11.1. Erinevad andmebaaside käsitluse mudelid

Enne kui hakata rääkima andmebaasikäsitluse optimeerimisest ja transaktsioonikäsitlusest tuleb mõista seda keskkonda (IS mudelit), milles me toimime. Andmebaasi käsitlusel põhinev infosüsteemi mudel on muutunud läbi aja pidevalt koos infosüsteemide riist- ja tarkvaralise keskkonna arenguga. Praeguseks oleme jõudnud nn. n-kihiliste mudelite faasi, mille aluseks on laia geograafilise paiknevuse ning mitmekesine riistvaraline ja tarkvaraline keskkond.

aja jooksul on olnud kasutusel mitmed erinevad andmebaasi käsitlemise mudelid

11.1.1. Ühe kihiline mudel

Andmebaaside arengu esimeses etapis, mis kestis kuni eelmise sajandi 80-ndate aastate teise pooleni (siis tekkis klient-teenindaja arhitektuur) valitses andmebaaside käsitlemisega tegelevates infosüsteemides ühe-kihiline mudel. Selle etapi alguses paiknesid andmebaasid ja programmid, mis andmebaasides olevaid andmeid käsitlesid, samas arvutis. See oli *main-frame'*de ja mini-arvutite aeg. Isegi personaalarvutite kasutusele võtmise alguses ei muutunud midagi - nii andmebaas, kui selle käsitlemiseks kirjutatud programmid paiknesid ikka veel samas arvutis. Muutuse tõi arvutivõrkude kasutusele võtmine ja faili-serverite tekkimine. Sellel hetkel lahutati andmebaasid ja nende töötluskeskkond üksteisest ja enam nad vast kokku ei lähegi. Andmebaasid "kolisid" fail-serverisse ja töötlusprogrammid jäid tööjaamadesse.

esimene mudel: ühe-kihiline mudel

Loomulikult jäävad personaalsed andmebaasid ilmselt igavesest ajast igavesti tööjaamadesse, kuid need pisikesed "andmebaasikesed" ei kõiguta kuidagi seda eelmist väidet. Personaalsetel andmebaasidel on ainult üks kasutaja ja nende tähtsus võrreldes globaalsete andmebaasidega on väga väike.

personaalsed, lokaalsed andmebaasid

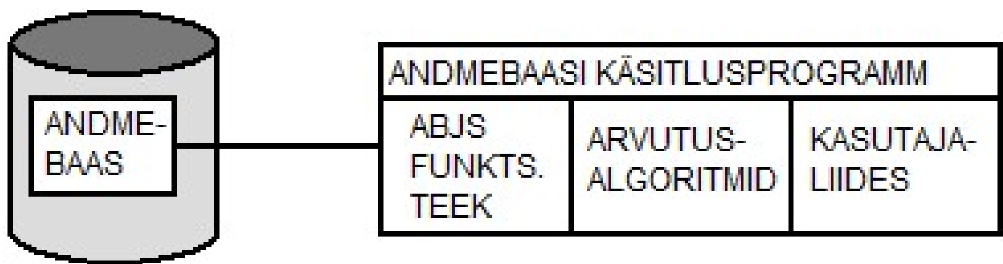
Milline siis oli 1-kihiline mudel? Mida mõeldakse üldse, kui räägitakse infosüsteemi arhitektuuri kihilisusest? Mida tähendab see "kiht"? Infosüsteemi mudeli kihiks nimetatakse ühte teistest tarkvarakomponentidest suhteliselt eraldatud tarkvara komponentide kogumist, millel on funktsionaalselt määratletud tähendus ja mis vahetab teiste analoogiliste komponentidega (teiste kihtidega) informatsiooni läbi standardiseeritud liidese *(standardized interface)* kaudu.

kihi definitsioon

Nagu sellest definitsioonist järeldada võib, pole andmebaaside käsitlemise ühe-kihilise mudeli puhul rohkem kui üks tarkvaraline komponent. Seega pole ka mingeid liideseid, mille kaudu teiste komponentidega informatsiooni vahetada, sest neid teisi kihte lihtsalt pole Kogu andmebaasi käsitlemise "tarkus" on

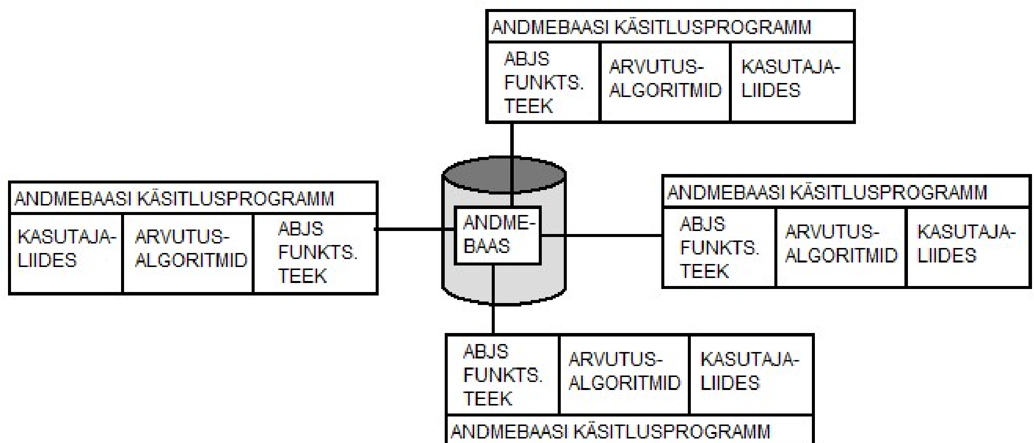
programm = kasuajaliides + arvutusalgoritmid + ABJS funktsioonide teek

kapseldatud ühte programmi. See programm peab hakkama saama nii kasutaja dialoogide juhtimisega, kui andmete füüsilise kirjutamisega andmebaasi. Selleks, et programmeerijad ei peaks iga kord kogu andmebaasi andmete kirjutamist programmeerima olid igal andmebaasisüsteemil standardfunktsioonide teegid, mis liideti (lingiti) iga selle andmebaasisüsteemi andmebaasi käsitlemiseks kirjutatud programmiga ja programmeerijad lihtsalt kutsusid, andmete baasi kirjutamiseks ja sealt nende lugemiseks, neid funktsioone välja *(call).* Iga andmebaasi käsitlemise programm oli kogum kasutaja liidese ja arvutusprotseduuride jaoks spetsiaalselt kirjutatud programmi ja andmebaasisüsteemi funktsioonide teegi kogum.



***multi-tasking* ja ühe-kihiline mudel**

Kuna andmebaas paiknes töötlus-programmiga samas arvutis ja korraga siia programmi kasutada ainult üks kasutaja, siis sellest mudelist täiesti piisas. Tegelikult tekkisid esimesed tõsisemad probleemid siis, kui tekkis nn. *multi-tasking ja* ajajaotus-süsteemid *(time sharing systems),* kus sama arvuti peal sai protsesse käivitada juba mitu kasutajat. Seega sai ka sama andmebaasikorraga käsitleda mitu kasutajat. Probleemid tekkisid seepärast, et kasutajad hakkasid üksteist segama - keegi kirjutas midagi andmebaasi, keegi kustutas selle ära, keegi kirjutas midagi teise jaoks väga vajalikku üle jne.

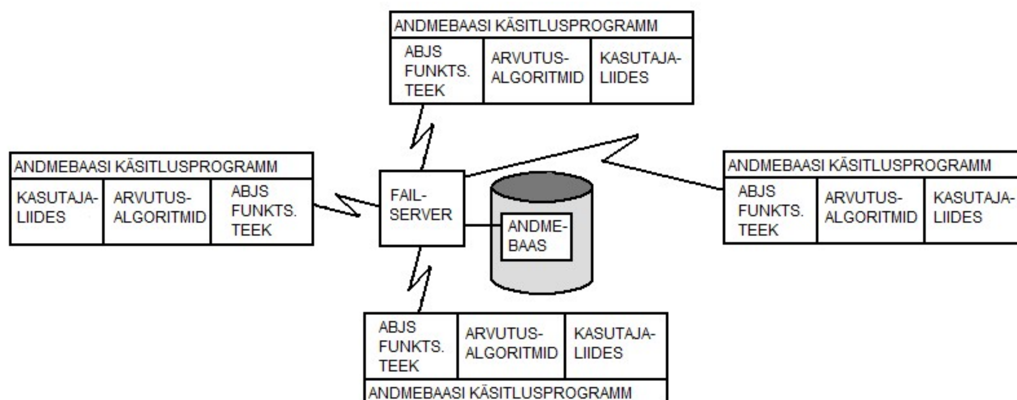


Mudel ise nägi välja selline. Unustada ei tohi seda, et kõik see toimis ühe arvuti piires. Et tekkinud probleeme lahendada, siis hakati andeid lukustama. Kui mõni protsess tahtis tagada, et andmed mingi, selle protsessi jaoks vajaliku aja jooksul ei muutuks, pani see vajalikele andmebaasi tabelite kirjetele luku. Lukk kirjutati kirje päisesse ja kui seda enam vaja polnud, kirjutati see tühja lukuga üle. Lukus oli kirjas see, milline protsess selle luku pani - ainult luku pannud protsess sai selle maha võtta (tühja lukuga üle kirjutada). Kui nüüd aga protsess juhtus "ära surema" ja protsessi poolt oli pandud aktiivseid lukke, siis jäidki need "igavesest ajast igavesti" kehtima ja takistasid teiste kasutajate tööd. Selleks, et "ära surnud" protsesside poolt pandud lukke maha võtta, kirjutati spetsiaalseid programme, mis teatava ajavahemiku tagant baasi üle vaatasid ja otsisid selliste protsesside poolt pandud lukke, mis ise "surnud olid". Kõige triviaalsemad lahendid olid sellised, kus teatud aja tagant lõpetati kõigi protsesside töö, hävitati kõik lukud ja siis jätkati tööd.

ühe-kihilise mudeli lukustusprobleemid

Hiljem võeti kasutusele lokaalsed arvutivõrgud ja andmebaas hakati hoidma fail-serverites. Mudel ise teisendus selliseks:

ühe-kihiline mudel ja lokaalsed arvutivõrgud



Midagi sellest paremaks ei läinud, muutus ainult füüsiline arhitektuur. Loogiline arhitektuur jäi samaks. Muutus ainult see, et "ketas, kus andmebaasi hoiti" nihkus tööjaamast kaugele, tegelikkuses aga "mängis fail-server tööjaamale ketast" nii, et tegelikult ei olnud tööjaamas aru saadagi, kas ketas, kus paikneb andmebaas on tööjaamas eneses või kusagil mujal. Muutus siiski üks asi - kasutajaid tuli veelgi juurde ja probleemid andmete lukustamisega muutusid veelgi hullemaks. Vaatamata sellele, et teoreetiliselt oleks selle arhitektuuri juures olnud võimalik sama andmebaasiga samaaegselt võinud tööd teha väga palju kasutajaid, pani andmete ühiskasutuse kohmakus reaalsele kasutajate arvule piiri peale.

andmete ühiskasutuse probleemide suurenemine

Vaja oli uut arhitektuuri, mis lahendaks selle probleemi. Uueks arhitektuuriks, mis andmete ühiskasutuse ja andmete lukustamise probleemi lahendas, oli klient/teenindaja (client/server) arhitektuur.

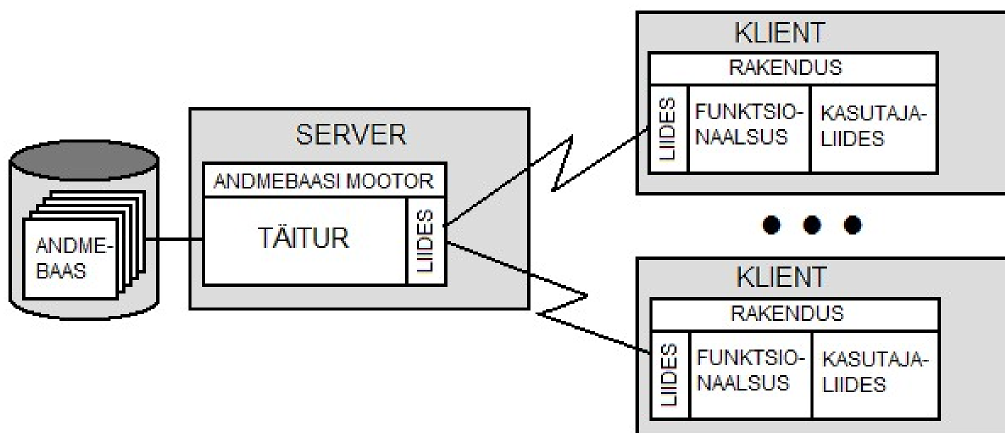
11.1.2. Kahe-kihiline andmebaaside käsitlusmudel - klient/teenindaja *(Client/Server)* mudel

klient/server - kahe-kihiline arhitektuur

Kahe-kihilise mudeli korral ei pöördu tööjaamad vahetult andmebaasi poole, vaid andmebaasi käsitleva rakenduse ja andmebaasi "vahel" on programm, mis täidab rakenduse korraldusi ja vahenda viimasele andmeid. Seda programmi nimetatakse andmebaasi mootoriks *(database engine)* Tegelikkuses võib olla selliseid andmebaasi poole pöörduvaid rakendusi kuitahes palju. Ainsaks piiranguks on siin see, kui suute jõudlust nõuab iga selline rakendus enda teenindamiseks andmebaasi mootorilt. Tegelikkuses seabki füüsilise piiri siin serveri riistvara konfiguratsioon - protsessori kiirus, mälu maht, ketaste maht, ketaste kiirus, kanali kiirus jne.

klient-teenindaja arhitektuur

Klient/teenindaja arhitektuur on järgmine:



andmebaasi mootor vs. rakendusprogramm

Iga andmeside protsess algatatakse kliendi arvutis. Kui seal töötav rakendusel on vaja kas andmebaasi midagi kirjutada või andmebaasist midagi lugeda, siis saadab ta korralduse andmebaasi mootorile, viimane täidab korralduse ja kui korralduse osaks oli andmete pärimine baasist, siis saadab need andmed rakendusele tagasi. Kuna siin on üks protsess andmebaasi poolel, andmebaasi mootor, mis koordineerib kõiki teisi protsesse, siis pole enam mingit vajadust lukke andmebaasi salvestada - andmebaasi mootor haldab kõiki lukke oma sisemiste meetoditega. Ühtlasi jälgib andmebaas, millises seisus on temaga

ühendust võtnud rakendused, ja kui mõni nendest peaks ootamatult töö lõpetama, siis hävitatakse koheselt ka kõik sellega seotud lukud..

Oluline on märkida veel seda, et konkreetse andmebaasi mootoriga suhtlev rakendus peab tundma seda liidest, mille kaudu andmebaasi mootorile korraldusi saata ja mille kaudu võtta vastu andmeid. Igal andmebaasisüsteemil on oma spetsiifiline liides, mida kutsutakse "*native link*" (eesti keelne tunnustatud termin puudub) ja kui mõni rakendus tahab andmebaasi poole pöörduda, siis peab tema koosseisus olema just selle andmebaasisüsteemi draiver, mille andmebaasi poole pöörduda soovitakse. Sama rakendusega võivad olla liidetud ka mitme andmebaasisüsteemi draiverid.

***native link* (ürgne, ainuomane liides)**

Üks andmebaasi mootor võib hallata korraga mitu andmebaasi ja harilikult see nii ongi.

üks mootor - mitu andmebaasi

On olemas ka universaalsed liidesed. MicroSoft' standardiks on ODBC *(Open Databas Connectivity)*. Enamik andmebaasisüsteeme tunnevad ka seda liidest. Selle liidesega suhtlemiseks peab rakendusega olema liidetud ODBC Draiver. ODBC draiveri suurimaks miinuseks on see, et ta ei ole väga kiire. Eriti aeglane on andmebaasiga ühendumine *(connect)*. Seepärast välditakse selle kasutamist, kui mingi muu võimalus on olemas.

ODBC

Viimaste aastate jooksul on ilmunud uus universaalne andmebaasi liidese standard OLE DB. See liides on väga hea ja ületab oma headelt omadustelt paljude andmebaasisüsteemide *native* liidese.

OLE DB

Klient/teenindaja arhitektuuris liiguvad korraldused ja andmed kliendi ja serveri vahel järgmise skeemi kohaselt:

infovahetus klient/teenindaja arhitektuuris

Kliendi poolt saadetakse andmebaasi mootori poole korraldus ja ka andmed, kui tegemist on andmete lisamise või uuendamise korraldusega. Andmebaasi mootor täidab korralduse. Pärast korralduse täitmist tagastab andmebaasi mootor kliendile veakoodi ja andmed kui vea-kood oli 0 /st. viga polnud). Kõik see käib teadete kaupa - üks pool saadab teate, teine pool võtab teate vastu, täidab selles

tehtud korraldused ja vastab teatega või teadete seeriaga, kui edastatavat informatsiooni on rohkem.

klient/teenindaja arhitektuur ühe arvuti piires

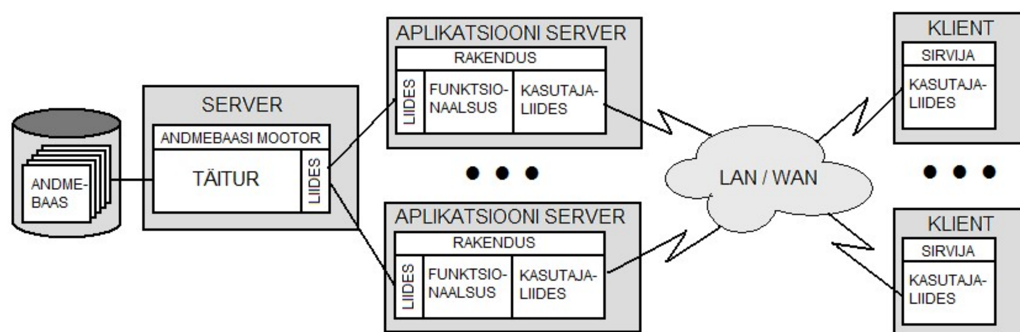
Nüüd veel natuke klient/teenindaja arhitektuuri kahe-kihilisusest. Klient/teenindaja arhitektuur on kahe-kihiline arhitektuur. Nendeks kaheks kihiks on andmebaasi serveris asuv andmebaasi mootor ja kliendi arvutis olev rakendus, mis suhtlevad standardiseeritud liides kaudu. Need mõlemad komponendid (kihid) võivad paikneda ka ühes arvutis - see ei muuda klient/teenindaja kahe-kihilisust millekski muuks. Ka ühe arvuti piires suhtlevad need kaks komponenti samuti nagu siis, kui nad on erinevates arvutites.

11.1.3. Kolme-kihiline mudel

kolme-kihilise ja kahe-kihilise mudeli erinevus

Kolme-kihilist andmebaasi käsitluse mudelit eristab see, et kasutaja liides on viidud rakenduses teemale. Rakendus ise töötab andmebaasi mootoriga täpselt samuti nagu kahe-kihilises mudeliski. Vahe on ainult selles, et kasutajaliides on viidud rakendusest välja:

kolme-kihiline arhitektuur



kolme-kihiline rakendus = andmebaasi mootor + rakendus + kauge kasutaja liides

Rakendused pannakse tavaliselt tööle selleks eraldi loodud keskkonnas, aplikatsiooni serveris, aga tegelikult ei ole see kohustuslik. Rakendus võib ka ise olla nii üles ehitatud, et ta suudab ilma aplikatsiooni s4rveri keskkonnata töötada. Oluline on see, et meil on kolm komponenti - andmebaasi mootor, mis tegeleb vahetu andmebaasi käsitlemisega, rakendus, mis juhib protsessi ja kasutajaliides, mille kaudu kasutaja suhtleb rakendusega. Andmebaasi serveri ja rakenduse vahel toimub korralduste ja andmete liigutamine läbi mõne *native* lingi, ODBC või OLE DB. Rakenduse ja kasutajaliidese vaheliseks liideseks on kas HTTP, HTTPS, XML (RPC), SOAP või mõne muu standardiseeritud liidese kaudu.

dialoogi käik

Dialoogi alustamiseks peab kasutaja pöörduma mingil URL-l. Seal registreerumisel (tavaliselt kasutaja nime ja võtmesõnaga) käivitatakse

aplikatsiooni serveris tema jaoks protsess, mille kasutajaliides saadetakse tema sirvijasse. Kui nüüd kasutaja teeb läbi kasutajaliidese rakendusega tegevusi, siis rakendus vahetab kõik vajalikud andmed ja korraldused andmebaasi mootoriga.

Kui me nüüd jätame ära kolmanda komponendi, kliendi, siis andmebaasi serveri ja rakenduse vahel toimib endiselt klient/teenindaja arhitektuurile vastav ühendus.

klient/server on kolme-kihilise mudeli sees

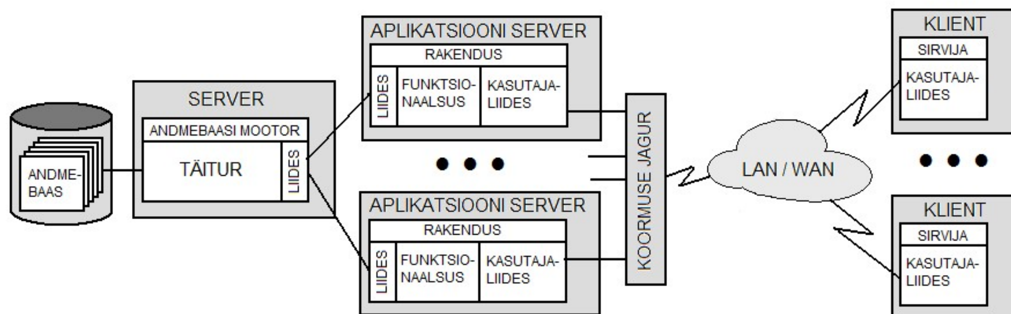
Kolme-kihilises arhitektuuris käib andmevahetus analoogiliselt kahe-kihilise mudeliga. Vahe on ainult selles, et korralduste initsialiseerimine ja andmete vastu võtmine toimub "kauge" kasutaja liidese kaudu:

infovahetus kolme-kihilises arhitektuuris

11.1.4. N-kihiline mudel

Tegelikkuses võib infosüsteemi arhitektuuris olla oluliselt rohkem kihte kui kolm. Kui vaadata suurte infosüsteemide arhitektuuri, siis nii see ongi. Näiteks juba kolme-kihilise mudeli puhul kasutatakse tihti veel ühte vahelüli, et ühtlustada süsteemi koormust:

kihte võib olla oluliselt rohkem kui kolm



Siin mudelis on mitu sarnast aplikatsiooni serverit, mis kõik suudavad sama tööd teha. Koormuse jagur kindlustab selle, et kõik need serverid oleks ühtlaselt koormatud ja sedasi tagatud klientide kõige parem teenindamine. See pole küll otseselt nelja-kihiline aplikatsioon aga "kolme ja poolene" küll.

serverite koormuse jagamine

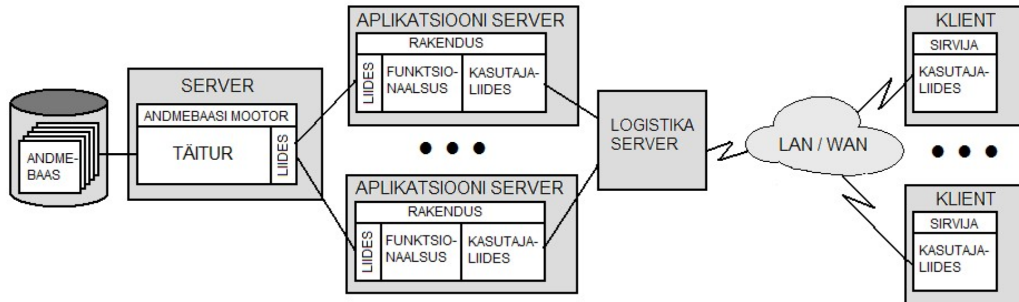
Kui infosüsteemile hakkab kihte lisanduma, siis lisanduvad need uued kihid alati aplikatsiooni serveri ja kliendi vahele. Seda seepärast, et kui me tekitaksime uue kihi aplikatsiooni serveri ja andmebaasi mootori vahele. siis me

kihid lisanduvad alati aplikatsiooni serveri ja kliendi vahele

lõhuksime ära seal toimiva klient/teenindaja mudeli. Paraku pole tänaseks ühtegi paremat andmebaasi käsitlemise mudelit välja mõeldud.

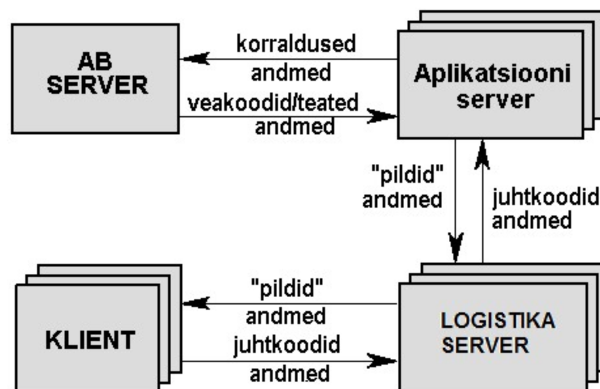
nelja-kihiline arhitektuur - logistika server

Toome näiteks ühe enam levinud nelja-kihilise infosüsteemi mudeli, kus aplikatsiooni serveri ja kliendi vahele on paigutatud logistika server, mis teostab süsteemi sisemist logistikat ja valida selle aplikatsiooni serveri, millel on see rakendus, mida klient vajab.



nelja-kihilise arhitektuuri andmeedastusmudel

Logistika server on ainult vahendaja, mis leiab õige rakenduse ja vahendab selle kliendile. Siiski võib siia serverisse olla lisatud ka teatav lisafunktsionaalsus näiteks andmete "tõlkimine" vajalikku formaati erinevate rakenduste vahel liikumisel:

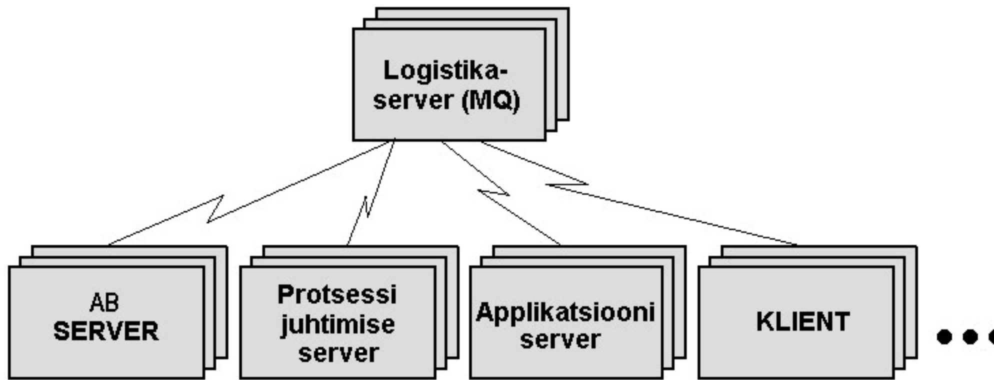


Uute, erinevat rolli omavate serverite, lisamist logistika serveri lähedusse piirab ainult infosüsteemi loojate fantaasia ja reaalsustaju - liiga paljusid kihte sisaldavate arhitektuuride haldamine võib osutuda liiga töömahukaks ja vahel isegi võimatuks.

11.1.5. Komponent mudel (0-kihiline mudel)

Komponent-mudeli puhul ei saa erilisest kihilisusest rääkida. Siin paiknevad kõik serverid samal tasemel ja suhtlevad üksteisega läbi logistika serveri või serverite. Serveritega samal tasemel paiknevad ka kliendid. Tegelikult on logistika server vaid üks serveritest, mis asub teiste serveritega samal tasemel. Joonisel on ta teistest serveritest eraldi joonistatud ainult jooniseülevaatlikuma paigutuse pärast:

kõik serverid ja kliendid on samal tasemel



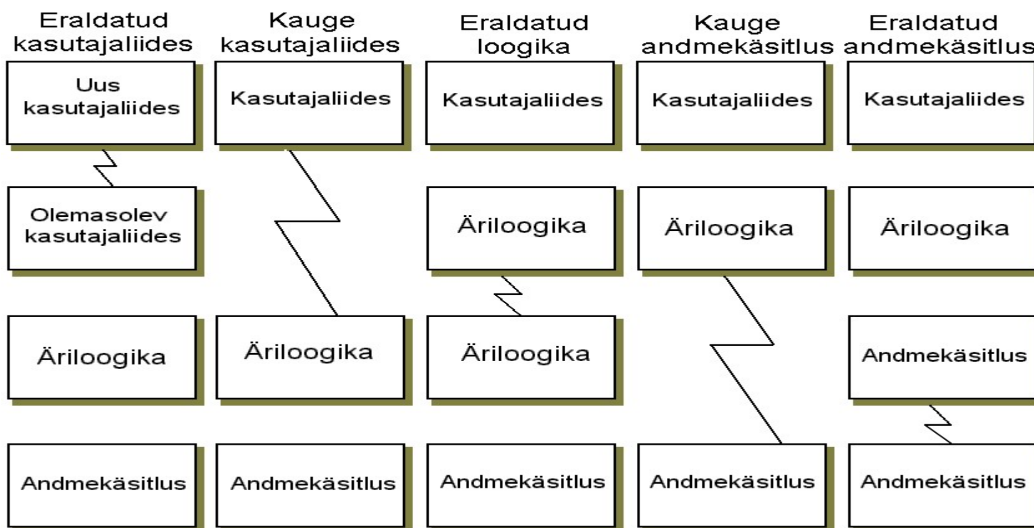
Sellise mudeli puhul suhtlevad kõik mudelid omavahe läbi ühe või mitme logistika serveri. Kui logistika servereid on mitu, siis sünkroniseerivad viimased oma vahelist tegevust ja vahendavad ühendusi teiste komponentidega. Tavaliselt valitakse selline mudel siis, kui sarnaseid süsteemi komponente (serverid) on rohkem kui üks ja kohati jagavad nad omavahelist koormust, kohati aga pakuvad erinevat funktsionaalsust. See arhitektuur sobib väga hästi keeruka arhitektuuriga ja dünaamilist laiendamist nõudvate infosüsteemide loomiseks.

arhitektuur keerukate ja dünaamiliste süsteemide tarvis

11.1.6. Infosüsteemi mudelite areng läbi ajaloo

Ühes Gartner Group'i 2001 aasta aruandes oli väga hea, infosüsteemide mudeli arengut kirjeldav joonis. Tegelikult vaatlesime kõike seda eelmistes jaotistes, kuid selliselt kokkuvõtlikuna ja natuke "teise nurga alt" annab see vaate terviklikkusele palju juurde:

joonis Gartner Group'lt (2001)



Pakett-töötluse mudeli korral toimub kogu töötlus samas arvutis. Mingit erilist suhtlemist kasutajaga ei toimu, kuna ülesanne (programm) "sisestatakse" arvutisse koos eelnevalt valmendatud andmetega ja kogu ülesanne lahendatakse korraga (paketina). Tulemiks on kas väljund-trükised või uus kogum andmeid. Kogu suhtlus arvutisüsteemiga toimub ühest kohast – operaatori terminalilt

pakett-töötlus

Kaug-terminalide mudeli korral toimub kogu töötlus samas arvutis. Vahe pakett-töötlusega oli esialgu ainult sellest, et operaatori terminale tekkis just kui juurde – andmete valmendus ja programmide käivitamine sai nüüd toimuda mitmest kohast korraga. Kogu ülesande lahendus toimus aga sama skeemi kohaselt – kui programm oli kord käivitatud, siis peatus see alles siis, kui lõpuni jõudis või vea tõttu peatus.

kaugterminalid

Programmeerimisvahendite arenedes tekkisid programmeerimissüsteemid ja vahendid, mis võimaldasid kirjutada selliseid programme, mis jäid seotuks selle käivitanud terminaliga ja võimaldasid "sekkuda" programmi täitumisse – tekkisid interaktiivsed süsteemid ja dialoog kasutajaga. Oma täieliku üleoleku saavutasid selline arhitektuur koos mini-arvutite võidukäiguga.

dialoogi tekkimine, kasutajaliides

Olenemata selle arhitektuuri arengu tasemest oli kõigis tema esinemise vormides terminal "rumal" (õhuke) sisend-väljund seade.

Personaalarvutite tulek tõi uut tüüpi arenduskeskkonnad, mis olid juba palju mobiilsemad – infosüsteemi osa oli võimalik kaasa võtta pea igale poole, kus oli elektrit. Siiski toimus mõningane tagasilöök eraldatuse suunas – infosüsteemid olid suutelised küll andmeid vahetama nö *off-line's*. Sama andmekomplekti üheaegselt nad siiski töödelda ei suutnud. Kogu andmete käsitlemise ja protsessi

personalarvutid

loogika ning vahendid paiknes tööjaamades eksisteerides paljude sarnaste koopiana, mille vahel vahetai andmeid failide kujul. Seda kas siis füüsilisi andmekandjaid ühest arvutist teise viies või siis telefonliinide ja modemite vahendusel.

lokaalvõrk

Selle probleemi lahendas lokaalvõrkude tulek, mis "pani" personaalarvutid omavahel suhtlema. Kogu loogika pöördus nüüd aga võrreldes eelneva "pea peale". Tekkis fail-server, mille ainuke ülesanne oli olla "kauge andmekandja kus paiknevad ühiskasutuses olevad andmed". Ise ta mingi "loogikaga ei tegelenud" ainult tegeles andmete töötlemisega kõige madalamal, *read/write* tasemel.

klient / server

Klient/serveri mudeli tulekuga oli loogiline mõte juba edasi arenenud – kogu füüsiline arhitektuur jäi samaks, muutus ainult loogiline arhitektuur. Andmete juurde tekkis andmebaasi-server (AB mootor), mille ülesandeks on "valvata" andmete terviklikkuse üle, "tagada" andmete säilimine ja muutmine vastavalt saabunud korraldustele ning väljastada andmeid vastavalt päringutele. Tööjaam nimetati ümber kliendiks ja tema juurest eraldati andmete füüsilise käsitluse protseduurid – need, mis anti üle andmebaasi serverile. Selles mudelis on mõlemad IS arhitektuuri komponendid (server ja klient) paksud (töökad). Serverile on jäetud andmete füüsiline haldamine, kliendile aga protsessiloogika juhtimine.

aeglased sidekanalid

Lokaalvõrgu ja piiratud arvu klientide juures töötab see mudel väga hästi – paremini kui ükski muu. Paraku võrkude hajususe suurenemise ja sellega seoses ka sama infosüsteemi kasutajate arvu kasvamine tekitas tarkvara levitamisel suuri raskusi – pidi ju muudatuste korral kliendi tarkvara "kopeerima" kõikidesse aktiivsete ja passiivsete kasutajate tööjaamadesse. Kuna kõikide kasutajate (klientide) juures ei ole aga tegevused sugugi samad (kasutaja profiilist lähtuvalt), siis suurenes paljusus veelgi, ning tarkvara levitamine hakkas käima üle jõu. Peamiseks takistuseks nagu öeldud võrkude aeglus.

kolme-kihiline mudel

õhuke klient

Siis jõutigi kolme-kihilise mudeli juurde, kus kasutajate aplikatsioonid töötavad spetsiaalsetes aplikatsioonserverites ja kliendil on ainult "aken" nendega suhtlemiseks. See "aken" on lahendatud tavaliselt kas läbi interneti sirvija või läbi mingi monitori programmi (N: Telnet). Sellises mudelis on kogu andmebaasiloogika ja käsitlus endiselt andmebaasiserveris (analoogiliselt klient/server mudeliga). Kasutaja aplikatsioonid on aga koondatud erinevatesse aplikatsioonserveritesse, kuhu spetsiifilise profiiliga kasutajad tekitavad omale "akna" pöörduses serveri poole enda protsessi käivitamiseks. Aplikatsioonserveri

ja kliendi vahel liiguvad ainult “ekraanipildid” ja andmed. Tegelik töötlus toimub aplikatsioonserveris, kus iga kliendi jaoks käivitatakse oma protsess. Sellise mudeli kohaselt on klient täiesti “õhuke” (laisk). Kogu protsessi juhtimine toimub aplikatsiooni serveris ja kogu andmete lõplik haldamine andmebaasiserveris.

n-kihiline mudel

N-kihiline arhitektuur ei muuda midagi andmebaasi serveri töös. Tavaliselt on aplikatsiooni serveri funktsionaalsusest eraldatud protsessi ja juhtimise üldine loogika ja jäetud sinna ainult konkreetsete tegevuste teostamise loogika. Kliendi toimimisloogikas ei muutu midagi.

komponent mudel

Komponent arhitektuur muutub võrreldes N-kihilise arhitektuuriga ainult komponentide sidumise meetodid – kõik komponendid pakuvad oma funktsionaalsust kasutamiseks, infologistilised meetodid võimaldavad valida komponente "vabalt". Suureneb ka komponentide variantsus. Õhukese kliendi kõrvale ilmub jälle ka paks klient. Mudeli arengu eelduseks on kaugete ja lühikeste sidekanalite kiiruse tõus.

paksu kliendi uus tõus

Viimasel kümnendil on, tänu laivõrkude kiiruse suurele tõusule hakanud arenema jälle paksud kliendid (appid). Enam ei ole probleemiks suhteliselt suurte programmimoodulite "nihutamine" suure maa taha. Laivõrkude suurenenud kiirus pakub selleks piisavalt häid võimalusi. Üheks põhjuseks paksu kliendi uuele tõusule on ka see, et nende abil õnnestub teha kliendi jaoks palju mugavamaid ja kliendi tegevust toetavaid kasutajaliidesest.

11.2. SQL-korralduse ettevalmistamine täitmiseks

viis etappi SQL-lause ettevalmistamisel täitmiseks

Iga andmekäsitlus-lause/korraldus, mis saadetakse rakendusest andmebaasi mootorile täitmiseks saadetakse läbib enne täitmist viis etappi:

1. leksiline analüüs (scanning) – keelekomponentide identifitseerimine
2. grammatiline analüüs (parsing) - keele süntaksi tunnistamine
3. kontroll (validating) – keele loogiliste konstruktsioonide (nimede, seoste jms.) õigsuse tunnistamine
4. optimeerimine (optimize, execution plan)– operatsioonide järjestuse määramine

5. kompileerimine (compile) – esitatud lause täidetava vahekoodi ehitamine

leksiline analüüs

Esiteks identifitseeritakse keele komponentide so. vaadatakse üle iga lause sõna ja kontrollitakse, kas selline sõna võib olla keele komponendiks. Kui leitakse mingeid "kummalisi" faase, mis ei ole keel komponendid, siis katkestatakse lekstiline analüüs veaga.

grammatiline analüüs

Teiseks viiakse läbi grammatiline analüüs so. "lammutatakse" SQL-lause tükkideks ja leitakse igale tükile tähendus - see on korralduse võtmesõna, see on tehtemärk, see on veeru nimi, see on muutuja, see on tabeli nimi jne. Süntaksi tunnistamise edukas lõpule viimine eeldab seda, et lause on tõe poolest süntaktiliselt korrektselt kirjutatud ja viimane kui komponent on ära tuntav. Kui lause ei ole süntaktiliselt õigesti kirjutatud, lõppeb grammatiline analüüs veaga.

kontroll

Kontrolli käigus vaadatakse, kas lauses kirjeldatud andmebaasi komponentide nimed (tabelite nimed, veergude nimed, protseduuride nimed jms. on ikka andmebaasis olemas ja kui on, kas nad siis on lubatud SQL-lause andmebaasi saatnud kasutajale kasutamiseks. Kui leitakse selliseid nimesid, mida baasis pole siis tagastatakse sellekohane veateade. Kui aga avastatakse õiguste rikkumine antakse sellest ühemõtteliselt teada. Mõlemal juhul lause läbi vaatamine katkestatakse.

optimeerimine

Optimeerimine on kõige keerukam etapp. Siin määratakse see, millises järjekorras tehakse päringuid baasi, milliseid indekseid kasutatakse, millised piirangud ja millises järjekorras rakendatakse jne. Selle etapi tulemuseks on SQL-lause täitmise algoritm. Siin etapis ei tohiks enam vigu tekkida, sest SQL-korralduse "kvaliteet" on juba eelmise kolme etapiga kontrollitud ja heaks kiidetud. Tegelikkuses juhtub vigu ka siin. Need ei ole aga enam programmeerija tingitud vead vaid andmebaasisüsteemi loojate vead - süsteemsed vead. Nendest mööda pääsemiseks on ainult üks võimalus. Proovige see sama asi, mida te andmebaasi serveril teha palusite, kirjutada üles kuidagi teisiti - lihtsalt "teiste sõnadega". Käituge nii nagu inimesega, kes teie poolt öeldust aru ei saa.

kompileerimine

Viimane etapp on kompileerimine. See tähendab seda, et võetakse eelmises etapis loodud algoritmi kirjeldus ja ehitatakse sellest andmebaasi mootorile arusaadav vahekood - selline mida saab täita. Ka siin võib juhtuda vigu ainult

samal põhjusel kui eelmises etapiski. Ka vea kõrvaldamise meetodid on samad.

Loomulikult tuleks süsteemsetest vigadest teavitada tootjat - äkki leiavad nad aega ja parandavad vea ära.

Kui etapp õnnestub, jätkatakse järgmise etapiga, kui etapp ei õnnestu, siis tagastatakse viga ja töötlus katkestatakse

Tihti vaadeldakse esimest kolme etappi ja kahte viimast etappi kui ühtseid tervikuid ilma neid täpsemalt eristamata – analüüs ja kompileerimine. Tihti käsitletakse ka kõiki viit etappi kui ühtset tervikut – kompileerimist.

analüüs ja kompileerimine

Kui kõik eelnev õnnestub, siis järgneb lause/korralduse kompileeritud kuju täitmine.

edukale analüüsile kompileerimisele järgneb lause täitmine

Täitmise tulemusena tagastatakse igal juhul vea-kood või/ja teated (veakood võib olla ka 0 st. OK – kõik õnnestus) ja kui SQL-lause oli andmeid tagastav siis saadetakse käsu saatjale tagasi ka andmed. Tagasi saadetav andmepuhver võib olla ka tühi, kui käsus kirjeldatud piirangutele baasis vastavaid andmeid ei leitud.

alati tagastatakse veateade

vahel ka andmed

11.3. Põhiliste SQL-lausete täitmise skeemid

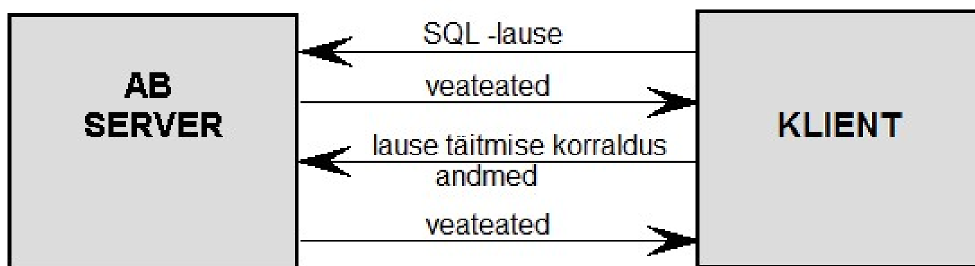
Iga SQL-lause kasutab andmebaasi serveri mootoriga suhtlemiseks mingisugust kliendi ja serveri vahelist teadete edastamise skeemi. Peamisi teadete vahetamse skeeme on neli:

1. INSERT-lause teadete edastusskeem
2. UPDATE-lause teadete edastusskeem
3. DELETE-lause teadete edastusskeem
4. SELECT-lause teadete edastusskeem
5. määratluse-lause teadete edastusskeem

INSERT-lause kliendist serverisse saatmisel analüüsitakse ja kompileeritakse serverisse saadetud lause. Selle õnnestumisel tagastatakse korralduse saatnud kliendile tagasi veateade - kas 0 (õnnestus) või nullist erinev so. tekkis/avastati

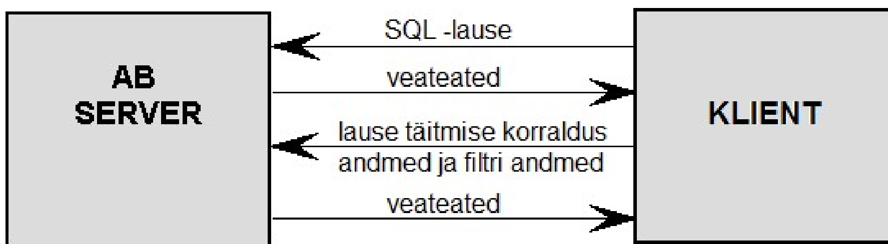
INSERT-lause täitmise skeem

mingi viga. Kui tekkis viga, siis korralduse täitmine siinkohal katkeb. Kui lause analüüs ja kompileerimine oli edukas saadab kliendi arvutis olev protseduur nüüd serverile teate korraldusega täita kompileeritud lause. Koos täitmis-korraldusega saadetakse serverisse ka andmed, mis tuleb andmebaasi lisada. Seejärel üritab server kompileeritud korraldust täita. Kui see õnnestub tagastatakse kliendile veateade 0 - kõik õnnestus. Kui korralduse täitmine ei õnnestunud, tagastatakse viga. Viga võib tekkida näiteks serveriga ühenduse kadumise tõttu või siis seetõttu, et serverisse saadetud andmete formaadid või tüübid olid valed. Näiteks üritati kuupäeva välja salvestada teksti või teksti välja sisse pikemat teksti kui veeru pikkus on tegelikult kirjeldatud (CREATE TABLE-lauses):



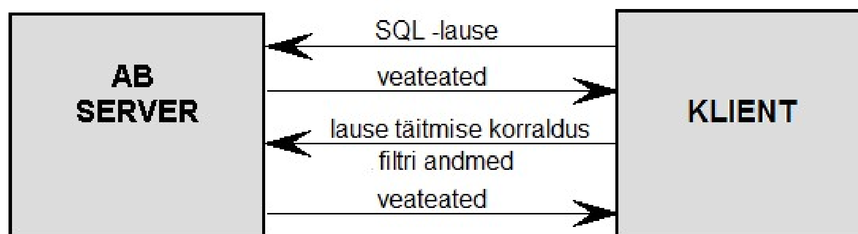
UPDATE-lause täitmise skeem

UPDATE-lause kliendist serverisse saatmisel analüüsitakse ja kompileeritakse serverisse saadetud lause. Selle õnnestumisel tagastatakse korralduse saatnud kliendile tagasi veateade - kas 0 (õnnestus) või nullist erinev so. tekkis/avastati mingi viga. Kui tekkis viga, siis korralduse täitmine siinkohal katkeb. Kui lause analüüs ja kompileerimine oli edukas saadab kliendi arvutis olev protseduur nüüd serverile teate korraldusega täita kompileeritud lause. Koos täitmis-korraldusega saadetakse serverisse ka andmed, mis tuleb uuendada. Kui lause WHERE tingimuses oli kasutatud programmi muutujaid, siis saadetakse serverisse ka need. Seejärel üritab server kompileeritud korraldust täita. Kui see õnnestub tagastatakse kliendile veateade 0 - kõik õnnestus. Kui korralduse täitmine ei õnnestunud, tagastatakse viga. Viga võib tekkida näiteks serveriga ühenduse kadumise tõttu või siis seetõttu, et serverisse saadetud andmete formaadid või tüübid olid valed. Näiteks üritati kuupäeva välja salvestada teksti või teksti välja sisse pikemat teksti kui veeru pikkus on tegelikult kirjeldatud (CREATE TABLE-lauses):



DELETE-lause täitmise skeem

DELETE-lause kliendist serverisse saatmisel analüüsitakse ja kompileeritakse serverisse saadetud lause. Selle õnnestumisel tagastatakse korralduse saatnud kliendile tagasi veateade - kas 0 (õnnestus) või nullist erinev so. tekkis/avastati mingi viga. Kui tekkis viga, siis korralduse täitmine siinkohal katkeb . Kui lause analüüs ja kompileerimine oli edukas saadab kliendi arvutis olev protseduur nüüd serverile teate korraldusega täita kompileeritud lause. Kui lause WHERE tingimuses oli kasutatud programmi muutujaid, siis saadetakse serverisse ka need. Seejärel üritab server kompileeritud korraldust täita. Kui see õnnestub tagastatakse kliendile veateade 0 - kõik õnnestus. Kui korralduse täitmine ei õnnestunud, tagastatakse viga. Viga võib tekkida näiteks serveriga ühenduse kadumise tõttu või siis seetõttu, et serverisse saadetud andmete formaadid või tüübid olid valed. Näiteks üritati filtris võrrelda kuupäeva tekstiga, arvu kuupäevaga vms.

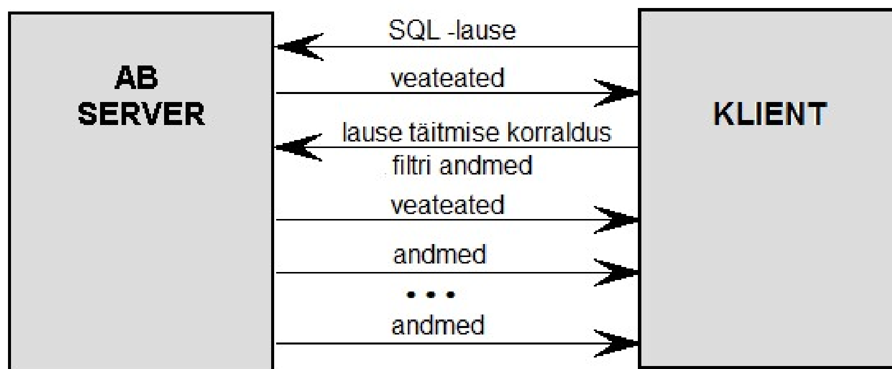


SELECT-lause täitmise skeem

SELECT-lause kliendist serverisse saatmisel analüüsitakse ja kompileeritakse serverisse saadetud lause. Selle õnnestumisel tagastatakse korralduse saatnud kliendile tagasi veateade - kas 0 (õnnestus) või nullist erinev so. tekkis/avastati mingi viga. Kui tekkis viga, siis korralduse täitmine siinkohal katkeb . Kui lause analüüs ja kompileerimine oli edukas saadab kliendi arvutis olev protseduur nüüd serverile teate korraldusega täita kompileeritud lause. Kui lause WHERE tingimuses oli kasutatud programmi muutujaid, siis saadetakse serverisse ka need. Seejärel üritab server kompileeritud korraldust täita. Kui see õnnestub tagastatakse kliendile veateade 0 - kõik õnnestus. Kui korralduse täitmine ei õnnestunud, tagastatakse viga. Viga võib tekkida näiteks serveriga ühenduse kadumise tõttu või siis seetõttu, et serverisse saadetud andmete formaadid või

tüübid olid valed. Näiteks üritati filtris võrrelda kuupäeva tekstiga, arvu kuupäevaga vms.

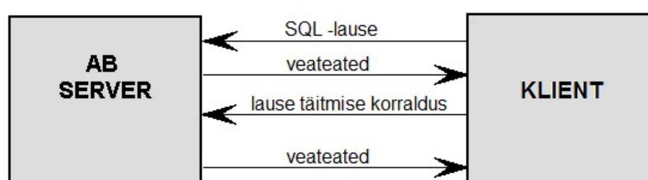
Kui korralduse täitmine õnnestus, kuid WHERE-filtrile vastavaid ridu ei leita, siis rohkem midagi kliendile ei tagastata. Kui baasist leiti WHERE-filtrile vastavaid ridu, siis tagastatakse ka need read. Kui filtrile vastavaid ridu leiti palju, võib neid teateid, millega andmeid tagastatakse, järgneda üksteisele palju:



määratluse-lause täitmise skeem

Määratlused on kõikvõimalikud korraldused, mis saadetakse andmebaasisüsteemile. Nende lausete hulka kuuluvad andmebaasi struktuuri loomise ja muutmise korraldused (CREATE TABLE, ALTER TABLE, DROP TABLE), kasutaja õiguste haldamise korraldused (CREARE USER, GRANT, REVOKE) ja kõik andmebaasi olekute/parameetrite sättimise korraldused.

Määratluse-lause kliendist serverisse saatmisel analüüsitakse ja kompileeritakse serverisse saadetud lause. Selle õnnestumisel tagastatakse korralduse saatnud kliendile tagasi veateade - kas 0 (õnnestus) või nullist erinev so. tekkis/avastati mingi viga. Kui tekkis viga, siis korralduse täitmine siinkohal katkeb . Kui lause analüüs ja kompileerimine oli edukas saadab kliendi arvutis olev protseduur nüüd serverile teate korraldusega täita kompileeritud lause. Seejärel üritab server kompileeritud korraldust täita. Kui see õnnestub tagastatakse kliendile veateade 0 - kõik õnnestus. Kui korralduse täitmine ei õnnestunud, tagastatakse viga. Viga võib tekkida näiteks serveriga ühenduse kadumise tõttu või siis seetõttu, et selline määratlus oli baasis juba varasemast kehtestatud vms.



11.4. Andmebaasi käsitluse optimeerimine

Nagu infosüsteemi erinevate mudelite skeemidelt selgub on andmebaasi serverid kriitiliseks ressursiks – olenemata protsesside iseloomust koondub enamike teiste süsteemikomponentide "huvi" andmebaasiserverites oleva informatsiooni muutmisele või selle pärimisele. Kuna "neid teisi komponente" on andmebaasiserveritega võrreldes palju, siis on äärmiselt oluline kasutada andmebaasiserverite ressurssi "väga kokkuhoidlikult".

andmebaasi protsessori aeg on infosüsteemi kõige kriitilisem ressurss

Andmebaasi serverite ressursi "väga kokkuhoidlik" kasutamine tähendab seda, et andmebaasiserverid ei tohi teha mitte midagi liigset ja ka seda "väga vajalikku", mida nad vältimatult tegema peavad, peavad nad tegema võimalikult väikese ressursi kuluga.

andmebaasi serveri ressurssi tuleb kasutada väga kokkuhoidlikult

1. Andmebaasi serveri poole tohib saata ainult korrektseid korraldusi – vigaste korralduste analüüsimine raiskab lihtsalt serveri aega. Seda aega, mida oleks võinud kasutada millegi asjalikuma tegemiseks.

korrektsed, silutud korraldused

2. Andmebaasi serveri poole tohib saata ainult vajalikke korraldusi – "igaks juhuks" saadetud korraldused raiskavad lihtsalt serveri aega. Sellega on mõeldud seda, et kui midagi on andmebaasiserveri käest juba küsitud, siis tuleb see programmis "meelde jätta", mitte küsida vajadusel uuesti. Mõtleme näiteks olukorrast, kus meil tehakse mingit päringut mingis protseduuris ühe korra asemel kolm korda. Kui meil on seda protseduuri täidetud 10 000 korda oleme teinud 20 000 asjatut päringut 10 000 asjaliku kohta. so. oleme 2/3 selle lause täitmise ajast raisanud asjata. Eriti oluline on seda meeles pidada internetis paljude kasutajate jaoks avatud infosüsteemide loomisel. 10 000 võib siin olla väga väike arv.

vainult vajalikud korraldused

3. Andmebaasi poole saadetud korraldus peab olema kirjutatud selliselt, et ta kulutaks minimaalselt serveri ressurssi. Siin loevad ainult kogemused ja koolitus. Sama asja on andmebaasis võimalik teha väga erinevaid SQL-konstruktsioone kasutades. Selleks et nendest välja valida need, mis võimalikult vähe protsessori aega kulutavad on vaja oskusi. Seda annavad haridus ja kogemused.

oskused ja kogemus

Esimest kahte kriteeriumit on suhteliselt lihtne järgida kasutades tunnetuslike ja kogemuslikke meetodeid s.t. programmid peavad olema silutud (1) ja protsessi

juhtimise algoritmid peavad olema optimaalselt projekteeritud/kirjutatud (2). Andmebaasi poole saadetavate päringute optimeerimisel on aga vaja tunda sügavamalt konkreetse andmebaasi käsitluskeele ja optimaiseri loogikat.

Järgnevalt vaatame SQL-keele optimeerimise “rusika-reegleid”, mis toimivad enamuses olenemata konkreetsest andmebaasi süsteemist. Vaatame neid eraldi INSERT, UPDATE, DELETE ja SELECT-lausete kontekstist lähtuvalt. Siiski on iga konkreetse andmebaasisüsteemi puhul omad "riukad" ja neid saab teada vaid "vanematelt kolleegidelt". Ise nende leiutamine võib liialt palju piinav olla.

11.4.1. INSERT-korralduse optimeerimine

INSERT on üldjuhul lihtne käsk, mis lisab andmebaasi tabelisse kas ühe või mitu rida. Enamikel juhtudel on tegemist “täiesti uute” andmetega, mis tulevad kusagilt “andmebaasi välisest keskkonnast”. Sellisel juhul on ainsaks optimeerimise reegliks see, et ühe INSERT-lausega tuleb kirjutada andmebaasi nii palju ridu andmeid, kui see on võimalik – soovituslikult kõik, mis meile antud hetkel on teada.

INSERT on lihtne käsk - optimeerimiseks palju ruumi pole, aga siiski

See tähendab seda, et andmebaasi poole tuleb saata teele INSERT- lause, see kompileerida ja seejärel saata andmebaasi kõik teada olevad andmed korraga lisamiseks me hoiame kokku korralduse ja veateadete edasi-tagasi saatmise aega .Mitme rea korraga baasi lisamine koormab serverit oluliselt vähem, kui sama arvu ridade lisamine eraldi INSERT-korraldustega.

kui võimalik lisa mitu rida korraga

Kui andmebaasi mootor seda võimaldab, võib kompileerida INSERT-lauset üks kord ja siis kasutada seda sama kompilatsiooni eraldi kõigi ridade ükshaaval baasi lisamiseks. See on küll aeglasem aga ikkagi veel piisavalt kiire võrreldes sellega, kui iga rea baasi kirjutamisel kompileerida INSERT-lause. Kõige suurem lollus, mis sellisel juhul tehakse, kui baasi on vaja lisada mitu korraga teada olevat kirjet, on see, et iga rida kirjutatakse baasi eraldi INSERT-korraldusega, mis iga rea lisamisel ka kompileeritakse.

kompileeri üks kord täida mitu korda

INSERT-korraldusega võib mingisse tabelisse kirjutada ka selliseid andmeid, mis andmebaasis on juba olemas (mingis teises tabelis). Siis ei tohi mingil juhul teha seda, et lugeda need andmed kõigepealt SELECT-lausega programmi ja seejärel asuda neid kas ühe või mitme kaupa baasi tagasi kirjutama. Sellisel juhul koormame me ilma asjata sidekanaleid. Õigem variant on kasutada

INSERT-SELECT - kopeerimine baasi sees

INSERT-korraldus, kus VALUES grupi kohapeal seisab SELECT-lause. Sellisel juhul toimub “filtri järgi kopeerimine” andmebaasi sees, ilma sidekanaleid koormamata (vt. INSERT-SELECT korraldus).

11.4.2. DELETE-korralduse optimeerimine

DELETE on ühest tabelist WHERE-filtri järgi andmete (kirjete) kustutamise korraldus. Siin ei ole olemas mingeid iseseisvaid optimeerimise reegleid. WHERE-tingimuse kirjeldamisel tuleb arvestada neid optimeerimisreegleid, mida SELECT-lause filtri koostamiselgi. On nad ju ühe asja kaks poolt – üks kustutab filtri järgi andmeid, teine väljastab neid andmebaasist filtri alusel. Kui küsimuses on andmete kustutamine tingimuse järgi, siis see filter määrab üpris jäigalt lause struktuuri ja mingid "vigurdamised" siin eriti ei aita

DELETE-lausel mingeid erilisi optimeerimisreeglei d pole

Teadma peab ainult seda, et kustutamise kiirendamiseks tuleb luua indeks, mis toetab kustutamise filtrit – ilma selleta võib suurest tabelist filtrile vastavate ridade otsimine väga kaua aega võtta.

indeksid kiirendavad kustutamist

11.4.3. UPDATE-korralduse optimeerimine

UPDATE on ühest tabelis WHERE-filtri järgi andmete (kirje või grupi kirjete) veergude väärtuste muutmise korraldus. Siin ei ole olemas mingeid iseseisvaid optimeerimise reegleid. WHERE-tingimuse kirjeldamisel tuleb arvestada neid optimeerimisreegleid, mida SELECT-lause filtri koostamiselgi. On nad ju ühe asja kaks poolt – üks uuendab andmeid filtri järgi , teine väljastab neid andmebaasist filtri alusel. Kui küsimuses on andmete uuendamine tingimuse järgi, siis see filter määrab üpris jäigalt lause struktuuri ja mingid "vigurdamised" siin eriti ei aita

UPDATE-lausel mingeid eriti optimeerimisreeglei d pole

Teadma peab ainult seda, et uuendamise kiirendamiseks tuleb luua indeks, mis toetab uuendamise filtrit – ilma selleta võib suurest tabelist filtrile vastavate ridade otsimine väga kaua aega võtta.

indeksid kiirendavad uuendamist

Nagu näete üsna on andmete uuendamise optimeerimine üsna sarnane andmete kustutamise optimeerimisega - tekstis tuli ainult mõned sõnad välja vahetada

UPDATE ja DELETE-lauset optimeerimine toimub samade reeglite alusel

11.4.4. SELECT-korralduse optimeerimine

SELECT- korralduse optimeerimisel on palju olulisi nüansse. Kõik need pole korraga kasutatavad, kuid "nippide" loend on oluliselt pikem kui eelmistel korraldustel. Kõigisse neisse tuleb suhtuda väga tõsiselt. Samas ongi enamik andmebaasi poole lähetatavaid korraldusi SELECT-laused.

SELECT-lausega annab kõvasti "nipitada"

1. SELECT-korralduse kompileerimise aeg sõltub WHERE-tingimuses seotavate tabelite vaheliste seoste tingimuste keerukusest. Siin saab palju ära teha andmemudeli projekteerimisel. Kasutada tuleks surrogaatvõtmeid (ID-d) - see tagab kõige lihtsamad kirjeldatud seosed tabelite vahel.

tingimuste keerukus ja surrogaatvõti

2. WHERE-tingimuse keerukus on otseses seoses tabelite/viewde arvust SELECT-lause FROM-komponendis – mida rohkem on tabeleid, seda rohkem on seoseid, seda kauem töötab optimaiser, seda keerulisem on koostatav päringu algoritm ja seda kauem võtab aega päringu enda täitmine. Siin võib olla abiks SELECT-lause ümber struktureerimine nii, et osa FROM-komponendis olevaid tabeleid õnnestub viia üle alampäringutesse. Üldjuhul aitab see SELECT-lause täitmise kiiremaks muuta.

WHERE-tingimuse keerukus ja alampäringud

3. Optimaiseri töö võtab seda rohkem aega, mida rohkem on SELECT-lausega seotud tabelitele koostatud indekseid. Seda annab mõnedes keeltes vältida andes optimaiserile (kompilaatorile) vihjeid, milliseid indekseid kasutada. Vastasel korral vaadatakse läbi kõik indeksid kõikides tabelites kõikides omavahelistes kombinatsioonides. Seega indeksid (nende liigne hulk) alati ei kiirenda tööd vaid vastupidi -võivad aeglustada seda.

indekseid ei tohi olla liiga palju

4. Vahel võib optimaiseri töö aeg ületada kümneid ja sadu kordi lause täitmise aja. Sellisel juhul annab parema tulemuse keeruka SELECT-lause osadeks jagamine ja nende osade käivitamine programmist üks haaval.

SELECT-lause tükkideks jagamine

5. Kui SELECT-lauses on FROM-grupis väga palju selliseid tabeleid, kust andmeid ei väljastata vaid neid kasutatakse ainult piirangute esitamiseks, siis on tavaliselt otstarbekas võtta need sealt välja ja kirjutada piirangute tingimused IN, NOT IN, EXISTS ja NOT EXISTS konstruktsioonidena kasutades alampäringuid.

FROM-grupi lühendamine

6. WHERE-tingimuses on alati soovitav kirjutada tabelite filtri-tingimused enne ja seose (JOIN-) tingimused alles pärast seda

enne filtri- siis seose-tingimused

7. FROM-grupis on tabelid mõtet kirjutada sellises järjestuses, nagu teie arvate et neist peaks pärima ja neid omavahel seostama. See võib juhtida optimaiseri lihtsamini õigele teele. Lihtne "talupoja mõistus" võib kogemuslikult "ära arvata" selle, mida arvuti peab välja arvutama.

FROM-grupi tabelite järjestus

8. Kui võimalik vältige OUTER-joini ja asendage see UNION-konstruktsioonidega, kus OUTER-joini erinevad võimalused on kirjutatud erinevatesse SELECT-lausetesse. Siiski on mõtet võrrelda kumb konstruktsioon töötab kiiremini. UNION konstruktsioonidest on kõige kiirem UNION ALL. Kasutage seda kui loogika lubab

ära kasuta OUTER-joini

9. Suurte lausete (tingimuste) korral katsetage erinevaid variante ja võrrelge neid stopperiga - milline töötab kiiremini.

võrdle erinevaid lauseid

10. Indekseerige kõik seosed

seosed tuleb indekseerida

11. Teadke, et päring, mis töötab täna kiiresti, võib seda enam mitte teha, kui andmebaasi maht on drastiliselt muutunud. Siis võib töötada kiiremini hoopis mingi teine lause, mille te omal ajal kõrvale heitsite. Seepärast on soovitav talletada süstemaatiliselt kõik katsetamise käigus osalenud laused - kunagi võib neid vaja minna. Siis kui peab hakkama uuesti katsetama.

    Kuidas saada teada, et lause on hakanud aeglaselt töötama. See on administraatorite ja süsteemi töö monitooringu ülesanne - leida need süsteemi osad, mille töö on aja jooksul oluliselt aeglasemaks muutunud.

Päringulausete täitmise kiirus muutub andmebaasi mahu suurenedes