produkts_3	01.01.2003.	Veilals_1
...		

OLAP datu bāze:



Katrā rutiņā tiek ierakstīts pārdošanas apjomu kopskaits attiecīgajai dienai, attiecīgajam produktam un attiecīgajam veikalam. 4 dimensija šādā gadījumā varētu būt datu veids kopējais skaits un kopējā summa.

19.5. HIERARHISKĀS DIMENSIJAS

Lai padarītu izmantošanas iespējas plašākas, gan datu glabātavas gan OLAP piedāvā iespējas iegūt hierarhisku skatījumu uz datu dimensijām. Tas nozīmē, ka datu dimensiju lielumi var tikai grupēti vairākos līmeņos un pēc tam atskaitēs var iegūt rezultātus ar dažādu detalizāciju.

Dimensiju dati tiek grupēti atsevišķās "dimensiju tabulās". Piemēram dimensiju dati – "Datums", kur katra vienība ir viens datums, bet datumi var tikt grupēt mēnešos un gados, līdz ar to parādās iespēja veikt analīzi gan pa datumiem, gan mēnešiem, gan pa gadiem. Līdzīgi var tikt grupēti arī produkti (piem. pēc piederības kādai produktu grupai vai ražotājam) un veikali (piem. pēc reģiona, kur tie atrodas). Pēc tam kombinējot dažādus datu dimensijas līmeņus viegli iespējams iegūt tieši to informāciju, kas nepieciešama lēmumu pieņemšanas procesā.

19.6. LIETOTĀJA INTERFEISS.

Relāciju datu bāzēs pieprasījumi pamatā tiek veidoti izmantojot SQL valodu. Lai to efektīgi pielietotu ir jāpārziņ programmēšana un datu bāzes struktūra. Tas neatbilst koncepcijai par datu glabātavām, kurā jābūt pieejamām bez šādām zināšanām. Šādiem mērķiem ir radīti programmatūras dažādi rīki. Microsoft ir izstrādājis Pivotabulu (pivottable) moduli, kurš tiek iekļauts daudzu Microsoft produktu sastāvā un kas ļauj lietotājam veikt datu analīzi vairākas dimensijās, izmantojot ērtu grafisko interfeisu.

19.7. PAPILDUS LIERATŪRA

1. Database Management Systems: 23. Decision support (677-704.lpp)

19.8. PAŠKONTROLES JAUTĀJUMI.

1. Kas ir datu glabātavas un kam tās izmanto.
2. Ar ko datu glabātavas atšķiras no tradicionālām datu bāzēm.
3. Kā tiek veidotas datu glabātavas, datu glabātavu arhitektūra.

20. Citi DB veidi.

Līdz šim kursa gaitā apskatījām vispopulārāko no patreiz izmantojamajiem DB veidiem – relāciju datu bāzes. Tomēr bez relāciju datu bāzēm eksistē vēl arī vairāki citi datu bāzu veidi, kuriem attiecībā pret relāciju datu bāzēm ir savas priekšrocības un savi trūkumi.

20.1. OBJEKTU UN OBJEKTU-RELĀCIJU DATU BĀZES.

Objektorientētai programmēšanai gūstot milzīgu popularitāti un kļūstot par "de facto" standartu aplikāciju izstrādē, radās jautājums – vai šādu koncepciju nevar pielietot arī veidojot datu bāzes. Sistēmu (tai skaitā DB sistēmu) loģiskās projektēšanas objektorientētu pieeju 90tajos gados izstrādāja zinātnieki J.Rambaugh, I.Jacobson un G.Booch. Viņi izveidoja UML valodu, ko varēja lietot objektorientētai sistēmu analīzei un objektorientētai datu modelēšanai. UML ir valodas, kas domāta programmatūras sistēmu specificēšanai, vizualizēšanai, konstruēšanai un dokumentēšanai, kā arī biznesa modelēšanai. Objektorientētā modelēšanā reālās pasaules objekti tiek attēloti kā klases, kas satur gan īpašības (atribūtus), gan arī metodes darbam ar šīm īpašībām.

Izmantojot tradicionālām relāciju datu bāzu vadības sistēmas ir problemātiski realizēt dažādus projektus, kur būtu jāveido ļoti sarežģītas datu struktūras, piemēram sistēmas zinātniskiem eksperimentiem, ģeogrāfiskās un multimēdiju informācijas sistēmas. Prasības šādās sistēmās atšķiras no prasībām, kas tiek izvirzītas parastām datu bāzu sistēmām, kas lielākoties tiek izmantotas biznesa procesu uzskaitē.

Objektorientētas pieejas pamatā ir 3 fundamentāli jēdzieni – iekapsulēšana (encapsulation), mantošana (inheritance) un polimorfisms (polymorphism).

Lai veidotu šiem modeļiem atbilstošas datu bāzes, tika izstrādātas vairākas Objektu datu bāzu vadības sistēmas (Object Data Base Management System-ODBMS). Literatūrā nākas sastapt arī nosaukumu Objektu orientēta datu bāzu vadības sistēma (Object Oriented Data Base Management System-OODBMS), kas ir sinonīms iepriekš minētajām Objektu datu bāzu vadības sistēmai.

Lai arī ir izstrādātas vairākas ODBVS, un uz tām balstītas datu bāzu sistēmas, tomēr pārāk plašu pielietojumu neviena no tām nav guvusi.

Paralēli tam notiek mēģinājumi apvienot tradicionālo Relāciju DBVS un Objektu DBVS stiprās puses un izveidojas Objektu Relāciju DBVS. Tomēr veicot šādu apvienošanu nākas iet arī uz kompromisiem, jo ir pietiekami liela atšķirība starp objektorientētu un relāciju datu bāzi, līdz ar to kaut ko no šīm stiprajām pusēm nākas zaudēt.

Liela daļa no populāriem relāciju DBVS izstrādātājiem (Oracle, Informix, Microsoft, Sybase) savos produktos ir iekļāvuši funkcionalitāti, kas ļauj tās izmantot kā Objektu Relāciju DBVS.