Kyselyjen käsittely ja optimointi

- Kyselyn käsittelyn vaiheet:
  - TKHJ ottaa vastaan kyselyn asiakasohjelmalta
  - Kysely selataan ja jäsennetään
    - tarkistetaan kyselyn rakenteellinen oikeellisuus
  - Jäsennetty kysely muunnetaan relaatiolausekkeiksi ja tallennetaan sisäisessä esitysmuodossa kyselypuuna
  - Laaditaan kyselyn toteutussuunnitelma
    - suunnitelman laadintaan liittyy valintaa eri toteutusvaihtoehtojen välillä 'optimointia'
  - Suoritetaan kysely suunnitelman mukaisesti
  - Toimitetaan tulokset asiakasohjelmalle

Kyselyjen käsittely ja optimointi

- Miksi optimoidaan
  - SQL-kysely määrittää halutun tuloksen, ei sitä miten tulos muodostetaan (deklaratiivinen kyselykieli)
    - useimmissa oliokannoissa kysely etenee navigoimalla proseduraalisesti – eli ohjelmoija kontrolloi kyselyn toteutustapaa, silloin ohjelmoijan pitää osata valita oikea tapa
  - Valitut tiedostorakenteet vaikuttavat siihen, miten operaatio kannattaa suorittaa
  - Taulujen koko ja arvojen jakautumat muuttuvat jatkuvasti, tämä saattaa vaikuttaa siihen, miten kysely kannattaa toteuttaa

Kyselyjen käsittely ja optimointi

- Kyselyn optimointi ei ole todellista matemaattista optimointia
  - tarkoitus ei ole löytää parasta vaan riittävän hyvä
  - parhaan etsiminen on laskennallisesti raskasta
  - taustatietoja ei yleensä ole riittävästi parhaan valintaan
- Myös SQL:ssä käyttäjä voi jonkin verran vaikuttaa kyselyn suoritustapaan
  - monissa SQL-murteissa käyttäjä voi liittää kyselyyn vihjeitä siitä, miten kysely pitäisi käsitellä
  - kyselyn esittämistapa voi myös vaikuttaa,
    - esim. jos taululla on indeksi sarakkeen Nimi perusteella Nimi like 'Vir%' käyttää indeksiä, mutta upper(Nimi) like 'VIR%' ei käytä

Kyselyjen käsittely ja optimointi

- Kyselyn optimoinnin periaatteet
  - Heuristinen optimointi (sääntöpohjainen optimointi)
    - käytössä on joukko sääntöjä, joiden mukaisesti operaatiot järjestetään tai suoritustavat valitaan
      - *'valinnat ennen liitoksia'*
      - *'käytä indeksiä aina kun mahdollista'*
      - *jne*
    - säännöt ovat yleiskäyttöisiä
    - suoritustapa ei välttämättä ole optimaalinen

Kyselyjen käsittely ja optimointi

- Kyselyn optimoinnin periaatteet
  - Kustannuslaskentaoptimointi (tilastollinen optimointi)
    - laaditaan vaihtoehtoisia suunnitelmia, esim. sääntöpohjalta
    - lasketaan vaihtoehdoille kustannukset
    - valitaan edullisin
    - edellyttää tilastoaineistoa ja sen uudistamista ajoittain

Kyselyjen käsittely ja optimointi

- SQL-kyselyn voi ajatella muodostuvat joukosta kyselylohkoja (query block) (select –from –where lohko)
  - yksi kyselylohko muodostaa kokonaisuuden, jonka suoritus voidaan suunnitella erikseen

```
select lname, fname
   from employee
   where salary >=
     (select max(salary)
      from employee
      where dno=5);
```

kaksi kyselylohkoa

Kyselyjen käsittely ja optimointi

- Kyselylohkot muunnetaan relaatioalgebran lausekkeiksi
- Lohko 1: $\pi_{lname,fname}(\sigma_{salary>c}(employee))$,
- Lohko 2: $F_{max\ salary}(\sigma_{dno=5}(employee))$

- *F* on laajennetun relaatioalgebran funktio-operaatio, tarvitaan esim. yhteenvetofunktioissa, ryhmittelyssä jne.

Kyselyjen käsittely ja optimointi

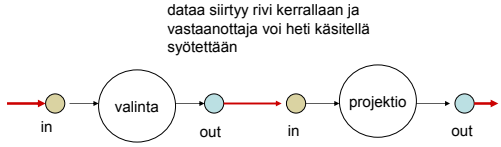
- Relaatioalgebran operaatioiden suoritukseen on useita vaihtoehtoja
- Vaihtoehtojen tehokkuus vaihtelee
- Tehokkuuden ensisijaisena kriteerinä tarvittavien levyoperaatioiden lukumäärä
  - luettaessa relaatioiden osia levyltä
  - kirjoitettaessa operaation tulosta levylle
    - tulos on joko välitulos tai lopputulos

Kyselyjen käsittely ja optimointi

- Välitulosten kirjoittamisen (ja edelleen lukemisen) kustannus voidaan usein jättää huomioimatta, jos tkhj käyttää putkitekniikkaa: (pipelining) operaation tulosjono ohjataan toisen syöttöjonoksi, jolloin levylle kirjoitusta ei tapahdu

Kyselyjen käsittely ja optimointi

- Operaation tulosjono ohjataan toisen syöttöjonoksi, jolloin levylle kirjoitusta ei tapahdu



Tiedoston järjestäminen

- Monet tietokantaoperaatiot (muutkin kuin order by) edellyttävät tietueiden järjestämistä
  - group by,
  - relaatiomallin mukainen projektio (select distinct)
  - yhdiste
  - erotus
  - järjestäminen voi luovuttaa ensimmäisen tulosrivin eteenpäin vastaa vastaanotettuaan viimeisen
- Jos käsiteltävät rivijoukot ovat isoja tarvitaan ulkoista järjestämistä (external sorting)

Tiedoston järjestäminen

- Ulkoisessa järjestämisessä käytetään yleisesti ns. lomitusjärjestämistä (merge sort):
  - Tiedosto luetaan osina keskusmuistiin (osien koon määrää järjestämiseen käytettävissä olevan keskusmuistin määrä)
  - Järjestetään kukin osatiedosto keskusmuistissa (esim. quicksort)
  - Kirjoitetaan järjestetty osatiedosto (run, jono) levylle.
  - Tehdään lomituskierroksia tarpeellinen määrä, jotta jäljellä on vain yksi järjestetty osatiedosto

Tiedoston järjestäminen

- Lomituskierros:
  - Luetaan tietueita m ($m \geq 2$) järjestetystä jonosta (osatiedostosta) - lukua m kutsutaan lomitusasteeksi
  - Lomitetaan tiedostot eli kirjoitetaan yhtä järjestettyä tulostiedostoa
    - kirjoitetaan avaimeltaan pienin jonojen vuorossa olevista tietueista tulosjonoon ja edetään tämän jonossa
  - Järjestettyjen osatiedostojen pituudet kasvavat kierroksella m-kertaiseksi.
  - Jos jonoja (osatiedostoja) oli alunperin n, tarvitaan lomituskierroksia roof($\log_m n$)

Tiedoston järjestäminen

- Oletetaan, että järjestämiseen on käytettävissä puskuritilaa 129*4KB (vähän yli 0.5MB). Tiedoston koko olkoon 2048 sivua= 8MB
  - Alkuperäiseen järjestämiseen voidaan käyttää 128 syöttöpuskuria ja 1 tulospuskuri, joten saadaan 128 sivun pituisia järjestettyjä jonoja 16 kappaletta.
  - Näiden järjestäminen hoituu 1 lomituskierroksella - 16 asteinen lomitus.
- Yhdellä lomituskierroksella selvittäisiin tällä puskurimäärällä 128*128*4KB (64MB) kokoisesta tiedostosta

Tiedoston järjestäminen

- Kullakin kierroksella kaikki sivut luetaan ja kirjoitetaan.
- Olkoon tiedoston koko N sivua ja muistitilaa käytettävissä B sivua. Levyoperaatioita tarvitaan tällöin
  $2N+2N*\text{roof}(\log_{B-1} \text{roof}(N/(B-1)))$
  - levyoperaatiot peräkkäiskäsittelyä
  - lomituksesta on olemassa erilaisia muunnelmia esim. muistitilan käytön suhteen,
    - *esim. kaksoispuskuroinnilla voidaan vähentää jonotusaikoja (toisiin puskureihin luetaan kun toisia käsitellään), mutta B puolittuu ja levyoperaatioiden määrä kasvaa vaikka kokonaisaika väheneekin*

Valintaoperaation toteutus

- Valinta:
  - select * from R where A op value
  - $\sigma_{\text{A op value}}$ (R)
- Valinnan vaihtoehdot riippuvat
  - valintaehdosta ja
  - käytössä olevista tiedostorakenteista

Valintaoperaation toteutus

- Tiedoston läpikäynti (scan)
  - soveltuu aina
  - käytettävä jos
    - sarake A ei ole mukana missään indeksissä
    - eikä tiedosto ole sarakkeen A:n perusteella järjestetty
    - sarake A on jonkin funktion argumenttina, esim. upper(A), myös substring(A,1,x) ?
    - voi hyödyntää peräkkäiskäsittelyä

Valintaoperaation toteutus

- Binäärihaku (binary search)
  - soveltuu, jos tiedosto on järjestetty sarakkeen A perusteella
  - etsitään ensimmäinen ehdon täyttävä, josta voidaan jatkaa eteenpäin
  - voi hyödyntää peräkkäishakua vasta ensimmäisen ehdon täyttävän löytymisen jälkeen

Valintaoperaation toteutus

- Indeksihaku (index scan)
  - soveltuu, jos käytössä on hakusarakkeeseen perustuva indeksi
    - pelkästään hakusarake
    - hakusarakealkuinen indeksointiperusta
    - hakusarake mukana indeksissä
      - *tämäkin saattaa olla käyttökelpoinen*
      - *edellyttää koko hakemiston läpikäyntiä, mutta toimii jos hakemisto on pieni suhteessa tiedostoon ja hakuehto on hyvin rajaava (selective) = jäljittää pienen tietuejoukon,*

Valintaoperaation toteutus

- Valintaehto muodostuu 'ja' ($\wedge$) ja 'tai' ($\vee$) operaatioilla yhdistetyistä alkeisehdoista.
- Ehdon konjunktiivinen normaalimuoto (conjunctive normal form) tarkoittaa, että ehto esitetään tai operaatioilla yhdistettyinä konjunktioina:

  $(e_1 \wedge e_2 \wedge ... \wedge e_n) \vee (e_m \wedge ... \wedge e_p) \vee ...$

Valintaoperaation toteutus

- Konjunktiota $(e_1 \wedge e_2 \wedge ... \wedge e_n)$ evaluoitaessa ei riitä, että jokin ehdoista täyttyy vaan kaikkien pitää täyttyä. Jos joillakin ehdoissa olevista sarakkeista on indeksit näitä pyritään käyttämään:
  - valitsemalla eniten rajaava indeksi, ja tutkimalla sen jälkeen tietueista muut ehdot, tai
  - muodostamalla hakemistojen leikkaus ja tutkimalla leikkausjoukon tietueista ne ehdot, joita ei voida ratkaista indeksien perusteella
    - tämä ratkaisu edellyttää tietueosoitteita
    - leikkausjoukon muodostamiseksi jäljitetyt osoitelistat järjestetään osoitteen perusteella ja lomitetaan
      - *Huom: tietueosoitteessa alussa sivuosoite.*

Valintaoperaation toteutus

- Olkoon taululle työntekijä määritelty indeksi sekä sukunimen että kotiosoitteen perusteella. Tällöin

```
select * from työntekijä
    where sukunimi='Löpönen' and
      kotiosoite='Teollisuuskatu 23 B446, Helsinki'
```

- voitaisiin toteuttaa
  - valitsemalla rajaavampi indeksi (kotiosoite), ja tutkimalla tietueet – keskimäärin vähän yli 1 (olisi tässä parempi vaihtoehto koska osoite on hyvin rajaava), tai
  - hakemalla kummankin indeksin perusteella (keskimäärin 30 ja 1+) ja muodostamalla näiden leikkaus. On tässä tapauksessa huonompi, mutta jos kumpikaan indeksi ei ole kovin tiukasti rajaava (vaikkapa kurssikoodi ja opiskelijanumero ilmoittautumistiedoissa), niin niiden leikkaus voi olla silti hyvin pieni)

Valintaoperaation toteutus

- Disjunktiivisen ehdon $e_1 \vee ... \vee e_n$ tulos vastaa erillisillä ehdoilla e_1 … e_n saatujen rivijoukkojen yhdistettä.
- Koko tiedosto on käytävä läpi, jos yksikin ehdoista perustuu sarakkeeseen, jolle ei ole indeksiä

Projektion toteutus

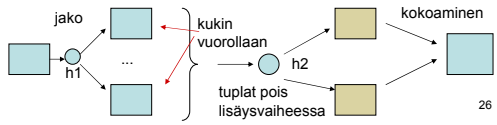
- Projektiolla on kaksi tehtävää
  - sarakkeiden poiminta ja
  - toistuvien vastausrivien karsinta
    - SQL-kyselyissä toistuvia rivejä ei välttämättä haluta karsia (kyselyssä ei DISTINCT-määrettä)
  - Sarakkeiden poiminta on suoraviivaista
    - otetaan tietueesta vain ne sarakkeet, jotka on lueteltu projisoitaviksi
    - jos kaikki luetellut sarakkeet löytyvät hakemistosta ei varsinaisia datarivejä tarvitse lukea lainkaan.

Projektion toteutus

- Toistuvien vastausrivien (duplikaattien) karsinta on hieman hankalampaa.
  - Tietueet voidaan järjestää käyttäen järjestysavaimena koko tietuetta (jos tulokselle on määritelty järjestys käytetään järjestysperustaa avaimen alussa). Näin duplikaatit saadaan peräkkäin ja ne voidaan helposti karsia yhdellä läpikäynnillä (tai lomituksen viimeisellä kierroksella)
    - jos vastaus on pieni, riittää keskusmuisti-järjestäminen, mutta ensimmäinen tulosrivi voidaan toimittaa eteenpäin vasta kun viimeinen syöttörivi on saatu.
    - isot vastaukset edellyttävät ulkoista järjestämistä.

Projektion toteutus

- Toistuvien rivien karsinta
  - Tietueet voidaan myös hajauttaa koko tietueesta lasketun hajautusavaimen perusteella. Tällöin duplikaatit sijoittuvat samaan soluun ja ne voidaan lisäysvaiheessa karsia. Jos osoiteavaruus on riittävän iso, tulee samaan soluun vähän tietueita ja vähän tutkittavaa.
  - Tyypillisesti käytetään kahta peräkkäistä hajautusta



Projektion toteutus

- Toistuvien rivien karsinta
  - Jos projektio sisältää taulun avaimen ei tulosrivejä tarvitse lajitella tai hajauttaa, tiedetään että toistuvia rivejä ei ole.

Liitoksen toteutus

- Liitos on tärkeimpiä ja yleisimpiä relaatioalgebran operaatioita
- Yleisin liitos on taulun pääavaimen ja siihen viittaavan viiteavaimen yhtäsuuruuteen perustuva liitos.
- Seuraavassa käytetään esimerkkinä taulujen opiskelija (500 sivua, kussakin 20 tietuetta) ja suoritus (2000 sivua, kussakin 50 tietuetta) liitosta opiskelijanumeron perusteella.

Liitoksen toteutus

- Liitos voidaan määritellä ristitulon ja valinnan yhdistelmäksi. Yksinkertaista olisi muodostaa ristitulo ja tehdä sen jälkeen valinta. Ratkaisu on kuitenkin tehoton.
- Liitoksen toteutukseen on tarjolla useita tekniikoita. Päätyypit:
  - sisäkkäiset silmukat (nested loop join)
  - hakemistoliitos (index nested loop join)
  - lomitusliitos (sort merge join)
  - hajautusliitos(hash join)

Liitoksen toteutus

- Tarkastellaan taulujen R ja S liitosta ehdolla R.x=S.y
- Sisäkkäisten silmukoiden tekniikassa perusidea on seuraava:

```
foreach tuple r in R do
  foreach tuple s in S do
    if r.x==sy then add  {r,s} to result
```

- **eli käydään läpi ulompi taulu ja jokaisen rivin kohdalla käydään läpi sisempi taulu**

Liitoksen toteutus

- Olkoon R:n sivumäärä pages(R) ja S:n sivumäärä pages(S) ja vastaavasti sivukoot tietueina psize(R) ja psize(S). Tällöin edellä kuvatun yksinkertaisen algoritmin kustannus levyhakuina olisi
  - op = pages(R)+psize(R)*pages(R)*pages(S)+ tuloksen kirjoitus
- Esim:
  - jos R= opiskelija ja S= suoritus, niin
    - operaatioita = 500+500*20*2000 =20 000 500 + tulos.
  - jos R= suoritus ja S= opiskelija, niin
    - operaatioita= 2000+2000*50*500 = 50 002 000 + tulos
- Hidasta

Liitoksen toteutus

- Parannetaan algoritmia hieman ja käsitellään koko sivu kerralla
  - Siis jokaista R:n sivua kohti käydään S kertaalleen läpi ja verrataan kaikkia R:n sivulla olevia tietueita vuorossa olevan S:n sivun tietueisiin.
  - operaatioita pages(R)+pages(R)*pages(S)+tuloksen kirjoitus
  - R= opiskelija, S= suoritus: 500+500*2000=1 000 500
  - R= suoritus, S= opiskelija: 2000+2000*500 = 1 002 000
  - satunnaishakuajalla 10ms laskettuna tämä veisi 10 005 s
- Mitä enemmän R:stä käsitellään kerralla sitä vähemmän kierroksia tarvitaan, niinpä parannetaan vielä

Liitoksen toteutus

- Jaksottaiset sisäkkäiset silmukat (Block nested loops)
  - versio joka yleensä tarjolla tkhj:ssä
  - Oletetaan, että liitoksen suorittamiseen on käytettävissä B syöttöpuskuria. Varataan näistä B-1 ulommalle taululle R ja 1 sisemmälle taululle S.
  - Käydään yhdellä vertailukierroksella läpi kaikki ulomman taulun puskurit ja verrataan tietueita sisemmän taulun vuorossa olevan sivun tietueisiin.
  - Sisempi taulu S joudutaan käymään läpi roof(pages(R)/(B-1)) kertaa
  - Vaikka levyoperaatioiden määrä on tässä mallissa pienin, paras käsittelyteho saavutetaan, jos puskurikapasiteetti jaetaan tasan kummankin liitostiedoston kesken – levyhakuja tulee enemmän, mutta voidaan hyödyntää peräkkäisyyttä

Liitoksen toteutus

- Esimerkki:
  - Varataan liitoksen toteutukseen puskuritilaa 101 syöttöpuskurille (404KB)
  - R= opiskelija, S= suoritus
    - operaatioita= 500 + (500/100) *2000 + tulos =10500+tulos
    - satunnaishakuajalla 10 ms laskettuna tämä olisi 105 s
      - *R voidaan kuitenkin lukea 100 sivun pätkinä, jolloin aikaa oikeastaan kuluisi 5*(satunnaishakuaika + 100 sivun siirto)+5*2000*(satunnaishakuaika+1 sivun siirto)=*
      - *5*(10ms +100*0.2 ms)+10000*(10 ms+0.2 ms)= 150 ms+102000ms= noin 102 s*
  - R=suoritus, S=opiskelija
    - operaatioita=2000 + (2000/100)*500= 12000

Liitoksen toteutus

- Jos jaettaisiin puskuritila puoliksi R:lle ja S:lle ja R = opiskelija ja S= suoritus niin
- hakuaika:

  10*(10ms+ 50 *0.2 ms)+
  10*2000/50*(10 ms+50 *0.2 ms)=
  10*11ms+10*40*(11ms)= 110 ms + 4400ms =4.51 s

  odotusajat lisäävät kokonaisaikaa

Liitoksen toteutus

- Hakemistoliitos (index nested loop)
  - Jos toisella taululla on hakemisto liitossarakkeeseen perustuen tehdään tästä sisempi taulu

  foreach tuple r in R do
  foreach tuple s in index_search(S,r.x) do
  add {r,s} to result

  - Jos hakemistona on rakenteeltaan B+ -puu rakenteinen oheishakemisto, tarvitaan indeksin kautta hakuun tyypillisesti 1-4 levyhakua isollakin tiedostolla.

Liitoksen toteutus

- Hakujen kokonaismäärä:
  pages(R)+pages(R)*psize(R)*(height(S)+matches(S))
- height(S):
  - levyhakujen määrä hakemiston kautta haettaessa, mukaan mahdolliset ylivuotoketjut ja oletukset siitä, mitä säilyy keskusmuistissa
- matches(S):
  - montako keskimäärin löytyy yhtä ulompaa kohti. Jos kyseessä on B+ -puuna toteutettu oheishakemisto, on jokaista tietuetta kohti oma hakemistotietue (jos tietueosoite on mukana avaimessa nämä ovat osoitejärjestyksessä), yksi haku / osuma riittänee

Liitoksen toteutus

- Esimerkki:
  - R= opiskelija S=suoritus
    - levyhakuja 500+500*20*(2+ 10) = 120 500
    - oletetaan tehollinen hakukorkeus 2, opiskelijalla 10 suoritusta
    - kaikki S:n haut satunnaishakuja -> noin 1200 s
  - R=suoritus, S=opiskelija
    - levyhakuja 2000+2000*50*(2+1)= 302 000
  - Huomaamme, että kokonaan läpikäytäväksi (ulommaksi) kannattaa valita pienempi tauluista

Liitoksen toteutus

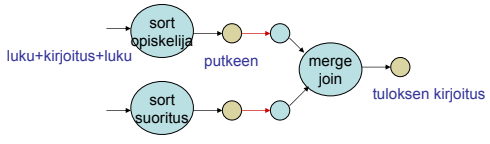
- Lomitusliitos (sort merge join)
  - soveltuu yhtäsuuruusliitokseen
  - Lomitusliitoksessa taulut järjestetään liitossarakkeen perusteella
  - Kun rivit ovat järjestyksessä on liitos muodostettavissa yhdellä läpikäynnillä
    - Rivit ovat järjestyksessä esim., jos tiedoston toteutusrakenteena on ISAM tai B+ -puu
    - Jos tiedostolla on B+ -puurakenteinen oheishakemisto, tietueet saadaan myös järjestyksessä (tosin hitaammin)

Liitoksen toteutus

- Elleivät rivit ole järjestyksessä, ne pitää ensin järjestää.
- Järjestämisen levyoperaatioiden määrähän oli
  $2N+2N*roof(\log_{B-1} roof(N/(B-1)))$
- tuloksen muodostus hoituu yhdellä läpikäynnillä.

Liitoksen toteutus

- Esim.
  - Olkoon B=101, Kumpikin esimerkkitaulu vaatii jonojen muodostuksen ja yhden lomituskierroksen sekä yhdistämisen liitoksen aikaansaamiseksi,
  - Oletetaan että lomituksen tulos putkitetaan liitokseen, tällöin operaatioita on 3* sivulukumäärä kummallakin



Liitoksen toteutus

- eli esimerkkitapauksessa levyoperaatioita
  - 3*500+3*2000+tulos = 7500 + tulos
  - jonojen muodostus sujuu peräkkäiskäsittelynä
- lomituksessa voidaan joutua odottamaan,
- Oletetaan, että jonoja luetaan lomituksessa10 ja 5 sivun jaksoissa (käytetään kaikki puskurit)
  opiskelija jonojen teko: 2*5*(10ms+100*0.2ms)+
  suoritus jonojen teko: 2*20*(10ms+100*0.2ms) +
  opiskelija jonojen luku: 500/10*(10ms+10*0.2 ms) +
  suoritus jonojen luku: 2000/5*(10ms+5*02.ms)=
  300ms+1200ms + 1500 ms + 8000 ms = 11s