CS-A1150 Tietokannat

15.4.2019

Oppimistavoitteet: tämän luennon jälkeen

- Tiedät, mitä tarkoitetaan hakemistolla ja mitä hyötyä hakemistosta on.
- Tiedät, miten voidaan arvioida, mitä hakemistoja tietokantaan kannattaa määritellä.
- Tiedät, miten ohjelmoija voi suojella tietokantaa erilaisilta häiriötilanteilta *transaktioiden* avulla.
- Osaat kertoa, mitä ominaisuuksia tietokannan hallintajärjestelmä takaa transaktioiden toteuttavan.
- Tiedät, millä eri tavoin transaktioilta vaadittavista ominaisuuksista voidaan joskus tinkiä tehokkuuden lisäämiseksi.

Esimerkkitietokanta

- Useimmat tämän luennon esimerkit käsittelevät aikaisempien luentojen esimerkkitietokantaa, joka koostuu seuraavista relaatioista

  `Customers(custNo, name, born, bonus, address, email)`
  `Products(number, prodName, description, price, manufID)`
  `Manufacturers(ID, manufName, phone)`
  `Orders(orderNo, deliver, status, custNo)`
  `BelongsTo(orderNo, productNo, count)`

Miksi tarvitaan hakemistoja?

- Tietokannan taulussa rivit ovat usein satunnaisessa järjestyksessä.

112233	Teemu Teekkari	1995	55
554422	Riina Raksalainen	1993	43
37856	Antti Virta	1970	12
77233	Nina Teekkari	1991	20
31355	Ville Virtanen	1997	14
224477	Teemu Teekkari	1998	22
43255	Sanna Konelainen	1995	17
44551	Ville Virtanen	1991	12
443311	Teemu Teekkari	1986	45

- Huom: oikeasti rivejä taulussa voi olla tuhansia tai kymmeniä tuhansia.
- Vaikka rivit järjestettäisiinkin yhden attribuutin mukaan (mikä vaikeuttaa taulun päivittämistä), ei siitä ole mitään apua, jos tehdään kysely jonkin muun kuin järjestämisessä käytetyn attribuutin arvon perusteella.

Miksi tarvitaan hakemistoja? (jatkuu)

- Ilman hakemistoja tavalliseen SQL-kyselyyn vastaaminen vaatii usein koko taulun läpikäymisen. Esimerkiksi

  ```
  SELECT *
  FROM Customers
  WHERE name = 'Ville Virtanen';
  ```

  vaatii sen, että käydään läpi kaikki *Customers*-relaation monikot ja tutkitaan niistä jokaisesta, toteuttaako monikko annetun ehdon.

Miksi tarvitaan hakemistoja? (jatkuu)

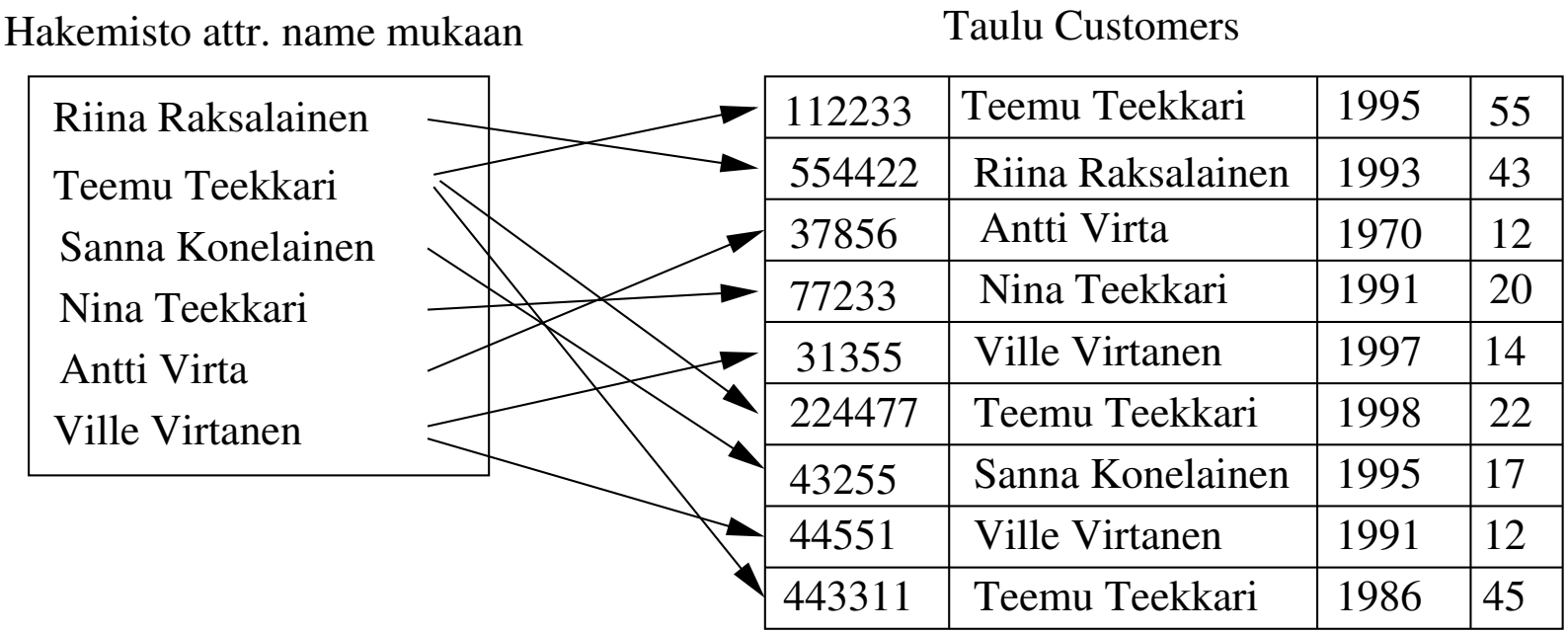
- Liitoksia tarvitsevassa kyselyssä tilanne on vieläkin hankalampi, esimerkiksi

```
SELECT orderNo, C.custNo, name, address
FROM Customers AS C, Orders AS O
WHERE C.custNo = O.custNo;
```

  ilman hakemistoa käydään *Orders*-relaatio kokonaan läpi jokaista *Customers*-relaation monikkoa kohti. Jos molemmat relaatiot eivät mahdu yhtä aikaa kokonaan keskusmuistiin, joudutaan niitä siirtelemään levy- ja keskusmuistin välillä.

Hakemistot

- Hakemiston (index) avulla voidaan löytää nopeasti monikot, joiden määrätyllä attribuutilla tai attribuuttien yhdistelmällä on haluttu arvo. Hakemistoja voidaan tehdä myös useamman attribuutin arvojen yhdistelmien suhteen.

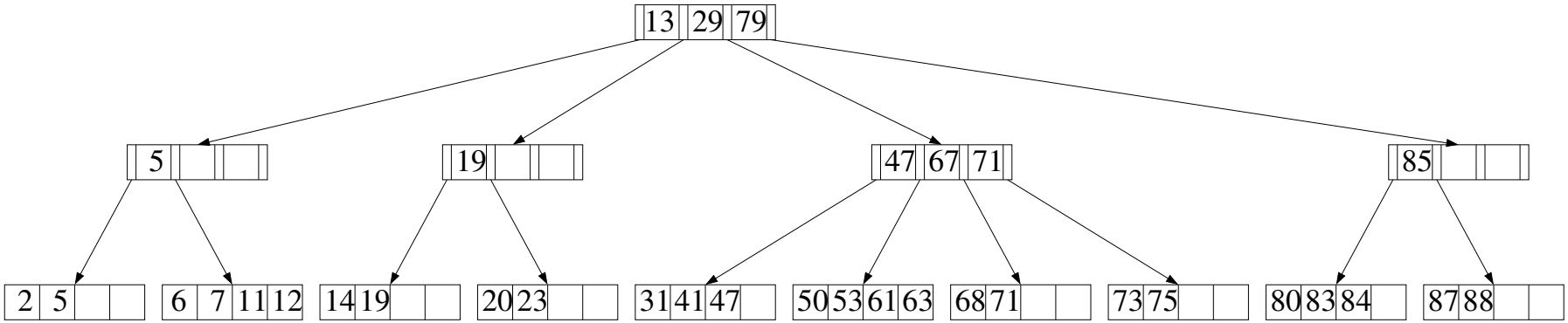


(Kuvan yksinkertaistamiseksi kaikkia `Customers`-relaation attribuutteeja ei näy kuvassa. Oikeasti itse taulussa on kaikki attribuutit, mutta hakemistossa vain se attribuutti, jonka mukaan hakemisto on tehty)

Hakemistot (jatkuu)

- Hakemistoissa tallennetun arvon yhteyteen on lisätty tieto siitä, mistä kohdasta/kohdista tietokannan taulusta arvo/arvot löytyy/löytyvät.
- Hakemiston rakenne on sellainen, että haettu arvo löytyy sieltä hyvin nopeasti.
- Useimmiten tietokannan hallintajärjestelmissä hakemistot on toteutettu B-puina (engl. B-tree).
- Toinen vaihtoehto hakemiston toteuttamiselle on hajautusrakenne (engl. hashing). Siinä hakemistoon tallennettavasta arvosta lasketaan jollain matemaattisella algoritmilla lukuarvo, jonka perusteella määräytyy se, mihin hakemistossa arvo tallennetaan.
- Hajautusrakenteeseen perustuvat hakemistot toimivat tehokkaasti, kun etsitään monikoita, jonka attribuutilla on täsmälleen haluttu arvo. B-puu-rakenteet tukevat hyvin myös sellaisia kyselyitä, joissa haetaan annetulla arvovälillä olevia attribuutin arvoja.

Esimerkki B-puusta



- Tässä kuvassa B-puun kullakin ei-lehtisolmulla on korkeintaan neljä lasta, mutta oikeasti lapsia on kymmeniä tai satoja.
- B-puun toiminta selitetään luennolla. Jos et osallistunut luennolle, voit tutusta aiheeseen esimerkiksi sivun `http://lipas.uwasa.fi/cs/kurssit/troo/tr/tr46.html` tekstin avulla. Olennaisinta on ymmärtää se, miten avaimen haku puussa etenee. Lehtisolmuissa jokaiseen arvoon liittyy tieto siitä, missä tämä arvo esiintyy tietokannan taulussa määrätyn attribuutin arvona.

Hakemiston luominen

- Hakemisto luodaan SQL:ssä käskyllä CREATE INDEX, esimerkiksi

  CREATE INDEX CustomerIndex ON Customers(name);

  luo *Customers*-relaatioon hakemiston, jonka avulla löydetään helposti monikot, joilla on haluttu arvo attribuutilla `name`
- Hakemiston voi luoda useamman attribuutin arvon perusteella seuraavasti:

  CREATE INDEX ProductIndex ON Products(description, manufID);

  luo hakemiston, jonka avulla löydetään nopeasti tuotteet, joilla on halutut arvot attribuuteilla `description` ja `manufID` (molemmilla yhtä aikaa). Tällaisesta hakemistosta ei ole mitään hyötyä, jos haku tehdään pelkästään jälkimmäisen attribuutin arvon perusteella.

Koska hakemiston luominen kannattaa?

- Sopiva hakemisto voi nopeuttaa selvästi kyselyitä, joissa ollaan kiinnostuneita sen attribuutin arvosta, jonka perusteella hakemisto on tehty. Kannattaako siis tehdä hakemistot taulujen kaikkien attribuuttien suhteen?
- Ei yleensä, koska hakemistoa on päivitettävä jokaisen päivityksen yhteydessä. Jos teemme *Products*-relaatioon hakemiston erikseen jokaista viittä attribuuttia varten, pitää päivittää viittä eri hakemistoa joka kerta, kun *Products*-relaatioon lisätään uusi monikko tai siitä poistetaan vanha.
- Lisäksi hakemistot vaativat paljon tilaa. Joskus hakemiston vaatima tila voi olla yhtä suuri kuin itse relaation vaatima tila.
- On siis valittava huolella ne hakemistot, joista on enemmän hyötyä kuin mitä aiheutuu lisätyötä niiden vaatimista päivityksistä. Seuraavilla kalvoilla käydään läpi niitä periaatteita, joiden avulla voidaan päätellä, mitkä hakemistot kannattavat.

Ratkaiseva tekijä hakemiston hyötyjä ja kustannuksia arvioitaessa

- Relaation monikot on yleensä tallennettu levymuistiin. Levymuisti on jaettu levysivuihin eli lohkoihin (disk page, block). Tyypillisesti yhden relaation monikoiden tallentamiseen tarvitaan useita levysivuja.
- Vaikka haluttaisiin tutkia vain yksi relaation monikko, on haettava keskusmuistiin koko se levysivu, jolla monikko sijaitsee. Kun levysivu on haettu keskusmuistiin, ei vie paljonkaan ylimääräistä aikaa, vaikka tutkittaisiin kaikki sillä olevat monikot.
- Yhden levysivun hakemiseen (levyhakuun) menee siis paljon aikaa verrattuna kaikkeen siihen, mitä keskusmuistissa yleensä tehdään.
- Kun tarkastellaan esim. kyselyn suorittamiseen kuluvaa aikaa, niin yleensä ratkaisevaa on tarvittavien levyhakujen määrä. Jos hakemiston avulla voidaan pienentää niitä selvästi, nopeutuvat kyselyt. Hakemiston käyttäminen vaatii kuitenkin sekin levyhakuja.

Milloin hakemiston luominen voi kannattaa, jatkoa

- Yleensä relaation avaimen mukaan kannattaa tehdä hakemisto, koska
  - Kyselyt, joissa avaimen arvo on määrätty, ovat yleisiä
  - Relaatiosta löytyy vain korkeintaan yksi monikko, jolla on haluttu avainarvo. Tällöin selvitään ko. monikon sisältävän levysivun lukemisella sen sijaan, että pitäisi käydä koko relaatio läpi. (Lisäksi tulevat kuitenkin hakemiston käyttämisen vaatimat levyhaut.)
- Jos jollain attribuutilla sama arvo esiintyy vain harvoilla monikoilla. Tällöin haluttu attribuutin arvo esiintyy todennäköisesti vain pienessä osassa relaation käyttämistä levysivuista, ja hakemistoa käyttämällä vältytään monien levysivujen lukemiselta.
- Jos relaation monikot on klusteroitu jonkin attribuutin arvon mukaan. Tällä tarkoitetaan sitä, että monikot, joilla tämän attribuutin arvo on sama, on tallennettu lähelle toisiaan. Tällöin tämän attribuutin mukaan tehdyn hakemiston avulla voidaan lukea vain osa relaation käyttämistä levysivuista.

Välitehtävä 1

- Oletetaan, että relaatio

  `Students(ID, name, program, year)`

  on tallennettu niin, että yhdelle levysivulle mahtuu keskimäärin 40 monikkoa. Tietokannasta haetaan usein määrättynä vuonna aloittaneita opiskelijoita, esimerkiksi

  ```
  SELECT *
  FROM Students
  WHERE year = 2017;
  ```

- Kannattaako (ja miksi) tehdä hakemisto `year`-attribuutin mukaan, jos
  1. eri vuosina aloittaneet opiskelijat ovat jakautuneet satunnaisesti relaation monikoiden kesken?
  2. relaation monikot on klusteroitu `year`-attribuutin arvon mukaan?

Ratkaisu välitehtävään

- Jos eri vuosina aloittaneet opiskelijat ovat jakautuneet satunnaisesti relaation monikoiden kesken, samana vuonna aloittaneita opiskelijoita on suunnilleen jokaisella relaation viemällä levysivulla. Hakemisto aloitusvuoden mukaan ei tällöin nopeuta hakua lainkaan, koska joka tapauksessa pitää hakea koko relaatio keskusmuistiin. Hakemisto ei siis kannata.
- Jos relaation monikot on klusteroitu `year`-attribuutin arvon mukaan, ovat samana vuonna aloittaneita opiskelijoita kuvaavat monikot relaatiossa peräkkäin. Kun hakemiston avulla löydetään yksi määrättynä vuonna aloittanut opiskelija, ovat kaikki muut samana vuonna aloittaneet heti sitä ennen tai sen jälkeen. Koko relaatiota ei siis tarvitse käydä läpi, vaan hakemiston avulla löydetään oikea aloituskohta, ja sen jälkeen tarvitsee vain käydä relaatiota läpi siitä molempiin suuntiin niin pitkälle kuin samana vuonna aloittaneita opiskelijoita riittää. Hakemiston luominen todennäköisesti kannattaa.

Mitä hakemistoja kannattaa luoda?

- Kun lähdetään selvittämään, minkä attribuuttien mukaan kannattaa tietokantaan määritellä hakemistoja, on ensin oltava arvio siitä, mikä kyselyjen ja päivitysten suhde tietokannan operaatioissa ja millaisia kyselyitä tietokantaan yleensä tehdään.
- Tämän jälkeen lasketaan erikseen, kuinka paljon levyhakuja tarvitaan keskimäärin eri operaatioissa eri hakemistoyhdistelmillä. Myös hakemiston käytön aiheuttamat levyhaut tulee ottaa mukaan.
- Kyselyn yhteydessä hakemiston käyttö aiheuttaa vähintään yhden levyhaun (yleensä vähintään B-puun alin taso on levymuistissa). Päivitysten yhteydessä levyhakuja tulee vähintään kaksi, koska muutettu B-puun solmu pitää myös kirjoittaa levymuistiin.
- Kun eri operaatioiden vaatimat levyhakujen määrät eri hakemistoyhdistelmille ovat selvillä, lasketaan operaatioiden esiintymistodennäköisyyksillä painotettu keskiarvo tarvittavista levyhauista kullekin hakemistoyhdistelmälle.

Esimerkki hakemistojen valinnasta

- Tarkastellaan relaatiota

  `BelongsTo(orderNo, productNo, count)`

  ja lasketaan, mitä hakemistoja relaatiolle kannattaa luoda, jos
  - Relaatioon tehdään kahdenlaisia kysyitä. Toisissa haetaan monikoita, joissa on määrätty tilausnumero. Näiden kyselyjen osuus kaikista relaatioon kohdistuvista tietokantaoperaatioista on p_1. Toisissa haetaan monikoita, joissa on määrätty tuote. Näiden kyselyjen osuus kaikista relaatioon kohdistuvista tietokantaoperaatioista on p_2.
  - Lisäksi relaatioon tehdään lisäyksiä, joiden osuus kaikista relaatioon kohdistuvista tietokantaoperaatioista on $1 - p_1 - p_2$.
  - Relaation monikot vievät 100 levysivua, joten koko relaation monikoiden tutkiminen vaatii 100 levyhakua.
  - Yhdessä tilauksessa on keskimäärin 3 tuotetta ja kukin tuote kuuluu keskimäärin 30 tilaukseen.
  - Monikoita ei ole klusteroitu minkään attribuutin arvon mukaan.

Esimerkki hakemistojen valinnasta, jatkoa

- Koska kyselyjen tuloksena tulevia monikoita on yleensä selvästi vähemmän kuin hakemiston vaatimia levysivuja, on useimmiten jokainen tulosmonikko taulussa (relaatiossa) eri levysivulla. (Tässä ei siis lasketa tarkkoja todennäköisyyksiä, vaan karkea arvio riittää.)
- Näin hakemistoa käytettäessä ensimmäiseen kyselyyn vastaaminen vaatii yleensä 3 levyhakua tulosmonikoiden hakemiseksi itse taulusta ja yhden levyhaun hakemiston käyttämiseen.
- Toiseen kyselyyn vastaaminen vaatii hakemistoa käytettäessä yleensä 30 levyhakua tulosmonikoiden hakemiseksi itse taulusta ja yhden levyhaun hakemiston käyttämiseen.
- Lisäysten yhteydessä jokaista hakemistoa kohti tarvitaan kaksi levyhakua (toinen B-puun alimman solmun lukemiseen ja toinen päivitetyn solmun kirjoittamiseen). Lisäksi tarvitaan yhteensä kaksi levyhakua itse päivitettävän taulun levysivun lukemiseen ja kirjoittamiseen.

Esimerkki hakemistojen valinnasta, jatkoa

- Lasketaan eri operaatioille tarvittavat levyhakujen määrät eri hakemistoyhdistelmien vaihtoehdoilla:
  1. Ei lainkaan hakemistoja (kyselyä suoritettaessa koko taulu pitää käydä läpi)
  2. Käytössä hakemisto tilausnumeron mukaan (orderNo Index): löydetään hakemiston avulla nopeasti taulusta monikot, joilla esiintyy haluttu tilausnumero.
  3. Käytössä hakemisto tuotenumeron mukaan (productNo Index): löydetään hakemiston avulla nopeasti taulusta monikot, joilla esiintyy haluttu tuotenumero.
  4. Käytössä molemmat yllä mainitut hakemistot

Esimerkki hakemistojen valinnasta, jatkoa

Action	No Index	orderNo Index	productNo Index	Both Indexes
Q_1	100	4	100	4
Q_2	100	100	31	31
I	2	4	4	6
Average	$2 + 98p_1 + 98p_2$	$4 + 96p_2$	$4 + 96p_1 + 27p_2$	$6 - 2p_1 + 25p_2$

- Merkinnät:
  1. Q_1 kysely, jossa haetaan monikoita, joissa esiintyy haluttu tilausnumero
  2. Q_2 kysely, jossa haetaan monikoita, joissa esiintyy haluttu tuote
  3. I relaatioon tehtävä lisäys
- Opimaalinen hakemistoratkaisu riippuu p_1:n ja p_2:n arvoista.
- Laskennassa on tehty yksinkertaistuksia. Ei ole esim. otettu huomioon sitä, että jos relaatio on tallennettu peräkkäisille levysivuille, voidaan koko relaatio hakea nopeammin kuin mitä sadan eri levysivun hakeminen vie aikaa.

Selityksiä ym.

- Edellisen kalvon taulukon viimeinen rivi on saatu kertomalla kunkin tapahtuman todennäköisyys ko. tapahtuman vaatimien levyhakujen määrällä ja laskemalla näin saadut tulot yhteen.
- Esimerkiksi tapaus *No index*:
  $100p_1 + 100p_2 + 2(1 - p_1 - p_2) = 2 + 98p_1 + 98p_2$
- Jos haetaan avaimen arvon perusteella ja kyselyt kohdistuvat lähes aina arvoihin, jotka löytyvät relaatiosta, ei ilman hakemistoa tarvitse käydä aina koko relaatiota läpi, vaan voidaan lopettaa, kun ollaan löydetty etsitty arvo. Tällöin ilman hakemistoa tarvitaan levyhakuja keskimäärin puolet koko relaation vaatimasta levysivujen määrästä.
  - Edellisessä esimerkissä kyselyitä ei tehty avainarvon perusteella, joten niissä koko relaatio jouduttiin käymään läpi, jos hakemistoa ei ollut käytössä.

Huomautus

- Myös muut seikat voivat vaikuttaa hakemistojen valintaan. Joissakin tietokannoissa voi olla tärkeintä, että määrätyt kyselyt voidaan suorittaa nopeasti, kun taas muiden kyselyjen tai päivitysten hidastuminen ei ole kriittistä. Tällöin hakemistot suunnitellaan niin, että ne tukevat aikakriittisiä kyselyitä.

Transaktiot

- Seuraavaksi tarkastellaan erilaisia ongelmatilanteita ja niiltä suojautumista *transaktioiden* (tapahtumien) avulla.

Ongelmatilanne 1

- Oletetaan, että pankin tilitiedot on tallennettu relaatioon

  `Accounts(acctNo, balance)`

- Tarkastellaan tilisiirtoa tililtä 286 tilille 354:

```
UPDATE Accounts
SET balance = balance − 1000
WHERE acctNO = 286;

UPDATE Accounts
SET balance = balance + 1000
WHERE acctNO = 354;
```

- Entä, jos ensimmäisen käskyn suorittamisen jälkeen ennen toista käskyä tulee sähkökatko tai laiterikko?

Ratkaisu: transaktiot

- Sähkökatkon tai muun häiriön aiheuttama ongelma voidaan välttää, jos ohjelmoija määrittelee, että molemmat edellisen kalvon käskyt kuuluvat samaan *transaktioon* (tapahtumaan, engl. transaction).
- Tietokannan hallintajärjestelmä pitää huolen siitä, että samaan transaktioon kuuluvat käskyt suoritetaan joko kaikki kokonaan tai yhtäkään niistä ei suoriteta.
- Tätä transaktioiden ominaisuutta kutsutaan *atomisuudeksi* (engl. atomicity).
- Atomisuuden lisäksi transaktioilta vaaditaan myös muita ominaisuuksia, joita käsitellään seuraavilla kalvoilla.

Miten tietokannan hallintajärjestelmä huolehtii atomisuudesta?

- Voiko tietokannan hallintajärjestelmä estää sähkökatkot ja laiterikot?
- Ei voi, mutta se voi varmistaa sen, että sähkökatkon tai laiterikon jälkeen tietokanta saadaan takaisin sellaiseen tilaan, jossa joko kaikki transaktioon kuuluvat käskyt on suoritettu tai mitään niistä ei ole tehty (eli osittain suoritetun transaktion tekemät muutokset on peruttu).
- Yleensä tähän käytetään *lokia* (engl. log), esimerkiksi:
  - Ennen kuin käskyt muuttavat itse relaatioiden monikkoja, tallennetaan lokiin päivitettävien monikoiden attribuuttien vanhat arvot.
  - Loki tallennetaan pysyvään muistiin ennen monikoiden päivittämistä.
  - Jos transaktion suoritus jää kesken, voidaan lokin tietojen avulla palauttaa attribuuttien vanhat arvot.

Ongelmatilanne 2

- Oletetaan, että lentoyhtiön lentojen paikanvaraustiedot on tallennettu relaatioon

  `Flights(fltNO, fltDate, seatNo, seatStatus)`

- Haetaan vapaa paikka halutulle lennolle:

```
SELECT seatNO
FROM Flights
WHERE fltNo = 'AY001' AND fltDate = DATE '2019-05-23'
    AND seatStatus = 'available';
```

- Oletetaan, että kysely palautti paikan 35C, jolloin se voidaan varata:

```
UPDATE Flights
SET seatStatus = 'occupied'
WHERE fltNo = 'AY001' AND fltDate = DATE '2019-05-23'
    AND seatNo = '35C';
```

Ongelmatilanne 2, jatkoa

- Edellisen kalvon käskyt toimivat hyvin, jos vain yksi käyttäjä käsittelee kerrallaan relaatiota `Flights`, mutta yleensä lentolippuja varataan useasta eri paikasta samanaikaisesti.
- Oletetaan seuraava suoritusjärjestys (aika kuluu ylhäältä alaspäin):

```
Käyttäjä 1 huomaa
paikan 35C olevan vapaa
```

```
                                    Käyttäjä 2 huomaa
                                    paikan 35C olevan vapaa
```

```
Käyttäjä 1 varaa
paikan 35C
```

```
                                    Käyttäjä 2 varaa
                                    paikan 35C
```

Sarjallistuvuus

- Vaaditaan, että transaktioiden pitää olla *sarjallistuvia* (serializable): Jos useita transaktioita suoritetaan samanaikaisesti, pitää niiden vaikutus tietokantaan olla sama kuin jos samat transaktiot olisi suoritettu peräkkäin yksi kerrallaan.
- Vastaava sarjallinen suoritusjärjestys saa kuitenkin olla mikä tahansa mahdollisista. Jos esim. samanaikaisia transaktioita on T_1 ja T_2, pitää lopputuloksen olla sama kuin jommassa kummassa seuraavista vaihtoehdoista:
  - T_1 suoritetaan kokonaan ennen T_2:ta
  - T_2 suoritetaan kokonaan ennen T_1:tä
- Jos ohjelmoija on määrännyt tiettyjen käskyjen kuuluvan samaan transaktioon, niin tietokannan hallintajärjestelmä pitää huolen eri transaktioiden sarjallistuvuudesta.
- Sarjallistuvuudesta voidaan pitää huoli esimerkiksi lukitsemalla transaktioiden käsittelemiä monikoita niin, että vain yksi transaktio voi käsitellä niitä kerrallaan.

Välitehtävä 2

- Tarkastellaan transaktioita T_1 ja T_2
  - T_1:

```
read(X)
X = X + 10
write(X)
```

  - T_2:

```
read(X)
X = X * 1.2
write(X)
```

- Oletetaan, että `X`:n arvo ennen transaktioiden suorittamista on 100. Mitkä ovat mahdollisia `X`:n arvoja transaktioiden suorittamisen jälkeen, jos transaktioiden suoritus on sarjallistuva?
- Mitkä olisivat mahdollisia `X`:n muita arvoja transaktioiden suorittamisen jälkeen, jos transaktioiden suoritus ei olisi sarjallistuva?

Ratkaisu välitehtävään

- Kaikissa sarjallistuvissa suorituksissa lopputulos (`X`:n arvo tietokannassa) on joko sama, joka se olisi silloin, kun T_1 suoritetaan kokonaan ennen T_2:sta, tai silloin, kun T_2 suoritetaan kokonaan ennen T_1:stä. Sarjallistuvan suorituksen jälkeen siis `X`:n arvo voi olla joko 132 tai 130 sen mukaan, kumpaa suoritusjärjestystä lopputulos vastaa.
- Jos suoritus ei ole sarjallistuva, on mahdollista, että T_1 ja T_2 vuorotellen lukevat ja päivittävät `X`:n arvoa niin, että toinen ei huomaa toisen jo muuttaneen arvoa aikaisemmin luetusta (samaan tapaan kuin lentokoneen paikanvarausesimerkissä). Tällöin X voi olla molempien transaktioiden suorituksen jälkeen joko 120 tai 110 sen mukaan, kumpi transaktioista päivittää X:ää viimeisenä.

Huomio sarjallistuvuudesta

- Sarjallistuvuus **ei tarkoita** sitä, että transaktiot pitäisi aina suorittaa peräkkäin yksi kerrallaan.
- Sarjallistuvuus tarkoittaa vain sitä, että lopputuloksen pitää olla sama kuin jos transaktiot olisi suoritettu peräkkäin (jossain järjestyksessä).
- Sarjallistuvuus ei myöskään tarkoita sitä, että ei ole väliä sillä, missä järjestyksessä yhden transaktion sisällä olevat käskyt suoritetaan (tenttivastausten perusteella tämä on yleinen virhekäsitys).

Ongelmatilanne 3

- Tarkastellaan vielä lentojen paikanvarausjärjestelmää. Oletetaan, että käyttäjä on varannut paikan 35C ja päivitys on tehty keskusmuistiin haettuun relaation osaan, joka sisältää päivitetyn monikon.
- Tapahtuu kuitenkin laiterikko tai sähkökatko, ennen kuin keskusmuistiin haettu päivitetty levysivu on tallennettu takaisin kovalevylle.
- Katoaako käyttäjän tekemä varaus?

Pysyvyys

- Varaus ei katoa, jos ohjelmoija on määritellyt varauksen tekevän ohjelman osan transaktioksi. Tällöin tietokannan hallintajärjestelmä pitää huolen siitä, että onnistuneesti suoritetun transaktion tekemät muutokset jäävät tietokantaan pysyvästi.
- Yksi transaktioilta vaadittava ominaisuus on siis *pysyvyys* (engl. durability).
- Asiasta voidaan huolehtia esimerkiksi lokin avulla.

Ongelmatilanne 4

- Tarkastellaan jälleen pankkitietokantaa, ja oletetaan sen sisältävän relaatiot

  `Accounts(acctNo, balance)`
  `Loans(loanNo, balance, custID)`
  `Customers(custID, name, acctNo)`

  Pankki haluaa siivota pois relaatiosta `Customers` ne asiakkaat, joilla ei ole lainkaan tiliä:

  ```
  DELETE FROM Customers
  WHERE acctNo IS NULL;
  ```

- Tässä saattaa kuitenkin tulla poistetuksi sellainen asiakas, jolla ei ole tiliä, mutta jolla on esimerkiksi 200 000 euron laina.

Eheys

- Edellisen kalvon ongelmatilanne vältetään sillä, että transaktioiden pitää toteuttaa *eheys*-ominaisuus (consistency): jos tietokannassa määritellyt eheysehdot ovat voimassa ennen transaktion suoritusta, niiden pitää olla voimassa myös transaktion suorituksen jälkeen.
- Oletetaan, että pankkitietokannassa on eheysehto, jonka mukaan `Loans`-relaation `custID`-attribuutin arvon pitää löytyä `Customers`-relaatiosta.
- Tietokannan hallintajärjestelmä ei salli sellaisia transaktioita, joiden jälkeen tietokannassa määritellyt eheysehdot eivät olisi voimassa. Jos lopputuloksena olisi tällainen tila, transaktio keskeytetään ja sen tekemät muutokset perutaan.
- Eheysehtoja voi olla erilaisia. Viite-eheys on vain yksi mahdollisuus. Yhtä hyvin pankkitietokannassa voi olla määriteltynä esim. eheysehto, jonka mukaan pankkitilin saldo ei saa olla negatiivinen.

Yhteenveto: transaktioilta vaadittavat ominaisuudet

- **Atomicity** (atomisuus): kaikki transaktion sisältämät käskyt suoritetaan tai mitään niistä ei suoriteta.
- **Isolation** (serializability, sarjallistuvuus): jos useita transaktioita suoritetaan samanaikaisesti, niin lopputulos on sama kuin jos samat transaktiot olisi suoritettu jossain järjestyksessä peräkkäin yksi kerrallaan.
- **Consistency** (eheys): jos tietokannassa määritellyt eheysehdot ovat voimassa ennen transaktion suoritusta, niin ne ovat voimassa myös sen jälkeen.
- **Durability** (pysyvyys): Jos transaktio on suoritettu onnistuneesti loppuun (transaktio on sitoutunut), niin sen vaikutukset eivät katoa tietokannasta.
- Näitä ominaisuuksia kutsutaan usein *ACID-ominaisuuksiksi*.

Transaktion määrittely SQL:ssä

- Transaktion määrittely SQL:ssä voidaan aloittaa ilmauksella

  BEGIN TRANSACTION

  Se ei ole kuitenkaan välttämätön, sillä SQL-standardin mukaan oletuksena ensimmäinen suoritettava SQL-käsky aloittaa uuden transaktion, samoin edellisen transaktion päättymisen jälkeen suoritettava seuraava SQL-käsky.
- Tämän jälkeen kirjoitetaan transaktioon sisältyvät käskyt
- Transaktio päätetään joko ilmauksella
  - COMMIT

    transaktio on suoritettu onnistuneesti ja sen vaikutukset tehdään tietokannassa pysyviksi eli transaktio *sitoutuu* tai
  - ROLLBACK

    transaktiota ei voitu suorittaa onnistuneesti (esimerkiksi tilillä ei ollut tarpeeksi rahaa nostoon tai tilisiirtoon), joten sen tekemät muutokset perutaan.

Transaktioiden eristyvyystasot

- Perustapauksessa vaatimuksena on se, että transaktiojoukon suoritus on sarjallistuva. Toisinaan tästä vaatimuksesta kuitenkin joustetaan, jotta saataisiin enemmän rinnakkaisuutta.
- Tällöin menetetään osa luettujen arvojen luotettavuudesta, mutta kaikissa sovelluksissa sillä ei ole väliä.
- Mahdolliset eristyvyystasot:
  - Read uncommitted (likaisen datan luku sallittu)
  - Read committed (luetaan vain sitoutuneiden transaktioiden kirjoittamaa tietoa)
  - Repeatable read (luvun toistettavuus)
  - Serializable (sarjallistuvuus)

Likaisen datan lukeminen

- *Likaisella datalla* (dirty data) tarkoitetaan arvoja, joiden kirjoittanut transaktio ei ole sitoutunut (commit).
- Jos yksi transaktio lukee toisen transaktion kirjoittamaa likaista dataa, on mahdollista, datan kirjoittanut transaktio myöhemmin keskeytyy.
- Tällöin datan lukenut transaktio toimii sellaisen tiedon pohjalta, jota ei koskaan virallisesti ole ollut tietokannassa.
- Joissakin sovelluksissa tämä kuitenkin halutaan sallia, koska suorituksen tehokkuutta pidetään tärkeämpänä kuin tietojen täsmällistä oikeellisuutta.
- Jos transaktiolle halutaan sallia likaiset luvut, määritellään se SQL-käskyllä

  SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

- Tälläkään eristyvyystasolla ei sallita sitä, että transaktio kirjoittaisi toisen transaktion kirjoittaman likaisen datan päälle.

Eristyvyystasot: sitoutuneiden luku

- Voidaan vaatia, että transaktio lukee vain sitoutuneiden transaktioiden kirjoittamaa dataa, mutta se ei silti ole välttämättä täysin sarjallistuva.
- Jos transaktion eristyvyystasoksi asetetaan READ COMMITTED, se ei lue likaista dataa, mutta jos transaktio lukee saman monikon arvoja useaan kertaan, voi arvo olla lukukertojen välillä vaihtunut, jos jokin toinen transaktio on sitoutunut lukukertojen välillä.
- Esimerkki:
  - Olkoon määritelty relaatio

    `Students(ID, name, credits)`
  - Oletetaan, että transaktio T_1 päivittää opiskelijan 11223F opintopistemäärää ja transaktio T_2 lukee saman opiskelijan opintopistemäärän kahteen kertaan.
  - Jos T_1 sitoutuu näiden lukukertojen välissä, niin T_2 lukee eri opintopistemäärät eri lukukerroilla.

Eristyvyystasot: luvun toistettavuus

- Jos transaktion eristyvyystasoksi asetetaan REPEATABLE READ, niin vaaditaan, että jos transaktio lukee saman monikon attribuuttien arvoja useaan kertaan, niin luettu arvo on joka kerralla sama.
- Edellisen kalvon opintopiste-esimerkissä T_2 lukee tällä eristyvyystasolla joka lukukerralla joko ennen T_1:n päivitystä olleen opintopistemäärän tai päivityksen jälkeen tallennetun opintopistemäärän, mutta ei yhdellä kerralla yhtä ja toisella kerralla toista.
- Eristyvyystaso ei kuitenkaan estä sitä, että toinen transaktio lisää käsiteltävään relaatioon uusia monikoita transaktion aikana. Se estää vain niiden monikoiden muuttamisen, jotka transaktio on jo lukenut.

Eristyvyystasot: luvun toistettavuus (jatkoa)

- Esimerkki:
  - T_3 laskee `Students`-relaation opiskelijoiden opintopistemäärän keskiarvon kahteen kertaan.
  - Samaan aikaan toinen transaktio T_4 voi lisätä `Students`-relaatioon uusia monikoita.
  - T_4:n lisäämien monikoiden vuoksi T_3:n laskema keskiarvo voi olla jälkimmäisellä kerralla ensimmäisestä kerrasta poikkeava.
- Monikoita, jotka on lisätty relaatioon T_3:n eri lukukertojen välillä, kutsutaan *haamuiksi* (phantoms).

Yhteenveto eristyvyystasoista

Eristyvyystaso	Likaiset luvut	Ei-toistettavat luvut	Haamut
Read uncommitted	mahdollinen	mahdollinen	mahdollinen
Read committed	ei	mahdollinen	mahdollinen
Repetable read	ei	ei	mahdollinen
Serializable	ei	ei	ei

- SQL-standardissa oletuksena on eristyvyystaso SERIALIZABLE, monissa järjestelmissä se on kuitenkin löysempi (esim. MySQL:ssa REPEATABLE READ, Oraclessa ja PostgresSQL:ssä READ COMMITTED).

Huomautus eristyvyystasoista

- Eristyvyystason määrittely koskee aina sen transaktion lukemia tietoja, jolle taso on määritelty.
- Jos esim. T_1:n taso on READ UNCOMMITTED ja T_2:n taso READ COMMITTED, niin T_1 voi lukea T_2:n kirjoittamaa likaista dataa, mutta T_2 voi lukea T_1:n kirjoittamaa dataa vasta sen jälkeen, kun T_1 on sitoutunut.
- Jotta transaktiojoukon suoritus kokonaisuudessaan olisi varmasti sarjallistuva, pitää jokaisen joukkoon kuuluvan transaktion eristyvyystaso olla SERIALIZABLE.

Vain lukuoperaatiota sisältävät transaktiot

- Sarjallistuvuuden kannalta ongelmia aiheuttavat transaktiot, jotka päivittävät relaatioita tai niiden monikoita.
- Jos kaksi tai useampi transaktio ainoastaan lukee tietokannassa olevia arvoja mitään muuttamatta, voi niiden operaatiot lomittua mielivaltaisella tavalla toisiinsa nähden ilman sarjallistuvuusongelmia.
- Ohjelmoija voi kertoa ennen transaktion aloittamista, että seuraava transaktio sisältää vain lukuoperaatioita ilmauksella:

  SET TRANSACTION READ ONLY;

- Vastaavasti voidaan kertoa, että transaktio sisältää myös kirjoitusoperaatioita (oletus muuten, paitsi READ UNCOMMITTED -eristyvyystasolla).

  SET TRANSACTION READ WRITE;

- Samalla käskyllä voidaan myös määritellä transaktion eristyvyystaso:

  SET TRANSACTION READ WRITE
  ISOLATION LEVEL READ UNCOMMITTED;