2. Tietokannan tallennusrakenteet

2.1 Levymuisti ja sen käyttö

Muistilaitteiden hierarkia: ainakin
- keskusmuisti
- levymuisti

(+ muita tukimuisteja, välimuistit, rekisterit)

Tietokannalle levymuisti keskeinen:
- suuri kapasiteetti
- tietojen pysyvyys
- riittävä nopeus
- keskusmuistia halvempaa

Tietokantojen koko vaihtelee todella paljon: MB .. GB ..

Tietokantaperiaate nojaa normaalisti levymuistin käyttöön, vaikka kanta mahtuisikin keskusmuistiin:
keskusmuistia hyödynnetään epäsuorasti, puskuroinnin välityksellä

Erikoistapaus: main memory database
- päätoiminta keskusmuistissa, pysyvyys hoidetaan erikseen (esim. backup-kopio levyllä)

Mikä on ‘very large database’?

Tietokannan tallennusrakenteen suunnittelun merkitys:
(‘physical database design’)

- käsittely (ohjelman suoritus)
  tapahtuu keskusmuistissa
- siirto levyltä ja levylle on hidasta; pyritään minimoimaan
  - sopivilla tiedostorakenteilla
  - muokkaamalla kyselyjä tehokkaiksi (‘optimoimalla’)

- merkitys eri rooleissa:

- tietokannan suunnittelija (asiantuntija)
  tkhj:n tarjoamien vaihtoehtojen valinta, parametrien asetuksia;
  loogisen ja fyysisen tason yhteensovitus
- tkhj:n suunnittelija
  tiedostojen perusrakenteiden ja käsittelyalgoritmien toteutus
- tietokannan hoitaja
  tietokantajärjestelmän suorituskyvyn seuranta, viritys (+ saantioikeudet, ohjelmisto- ja laitteistoresurssien hallinta, eri tietokantojen yhteydet)
- tietokannan käyttäjä: perusasioiden ymmärrys
  kyselyt, menetelmien valinta; realismi

Peruskäsitteitä:

Fyysinen tietokanta = levymuistiin sijoitettu kokoelma sivuja, jotka sisältävät tietueita.

Sivu (page) = vakiokokoinen alue, joka on sijoitettu levyjaksoon. Termejä jakso (block) ja sivu käytetään yleensä vaihtoehtoina.

Tietue (record) jakaantuu kenttiin (field), jotka sisältävät merkki- tai lukuarvoja (merkitys voi olla myös osoitin, lähinnä hakemistoissa).

Tietokannan tietosisältöä tarkastellaan eri tasoilla:
tiedoston (taulun) nimi tai muu tunniste,
sivunumero, tietuenumero, kentän tunniste

Relaatio (taulu) sisältää niukasti rakennetta: tietyn tiedon osoittamisessa tietuenumero (tunniste) periaatteessa riittävä; sijoittelun takia myös sivunumero yleensä käytössä.
Osoitteena on usein sivunumero tai pari (sivu, tietue).

Levymuistin rakenne: levykkö, levypinta, sylinteri, ura, sektori (merkki)

Levyjakso t. lohko (disk block) = yhdestä tai useammasta peräkkäisestä sektorista koostuva alue levypinnan uralla.

Jaksot määräytyvät levyn formatoinnin yhteydessä: kiinteä koko, tyypillisesti 1 KB - 16 KB.

Sektori = pienin osoitettavissa oleva yksikkö;
sektorin fyysinen osoite = (levypinta, ura, sektori).

Levyhaku (disk access) = levymuistin käytön perussuure: joko jakson luku (kopiointi keskusmuistin puskuriin) tai jakson kirjoitus puskurista levylle.

Pienempien osien siirtämiseen kuluva aika voidaan vain laskea keskiarvona; ideana on jakaa levyhaun aika mahdollisimman monelle operaatiolle - ja tavallaan mahdollisimman suurelle määrälle tietoa (pitkät jaksot). Jakson pidentämistä rajoittaa puskuritilan tarve sekä tarpeettomien osien siirto.

Levyhaun vaatima aika: saantiaika l. hakuaika (access time) =
kohdistusaika + pyörähdysviive + siirtoaika.

Kohdistusaika s (seek time): luku/kirjoituspää viedään oikealle uralle (sylinterille).

Sylintereitä esimerkiksi 8000, kohdistusaika välillä 0-s_{max}; esimerkiksi keskimäärin 5 ms, välillä 0 - 15 ms.

Pyörähdysviive rd (rotational delay, latency): odotetaan levyn pyörähdystä jakson alkukohdalle.

- kiinteä pyörimisnopeus esim. 10000 rpm
- keskimääräinen odotus puoli kierrosta

→ viive rd = 60000 / 2*10000 ms = 3 ms
(yleensä luokkaa 3 - 8 ms)

Jakson siirtoaika bt (block transfer time): yhden jakson kopiointi puskuriin tai päinvastoin.

- riippuu jakson ja uran koosta sekä pyörimisnopeudesta:

esim. jakso 4KB, ura 128KB:
bt = (4/128) * (60/10000) s = 0.19 ms

Todellinen hakuaika monimutkainen laskea (sijoitteluvaihtoehdot, jaksovälit ym. kontrollitiedot); yleensä riittää keskimääräinen hakuaika

s + rd + bt

Suuret komponentit s ja rd pyritään eliminoimaan:

- s = 0, jos luetaan samalta sylinteriltä kuin edellinen jakso
- rd = 0, jos luetaan peräkkäisiä jaksoja (samalta uralta)

eli k jakson luku: s + k * (rd + bt) tai s + rd + k * bt

Puskuri = keskusmuistialue, jossa on tilaa joukolle jaksoja (sivuja; page frame). Tkhj:lla yleensä oma puskurinhallitsin, joka tietää (kontrollitiedoista)

- mitkä tietokannan sivut ovat puskurissa,
- onko puskurissa olevan sivun sisältöä muutettu.

Puskurissa olevaa sivua voi käyttää moni operaatio (transaktio) - ilman moninkertaista levyltä lukua (ja välillä tapahtuvaa kirjoitusta). (Tapahtumanhallinta hoitaa ...)

- puskurista haku esim. vain 10-100 ns
- levyhaun aika jakaantuu monelle operaatiolle

Puskurin käyttö on erityisen tehokasta, jos monet transaktiot tarvitsevat samoja sivuja.

Esim. kursseille ilmoittautuminen: (suosittujen) kurssien sivut.

Puskurien käyttö loogisesti:
- operaatiot
  bufferfix(P): tuo sivu P puskuriin
  bufferunfix(B): vapauta puskuri(kytkentä)
  (jaksoa ei välttämättä siirretä heti levylle
  eli tilaa ei vapauteta)

Esim. update employee
set salary = salary + 500;

- yksinkertaistettu toteutus:

```
for jokainen employee-taulun sivu p do
  B:=bufferfix(P);
  for jokainen puskurisivulla B oleva
      employee-monikko e do
          e.salary := e.salary + 500;
  aseta B:n päivitysbitti ('muutettu');
  bufferunfix(B);
end.
```


2.2 Tietokannan tiedostorakenne

Tiedosto = tietuejoukon muodostama kokonaisuus (tietokantasivuja).

Yleensä tiedosto muodostetaan samantyyppisistä tietueista, esim. taulusta. Myös erityyppisiä (eri taulujen) tietueita voidaan säilyttää yhdessä, lähekkäin.

Tiedoston alue levyllä: joukko varausyksikköjä (segment, cluster, extent).
Varausyksikkö = peräkkäisten jaksojen muodostama yhtenäinen alue (vrt. peräkkäin luku).

Tilanvaraus riippuu tiedoston koosta ja järjestelmän piirteistä.

Esim. Oracle: monta sivua = extent, monta extent-aluetta muodostaa segmentin, segmentit kuuluvat taulukkoalueeseen (tablespace).
- eri segmentit datalle, hakemistoille, historiatiedolle, tilapäiseen käyttöön

- aluksi varataan yksi (data)extent, tarvittaessa lisää; koot create table -lauseen parametreina (käytännössä yleensä DBA:n toimesta taulukkoalueittain).

Tiedostokuvaaja (file descriptor) edustaa tiedostoa (ohjelmille): sisältää hallintatietoa + tiedoston levyalueiden osoitteet, esimerkiksi luettelona tai hierarkiana.

Unix: i-node:
- 10 suoraa levyosoitetta
- max. 3 epäsuoraa osoitetta (osoitelohkoihin)

Sivun sisäinen organisointi:

- kiinteänmittaiset tietueet: jaksossa on f tietuetta
  f = jaksotuskerroin, kiinteät tietuepaikat

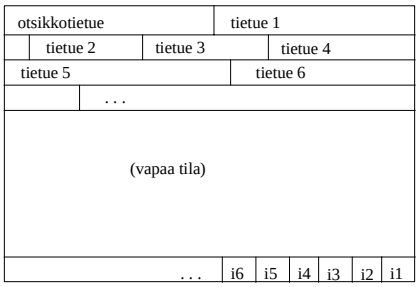
- vaihtuvanmittaiset tietueet: tietueiden (paikkojen) hallinta monimutkaistuu

Esim. seuraava organisaatio:
- otsikkotietue (page header)
- tietueet (monikot) peräkkäin luontijärjestyksessä
- tietuehakemisto: tietueiden (suhteelliset) osoitteet

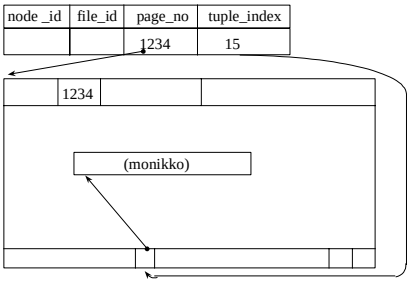
Sijoittamalla tietuehakemisto kasvamaan sivun lopusta alkua kohti voidaan sivun tila käyttää optimaalisesti hyväksi.

Otsikkotietueessa on hallintatietoa, tunnisteita, viimeisin päivitys-/käyttöaika jne.

Esimerkki sivun organisoinnista (vaihtuvanmittaiset tietueet)



Tekninen tietuetunniste (monikkotunniste):



Kiinteänmittaisten tietueiden paikat voidaan laskea ilman hakemistoa.
Tietueen osoite = (p,i), missä p on sivun numero ja i on tietuehakemiston indeksi (tai tietueen osoite, jos ei ole hakemistoa).

Relaatiotietokannassa osoitteita ei esiinny datasivuilla, vaan vain hakemistosivuilla.
(Olio- ja verkkotietokannoissa oliot (tietueet) voivat sisältää toisten olioiden (tietueiden) osoitteita (osoittimia).

Yleisessä tapauksessa tietueella on täydellinen tietuetunniste (record identifier; myös monikko- tai rivitunniste):
- tietoverkon pisteen (site) tunniste
- tiedoston tunniste
- osoite (p,i).

Yleensä jaksossa on monta kokonaista tietuetta (bfr = jaksotuskerroin, usein myös f).

Jaksorajan yli jatkuva tietue (spanned record)
- käytetään tila tarkemmin
- käsittely monimutkaistuu
- joskus välttämätön (blob)

jakso i	tietue j-2	tietue j-1	tietue j (alkuosa)

jakso i+1	tietue j (loppuosa)	tietue j+1	. . . . .

Tietueen esitysmuoto

Tietueen kentät voivat olla kiinteän- tai vaihtuvanmittaisia.
(Tietueet voivat olla myös erityyppisiä;
relaatiotietokannassa yleensä taulu/tiedosto
- paitsi muodostettaessa klustereita)

Esim. relaation employee(name, ssn, salary) monikon
('Smith, John', '010263-189F', 17000)
esittäminen:

1. kiinteänmittaiset kentät

Smith, John |010263-189F|	17000
20 tavua	11 tavua	4 tavua

Kokonaisluku ehkä aloitetaan sanarajalta; mahdollisesti edessä tyhjä tavu.

2. vaihtuvanmittaista NAME-kenttää käyttäen:

| 17000 |010263-189F|11|Smith, John|

- Vaihtuvanmittainen kenttä voidaan esittää myös erotinmerkkien avulla (tässä etu- ja sukunimi erikseen):

  |010263-189F|*|Smith|*|John|*|

Oraclen tietueen (monikon) esitys: kaikki tiedot periaatteessa vaihtuvanmittaisia, esityksenä attribuutin tunniste, arvon pituus ja itse arvo merkkeinä tai lukuna.

Esim. relaation R(A, B, C) monikko ('aaa', NULL, 'cc'):

| 301 |19| . . . |1|3|a|a|a|3|2|c|c|

- 301 = monikon järjestysnumero (lisäysjärjestyksessä)
- 19 = tietueen pituus
- seuraavana hallintatietoa (voisi olla esim. monikon poistomerkintä, tietueen tyyppi (jos eri tyyppejä))
- attribuuttien A ja C tunnisteet ovat 1 ja 3, pituudet 3 ja 2
- NULL-arvoa ei esitetä

Tietokantasovelluksen suorituskykyä voidaan arvioida kahdella tasolla:
- karkeasti levyhakujen lukumäärällä (hajautetussa tapauksessa verkon yli siirrettävän tiedon määrällä)
- ottamalla huomioon myös puskurien koot, levymuistin ominaisuudet, verkkoyhteyksien nopeus jne

Levyhakujen määrää käytettäessä tulokset ovat laitteistosta riippumattomia.

Levyhakujen maksimimäärä vai keskiarvo?
(vrt. tietorakennealgoritmien analyysi …)

Analyysi on selkeintä algoritmikohtaisesti; esimerkiksi N-jaksoisen tiedoston lajittelu (järjestäminen) vaatii enintään

$2N\lceil \log_M N \rceil$ levyhakua,

kun käytettävissä on M+1 jakson kokoista puskuria.

Puskuritilan määrä on yhteydessä myös siihen kohdistetun operaation suoritusnopeuteen (ja siirrännän organisointiin: erillinen i/o-prosessori vai ei, monet yksityiskohdat).

Esim. optimitapaus tiedoston läpiluvussa: i/o-prosessori täyttää puskureita samassa tahdissa kuin niitä prosessoidaan (esim. etsitään monikkoja 'where x = 'y').

Tiedosto-operaatioista

- haku; lisäys, poisto
  (puskurin sisältö ei muutu / muuttuu)

- tietuekohtaiset / tiedostokohtaiset
  'hae ainoa' 'käsittele kaikki'

- perusoperaatioita

open	avaa tiedosto
reset	aseta tietueosoitin alkuun
find	etsi seuraava ehdon täyttävä
get (read)	ota vuorossa oleva käsittelyyn (aseta tietueosoitin seuraavaan)
findnext	
delete	poista (t. poistomerkitse)
modify	muuta sisältöä
insert	hae paikka, lisää
close	sulje tiedosto

Moneen operaatioon liittyy eri tilanteita, esim.
- seuraava tietue on puskurissa / levyllä
- uuden tietueen paikka voi olla puskurissa/ levyllä, määritys voi olla monimutkaista

(tietuekohtaisen operaation suoritustilanne voi riippua vahvasti muista operaatioista!)

Tiedostotasolla on monia erilaisia tietojen organisointimahdollisuuksia:
- missä järjestyksessä monikot ovat levyllä
- mitkä monikot ovat lähekkäin (erityisesti samalla sivulla)
- mitä yhteyksiä tietueiden ja tiedostojen välillä ylläpidetään osoittimilla

Tiedoston (perustietojen eli tietueiden) organisoinnin lisäksi voidaan käyttää apurakenteita (saantimenetelmiä, hakemistoja), jotka
- nopeuttavat operaatioita
- vievät lisää levytilaa
- vaativat (omat?) puskurialueensa
- tekevät suorituskyvyn analysoinnin monimutkaisemmaksi

'Lopullinen' vaikeus tiedostorakenteen suunnittelussa ja suorituskyvyn arvioinnissa: sovellusten erilaiset käyttötavat ja niiden tärkeys (frekvenssi ym.):
- ei tiedetä tarpeeksi etukäteen
- sovellusten tarpeet keskenään ristiriitaisia

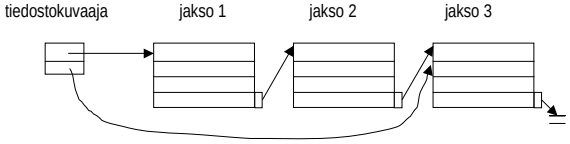
Yksinkertainen esimerkki:
- henkilötiedoston käyttö henkilöittäin (”muuta osoite”)
- kaikkien henkilöiden systemaattinen palkankorotus

Ja vielä: staattiset vs. dynaamiset tiedostot …

2.3 Järjestämätön peräkkäistiedosto (kasa)

Tietueet sijoitetaan peräkkäisille sivuille siinä järjestyksessä kuin ne muodostetaan.

Tiedostoon kuuluvien sivujen hallinta: esimerkiksi ketjurakenteena:



Relaatiotietokannan relaation oletusrakenne on yleensä kasa.
(Vrt. relaation määritelmä; entä yleiset käsittelytavat?)

Esim. Oracle:
create table employee (…)
luo tyhjän kasan.

insert into employee values (…)
sijoittaa uuden monikon kasan viimeiseen jaksoon.

Insert-operaatio on kasaan sovellettuna tehokas: tarvitaan yleensä vain kaksi (korkeintaan 3) levyhakua, kasan koosta riippumatta. Aikavaativuus on siis vakiokertaluokkaa: O(1).

Avainrajoitteen tarkistus vaatii kuitenkin pahimmassa tapauksessa kaikkien jaksojen lukemisen eli jopa N+2 levyhakua.

Tietueen haku vaatii keskimäärin N/2 levyhakua (kun tietue löytyy).

Tietueen poisto avaimen avulla: tietueen paikannus (haku), poisto (tai ainakin poistomerkintä) ja mahdollisesti muutetun sivun kirjoitus takaisin levylle.

Poisto-operaatio yleisemmässä tapauksessa, esim.

delete from employee where dno=5 :

```
for jokainen employee-taulun sivu P do
    B:=bufferfix(P);
    for jokainen B:n monikko e do
        if e.dno = 5 then
            merkitse e poistetuksi;
            aseta sivun B päivitysbitti ← 1
        endif;
    bufferunfix(B);
end.
```


Poisto vaatii pahimmassa tapauksessa 2N levyhakua: jokaisen sivun haun ja kirjoituksen.

Poistomerkintää käytettäessä tuhlataan tilaa - ja hakualgoritmin tulee ottaa merkinnät huomioon.

Vaihtoehto poistomerkinnöille:
- poistetaan tietue, vapautetaan tietueen paikka
  → jaksoihin jää tyhjiä paikkoja
  uudelleen käyttö mahdollista, mutta monimutkaista

Tietueen päivitys: haetaan puskuriin, muutetaan, kirjoitetaan (ei välttämättä heti) levylle samaan paikkaan.

Vaihtuvanmittainen tietue?
- ei mahdu välttämättä entiselle paikalle
  → poistetaan vanha, lisätään uusi kasan loppuun

=> On mahdollista, että kasa muuttuu harvaksi, jolloin se on (joskus) organisoitava uudelleen (reorganisation): luetaan kasan tietueet, viedään ne uuteen tyhjään kasaan ja hävitetään vanha kasa.

Uudelleenorganisointi on normaali toiminto tietokannan ylläpidossa:
- kallis (pyritään välttämään)
- ajankohtaa ei aina ole helppo löytää

Esimerkkejä.

1) Kyselyn select * from employee where ssn ='123456789' toteutus kasarakenteessa:

```
for jokainen employee-taulun sivu P do
    B:=bufferfix(P);
    for jokainen B:n monikko e do
        if e.ssn = '123456789' then tulosta e;
    bufferunfix(B);
end.
```

Kyselyn aikavaativuus on N levyhakua, jos ei ole ssn-avainrajoitetta.

Jos on ssn-avainrajoite, tarvitaan keskimäärin N/2 levyhakua, pahimmassa tapauksessa edelleen N.

2) Olkoon kasarakenteisessa tiedostossa n = 1 milj. tietuetta á 200 tavua, ja jakson koko 4KB (4096 tavua).

Jaksotuskerroin on $f = \lfloor 4096/200 \rfloor = 20$.

Jaksoon jää 'ylimääräistä' tilaa 96 tavua, mikä ehkä riittää jakson sisäiseen organisointiin linkeille ym.

Jaksoja on 1000 000 / 20 = 50 000 kpl. Jakson saantiajalla 10 ms tietueen haku vie enintään 50 000 * 10 ms = 8.3 min!

2.4 Järjestetty peräkkäistiedosto

(ordered file, sequential file)

Tietueet säilytetään tiedostossa (jaksoissa) jonkin järjestyskentän (järjestysavaimen) mukaan (nousevassa) järjestyksessä.

Esim. employee-relaatio järjestysavaimena name:

	NAME	SSN	BDATE	SALARY	...
jakso 1	Aaron, Ed	...	...	...	
	Abbott, Diane	...	...	...	
	...				
	Acosta, Marc	...	...	...	
jakso 2	Adams, John	...	...	...	
	Adams, Robin	...	...	...	
	...				
jakso 3	Allen, Sam	...	...	...	
	Andre, Marc	...	...	...	
	...				
jakso N	Wright, Pam	...	...	...	
	...				
	Zimmer, Byron	...	...	...	

Järjestys saattaa vastata intuitiivisesti joitakin käsittely- tai tulostustarpeita - mutta ei kaikkia …
Tiedosto voidaan järjestää vain yhden kentän mukaan kerrallaan.

Operaatioiden aikavaativuudesta:

· Koko tiedoston läpikäynti järjestysavaimen mukaisessa järjestyksessä hyvin nopeaa. Seuraavaan tietueeseen siirtyminen vaatii uuden levyhaun vain 1/f tapauksessa (f = jaksotuskerroin).

· Tietueen haku järjestysavaimen mukaan: voidaan käyttää binäärihakua (‘haarukointia’):

1. Etsi tietuetta keskimmäiseltä sivulta $\lfloor N/2 \rfloor$.

2. Jos ei löydy, etsi samalla periaatteella
   sivuilta 1, 2, … , $\lfloor N/2 \rfloor$ -1 (kun tutkitun sivun avaimet ovat suurempia kuin haettava avain),
   tai
   sivuilta $\lfloor N/2 \rfloor$ +1, … ,N (kun tutkitun sivun avaimet ovat pienempiä kuin haettava avain).

3. Haku päättyy tuloksettomana, kun
   (1) tutkittava tiedoston osa on yksi sivu eikä haettua tietuetta ole tällä sivulla,
   (2) tutkittava tiedoston osa on tyhjä.

Binäärihaun kustannus on enintään $\lceil \log_2 N \rceil$ levyhakua (kun järjestysavain on yksikäsitteinen).

Esim. (n = 1000 000) N = 50 000, $\lceil \log_2 N \rceil$ = 16 levyhakua,

saantiajalla 10 ms aikaa kuluu siis 0.16 s.

Muita ominaisuuksia:

· Haku erisuuruusehdon <, >, =>, <= perusteella on suhteellisen tehokasta: ehdon täyttävät tietueet ovat lähekkäin ja haku voidaan keskeyttää selaamatta koko tiedostoa läpi. (myös muunnettu binäärihaku ?)

· Haku muun kuin järjestysavaimen mukaan ei nopeudu kasaan verrattuna yhtään. (yksittäisen tietueen haku: mahdollisesti läpiluku; kaikki järjestyksessä -> ensin lajittelu hakuavaimen mukaan)

· Lisäys, poisto?
  Tiedoston tulee olla operaation jälkeen järjestyksessä.

Lisäys:
1. On löydettävä oikea lisäyskohta.
2. Jos jaksossa ei ole tilaa, on lisäyksen jälkeen siirrettävä keskimäärin puolet tietueista (yhden tietuepaikan verran eteenpäin).

Poisto: Poistettavan paikannus kuten lisäyksessä, poistomerkintää käytettäessä siirtoja ei tarvita.

Vaihtoehtoja lisäyksen vaatimille siirroille:
- Jätetään jaksoihin varatilaa lisäyksiä varten (muutamalle tietueelle).
- Linkitetään jaksoihin ylivuotojaksoja.

Vaihtoehdot lisäävät monimutkaisuutta.
Uudelleenorganisointi tulee joka tapauksessa jossain vaiheessa tarpeelliseksi.

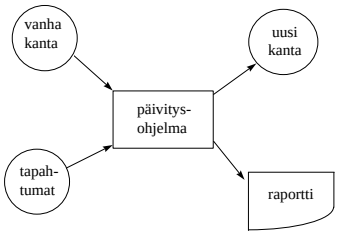
Järjestetty peräkkäistiedosto ei ole sellaisenaan yleinen tietokannan hallinnassa. Lisättäessä järjestetyn perustiedoston ‘päälle’ hakemistorakenne saadaan ns. IS-tiedosto, joka on hyvin tehokas myös yksittäisten hakuoperaatioiden kannalta (luku 3).

Järjestykseen perustuva käsittely on hyvin tehokasta silloin, kun se on mahdollista. Tyypillisiä tilanteita ovat esim.

· laajan aineiston läpikäynti yleensä
· usean järjestyksessä olevan tiedoston lomitus (mm. liitosoperaation toteutus)

Lomitus eli järjestysavaimen mukaan 'tahdistettu' käsittely on hyvin yleiskäyttöinen algoritmi.

Klassinen esimerkki:
(järjestetyn) peräkkäistiedoston (erä)päivitys:



Kantatiedosto järjestyksessä (esim. tilinumero), tapahtumat on järjestettävä ennen päivitysajoa samaan järjestykseen. Kunkin tiedostosta jokainen jakso käsitellään vain kerran. Tietue- tai jaksokohtaisia hakuja ei tarvita, vaan käsitellään aina 'seuraava jakso'.

Muita sovelluksia: duplikaattien poisto

2.5 Hajautukseen perustuva tiedostorakenne

- tavoite: saada yksittäinen tietue käsittelyyn nopeasti
→ tarvitaan suorasaantitiedosto (random access file)
-- hajatiedosto, hajautettu tiedostorakenne –

- suoraan levyosoitteilla?
nopein, mutta joustamaton

Hajautuksessa (hashing) tietue paikannetaan
'suoraan' (= selaamatta ympäristöä)
'epäsuorasti': ei levyosoitteella, vaan

säilyttämällä levyjaksojen osoitteita hajautusalueella.

Keskusmuistirakenteissa hajautuksen tavoite sama: selaamisen välttäminen. Levymuistia käytettäessä pyritään minimoimaan sivujen (levyjaksojen) lukemista; jakson sisältä voidaan etsiä selaamallakin …

Hajatiedosto jakaantuu loogisesti soluihin - oikeastaan solutiloihin (sankoihin; bucket, cell, slot).

Solu = yksi tai useampi jakso, rakenne yleensä kasa.

Tietueet jaetaan soluihin hajauttimen l. hajautusfunktion h avulla: solun kotiosoite ha saadaan kaavasta
ha = h(k),
missä k on hajautusavain, yleensä tietueen tunniste (tai sen osa).

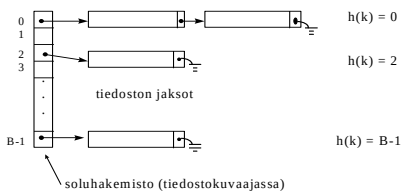
Hajautusavaimen arvoalue on yleensä paljon laajempi kuin hajautusalueen koko, ts. monella tietueella on sama kotiosoite.

> Keskusmuistirakenteissa tästä aiheutuvat yhteentörmäykset vaativat tavallaan välitöntä ratkaisua (ylivuotoalue, ketjutus jne).

Kotiosoitetta vastaavaan levyjaksoon mahtuu jokin määrä tietueita. Sen perään voidaan ketjuttaa ylivuotojaksoja (vrt. kasan ketjutettu toteutus).
Jaksorakenteen takia yhteentörmäyksellä ei ole aivan samaa merkitystä kuin keskusmuistihajautuksessa.

Hajautusavaimen arvolla k varustetun tietueen haku: lasketaan h(k) ja etsitään tietuetta kotiosoitteesta alkavasta jaksoketjusta kuten kasasta.

Hajautuksen optimitilanne: jokaisen ketjun pituus = 1.
Jos on aito ketju (enemmän kuin yksi jakso / kotiosoite), tässä **ei** toteudu aivan suorasaantitiedoston idea löytää tietue suoraan yhdellä levyhaulla. Toisaalta ero kasan tai järjestetyn tiedoston etsintään on selvä:
hajautuksessa ehkä keskimäärin 1.x levyhakua



Lisäys hajautettuun tiedostoon:
- lasketaan lisättävän tietueen kotiosoite kuten haussa
- lisätään tietue kuten kasaan

Poisto: paikannus kuten haussa, poistetaan tietue jaksosta joko poistomerkinnällä tai tiivistämällä samalla kasaa.

Hajauttimella on levymuistiin perustuvassa rakenteessa samat ominaisuudet kuin keskusmuistirakenteissa:
- avainarvot (tietueet) tulee hajauttaa mahdollisimman tasaisesti kotiosoitteisiin (osoiteavaruuteen)
- h(k) tulee voida laskea nopeasti (vähemmän tärkeä)

Yleinen hajauttimen muoto: h(k) = k mod B.

Jos k ei ole kokonaisluku, siitä voidaan muodostaa yksinkertaisella muokkauksella kokonaisluku (merkkien koodiarvojen summa, merkkien tai bittien poiminta arvon keskeltä tms.). (Eikä mod-operaatio ole ainoa tapa.)

Esimerkki suorituskyvystä:

Oletetaan, että N-sivuinen tiedosto on hajautettu tasaisesti B soluun. Soluhakemisto on luettu kokonaisuudessaan keskusmuistiin (esim. peräkkäinen asiakastietueiden käsittely asiakasnumeron perusteella, ei numerojärjestyksessä).

Tietueen lisäys vie enintään 3 levyhakua (kun ei ole avainrajoitetta).

Tietueen haku: enintään $\lceil N/B \rceil$ levyhakua.
Tietueen poisto: enintään $\lceil N/B \rceil + 1$ levyhakua.
Poisto-operaatio muulla kuin taulun avaimeen liittyvällä ehdolla voi vaatia $2\lceil N/B \rceil$ levyhakua.

Esim. N = 50000, B = 5000, saantiaika 10 ms
→ operaatiot maksimissaan 0.1-0.2 s.

Päivitysoperaatio: jos muutetaan muuta kuin avainkentän arvoa, muutettu tietue sijoitetaan samaan jaksoon; muutettaessa avainkentän arvoa tietue poistetaan kotisolustaan ja lisätään uuteen kotisoluunsa.

Parametrien yhteys:

- suuri hajautusalue (ja hyvä hajautin) → hyvä hajautus (lyhyet ketjut); voi olla paljon vajaita jaksoja
- pieni hajautusalue tai huono hajautin → pitkiä ketjuja

Parametrien valinnassa tarvittaisiin tietoa tiedoston dynaamisuudesta ja operaatioista. Yleensä pyritään siihen, että tiedoston luontivaiheessa
· jokainen solu olisi yksisivuinen (eikä aivan täynnä).

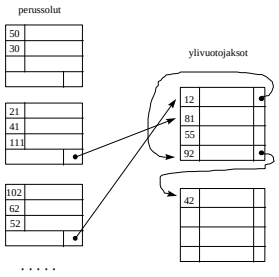
Hajautusrakenteen käyttö kyselyissä:
· (taulun) avaimen tarkkaan arvoon perustuva haku nopea
select * from employee where ssn = '123456789';

· muuhun arvoon perustuva haku yhtä tehoton kuin kasasta (kaikkien kotisolujen kaikki ketjut …)
select * from employee where name = 'Smith';

· likimääräiseen arvoon perustuva haku kuten kasasta
select * from employee where name like 'Sm%';
(vaikka name olisi avain)

Hajautusrakenne on luonteeltaan staattinen: solujen lukumäärä on kiinteä. Usein solujen tasapaino heikkenee vähitellen - koko tiedosto on organisoitava uudelleen (B:n muutos? hajauttimen muutos?).

Joskus käytetään yhteisiä ylivuotojaksoja useille (t. kaikille) soluille. Tällöin on luontevaa käyttää jakso-osoitteiden sijasta tietueosoitteita ylivuotojaksoissa:



Hajautuksen erikoistapauksia:

· täydellinen hajautin: ei samoja soluosoitteita eri avaimille
(ei ole mielekäs, kun jaksoon mahtuu useita tietueita; tasainen jakauma riittää)

· järjestyksen säilyttävä hajautin
sovellusten kannalta toivottava; saataisiin sekä nopea yksittäin haku että järjestyksessä tapahtuva käsittely samasta tiedostosta

Esim. (juoksevan) tilausnumeron alkuosa hajautusavaimena

· dynaamiset hajautusmenetelmät: muutetaan hajautusaluetta tai hajautinta
→ joustavuutta tiedoston kasvulle

Oracle:
· hajautus liittyy klusterointiin: yhden tai useamman taulun rivit säilytetään yhdessä
· hajautusfunktio voidaan antaa; oletus '… mod alkuluku'
· datalohkojen ketjutusta kehotetaan välttämään
· sopivuus:
- yhtäsuuruuskyselyt (koko avaimelle)
- tiedosto staattinen tai maksimikoko ennustettavissa