**SQL-OHJELMOIJAN MUISTILISTAN LIITE(DB2 V6R1)**

MIS = Matching Index Scan

MIA = Multiple Index Scan

Lause "... Matching Index Scan (MIS) -saantipolku ei ole mahdollinen" tarkoittaa, että kyseisissä tilanteissa päädytään joko taulun läpilukuun tai Non-Matching Index Scan -saantipolkuun.

## A. Yllättävät saantipolut

1. Käytä muuttujalle ja literaalille sarakkeen tietomuotoa ja pituutta.

   Sama sääntö pätee myös liitossarakkeille.

   Esimerkkeinä eri muodoista ovat:
   - ?? eri tietotyypit kuten INTEGER ja DECIMAL tai INTEGER ja SMALLINT
   - ?? eri pituudet kuten DECIMAL(7,2) ja DECIMAL(6,2)
   - ?? eri tarkkuudet DECIMAL(7,2) ja DECIMAL(7,3)

   Näissä tapauksissa optimoija ei yleensä voi hyödyntää Matching Index Scan -saantipolkua.

   MIS-saantipolku on mahdollinen
   - ?? jos CHAR-sarakemääritys on pidempi kuin CHAR-muotoinen isäntämuuttuja tai literaali
   - ?? yhtäsuuruusliitoksissa eripituisten ja –tyyppisten CHAR- ja VARCHAR –liitospredikaattien välillä

   Tämä tarkennus ei kuitenkaan kumoa pääsääntöä, koska näissäkin tilanteissa tapahtuu tarpeettomia tieto-konversioita.

   Muuttujien osalta näitä virheitä voi välttää käyttämällä aina **DCLGEN**-määrityksiä.

2. Vältä skalaarifunktioita WHERE-ehdossa niille hakemistosarakkeille, joita halutaan käyttää haussa. Esim. SUBSTR voidaan usein korvata BETWEEN:llä.

   Tyypillisiä skalaarifunktioita ovat SUBSTR, konkatenointi ja erilaiset tietomuodon muunnosfunktiot.

   Esimerkiksi SQL-lauseen

   ```
   MOVE  'A'        TO HV.

   SELECT           DEPTNAME
   FROM             DSN8610.DEPT
   WHERE            SUBSTR(DEPTNO,1,1) = :hv
   ```

   tapauksessa optimoija ei voi käyttää Matching Index Scan -saantipolkua DEPTNO-hakemiston osalta.

   Optimoija saadaan käyttämään Matching Index Scan-saantipolkua, jos skalaarifunktiota vastaava toimenpide siirretään SQL-lauseen ulkopuolelle tai korvataan tehokkaammalla SQL-lauseella.

   Esimerkin tapauksessa SUBSTR-skaalarifunktio voidaan korvata BETWEEN tai LIKE -rakenteella

   ```
   MOVE 'A'         TO ALKU.
   ```

```
MOVE 'A99'      TO LOPPU.

SELECT          DEPTNAME
FROM            DSN8610.DEPT
WHERE           DEPTNO BETWEEN :ALKU AND :LOPPU
```

Tai

```
SELECT          DEPTNAME
FROM            DSN8610.DEPT
WHERE           DEPTNO LIKE 'A%'
```

?? LIKE –ehdossa on huomioitava, ettei jokerimerkki saa olla ensimmäisenä merkkinä eikä ehtosarakkeelle määritelty FIELD PROC -rutiinia (Huom. FIELD PROC -rutiini on yleinen nimisarakkeiden yhteydessä).

?? avainsana: SCALA R FUNCTION

3. Vältä aritmeettisten lausekkeiden käyttöä WHERE-ehdossa. Laske arvo valmiiksi muuttujaan.

```
muuttuja + 3   <- ok      (non-column expression on indeksoituva)
sarake + 3     <- ei-ok   (column expression ei ole indeksoituva)
```

Mikäli column expression:a käytetään WHERE-ehdossa, niin optimoija ei voi vastaavien sarakkeiden osalta hyödyntää Matching Index Scan -saantipolkua.

Suositeltavin ja varmin tapa on laskea arvot valmiiksi muuttujaan ennen SQL-lausetta.

```
ADD 100         TO HV1.

SELECT          ACSTAFF
FROM            DSN8610.PROJACT
WHERE           PROJNO = :hv1
AND             ACTNO = :hv2
AND             ACSTDATE = :hv3
```

Hyväksyttävä tapa on myös laskea arvo muuttujaan SQL-lauseessa. Tässä tapauksessa hakemiston (PROJNO, ACTNO, ACSTDATE) osalta saantipolku on Matching Index Scan (MC=3).

```
SELECT          ACSTAFF
FROM            DSN8610.PROJACT
WHERE           PROJNO = :hv1 + 100
AND             ACTNO = :hv2
AND             ACSTDATE = :hv3
```

Huono ja kielletty tapa on muodostaa lauseke sarakkeen kanssa; tällöin hakupoluksi tulee non-matching index scan.

```
SELECT          ACSTAFF
FROM            DSN8610.PROJACT
WHERE           PROJNO – 100 = :hv1
AND             ACTNO = :hv2
AND             ACSTDATE = :hv3
```

4. Käytä ehtoparia >=, <=, kun arvovälikyselyssä vertaat muuttujaa kahteen sarakkeeseen. Esim: Verrattaessa saraketta kahteen muuttujaan, on BETWEEN yleensä yhtä tehokas kuin ehtopari >=,<=. Esim: WHERE COL BETWEEN :m1 AND :m2.

   Esimerkiksi

```
SELECT    PROJNO, PROJNAME
FROM      DSN8610.PROJ
WHERE     :TARKPVM >= PRSTDATE
AND       :TARKPVM <= PRENDATE
```

   Oletetaan, että hakemisto (PRSTDATE,PRENDATE) mahdollistaisi Matching Index Scan -saantipolun.

   Vastaava erittäin huono ja vaarallinen BETWEEN-rakenne

```
….
WHERE     :TARKPVM BETWEEN  PRSTDATE
          AND PRENDATE
```

   Tämä on tason 2 (STAGE 2) -ehto ja sulkee siten Matching Index Scan -saantipolun pois.

   Verrattaessa saraketta kahteen muuttujaan, BETWEEN ja ehtopari >= , <= ovat useimmiten yhtä tehokkaita. Ainoana erona on optimoijan käyttämät oletusarvot sen laskiessa läpäisykertoimia. Niiden läpäisykertoimet hieman eroavat toisistaan, jolloin tapauskohtaisesti toinen voi olla indeksoituva, toinen ei.

   ?? avainsana: FILTER FACTOR

5. Vältä negaatiota WHERE-ehdossa niille hakemistosarakkeille, joita halutaan käyttää haussa.

   Jos hakemistosarakkeelle käytetään WHERE-ehdossa esimerkiksi NOT BETWEEN tai NOT IN -predikaattia, optimoija ei valitse saantipoluksi Matching Index Scania.

   Esimerkki huonosta tavasta:

```
SELECT    LASTNAME,
          EMPNO
FROM      DSN8610.EMP
WHERE     EMPNO NOT BETWEEN '000100' AND '000350'
```

   SQL-lauseen saantipoluksi tulee Table Space Scan tai Non-Matching Index Scan tulosjoukon ja taulukon rivimäärän suhteesta riippuen.

   NOT BETWEEN voidaan korvata kahdella lauseella tai jos vastausjoukko on pieni, UNION:lla seuraavasti:

```
SELECT    LASTNAME, EMPNO
FROM      DSN8610.EMP
WHERE     EMPNO < '000100'
UNION ALL
SELECT    LASTNAME, EMPNO
```

```
FROM        DSN8610.EMP
WHERE       EMPNO > '000350'
```

SELECT-lauseiden saantipoluksi tulee Matching Index Scan, jos ehdon täyttävien rivien lukumäärä on pieni suhteessa taulukon rivimäärään.

NOT IN -predikaatti on kierrettävissä vain muuttamalla se IN-listaksi.

6. Muista, että alikyselyn kirjoitustapa määrää suoritustavan ja –järjestyksen.

Alikyselyitä on kahdenlaisia: korreloitu ja korreloimaton.

KORRELOITU alikysely viittaa ainakin yhteen isä lauseensarakkeeseen ja sen suoritus tapahtuu ylhäältä alaspäin seuraavasti: Ensin suoritetaan uloin isälause ja jokaisen ehdokasrivin kohdalla suoritetaan alikysely, jonka ehtolauseessa viitataan em. ehdokasriviin eli alikysely suoritetaan yhtä monta kertaa kuin isälauseen ehdokasrivejä on.

Jos SQL-lauseessa on sisäkkäisiä alikyselyitä, jatko tapahtuu seuraavasti: Uloimman alikyselyn jokaisen ehdokasrivin kohdalla suoritetaan seuraava alikysely, jonka ehtolauseessa taas viitataan em. ehdokasriviin ja näin edetään, kunnes sisin alikysely on suoritettu. Uloimman isälauseen jokaisen ehdokasrivin kohdalla saatetaan suorittaa kukin alikysely kertaalleen.

Esimerkki:

```
SELECT EMPNO,                                   <== isälause
       LASTNAME,
       WORKDEPT,
       EDLEVEL
FROM   DSN8610.EMP CURRENT_ROW
WHERE  EDLEVEL >
      (SELECT AVG(EDLEVEL)    <==alikysely
       FROM   DSN8610.EMP                       (suoritus n kertaa)
       WHERE WORKDEPT = CURRENT_ROW.WORKDEPT)
```

?? jos WORKDEPT:n keskimääräinen EDLEVEL on laskettu jo aiemmin, alikyselyä ei suoriteta uudelleen

KORRELOIMATON alikysely on riippumaton isälauseesta ja sen suoritus tapahtuu alhaalta ylöspäin seuraavasti: Ensin suoritetaan kertaalleen sisin alikysely (kohdistimen avausvaiheessa) ja sen arvoa tai arvojoukkoa verrataan alikyselyn isälauseen jokaiseen ehdokasriviin.

Jos SQL-lauseessa on sisäkkäisiä alikyselyitä, jatko tapahtuu seuraavasti: Sisimmästä isä/alikyselyparista saatua arvoa tai arvo-joukkoa verrataan edelleen seuraavan isälauseen tulosjoukkoon jne, kunnes uloin isälause on suoritettu.

Esimerkki:

```
SELECT  EMPNO,                                  <== isälause
        LASTNAME,
        WORKDEPT,
        EDLEVEL
FROM    DSN8610.EMP
WHERE   EDLEVEL >
```

```
(SELECT AVG(EDLEVEL)  <== alikysely
FROM    DSN8610.EMP)  (suoritus kerran)
```

Jos alikyselyn tuloksena saatu arvojoukko on suuri, SQL-lauseen suoritus saattaa olla raskas. Korreloimattoman alikyselyn arvojoukko lajitellaan ja talletetaan työtaulukkoon, joka luetaan läpi isälauseen jokaisen ehdokasrivin kohdalla erityisen karkean hakemiston (sparse index) kautta.

7. Käytä OPTIMIZE FOR n ROWS, kun ohjelma ei lue koko tulosjoukkoa. Jos SELECT-lauseen tuloksena syntyy suuri tulosjoukko, mutta ohjelma tarvitsee siitä vain pienen osan, SELECT-lauseeseen voidaan lisätä: OPTIMIZE FOR n ROWS. Lisäyksen jälkeen optimoija saattaa valita ohjelman kannalta tehokkaamman saantipolun.

Esimerkki:

```
SELECT      LASTNAME, EMPNO, WORKDEPT
FROM        DSN8610.EMP
WHERE       WORKDEPT LIKE 'D%'
ORDER BY WORKDEPT
```

?? oletetaan, että WORKDEPT ei ole C-hakemisto

SELECT-lauseen saantipoluksi todennäköisesti tulee LIST PREFETCH (Matching Index Scan + PREFETCH table space scan through a page list) eli kohdistimen avausvaiheessa hakemistosta luetaan kaikkien ehdon täyttävien rivien RID-osoitteet, jotka lajitellaan ja rivit luetaan työtaulukkoon sekä lajitellaan. (Kun lajittelun hinta on versioiden myötä halventunut, niin niitä tulee entistä herkemmin.)

Kun SELECT-lauseeseen lisätään OPTIMIZE FOR n ROWS seuraavan esimerkin mukaan

```
SELECT      LASTNAME, EMPNO, WORKDEPT
FROM        DSN8610.EMP
WHERE       WORKDEPT LIKE 'D%'
ORDER BY WORKDEPT
OPTIMIZE FOR 2 ROWS
```

SELECT-lauseen saantipoluksi tullee Matching Index Scan (ei lajitteluja).

Hajautetussa ympäristössä OPTIMIZE FOR N ROWS vaikuttaa blokin kokoon. Jos teet tätä client-server –järjestelmässä, katso kohta H Client-server.

8. Muista, että OR-rakenteesta, jota ei voi konvertoida IN-listaksi (esim.WHERE x < ...OR x > ...), seuraa parhaimmillaankin MIA-saantipolku.

Jos hakemistosarakkeelle käytetään WHERE-ehdossa OR-predikaattia, Matching Index Scan ei ole mahdollinen (optimoija valitsee usein saantipoluksi Table Space Scanin tai Non-Matching Index Scanin). Parhaimmillaan OR-rakenne tuottaa Multiple Index Access -saantipolun, josta on muistettava, ettei Index Only ole mahdollinen.

Esimerkiksi

```
SELECT      LASTNAME, EMPNO
FROM        DSN8610.EMP
WHERE       EMPNO < '000100' OR EMPNO > '000350'
```

OR voidaan korvata UNION:lla seuraavasti:

```
SELECT      LASTNAME, EMPNO
FROM        DSN8610.EMP
WHERE       EMPNO < '000100'
UNION ALL
SELECT      LASTNAME, EMPNO
FROM        DSN8610.EMP
WHERE       EMPNO > '000350'
```

SELECT-lauseiden saantipoluksi tulee Matching Index Scan, jos ehdon täyttävien rivien lukumäärä on pieni suhteessa taulukon rivimäärään.

9. Jos DB2 yhden rivin haussa INDEX ONLY -hakupolun sijaan lukee tiedot taulusta, lisää SELECT-listaan, jos mahdollista, sarakefunktio.

DB2:n mielestä WHERE-ehtoihin täysin täsmäävä unique –indeksi on aina muita parempi jopa siinä määrin, ettei se tarkastakaan muita indeksivaihtoehtoja.

Esimerkiksi

Indeksit IX1 = C1
IX2 = C1, C2

```
Lause     SELECT      C2
          FROM        T1
          WHERE       C1 = :hv

Korjaus   SELECT      MIN(C2)
          FROM        T1
          WHERE       C1 = :hv
```

10. Tarkista MIN- ja MAX-funktioiden hakupolku EXPLAIN:sta.

Yhden rivin INDEX ONLY –haku on tehokkain hakupolku. MIN- ja MAX-funktioiden kanssa se on mahdollinen silloin, kun seuraavat edellytykset on voimassa:

- ?? Käsitellään vain yhtä taulua.
- ?? Lauseessa on vain yksi sarakefunktio (MIN tai MAX).
- ?? Kaikki predikaatit ovat täsmääviä predikaatteja.
- ?? Ei käytetä GROUP BY -funktiota.
- ?? MIN -funktiolle on kasvavassa, MAX -funktiolle laskevassa järjestyksessä oleva indeksi.
- ?? Sarakefunktio kohdistuu:
  - ?? Ensimmäiseen indeksisarakkeeseen, jos predikaatteja ei ole
  - ?? Viimeiseen match-sarakkeeseen, jos viimeinen täsmäävä predikaatti on arvovälipredikaatti.
  - ?? Viimeistä match-saraketta seuraavaan sarakkeeseen, jos kaikki täsmäävät predikaatit ovat yhtäsuuruuspredikaatteja.

11. Jos NOT NULL –saraketta verrataan alikyselyn tulokseen, joka voi saada NULL-arvon, käytä alikyselyssä COALESCE (VALUE) –funktiota.

Em. funktiolla varmistetaan, että alikysely ei palauta NULL -arvoa silloin, kun alikyselyn tulosta verrataan NOT NULL -sarakkeeseen.

Esimerkiksi

```
CREATE TABLE T1
      (S1 INTEGER NOT NULL,
       S2 INTEGER);
CREATE INDEX X1 ON T1
      S1 ASC);
CREATE INDEX X2 ON T1
      S2 ASC);
```

S1:lle ei voida käyttää X1:stä, koska alikysely voi palauttaa NULL -arvon.

```
SELECT .....
FROM          T1
WHERE         S1 =
              (SELECT     MIN(S1)
              FROM        T1
              WHERE ….                  );
```

S2:lle voidaan käyttää X2:sta, koska S2 voi saada NULL –arvoja.

```
SELECT .....
FROM          T1
WHERE         S2 =
              (SELECT     MIN(S2)
              FROM        T1
              WHERE ….                  );
```

Viimeisessä vaihtoehdossa S1:lle voidaan käyttää X1:stä, koska VALUE -funktio palauttaa aina NOT NULL -arvon (999999 on arvo, joka ei esiinny kannassa).

```
SELECT .....
FROM          T1
WHERE         S1 =
              (SELECT     VALUE(MIN(S2),999999)
              FROM        T1
              WHERE ….                  );
```

12. Älä tutki isäntämuuttujien arvoa WHERE -predikaatilla.

Samoin vältä turhien WHERE –ehtojen asettamista; kaikkea ei tarvitse laittaa SQL-lauseeseen.

Turha ehto

```
….
WHERE         :hv1 = :hv2
AND           NOT :hv1 = SPACE
```

Toinen turha ehto

```
….
WHERE     sar = :hv1
AND       sar > 0
```

Siirrä em. blankkotiedon tarkistus logiikkaan eli pois SQL-lauseesta. SQL-lause suoritetaan vain tarvittaessa.

**B. Turhat lajittelut**

1. Käytä ORDER BY –listassa hakemiston järjestystä, jos mahdollista.

   Jos DB2 päättää käyttää ORDER BY –listan mukaista hakemistoa, haetaan rivit oikeassa järjestyksessä eikä tulosjoukkoa lajitella.

   Seuraavan kyselyn tulosjoukko halutaan käsitellä järjestyksessä (sarake1, sarake2). DB2 voi käyttää rivien järjestämiseen sellaisia hakemistoja, joiden ensimmäisinä sarakkeina ovat lajittelusarakkeet. Kelvollisia hakemistoja ovat esim. (sarake1, sarake2) tai (sarake1, sarake2, sarake3):

   ```
   SELECT      sarake1, sarake2, sarake3
   FROM        taulu
   WHERE       sarake1 BETWEEN :hv1 AND :hv2
   ORDER BY    sarake1, sarake2
   ```

   Huomaa kuitenkin, että LIST PREFETCH -hakua käytettäessä ORDER BY aiheuttaa tulosrivien lajittelun aina. MIA sekä peräkkäishaku muun kuin cluster -hakemiston mukaan käynnistävät LIST PREFETCH -haun.

2. Muista, että OPEN CURSOR –lauseen suoritusvaiheessa kaikki hakuehdon täyttävät rivit lajitellaan tilapäistaulukkoon, mikäli tarvittavaa järjestystä ei saada hakemiston avulla.

   Kohdistinkäsittelyssä DB2 pyrkii hakemaan rivit taulusta yksi kerrallaan.

   Kuitenkin jos lajittelua (ORDER BY, GROUP BY, DISTINCT, UNION, liitos) ei voida suorittaa hakemiston avulla, tai jos optimoija päättää valita saantipolun, jossa on lajittelu, kaikki hakuehdon täyttävät rivit luetaan tilapäistaulukkoon OPEN CURSOR -lausetta suoritettaessa. Myös korreloimattoman alikyselyn tulosjoukko lajitellaan kohdistimen avauksessa. Jos tulosjoukko on suuri, tosiaikaohjelmissa näytön haku kestää kohtuuttoman kauan.

   Lajittelun poistamiseksi voidaan harkita indeksin lisäämistä tai lajittelujärjestyksen muuttamista, lajittelun pienentämiseksi vastausjoukon rajoittamista lisäehdoilla. Jos optimoija on valinnut huonon hakupolun, OPTIMIZE FOR n ROWS saattaa auttaa.

   Eräohjelmissa lajittelu voidaan ehkä sallia,jos ohjelma käsittelee kaikki hakuehdon täyttävät rivit. Kohdistin määritellään WITH HOLD -optiolla, jotta sen positio säilyy COMMIT-pisteen yli eikä aineistoa lajitella moneen kertaan. On kuitenkin huolehdittava siitä, että COMMIT-väli ei ison lajittelun vuoksi veny liian pitkäksi.

   Jos taululla ei ole sopivaa hakemistoa (sarake1, sarake2), toimii taulun selaus seuraavasti:

   ```
   DECLARE     kursori1 CURSOR FOR
   ```

```
SELECT      sarake1, sarake2
FROM        taulu
WHERE       sarake1 > :hv1
ORDER BY    sarake1, sarake2

OPEN        kursori1
```

?? kaikki ehdon täyttävät rivit luetaan aputaulukkoon sekä lajitellaan kohdistimen avauksessa!

```
FETCH       kursori1
INTO        :sarake1, :sarake2
```

?? FETCH hakee rivin kerrallaan kohdistimen avauksessa muodostetusta aputaulukosta

3. Älä käytä DISTINCT –määrettä, jos se ei ole tarpeellista.

SELECT DISTINCT aiheuttaa lajittelun duplikaattirivien poistamiseksi.

Funktiokyselyssä DISTINCT ei aiheuta lajittelua, jos käsiteltävä sarake on minkä tahansa hakemiston ensimmäinen sarake. Seuraava kysely voi käyttää esim. hakemistoa (sarake1, sarake2, sarake3):

```
SELECT COUNT (DISTINCT sarake1)
FROM taulu
```

4. Määrittele yhdiste UNION ALL, paitsi, jos duplikaattien poisto on välttämätöntä.

UNION aiheuttaa aina tulosjoukon lajittelun duplikaattirivien poistamiseksi, myös silloin, kun tulosjoukot ovat toisensa poissulkevia eikä duplikaatteja ole.

Kun yhdiste määritellään UNION ALL, duplikaattirivejä ei poisteta. Duplikaattirivit eivät kuitenkaan ole tulosjoukossa peräkkäin.

Kummankin UNION –tavan yhteydessä ORDER BY aiheuttaa aina lajittelun.

5. Huomioi, että ylimääräisen, hakemistosta puuttuvan sarakkeen lisääminen lajittelutekijäksi aiheuttaa lajittelun, vaikka sarakkeeseen olisi voimassa yhtäsuuruusehto.

Esimerkki:

Indeksi A, B
ja tehdään lause

```
…..
WHERE       A = :hv
AND         B > :hv
AND         C = :hv
```

OK order by -lauseet

```
ORDER BY    B
ORDER BY    A, B
```

Huono order by -lause

```
ORDER BY    A, B, C
```

6. Jos yhdistelyssä (JOIN) lajittelutekijät kohdistuvat useampaan kuin yhteen tauluun tai sisempään tauluun, DB2 voi tehdä turhan lajittelun.

Hakemistot:
T1: sar1, sar2
T2: sar3

```
SELECT      A.sar1, A.sar2, Bsar3, B.sar4
FROM        T1 A, T2 B
WHERE       ….
ORDER BY    A.sar1, A.sar2, B.sar3
```

Tämä ORDER BY aiheuttaa lajittelun. Jos lajittelu on ongelma tässä tilanteessa, harkitse kursorin hajoittamista kahdeksi SQL-lauseeksi.

## C. CPU-ajan säästö

1. Massalisäyksessä LOAD-apuohjelma on tehokkaampi kuin INSERT-lause. Huomaa, ettei taulut / kanta ole käytettävissä LOADin aikana (24 h käytettävyys).

LOAD-apuohjelma lisää rivit aina taulukon loppuun saapumisjärjestyksessä. Tämän vuoksi on suositeltavaa lajitella rivit clustering-avaimen mukaiseen järjestykseen.

Tämän vuoksi input-tiedoston rivit on lajiteltava CLUSTER -avaimen mukaiseen järjestykseen ennen latausta. Aineiston latauksessa käytetään parametria LOG NO (lokia ei kirjoiteta) ja taulukko varmuuskopioidaan. Latauksen aikana muut työt eivät voi käyttää ko. taulukkoa. CPU-säästön lisäksi ohjelmointityö jää pois.

INSERT:ä käytettäessä DB2 pyrkii lisäämään rivit ensisijaisesti sivulle, jossa on lisättävän rivin clustering-avainta edeltävä arvo ja tähän lisäysmekanismiin kuluu enemmän CPU-aikaa kuin LOAD-apuohjelman lisäysmekanismiin. INSERTiä käytettäessä lisättävät rivit talletetaan aina lokille. INSERT-lause (päinvastoin kuin LOAD-apuohjelma) ei huomioi PCTFREE ja FREEPAGE -parametreja.

2. Jokerimerkki (%’, ’_’) ensimmäisenä merkkinä LIKE-ehdossa sulkee pois MIS-saantipolun. Sen käyttö on silti tehokkaampaa kuin rivien luku FETCH:llä ja ehdon tarkistaminen ohjelmassa.

Jos hakemistosarakkeen ensimmäisenä merkkinä LIKE-ehdossa käytetään '%' tai '_' -merkkiä, optimoija ei valitse saantipoluksi Matching Index Scania.

Esimerkiksi:

```
SELECT      EMPNO, LASTNAME, WORKDEPT
FROM        DSN8610.EMP
WHERE       WORKDEPT LIKE '%1'      <<--- Tablespace Scan
```

Jos ohjelma kuitenkin tarvitsee kaikki oheisen ehdon käyttävät rivit, LIKE:n käyttö on tehokkaampaa kuin kaikkien edon täyttävien rivien haku FETCH:llä ja ehdon tarkistaminen ohjelmassa.

```
SELECT      EMPNO, LASTNAME, WORKDEPT
FROM        DSN8610.EMP
WHERE       WORKDEPT LIKE  'A%'  <<---Matching Index Scan
```

Jos haettava merkkijono sisältää '%' tai '_'-merkkejä, ne voidaan hakea ESCAPE-määreen avulla.

Esimerkki:

```
SELECT    sarake1
FROM      taulu
WHERE     sarake2 LIKE '10+% KOROTUS%'
          ESCAPE '+'
```

?? lause hakee merkeillä '10% KOROTUS' alkavan jonon

3. Määrittele kohdistin eräajoissa WITH HOLD –optiolla.

Jos eräohjelmassa käytetään kohdistimen määrittelylauseessa WITH HOLD -määrettä, COMMIT/CHECKPOINT-lause voidaan suorittaa kohdistimen pysyessä koko ajan avoimena, jolloin voidaan jatkaa FETCH-lauseella. WITH HOLD -määrettä ei voi käyttää CICSin valekeskustelevissa (=tosiaikaisissa) ohjelmissa.

4. Käytä olemassaolon tarkistukseen **KOHDISTIN**ta, jos tulosjoukko voi olla suurempi kuin yksi rivi.

Muita hyviä tapoja ovat
?? viite-eheyden hyväksikäyttö eli uskotaan viite-eheyden hoitavan asiaan (lapsirivin lisäys, isärivin poisto)
?? EXISTS

Huonoja tapoja ovat
?? Suora SELECT
?? SELECT COUNT(*)

Suora SELECT hakee kaikki ehdon täyttävät rivit ja vasta sen jälkeen ilmoittaa liian monesta rivistä.

Vaikka EXISTS on mainittu hyväksi, huomioi, että muiden hakuehtojen on oltava tehokkaat.

5. Luettele SELECT –lauseessa vain ne sarakkeet, joiden tietoa tarvitset. WHERE –ehtojen sarakkeita ei pidä turhaan toistaa.

Vertaa myös kohta G1 / ORDER BY –sarakkeet.

?? ei esimerkkiä

6. Vältä UPDATE –lauseessa tietoarvoltaan muuttumattomien sarakkeiden luetteleminen. Ole tarkka varsinkin avain-, viiteavain- ja muiden hakemistosarakkeiden kohdalla.
?? ei esimerkkiä

7. Vältä turhien SQL-lauseiden suorittamista.

Lue pienet, usein käytettävät kooditaulut muistialueelle, älä lue näitä useaan kertaan ohjelmassa.

Esimerkiksi tämä on **huono tapa** hakea tai päivittää päivämäärää

```
SELECT      CURRENT DATE
FROM        SYSIBM.SYSDUMMY1
```

**Oikea tapa** suorittaa nykyhetken päivitys, kun et tarvitse jatkossa saadun nykyhetken arvoa

```
UPDATE      taulu
SET         sar = CURRENT DATE
….
```

**Oikea tapa** suorittaa nykyhetken päivitys, kun tarvitset jatkossa saadun nykyhetken arvon. Ensin asetat nykyhetken isäntämuuttujaan ja sen jälkeen päivität saraketta samaisen isäntämuuttujan arvolla.

```
SET         :hv1 = CURRENT DATE

UPDATE      taulu
SET         sar = :hv1
….
```

V6:n myötä on päivämäärien laskenta helpottuu

```
SET :HV1 = CURRENT DATE + 1 MONTH
```

tai

```
VALUES(CURRENT DATE + 1 MONTH) INTO :HV1
```

## D. Turvallisuus

1. Käytä rivien päivityksessä ja poistossa kohdistinta. Poikkeus: käsitellään vain yhtä riviä, jonka perusavain tunnetaan.

Kohdistinta käyttämällä varmistat, että päivitykset ja poistot kohdistuvat vain haluttuihin riveihin. Kohdistimen käyttämistä puoltavat lisäksi lukitusnäkökohdat (katso F1).

```
DECLARE CURSOR kohdistin1 CURSOR FOR
SELECT      sarake1, sarake2, sarake3
FROM        taulu1
WHERE       sarake1 = :hv1
FOR UPDATE OF   sarake3

.....

UPDATE      taulu1
SET         sarake3 = :hv3
WHERE CURRENT OF kohdistin1
```

Poikkeuksena pääsääntöön on tilanne, jossa käsitellään yhtä riviä, jonka perusavain tunnetaan. Tällöin voidaan käyttää suoraa päivitystä.

```
UPDATE      taulu1
SET         sarake3 = :hv3
WHERE       perusavain = :hvpk
```

Kursorin käyttö päivityksissä on jonkin verran raskaampaa kuin suoran UPDATE -lauseeen käyttö.

Huom: Kursoripäivitystä ei voida tehdä, jos tulostaulu on read-only. Esim. ORDER BY, UNION, UNION ALL, DISTINCT, sarakefunktiot ja liitos aiheuttavat sen, että tulostaulua voidaan vain lukea.

2. Jos käytät näkymän määrittelyssä WHERE –ehtoja ja näkymää käytetään rivien päivitykseen / lisäykseen, määrittele näkymä WITH CHECK OPTION.

WITH CHECK OPTION -määre takaa, että lisätyt ja päivitetyt rivit 'näkyvät' ao. näkymän kautta. Tämä vähentää osaltaan virheiden määrää.

Esimerkiksi

```
CREATE VIEW DSN8610.VPROJ01
      (PROJNO,PROJNAME,PROJDEP,RESPEMP)
 AS SELECT
      PROJNO,PROJNAME,PROJDEP,RESPEMP
      WHERE          PROJDEP LIKE 'AA%'
      WITH CHECK OPTION
```

Esimerkin tapauksessa WITH CHECK OPTION varmistaa, että näkymän kautta lisätyille ja päivitetyille riveille pätee aina ehto PROJDEP LIKE 'AA%'.

WITH CHECK OPTION määrettä ei voi käyttää, jos näkymä ei ole päivitettävä (eli se on ns. 'read-only view' ) tai jos näkymä sisältää alikyselyn.

3. Käytä aina **:** -merkkiä isäntämuuttujan edessä. DB2 V6:ssa se on pakollinen ja vanhat merkinsäästöt on korjattava sitä ennen.

?? ei esimerkkiä

## E. Selkeys ja ylläpidettävyys

1. Älä käytä SELECT * -lausetta.

Jos SELECT-lauseeseen kirjoitetaan aina näkyviin halutut sarakkeet (ei siis * -määrettä), niin SELECT-lauseen rakenne on riippumattomampi taulun rakenteesta. Tämä ns. tietoriippumattomuus palvelee taulun ylläpidettävyyttä.

Lisäksi SELECT-lause on tällöin itsedokumentoituva eli sarakkeiden ja vastaavien ohjelmamuuttujien vastaavuus näkyy suoraan SELECT-lauseesta.

Tarpeettomien sarakkeiden määritteleminen kuluttaa ylimääräistä CPU-aikaa ja saattaa tietyissä tilanteissa aiheuttaa myös tarpeettomia taululukuja.

2. Kirjoita SELECT –lauseeseen vain ne sarakkeet, joita todella tarvitset.

Katso kohta E1 eli edellinen kohta.

3. Luettele INSERT-lauseessa ne näkymän sarakkeet, joihin viet tietoa.

   Tällä takaat, että

   ?? INSERT-lause on itsedokumentoituva.
   ?? Syötettävien tietojen vastaavuus sarakejärjestyksen kanssa voidaan tarkistaa suoraan INSERT-lauseesta.
   ?? INSERT-lause on riippumaton taulun rakenteesta.

4. Määrittele FOR UPDATE OF –sarakelistaan vain ne sarakkeet, joita aiot päivittää. Jos rivejä poistetaan, luettele perusavaimen sarakkeet.

   Kun määrittelet päivittävää kohdistinta, luettele FOR UPDATE OF -sarakelistassa ainoastaan päivitettävät sarakkeet. SELECT-listassa päivitettäviä sarakkeita ei tarvitse luetella. Kun taulun perusavain on (sarake1) ja saraketta 3 päivitetään:

```
DECLARE CURSOR kursori1 CURSOR FOR
SELECT          sarake1, sarake2
FROM            taulu
WHERE           sarake1 = :w1
FOR UPDATE OF   sarake3
```

   Jos aiot poistaa rivejä, luettele FOR UPDATE -listassa perusavaimen sarakkeet:

```
DECLARE CURSOR kursori2 CURSOR FOR
SELECT          sarake1, sarake2
FROM            taulu
WHERE           sarake1 = :w1
FOR UPDATE OF sarake1
```

5. Käytä mieluummin IN-listaa usean OR-ehdon sijaan.

   DB2 konvertoi peräkkäiset OR-ehdot automaattisesti IN-listaksi. IN-lista on kuitenkin selkeämpi.

```
SELECT          sarake1, sarake2
FROM            taulu
WHERE           sarake1 IN ('A1', 'B2', 'C3')
```

6. Käytä isäntämuuttujina DCLGEN-struktuureja.

   DCLGEN muodostaa taulun sarakkeista oikeantyyppiset ja -pituiset ohjelmamuuttujat.DCLGENillä saadaan ohjelmaan myös sarakkeille määritellyt LABEL-kuvaukset, jotka lisäävät ohjelman selkeyttä ja ylläpidettävyyttä.

7. Luettele ORDER BY –sarakelistassa sarakkeet nimeltä.

   Sarakkeiden nimeäminen ORDER BY -listassa lisää ohjelman selkeyttä ja turvallisuutta erityisesti ohjelmamuutosten yhteydessä.

   On kuitenkin tilanteita, joissa sarake on ‘nimetön’:

?? SQL-lause on yhdiste (UNION, UNION ALL). Tällöin kaikki tulostaulun sarakkeet ovat 'nimettömiä'.
?? Lajittelusarake on vakio tai lauseke.
?? Lajittelusarakeeseen käytetään sarakefunktiota.
?? Vakiot

Nimettömille sarakkeille voi antaa nimen AS –lauseella. Jos tätä ei halua tehdä, on sarakkeeseen viitattava järjestysnumerolla. Suositellaan AS –lauseen käyttöä, numeron käyttö on huono tapa koodata.

Katso myös G1.

8. Sarakkeen nimen on hyvä olla kuvaava. Näe samalla kertaa vaivaa ja kommentoi sekä anna otsakenimi (LABEL) kullekin sarakkeelle. Käytä samasta tiedosta aina samaa nimeä, jolloin DB2-katalogi toimii ikäänkuin köyhän miehen tietohakemistona.

?? ei esimerkkiä

## F. Lukitukset

1. Auki olevan **LUKEVAN** kursorin kohteena olevien taulujen päivittäminen kursorin ulkopuolelta samassa yhteydessä (UOW) voi häiritä kursorin etenemistä. Rivi on päivitystä varten luettava SELECT FOR UPDATE –optiolla. Ohjeen noudattamatta jättämisellä voit saada aikaan ns. kengurukursorin. Tällöin päivitykset ’siirtävät’ rivin paikkaa. Oireina ovat mm. hypyt rivien yli, saman rivin uudelleen luku, pahimmillaan ikikieriö (LOOP).

Jos päivität tai poistat selattavia rivejä, käytä SELECT FOR UPDATE –kohdistinta lukkiutumien välttämiseksi.

Käytä päivittävää kohdistinta silloinkin, kun luet vain yhden rivin, joka mahdollisesti päivitetään tai poistetaan. Kun PLAN tai PACKAGE on sidottu (BIND) ISOLATION-parametrilla CS, päivittävä kohdistin lukitsee selattavan sivun U-lukolla, joka estää muut samanaikaiset päivitykset. Päivitystä ja poistoa varten lukko muutetaan X-lukoksi.

SELECT FOR UPDATE -kohdistinta käytettäessä poisto ja päivitys tehdään aina rivi kerrallaan. Ja lukot voidaan vapauttaa sopivin väliajoin COMMIT:lla. Sen sijaan päivitys- tai poistolause ilman kohdistinta voi yhdellä kertaa päivittää/poistaa suuren joukon rivejä ja pitää suuren määrän sivuja lukossa liian pitkään.

```
DECLARE CURSOR kursori1 CURSOR FOR
SELECT          sarake1, sarake2
FROM            taulu
WHERE           sarake1 = :hv1
FOR UPDATE OF   sarake3
...

UPDATE          taulu1
SET             sarake3 = :hv3
WHERE CURRENT OF kursori1
```

Huom: Jos kohdistimen määrittelemä tulostaulu on read-only, ei päivittävää kohdistinta voida käyttää. Esim. ORDER BY, DISTINCT, UNION, UNION ALL, sarakefunktiot ja liitos aiheuttavat sen, että tulostaulua voi vain lukea.

?? avainsana: UNPREDICTABLE

2. Samanaikaisesti tosiaikaohjelmien kanssa suoritettavissa eräohjelmissa voi deadlock- ja timeout-paluukoodin jälkeen (-911) harkita kesken jääneen LUW:n suoritusta uudelleen muutamia kertoja. Toistuvasti lukkiumaan johtavan tapahtuman voi ehkä ohittaa ja siirtää uudelleensuoritettavaksi.

   Deadlock- ja timeout-tilanteissa eräohjelmalle palautetaan SQL-koodi -911 ja tehtävä palautetaan automaattisesti edelliseen COMMIT-pisteeseen. (Tosiaikaohjelmissa ohjelmalle voidaan systeemimäärityksistä riippuen palauttaa vaihtoehtoisesti SQL-koodi -913, jolloin DB2 ei tee palautusta). Jos ohjelman suoritusta jatketaan, on kaikki edellisen COMMIT-pisteen jälkeen suoritetut lauseet suoritettava uudestaan. Muista, että edellisen COMMIT- pisteen jälkeen SQL-lauseita on voitu suorittaa useista moduleista. Ne kaikki on suoritettava uudestaan.

3. Käytä kohdistimen määrittelyssä FOR FETCH ONLY –määrettä, jos rivejä aiotaan vain lukea.

   DB2 ei ota lukkoja silloin, kun tiedon eheys voidaan muulla tavoin varmentaa (lock avoidance). Vaikka lock avoidance on käytössä, lämpimille sivuille otetaan lukkoja. Siksi COMMIT on hyödyllinen myös lukevissa eräohjelmissa.

   Lock avoidance edellyttää, että:
   ?? Isolation Level (CS)
   ?? Currentdata(NO) Huom! Oletus on Currentdata(YES).
   ?? Ei päivittävä -kohdistin
   ?? Riviin tai sivuun ei kohdistu keskeneräisiä päivityksiä (cold row)

   FOR FETCH ONLY -määre kertoo DB2:lle, että kohdistimella vain luetaan.

4. Harkitse lukevissa lauseissa UR –optiota. Tämä parantaa samanaikaisuutta kyselijöitten ja päivittäjien välillä, mutta voi tuottaa lukevalle ohjelmalle loogisesti epäyhtenäisen tuloksen.
   ?? ei esimerkkiä

5. Tee commit eräohjelmassa 2-5 sekunnin välein, jolleivat painavat syyt muuta vaadi.

   Älä tee useammin kuin kerran sekunnissa, sillä silloin itse commitin teosta tulee pullonkaula. Suunnittele uudelleenaloitus.

**G. Yllättävät tulokset**

1. Käytä ORDER BY –määrettä aina silloin, ja vain silloin, kun tulosjoukko halutaan tietyssä järjestyksessä.

   ORDER BY -määre on ainoa keino varmistaa tulosjoukon järjestys.

ORDER BY -sarakkeita ei tarvitse luetella SELECT –listassa, paitsi jos lauseessa on
?? UNION / UNION ALL
?? GROUP BY
?? sarakefunktio SELECT –listassa
?? DISTINCT SELECT -listassa

Älä kuitenkaan käytä ORDER BY -määrettä turhaan, jos tulosjoukon järjestyksellä ei ole väliä, sillä rivien lajittelu voi olla raskasta.

2. Varo väärää päätelmää sarake-funktioiden luvusta.

Seuraavien sarakefunktioiden (AVG, MAX, MIN, SUM) tulosten tulkinnassa kannattaa olla huolellinen.

Jos ehto jää toteutumatta, niin SQL-koodi = 0, vastauksena yksi rivi, jossa NULL -arvo. Tämän voi korjata viemällä luetun tutkimalla isäntämuuttujan NULL-indikaattoria.

Esimerkiksi

```
SELECT      MIN(KONTNO)
INTO        :KONTNO:KONTNO-IND
FROM        TAULU
WHERE       KONTNO > :hv1
```

Lauseessa SUM(SALARY + BONUS), jos bonuksen arvo on NULL, jää palkkakin saamatta. Lauseen voi korjata käyttämällä COALESCE –funktiota tai tarkis tamalla NULL-indikaattoria.

```
SELECT      SUM(SALARY + COALESCE(BONUS, 0))
….
```

HUOM! Jos funktioon kohdistuvalle sarakkeelle on luotu indeksi ja se on laskevassa järjestyksessä ja kentässä voi olla NULL-arvoja, niin kaikki ne NULL-arvot ovat indeksin alussa.

Esimerkiksi taulu, jossa on avain ja kaksi datasaraketta, joista toisessa NULL on sallittu. Tauluun vietiin muutamia rivejä (myös NULL-arvoja) ja tehtiin kaikki mahdolliset kyselyt. Tulos (hiukan yllättävä) on alla olevassa taulukossa. WHERE-ehto aiheuttaa NULL-vastauksia, mutta todelliset NULL-arvot vain jätetään aina laskuista pois.

| Funktio | Ei löydy rivejä Not null -sarake | Ei löydy rivejä Null -sarake | Löytyy rivejä Not null -sarake | Löytyy rivejä Null -sarake |
|---|---|---|---|---|
| AVG | NULL | NULL | Arvo | Arvo * |
| MAX | NULL | NULL | Arvo | Arvo ** |
| MIN | NULL | NULL | Arvo | Arvo ** |
| SUM | NULL | NULL | Arvo | Arvo *** |

* = todellisten arvojen summa / todellisten arvojen lkm
** = todellinen suurin/pienin arvo
*** = todellisten arvojen summa

3. Jos sarakkeen arvo on NULL, pätee siihen vain IS NULL, eikä mikään muu ehto. Kaikista muista kyselyistä rivi jää huomioimatta.

4. Ole varovainen WHERE –ehtojen kanssa ulkoliitoksen yhteydessä.

Esimerkki:

(taulu T1 arvot 1-4 ja 5 ja 7, taulu T2 arvot 1 ja 3 ja 5-8)

```
SELECT          T1.SAR1 AS T1_SAR1,
                T2.SAR1 AS T2_SAR1
FROM            T1 LEFT JOIN T2
                ON T1.SAR1 = T2.SAR1
WHERE           T2.SAR1 > 0

T1_SAR1  T2_SAR1
---------+---------+---------+---------
      1         1
      3         3
      5         5
      7         7
```

Tulos on yllättävä, koska tulosjoukkoon odotettiin rivejä 1-4 (left join) ja yhteiset 5, 7.
Syynä on **liitoksen teko ennen where –ehtoa.** Kun liitoksessa T2.SAR1 saa NULL-arvot T1-riveille 2 ja 4 , niin where-ehto on näille epätosi.
Asia voidaan korjata ON-ehdolla eli siirretään where-predikaatti sinne:

```
SELECT          T1.SAR1 AS T1_SAR1,
                T2.SAR1 AS T2_SAR1
FROM            T1 LEFT JOIN T2
                ON T1.SAR1 = T2.SAR1
                   AND T2.SAR1 > 0

---------+---------+---------+---------+---
T1_SAR1  T2_SAR1
---------+---------+---------+---------+---
      1         1
      2  -------
      3         3
      4  -------
      5         5
      7         7
```

Muita jo V5:ssä mahdollisia korjaustapoja ovat b) null-ehdon kysyminen erikseen tai c) taululauseke.
SUOSITUS: käytä ON-korjausta.

Korjaus b): NULL kysytään erikseen

```
SELECT          T1.SAR1 AS T1_SAR1,
                T2.SAR1 AS T2_SAR1
FROM            T1 LEFT JOIN T2
                ON T1.SAR1 = T2.SAR1
WHERE T2.SAR1 > 0
```

```
                OR T2.SAR1 IS NULL
```

Korjaus c): taululauseke

```
SELECT      T1.SAR1 AS T1_SAR1,
            T2.SAR1 AS T2_SAR1
FROM        T1 LEFT JOIN
            (SELECT    SAR1
             FROM      T2
             WHERE     SAR1 > 0) AS T2
            ON T1.SAR1 = T2.SAR1
```

## H. Client-Server

1. OPTIMIZE FOR N ROWS

   N vaikuttaa blokin kokoon mentäessä koneen yli.

2. Käytä hajautetussa ympäristössä kohdistimen määrityslauseessa FOR FETCH ONLY –määrettä, jos rivejä aiotaan vain lukea.

   Hajautetussa ympäristössä tehokkaimpaan (vähiten sanomia verkossa ja vähiten yhteydenottoja) käsittelyyn päästään käyttämällä nk. Block Fetchiä.

   Block Fetchin käyttö edellyttää, ettei SQL-lauseessa suoriteta päivityksiä. Staattista SQL:ää käytettäessä DB2 voi päätellä, milloin päivityksiä tehdään, mutta dynaamista SQL:ää käytettäessä päätös perustuu kursorimäärittelyyn. Käyttämällä kohdistimen määrityslauseessa FOR FETCH ONLY -määrettä, ‘server-DB2’ saa tiedon, että päivityksiä ei tehdä ja lähettää kerralla yhden blokin verran rivejä paikalliselle DB2:lle.

## I. Data Sharing

1. Commit-väli & LOCK AVOIDANCE –optio.

2. Ohjelman suunnitttelu mukaan; hyvä ei riitä, pitää olla loistava.

3. Käytä ehdottomasti LOCK AVOIDANCE –optiota, vaikutus tuntuu paljon laajemmin käydessään kaikkien koneiden läpi.