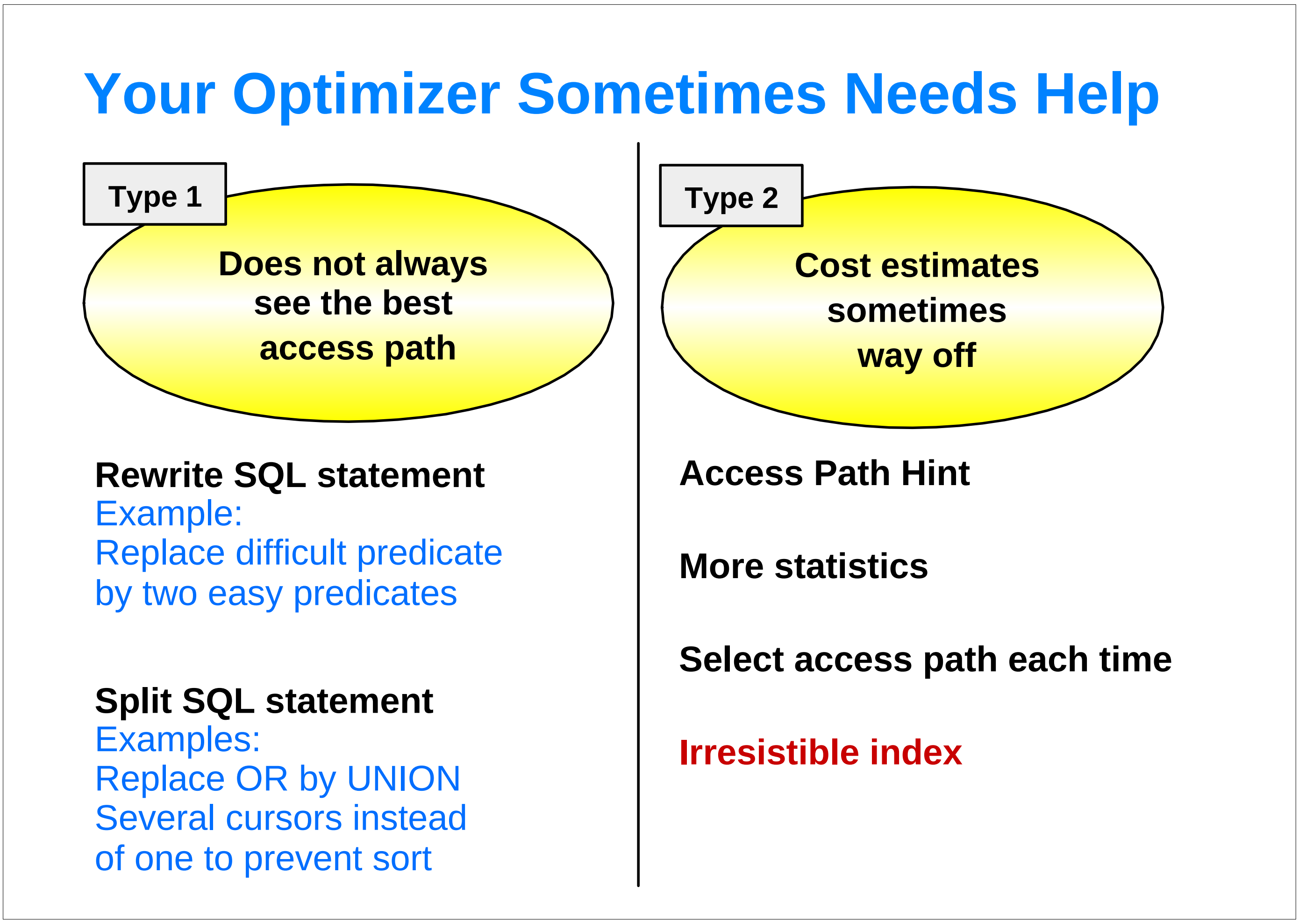
Liitoksista ja optimoijan auttamisesta

Tapio Lahdenmäki
tapio.lahdenmaki@siol.net

Huhtikuu 2005



Your Optimizer Sometimes Needs Help

Type 1

Does not always see the best access path

Rewrite SQL statement
Example:
Replace difficult predicate by two easy predicates

Split SQL statement
Examples:
Replace OR by UNION
Several cursors instead of one to prevent sort

Type 2

Cost estimates sometimes way off

Access Path Hint

More statistics

Select access path each time

Irresistible index

- **Type 1 problems** can only be solved by rewriting the SQL in a more optimizer-friendly way (exception: Oracle Function Based Indexes; you may create an index whose key is a column function, A CONCAT B, for instance)
- Some optimizers rewrite many queries before access path selection. Simple DB2 example: C1 = :hv1 OR C1 = :hv2 is rewritten as C1 IN(:hv1,:hv2). C1 may then be a matching column.
- To avoid manual rewrite, all SQL programmers should know the most common optimizer pitfalls (mainly which predicates can never be matching predicates & the concept of non-BT predicates). The pitfall list is product and release dependent.
- **Type 2 problems** are less predictable but easier to fix
- Access path hint is a good quick fix (to return to a previous access path, for instance)
- Note that hints do not help with type 1 problems
- Optional statistics -- such as column or column group cardinality -- may make the FF estimates good enough even when the access path is selected once
- Extensive statistics, such as histograms -- together with selecting access path each time -- are necessary for complex queries when the optimal access path depends on input, especially when an user may generate many different predicates
- Index improvement is sometimes the best solution

Good Question

I've read a lot of papers on index , at each time every thing is clear in my mind , but some days after but i am still not 100% sure with the comprehension of the concept , so just want to verify with you

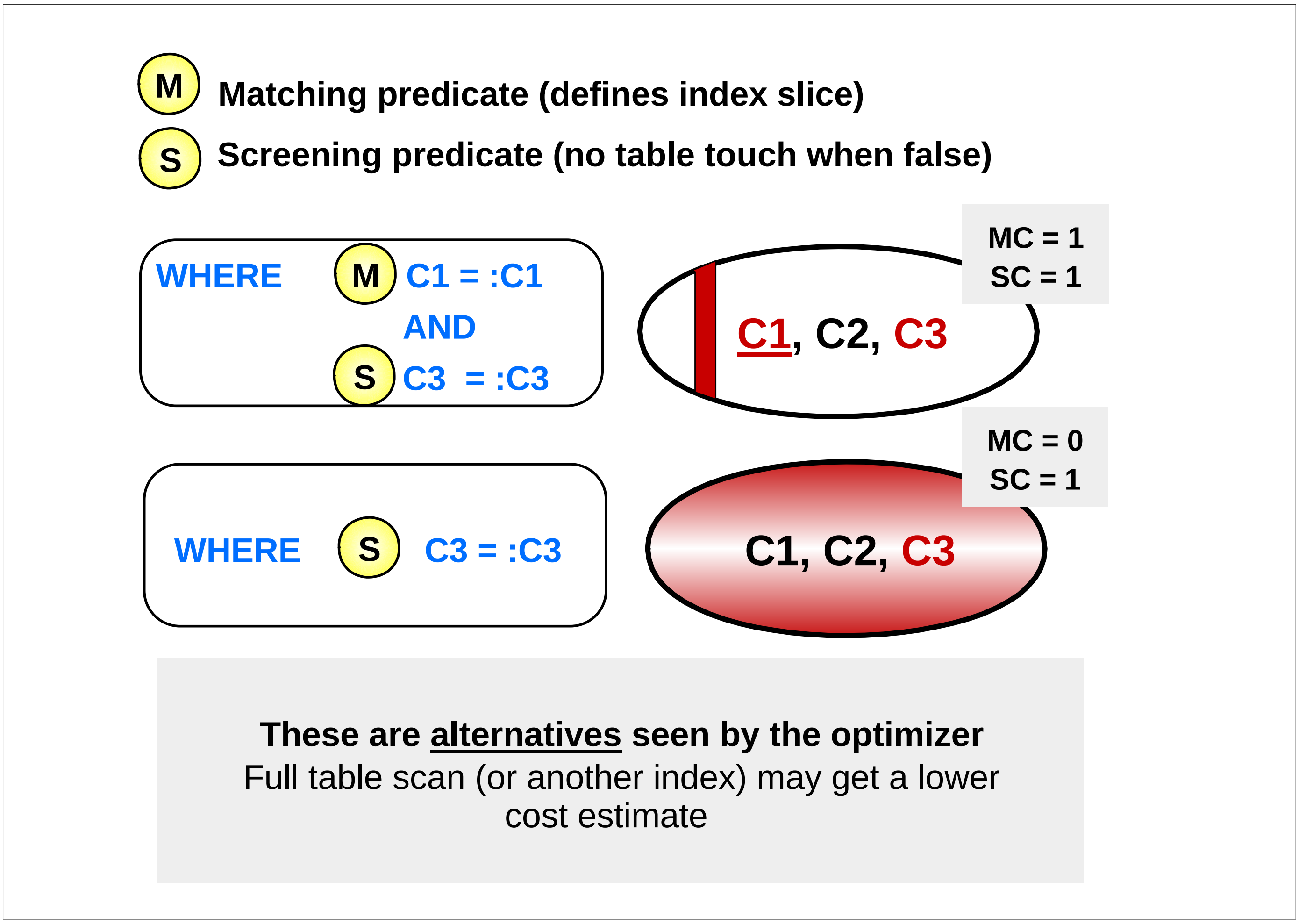
INDEX(col1,col2,col3)

select xxxx from xxx
where
col1= aa AND
col2= aa
=> Index matching , right ?

where
col1= aa AND
col3= aa
=> Index screening , right ?

where
col3= aa
=> index screening ? (in my opinion No , as not restrictive enough)

- From IDUG DB2 L



M Matching predicate (defines index slice)

S Screening predicate (no table touch when false)

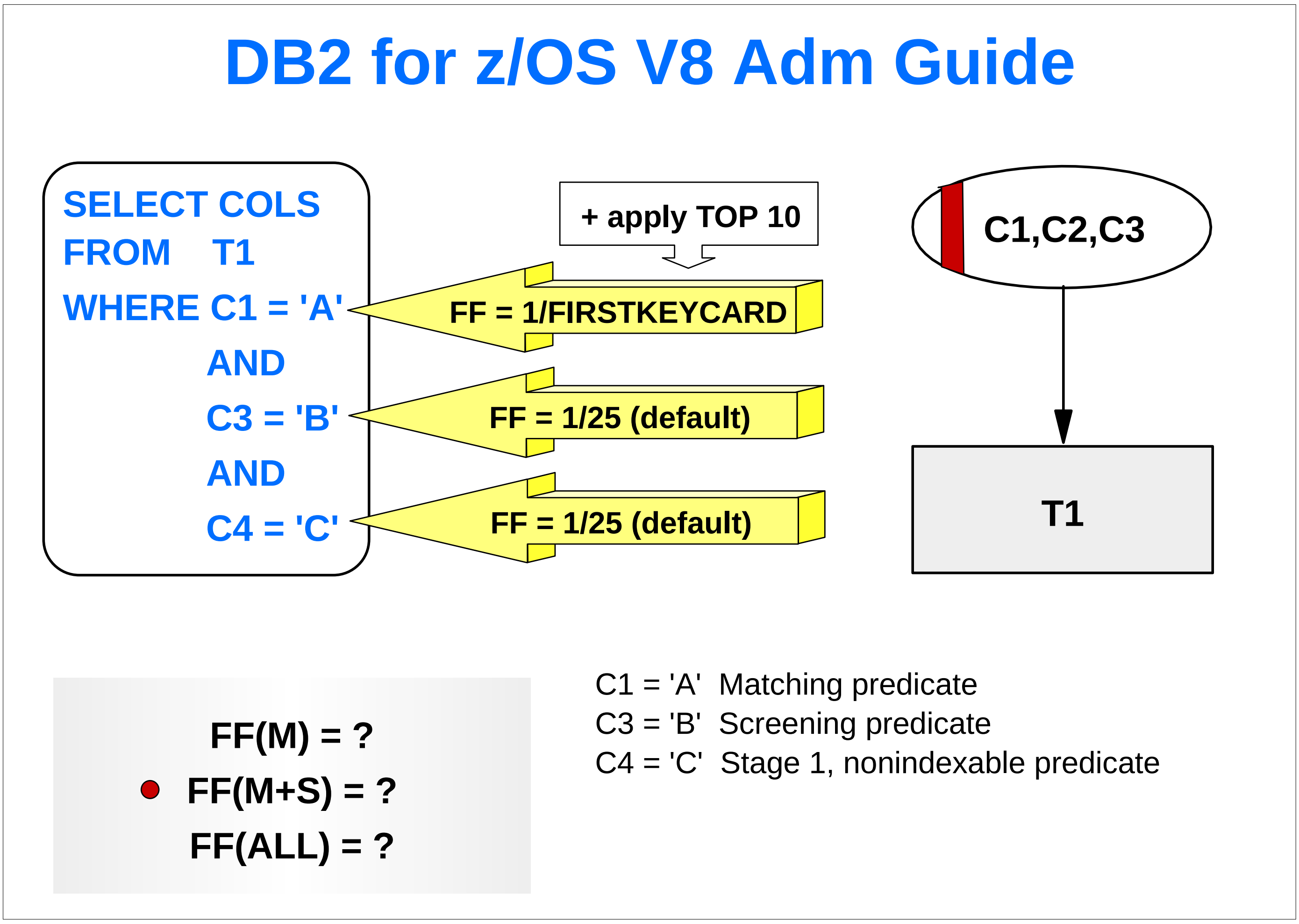
These are alternatives seen by the optimizer
Full table scan (or another index) may get a lower cost estimate

- Why the confusion?
- (1) Matching and screening are attributes of simple predicates (as opposed to compound predicates) in relation to a given index: **which predicates are matching or screening if that index is used with that WHERE clause**
- (2) When the optimizer evaluates an index candidate, it identifies matching and screening predicates with that candidate; it then evaluates filter factors and the number of touches and chooses the access path that has the lowest cost estimate
- Thus, in the last case, when the optimizer **considers** index (C1,C2,C3) for WHERE C3 = :C3, it sees that the predicate is a screening predicate

----

Compare with predicates that are too difficult for the optimizer:

- No matching (DB2 for z/OS: non-indexable)
- No matching, no screening (DB2 for z/OS: Stage 2)



DB2 for z/OS V8 Adm Guide

SELECT COLS
FROM T1
WHERE C1 = 'A'
AND
C3 = 'B'
AND
C4 = 'C'

+ apply TOP 10

FF = 1/FIRSTKEYCARD

FF = 1/25 (default)

FF = 1/25 (default)

C1,C2,C3

T1

FF(M) = ?

● FF(M+S) = ?

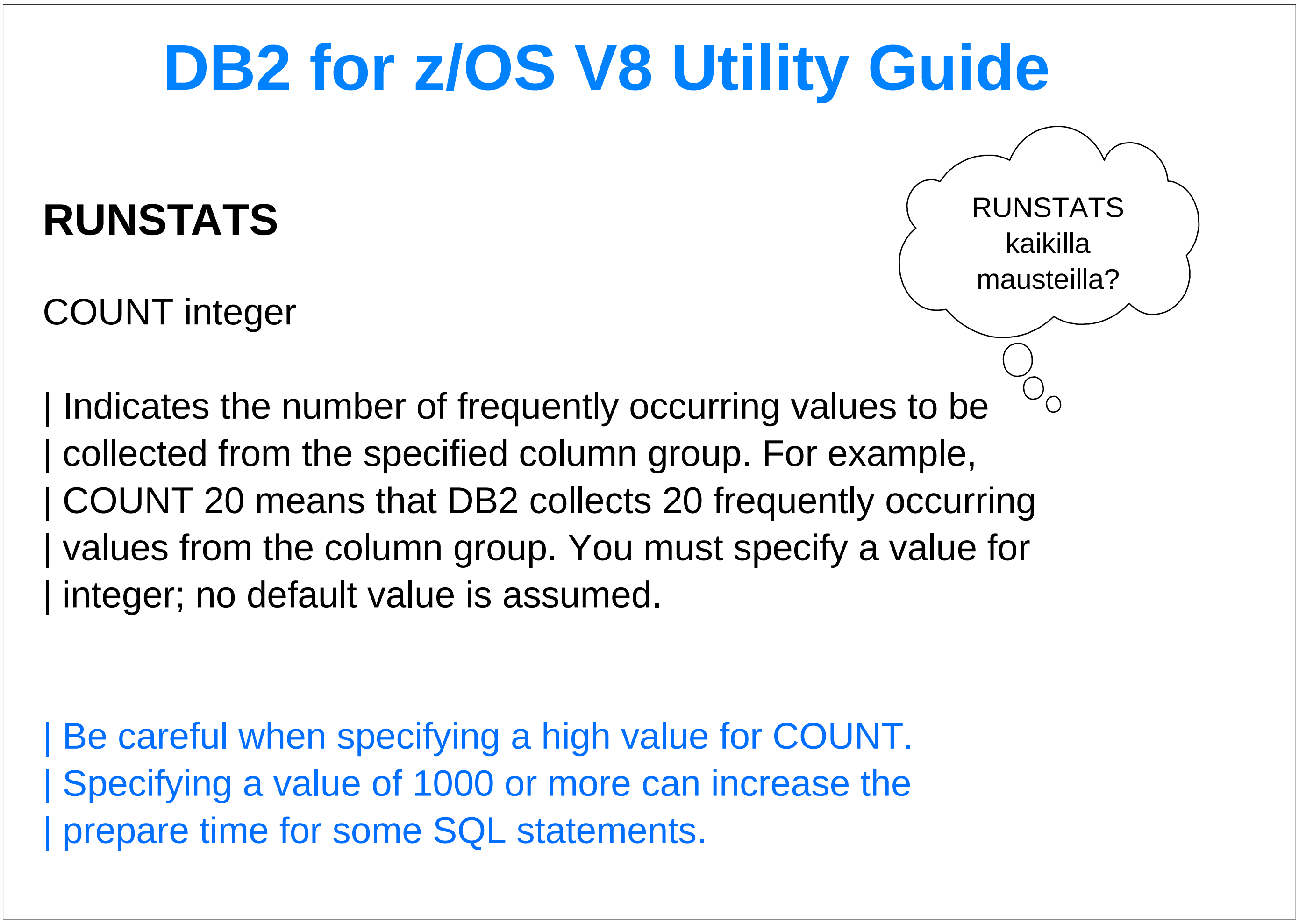
FF(ALL) = ?

C1 = 'A' Matching predicate
C3 = 'B' Screening predicate
C4 = 'C' Stage 1, nonindexable predicate

- 'To determine the cost of accessing table T1 through index I1, DB2 performs these steps:
- Estimates the matching index cost. DB2 determines the index matching filter factor by using single-column cardinality and single-column frequency statistics because only one column can be a matching column. (TL: in this case)
- Estimates the total index filtering. This includes matching and screening filtering. If statistics exist on column group (C1,C3), DB2 uses those statistics. Otherwise DB2 uses the available single-column statistics for each of these columns. DB2 will also use FULLKEYCARDF as a bound. Therefore, it can be critical to have column group statistics on column group (C1, C3) to get an accurate estimate.
- Estimates the table-level filtering. If statistics are available on column group (C1,C3,C4), DB2 uses them. Otherwise, DB2 uses statistics that exist on subsets of those columns.
- Important: If you supply appropriate statistics at each level of filtering, DB2 is more likely to choose the most efficient access path.
- You can use RUNSTATS to collect any of the needed statistics.'

The arrows on the visual show the FF estimates for the simple predicates with default statistics

And by the way...FF = Filter factor = Number of passing (qualifying) rows divided by the number of source rows, 0...1 or 0...100%...a property of a predicate (simple or compound)...depends on the actual value distributions in the predicate columns



DB2 for z/OS V8 Utility Guide

RUNSTATS
kaikilla
mausteilla?

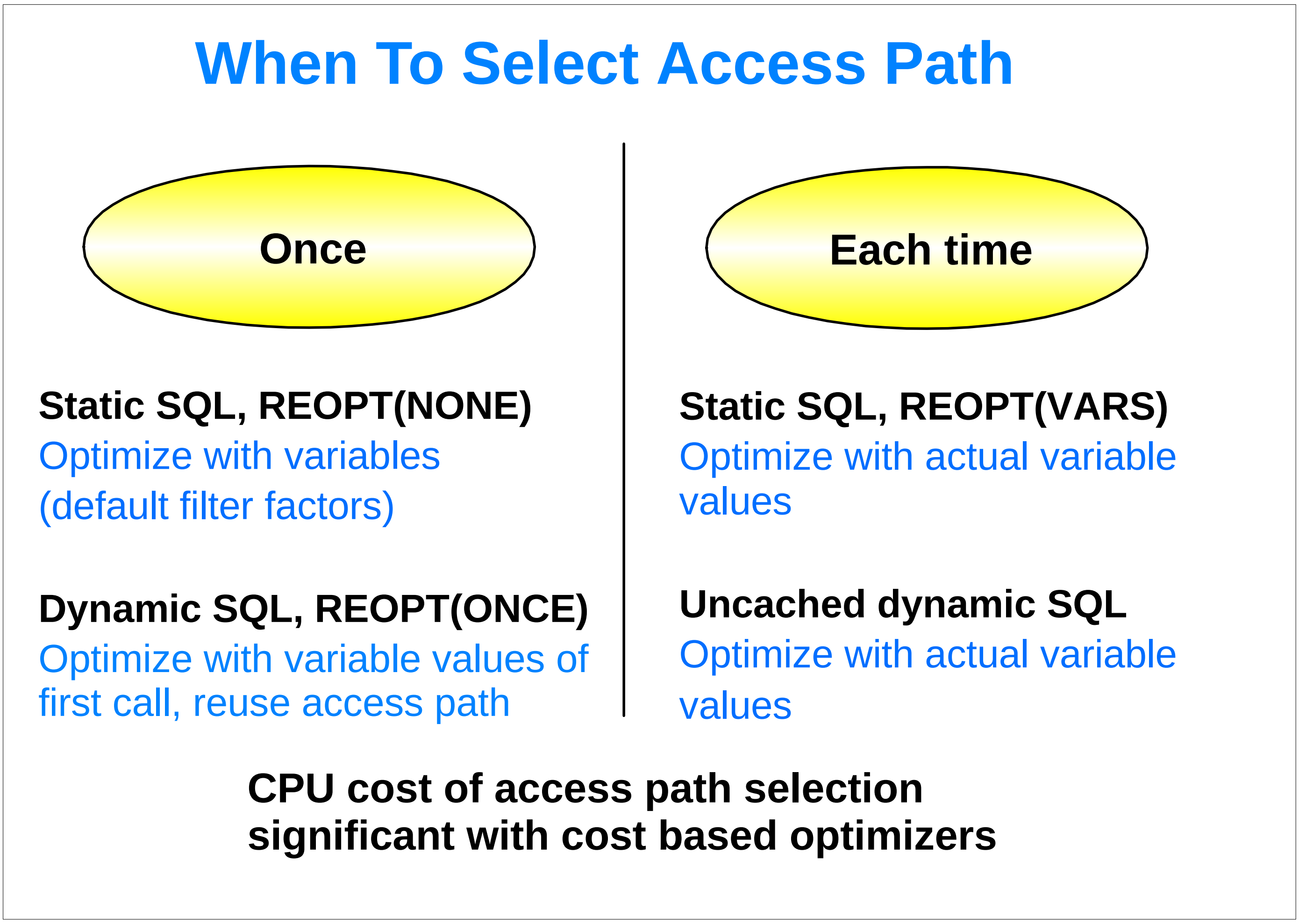
RUNSTATS

COUNT integer

| Indicates the number of frequently occurring values to be
| collected from the specified column group. For example,
| COUNT 20 means that DB2 collects 20 frequently occurring
| values from the column group. You must specify a value for
| integer; no default value is assumed.

| Be careful when specifying a high value for COUNT.
| Specifying a value of 1000 or more can increase the
| prepare time for some SQL statements.

- Cardinality of all predicate columns?
- Cardinality of all possible predicate column combinations?
- + TOP 1000 and BOTTOM 1000 for all above?
- Or only when a cost estimate problem found?



When To Select Access Path

Once

Static SQL, REOPT(NONE)

Optimize with variables (default filter factors)

Dynamic SQL, REOPT(ONCE)

Optimize with variable values of first call, reuse access path

Each time

Static SQL, REOPT(VARS)

Optimize with actual variable values

Uncached dynamic SQL

Optimize with actual variable values

CPU cost of access path selection significant with cost based optimizers

- Estimating the costs of the alternatives each time a query is executed leads to best estimates
- However, in operational applications, access path selection should seldom occur at run time because of the high CPU overhead
- When the WHERE clause contains no variables (WHERE COL = 1), the same access path remains optimal until one of the following takes place:
- (1) Indexes are modified
- (2) The number of rows increases a lot
- (3) The distribution of column values changes a lot
- When the WHERE clause contains variables (WHERE COL = :COL), the optimal access path may depend on the values moved to the variables
- In such cases, it may be necessary to do access path selection each time with the actual values (COL = value)
- It is often possible to design indexes that make the optimal access path for a SELECT independent of variable values; then the access path can be selected once and reused many times
- The implementation of **Once / Each time** is product dependent

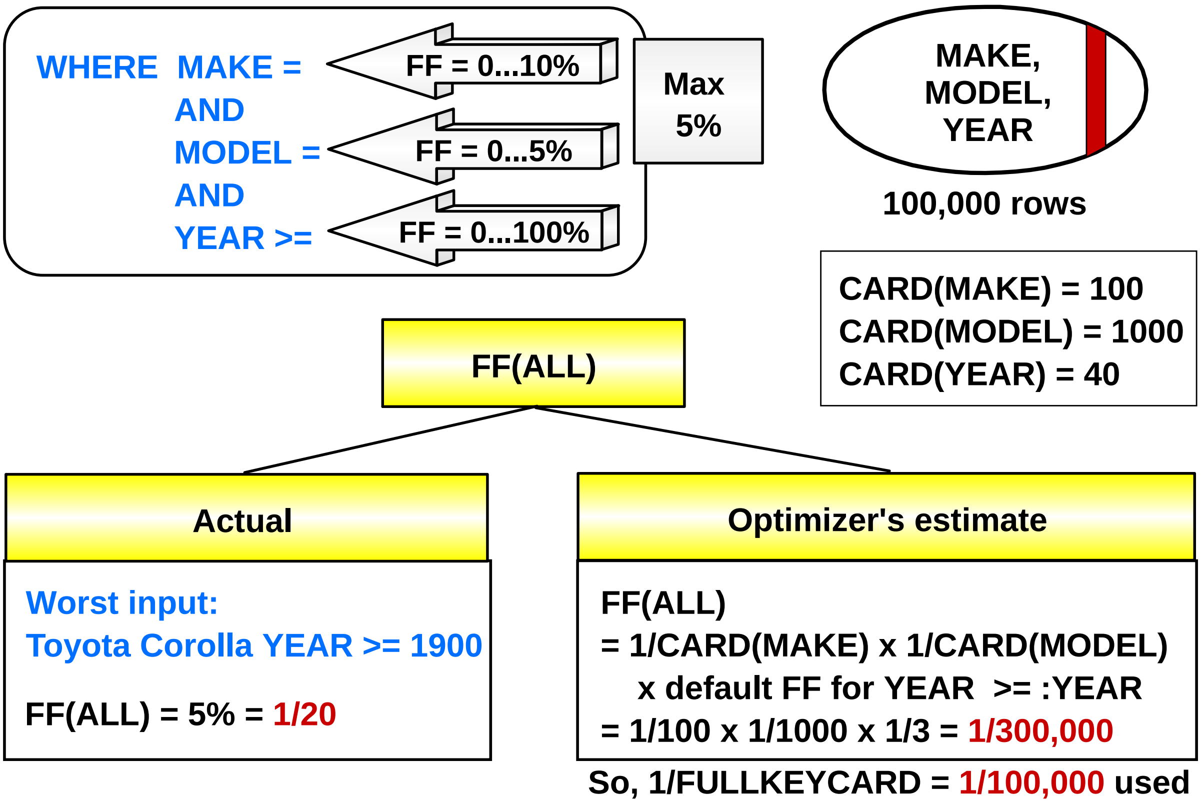
Table 94. Default filter factors for interpolation

COLCARDF	Factor for Op	Factor for LIKE or BETWEEN
>=100000000	1/10,000	3/100000
>=10000000	1/3,000	1/10000
>=1000000	1/1,000	3/10000
>=100000	1/300	1/1000
>=10000	1/100	3/1000
>=1000	1/30	1/100
>=100	1/10	3/100
>=2	1/3	1/10
=1	1/1	1/1
<=0	1/3	1/10

Note:
Op is one of these operators: <, <=, >, >=.

- DB2 for z/OS V8 Administration Guide

Optimizer Estimates Once With Variables



WHERE MAKE =
AND
MODEL =
AND
YEAR >=

FF = 0...10%
FF = 0...5%
FF = 0...100%

Max
5%

MAKE,
MODEL,
YEAR

100,000 rows

CARD(MAKE) = 100
CARD(MODEL) = 1000
CARD(YEAR) = 40

FF(ALL)

Actual

Worst input:
Toyota Corolla YEAR >= 1900

FF(ALL) = 5% = 1/20

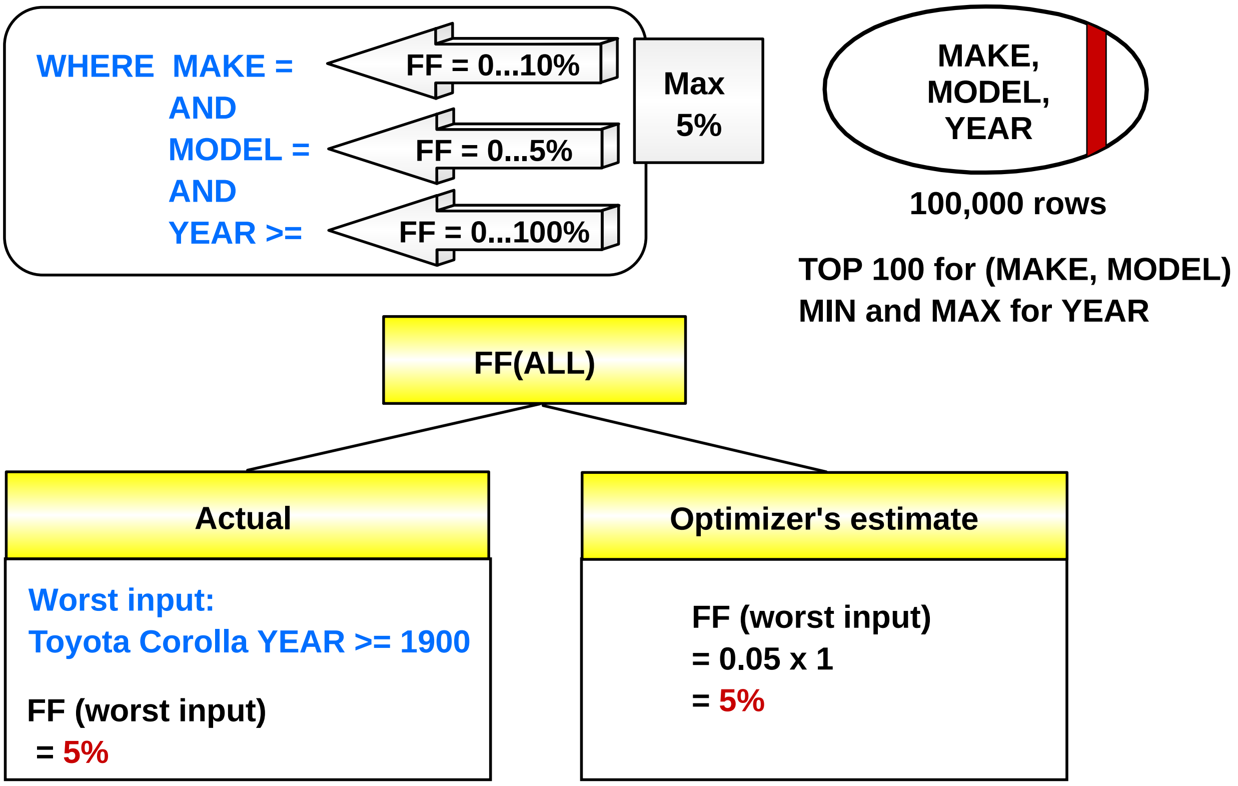
Optimizer's estimate

FF(ALL)
= 1/CARD(MAKE) x 1/CARD(MODEL)
x default FF for YEAR >= :YEAR
= 1/100 x 1/1000 x 1/3 = 1/300,000

So, 1/FULLKEYCARD = 1/100,000 used

- In this example, the worst input is the most common MAKE, MODEL together with a low value in YEAR
- The FF values next to the WHERE clause are **actual** filter factors
- FF(M) = FF(M+S) = FF(ALL) because all predicates are matching
- TOYOTA COROLLA is the most common value combination in columns MAKE, MODEL
- If the optimizer does not know that columns MAKE and MODEL are highly correlated, the FF estimate is way too low
- If the optimizer knows CARD(MAKE, MODEL) = 1000, the FF estimate is better (1/3000) -- not bad for average input
- FFest = 1/3000 would probably lead to index slice scan with table touches -- not good with the worst input unless the order of table rows is the same as the order of index rows in (MAKE,MODEL,YEAR)
- **The cost estimates must be made each time (based on user input) or we have to design an index that performs well enough with any input**

Optimizer Estimates Each Time



WHERE MAKE =
AND
MODEL =
AND
YEAR >=

FF = 0...10%

FF = 0...5%

FF = 0...100%

Max
5%

MAKE,
MODEL,
YEAR

100,000 rows

TOP 100 for (MAKE, MODEL)
MIN and MAX for YEAR

FF(ALL)

Actual

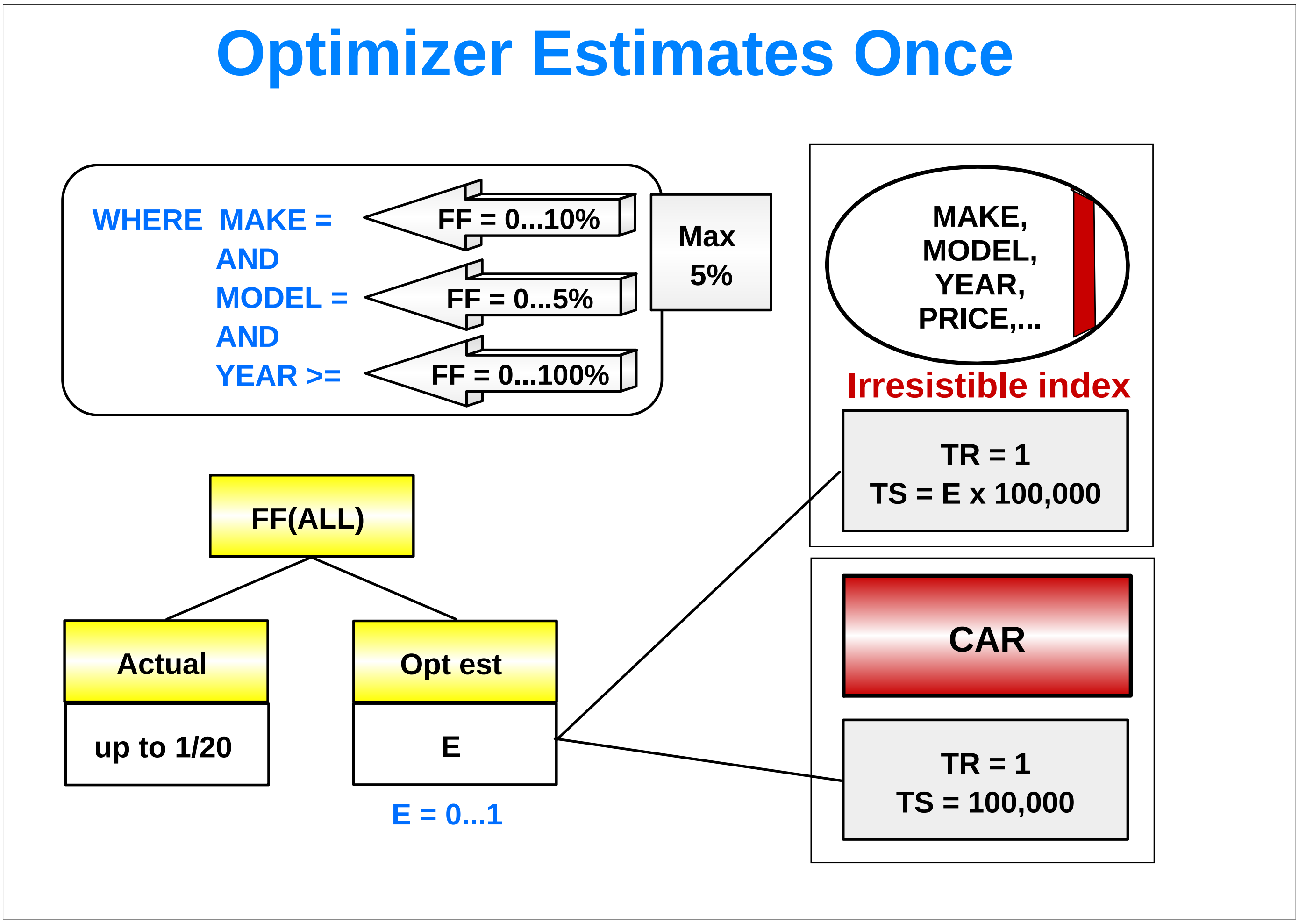
Worst input:
Toyota Corolla YEAR >= 1900

FF (worst input)
= 5%

Optimizer's estimate

FF (worst input)
= 0.05 x 1
= 5%

- Now the optimizer probably chooses full table scan when the input is TOYOTA COROLLA 1900 or later and index slice scan with table touches when the input is MINI COOPER S 2003 or later
- Note the two prerequisites:
- (1) Statistics options tailored for this SELECT
- (2) Run-time cost estimates each time: high CPU time, latch waits in some environments -- **not feasible with high transaction rates**



Optimizer Estimates Once

- **With a fat index, an index slice scan is the best alternative with any input**
- It also performs quite well enough with any input: TR = 1 and TS = 5,000 with the worst input
- **The cost estimates need to be made only once -- either before the first execution or at the first execution**
- Minimal statistics are adequate for this SELECT; the optimizer will get a lower cost estimate for index slice scan than for a full table scan
- However, at least CARD(MAKE, MODEL) and CARD(MAKE,MODEL,YEAR) are recommendable options anyway -- think what might happen when table CAR is joined with table DEALER and the optimizer makes a bad estimate for FF(MAKE = AND MODEL =)

Review

1. List possible reasons for the optimizer problem on the next visual

   Wrong index chosen after statistics created for a new table and its indexes

2. Propose possible remedies

   Better than the current one (cheating the optimizer by reducing CLUSTERRATIO of wrong index)

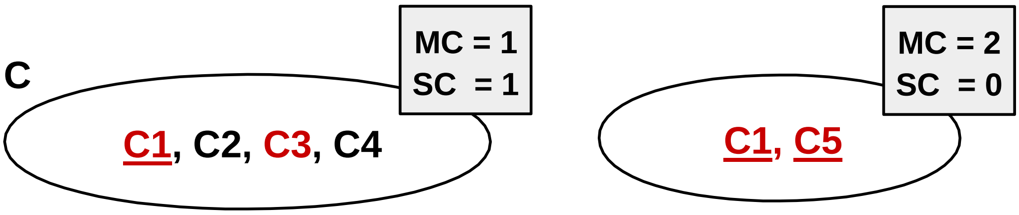
   Assume choosing access path each time too expensive

- When the new table had very few rows, the statistics were not created because the optimizer probably would have chosen a full table scan. With the default filter factors, the optimizer chose index (C1,C5)
- Later, when the table had grown, the statistics were created. Now the optimizer chose index (C1, C2, C3, C4), which is the clustering index . The new access path was observed to be slower than the old one.

We have a new partitioned table, say T1(C1,C2,C3,C4,C5,...),with a clustering index (C1,C2,C3,C4) and 2 other indexes, one is (C1,C5), and the other one, you can forget it :-) And the predicates of the problem query, is **"C1=:hv and C5=:hv and C3<:hv"**.

(Table growing, statistics updated with RUNSTATS)... After that, we suffered a high volume I/O on T1 and transaction response time slow down. We checked PLAN_TABLE and found the access path now is PI(C1,C2,C3,C4), which having only 1 matchcolumns... So index-screening is played, right?

We RUNSTATS and REBIND it again, still can't help optimizer do a correct choice. Finally, we cheated DB2 by tuning down the CLUSTERRATIO value of (C1,C2,C3,C4), and then (C1,C5) was chosen back. But it is not a good idea to update CATALOG manually, so we are now looking for a better solution. Of course, optimization hint is useful, but we didn't set this ZPARM on. Any other advice, such as special RUNSTATS method and so on? I am still curious about the arithmetic of filter factor on this kind of index-screening. Why the ineffective PI would be chosen? You know there is a perfectly matched index!


C

C1, C2, C3, C4

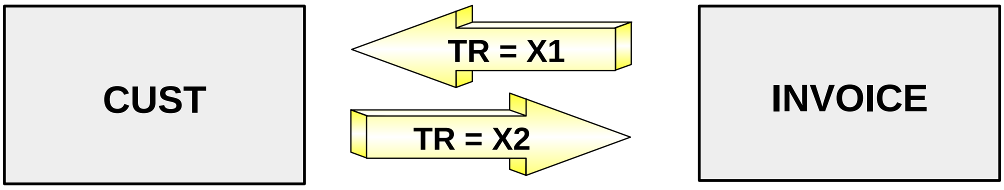
MC = 1
SC = 1

C1, C5

MC = 2
SC = 0

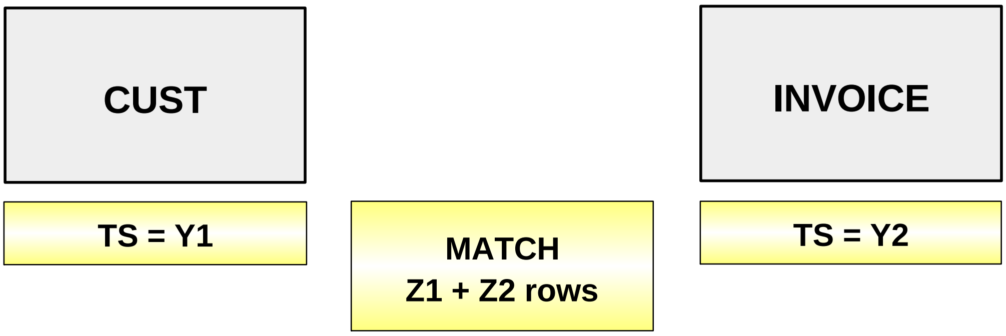
- RUNSTATS = Create statistics
- REBIND = Optimize with current statistics
- CATALOG contains the statistics used by the optimizer. The statistics can be updated manually with SQL (to make test table look like production table, for instance)
- Access path hint is disabled (a system parameter, ZPARM)
- INDEX CLUSTERRATIO = 100% if the index rows in the same order as the table rows. CLUSTERRATIO affects the I/O time of table touches (actual & optimizer estimates)
- PI = Partitioning index (C1,C2,C3,C4)

Nested Loop Or Merge Scan/Hash Join?

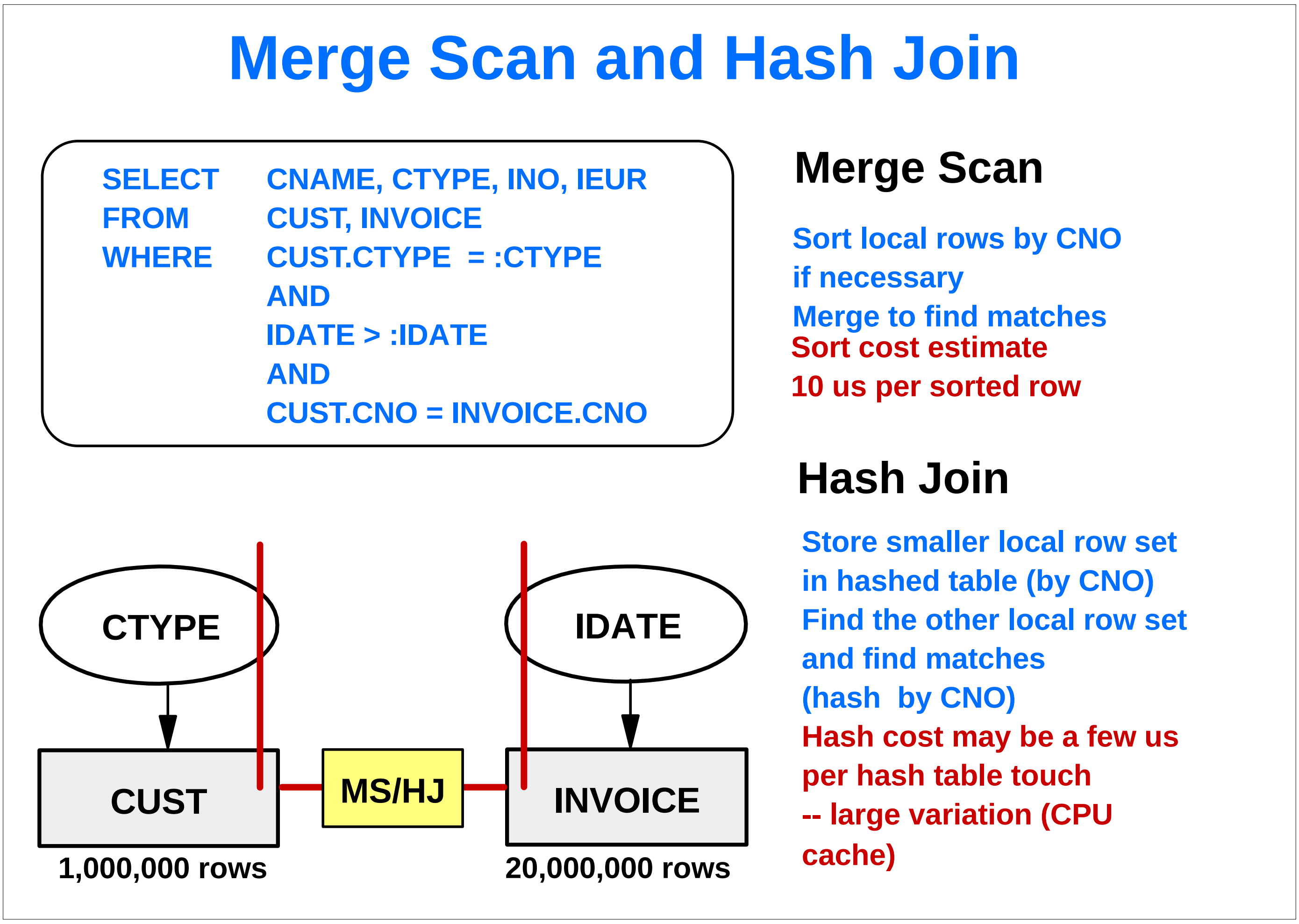


X1 = NLR(INVOICE) if fat index on CUST

X2 = NLR(CUST) if fat index on INVOICE



- The upper part of the visual shows the justification for the rule of thumb 'The outer table in nested loop should probably be the one with a lower number of local rows''
- NLR = Number of local rows (rows that satisfy all local predicates)
- When both X1 and X2 are fairly high, MS/HJ may be a better join method
- Note that X1 and X2 are determined by FF(ALL)
- Y1 + Y2 is the number of rows in the index slices defined by the local matching predicates -- determined by FF(M)
- Z1 + Z2 is the number of local rows -- NLR(CUST) + NLR(INVOICE) -- determined by FF(ALL)
- TR = Number of random touches
- TS = Number of sequential touches



Merge Scan and Hash Join

```
SELECT    CNAME, CTYPE, INO, IEUR
FROM      CUST, INVOICE
WHERE     CUST.CTYPE  = :CTYPE
          AND
          IDATE > :IDATE
          AND
          CUST.CNO = INVOICE.CNO
```

Merge Scan

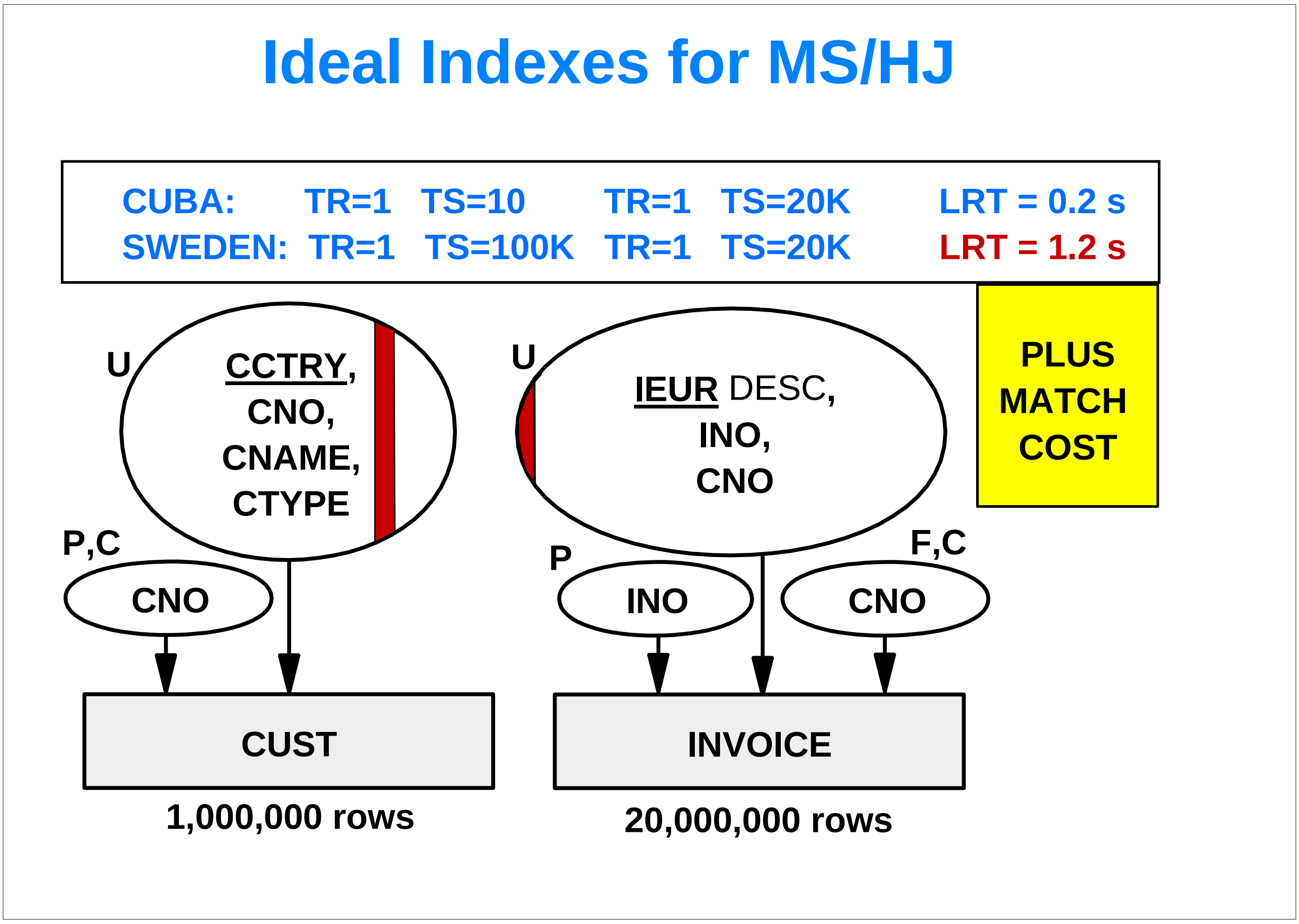
Sort local rows by CNO
if necessary
Merge to find matches
Sort cost estimate
10 us per sorted row

Hash Join

Store smaller local row set
in hashed table (by CNO)
Find the other local row set
and find matches
(hash by CNO)
Hash cost may be a few us
per hash table touch
-- large variation (CPU
cache)

- Nested loop join sometimes leads to an excessive number of random touches, even with ideal indexes
- This is why the optimizers have an alternative method that reads the qualifying rows from each table and then match them either by merging (after sorting, if necessary) or by hashing (randomizing)
- Some products provide one of these join methods (MS or HJ), some provide both
- The ideal indexes for MS/HJ may be different from those derived for nested loop join: the local predicate columns should be in front; the position of the join columns matters only if sorting can be avoided for MS

- The optimizers choose MS/HJ more often than they used to, especially with fat indexes. This is because the cost gap between random and sequential touches has widened.
- The cost of matching the rows with MS can be estimated by counting the touches to the work file and the number of rows to be sorted
- The cost of matching the rows with HJ is difficult to predict because the cost of a random touch to the hash table is very variable. It is extremely low (roughly 1 us) if the hash table remains in the CPU cache (L2). Otherwise, it can be 10 us or more.



Ideal Indexes for MS/HJ

CUBA:	TR=1	TS=10	TR=1	TS=20K	LRT = 0.2 s
SWEDEN:	TR=1	TS=100K	TR=1	TS=20K	LRT = 1.2 s

U

CCTRY, CNO, CNAME, CTYPE

P,C

CNO

CUST

1,000,000 rows

U

IEUR DESC, INO, CNO

P

INO

F,C

CNO

INVOICE

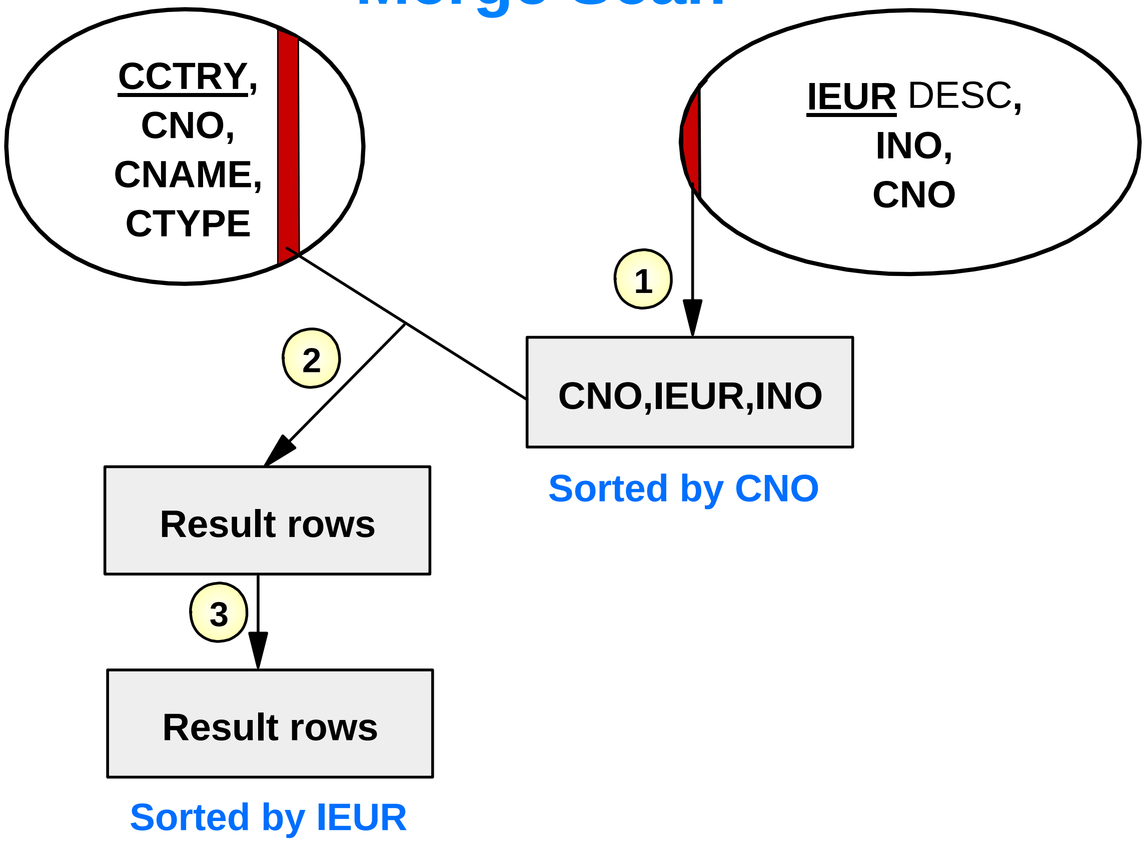
20,000,000 rows

PLUS MATCH COST

```
SELECT      CNAME, CTYPE, INO, IEUR
FROM        CUST, INVOICE
WHERE       CCTRY = :CCTRY (FF up to 10%)
            AND
            IEUR > :IEUR (FF = 0.1 %)
            AND
            CUST.CNO = INVOICE.CNO
ORDER BY IEUR DESC
OPTIMIZE FOR 20 ROWS
```

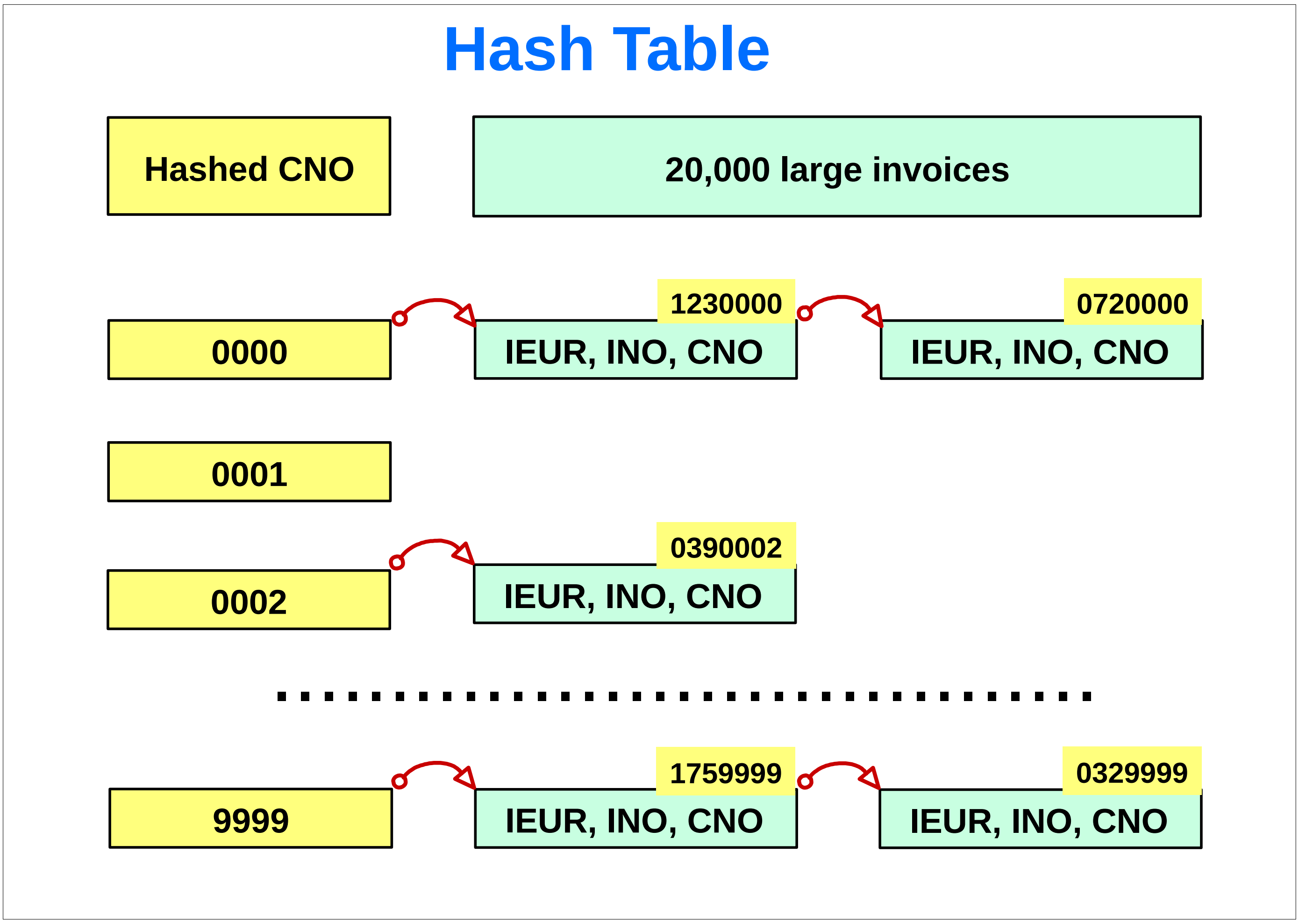
▸ SORT = Y, so perhaps the whole result should be fetched in one transaction

Merge Scan



1. Scan index slice, build temporary table 1, sort by CNO
2. Scan index slice (already in CNO sequence), merge with the rows in the temporary table 1; when a match found in step 2, store the result row in temporary table 2
3. 3ort the result rows by IEUR (descending)

Note: This is an 1:M join, so a temporary table for qualifying CUST rows is not necessary



Hash Table

Hashed CNO

20,000 large invoices

0000

1230000

IEUR, INO, CNO

0720000

IEUR, INO, CNO

0001

0002

0390002

IEUR, INO, CNO

9999

1759999

IEUR, INO, CNO

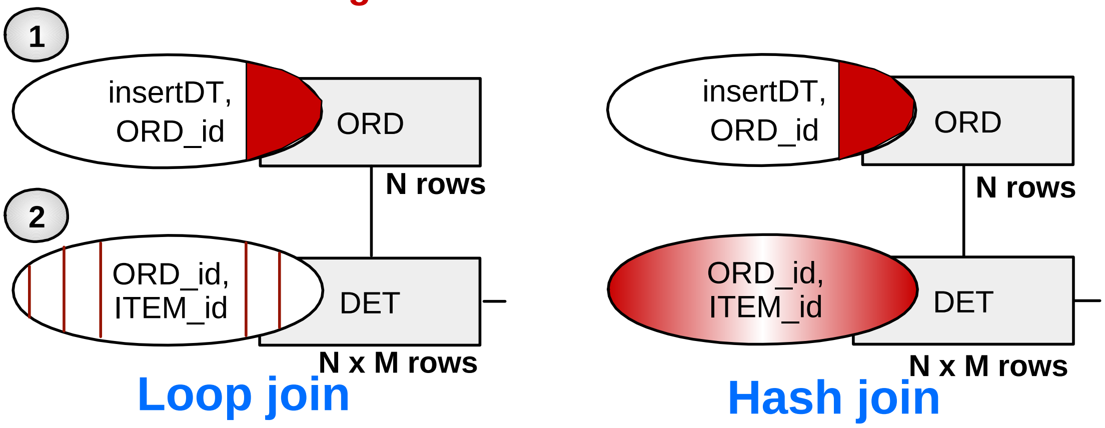
0329999

IEUR, INO, CNO

- With HJ, the selected columns from large invoices are first stored in a hash table. The anchor points are hashed CNO values.
- Next, the index slice containing the customers from the given country is scanned. For each of these rows, the CNO is hashed to find if there are matching rows in the hash table (any large invoices with this CNO?)

WHERE ORD.insertDT BETWEEN DATEADD(d, -3, GETDATE()) AND GETDATE()

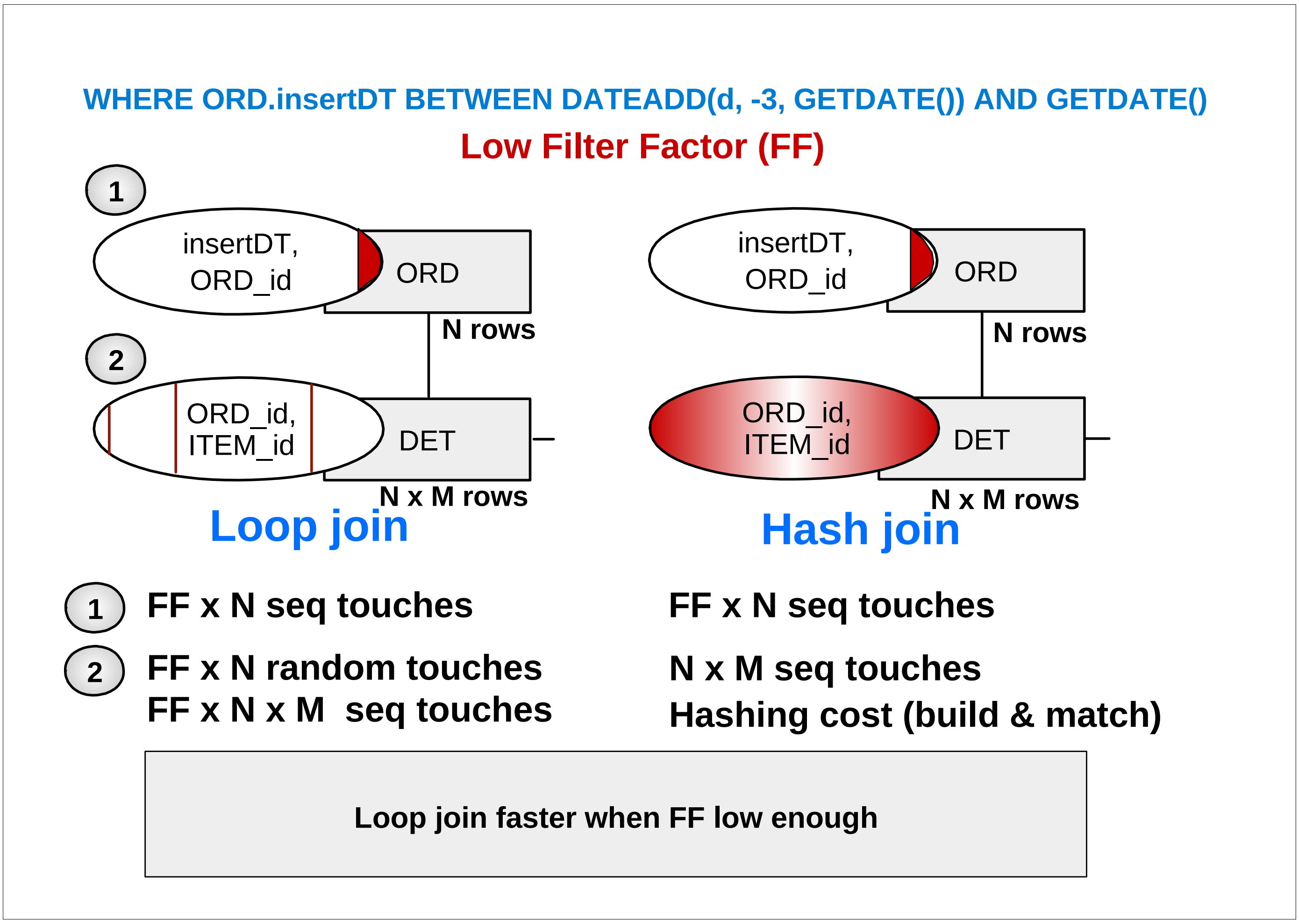
High Filter Factor Estimate



	Loop join	Hash join
1	**FF x N seq touches**	**FF x N seq touches**
2	**FF x N random touches** **FF x N x M seq touches**	**N x M seq touches** **Hashing cost (build & match)**

Hash join seems faster

- Joins are often very sensitive to filter factors
- This is a part of a simple five-table join, basically looking for item data for recent orders
- The platform is SQL Server 2000 (Courtesy of Chris Dickey, http://www.tunesqlserver.com/notes/CostEstimatePuzzle.1.aspx)
- The SELECT was very slow because the optimizer assumed a high FF for BETWEEN (default FF); it then got a lower cost estimate for hash join than nested loop
- **The indexes for tables ORD and DET are ideal for both loop join and hash join**, so we cannot influence the optimizer by index improvement (only ORD_id is SELECTed from table ORD, only ITEM-id from table DET; both indexes are fat)
- All touches on the visual are index touches
- The first random touch to start an index slice scan (index 1) is ignored
- The first random touch to start a full index scan (index 2) is ignored
- Note: The only local predicate on table ORD is matching; therefore FF(M) = FF(M+S) = FF(ALL)



WHERE ORD.insertDT BETWEEN DATEADD(d, -3, GETDATE()) AND GETDATE()

Low Filter Factor (FF)

1

insertDT, ORD_id

ORD

N rows

2

ORD_id, ITEM_id

DET

N x M rows

Loop join

insertDT, ORD_id

ORD

N rows

ORD_id, ITEM_id

DET

N x M rows

Hash join

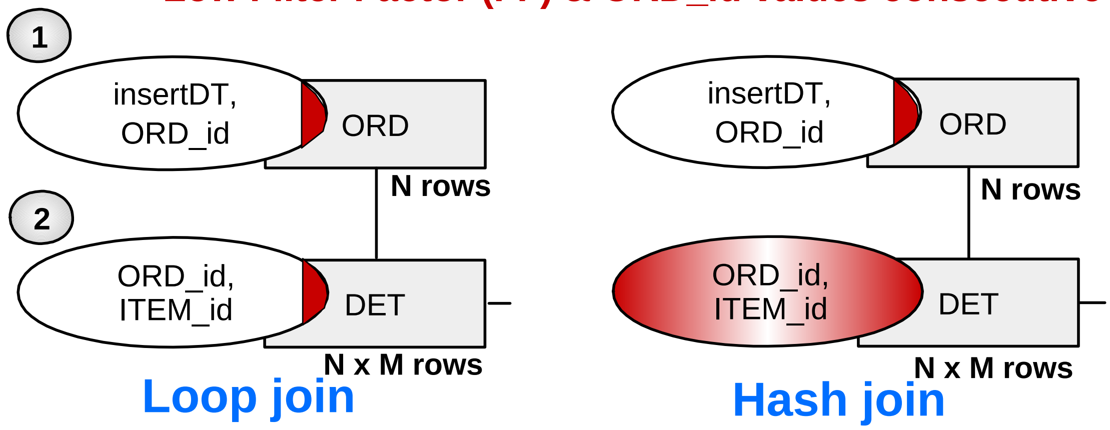
	Loop join	Hash join
1	FF x N seq touches	FF x N seq touches
2	FF x N random touches FF x N x M seq touches	N x M seq touches Hashing cost (build & match)

Loop join faster when FF low enough

- When the query was modified to enable parameter sniffing (equivalent to REOPT (ONCE)), the FF estimate for the BETWEEN predicate was close to the actual value (which was consistently quite low)
- Now the optimizer correctly estimates that loop join is faster

WHERE ORD.insertDT BETWEEN DATEADD(d, -3, GETDATE()) AND GETDATE()

Low Filter Factor (FF) & ORD_id values consecutive



Loop join

1 **FF x N seq touches**

2 **FF x N x M seq touches**

Hash join

FF x N seq touches

N x M seq touches

Hashing cost (build & match)

Loop join dramatically faster

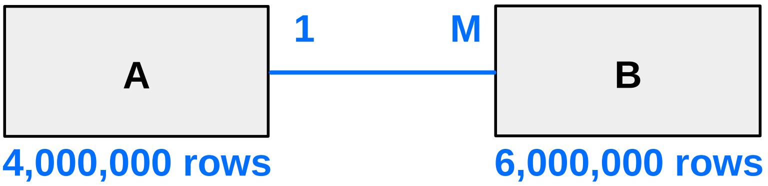
- The improvement was dramatic (1:75)
- When ORD_id is ever-increasing, the touches to the second index are actually sequential
- This is an example of a factor that is virtually impossible for an optimizer to find out by just looking at the statistics

Two Big Questions

Does the
optimizer know
the filter factors?

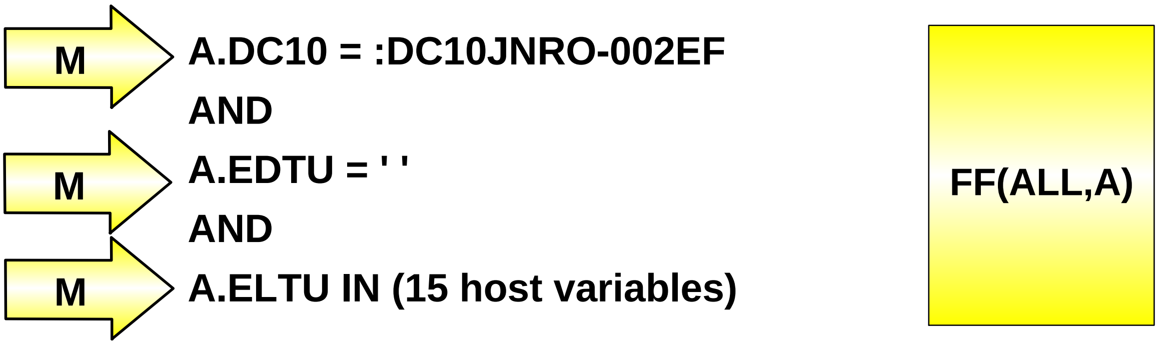
Do we know
the filter factors?

Join Case Study 1



- A step in a batch job to produce statistics
- SQL generated by CoolGen
- A programmer might write a subquery
  - Sort of result rows unavoidable (DISTINCT)
- A variation of CUST/INVOICE join
  - Customers with given characteristics that have at least one invoice with given characteristics
- Many predicates too difficult for the optimizer
  - Non-BT
  - COL <> literal
- Design indexes
  - Filter factors not known at this point

Local Predicates on Table A



A.DC10 = :DC10JNRO-002EF

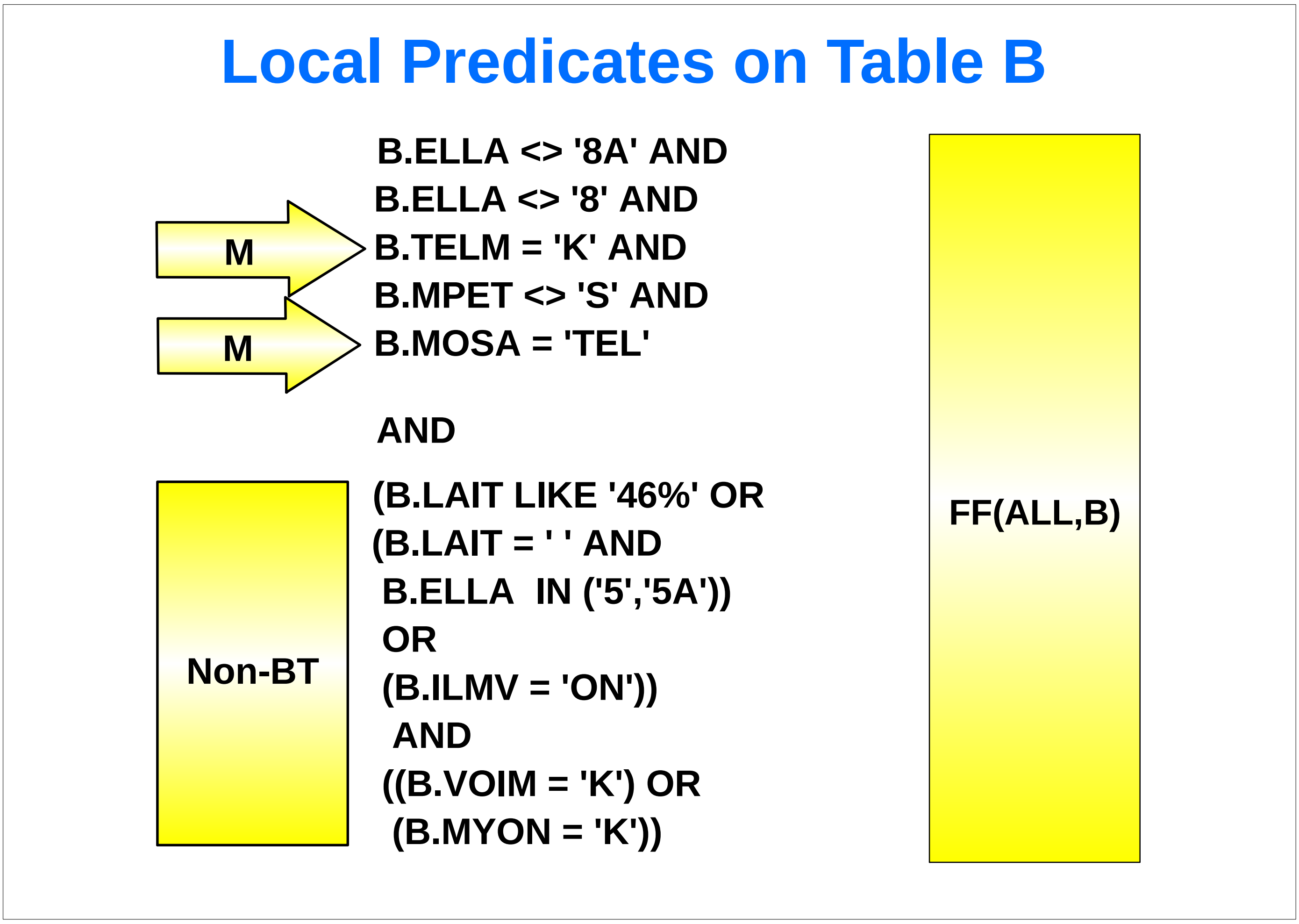
AND

A.EDTU = ' '

AND

A.ELTU IN (15 host variables)

- FF(ALL) = is the FF of this compound predicate
- M = potential matching predicate



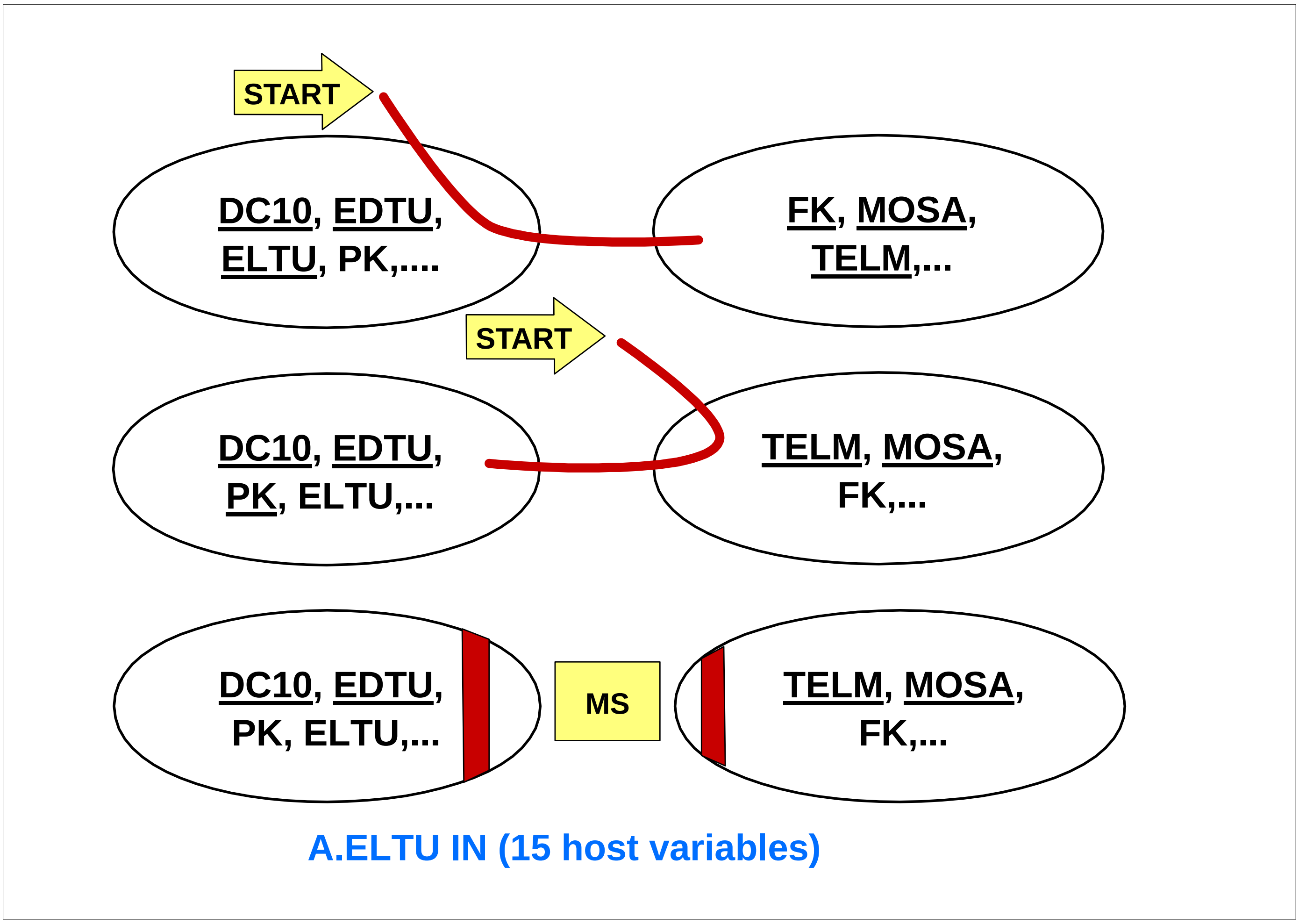
Local Predicates on Table B

B.ELLA <> '8A' AND
B.ELLA <> '8' AND
B.TELM = 'K' AND
B.MPET <> 'S' AND
B.MOSA = 'TEL'

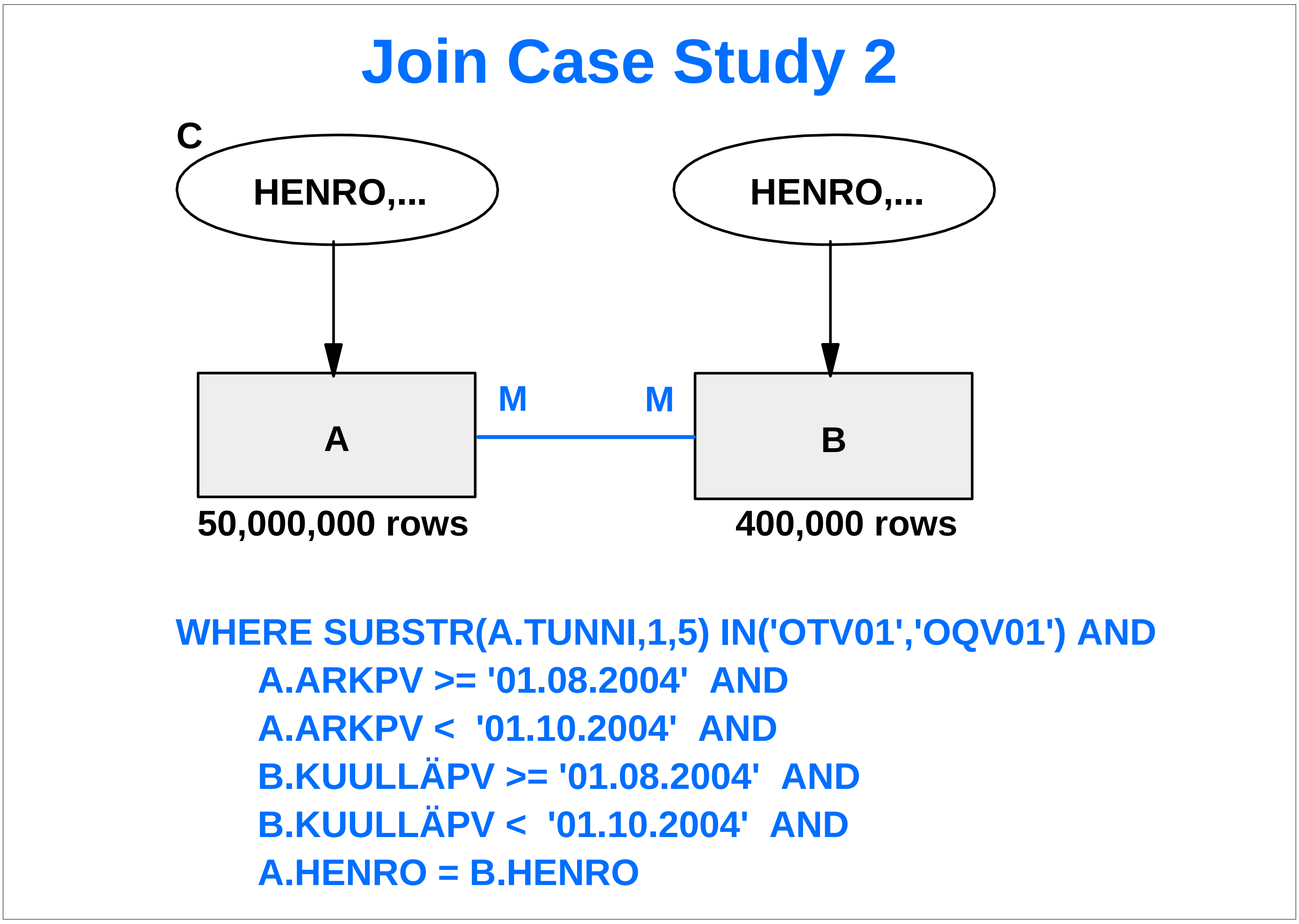
AND

(B.LAIT LIKE '46%' OR
(B.LAIT = ' ' AND
B.ELLA IN ('5','5A'))
OR
(B.ILMV = 'ON'))
AND
((B.VOIM = 'K') OR
(B.MYON = 'K'))

- M = Potential matching predicate
- Index designer must know at least FF(ALL) for the two tables
- Three alternatives: indexes for A-driven NL, B-driven NL or MS/HJ



- If FF(ALL,A) < FF(ALL,B) x 1.5, design indexes for A-driven nested loop
- If FF(ALL,A) > FF(ALL,B) x 1.5, design indexes for B-driven nested loop
- If both FF(ALL,A) and FF(ALL,B) fairly high, design indexes for merge scan (without sort) -- or no indexes (full table scans with sorts)
- Semi-fat or fat indexes? It depends on FF(M) and FF(S)
- Then, if the optimizer chooses wrong access path, create optional statistics (CARD, TOP N, BOTTOM N at least for single columns)



Join Case Study 2

C

HENRO,...

HENRO,...

A

M

M

B

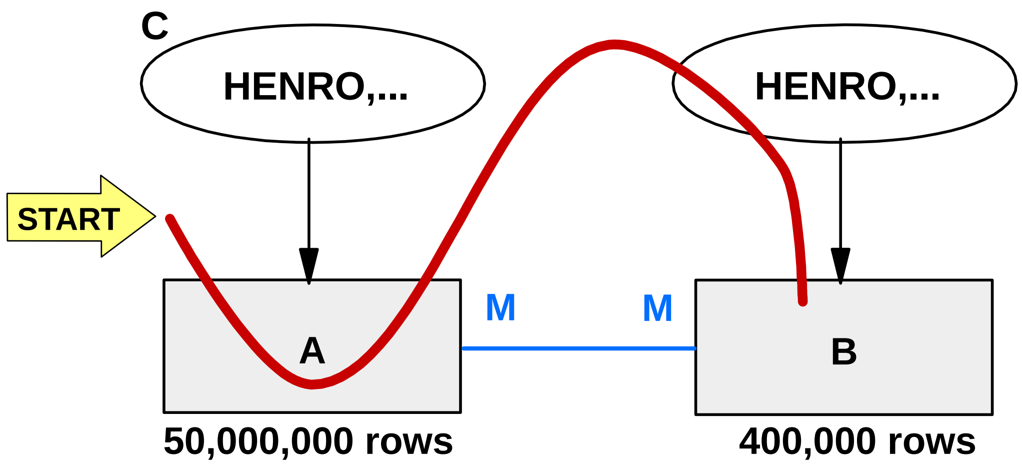
50,000,000 rows

400,000 rows

```
WHERE SUBSTR(A.TUNNI,1,5) IN('OTV01','OQV01') AND
       A.ARKPV >= '01.08.2004'  AND
       A.ARKPV <  '01.10.2004'  AND
       B.KUULLÄPV >= '01.08.2004'  AND
       B.KUULLÄPV <  '01.10.2004'  AND
       A.HENRO = B.HENRO
```

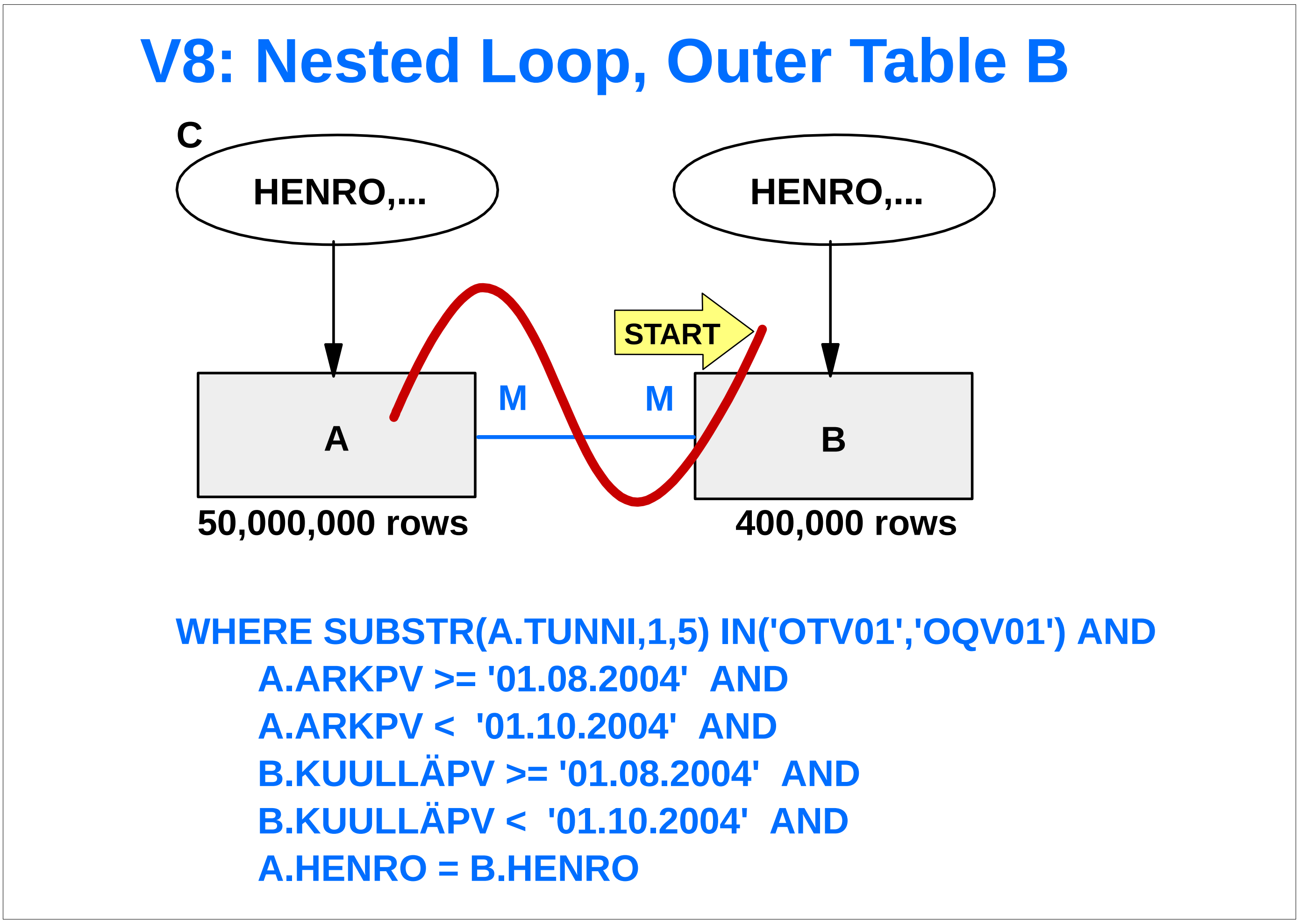
- Standard statistics (nothing on non-indexed columns)
- Local predicate columns not indexed
- Default FF for SUBSTR IN = ?
- Default FF for others 1/3 -- but probably only one per column (choose the most selective predicate for FFest when multiple predicates with the same column)
- FFest(ALL) for table A = ? x 1/3
- FFest(ALL) for table B = 1/3
- FFest means optimizer estimate
- Note: Date predicates have host variables in the program, literals used in EXPLAIN (should not make a difference in this case because no statistics on these columns)

V7: Hybrid Join, Outer Table A



WHERE SUBSTR(A.TUNNI,1,5) IN('OTV01','OQV01') AND
A.ARKPV >= '01.08.2004' AND
A.ARKPV < '01.10.2004' AND
B.KUULLÄPV >= '01.08.2004' AND
B.KUULLÄPV < '01.10.2004' AND
A.HENRO = B.HENRO

1. Looks smart
2. NL with B as the outer table would imply 1/3 x 400,000 random touches if FF(ALL) for table B is 1/3
3. The touches to table B skip sequential (with short skips if FF(ALL) for table A fairly high)

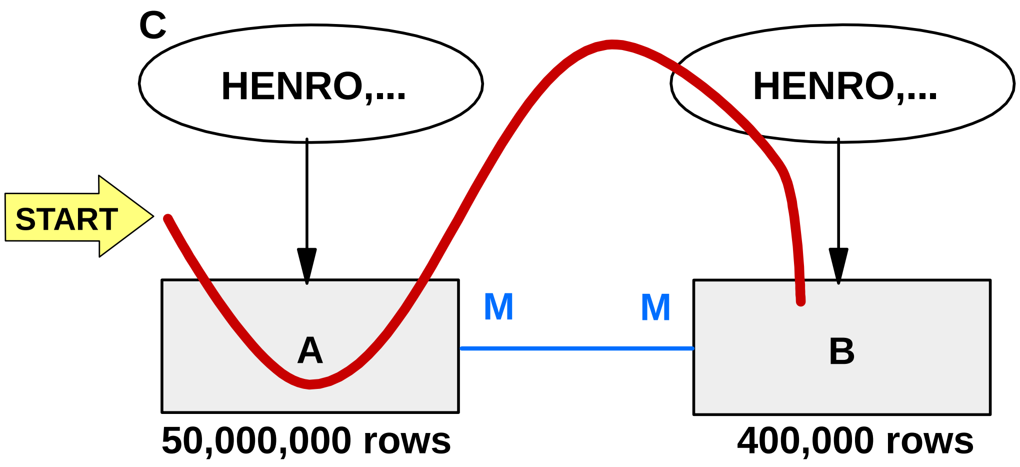


V8: Nested Loop, Outer Table B

WHERE SUBSTR(A.TUNNI,1,5) IN('OTV01','OQV01') AND
A.ARKPV >= '01.08.2004' AND
A.ARKPV < '01.10.2004' AND
B.KUULLÄPV >= '01.08.2004' AND
B.KUULLÄPV < '01.10.2004' AND
A.HENRO = B.HENRO

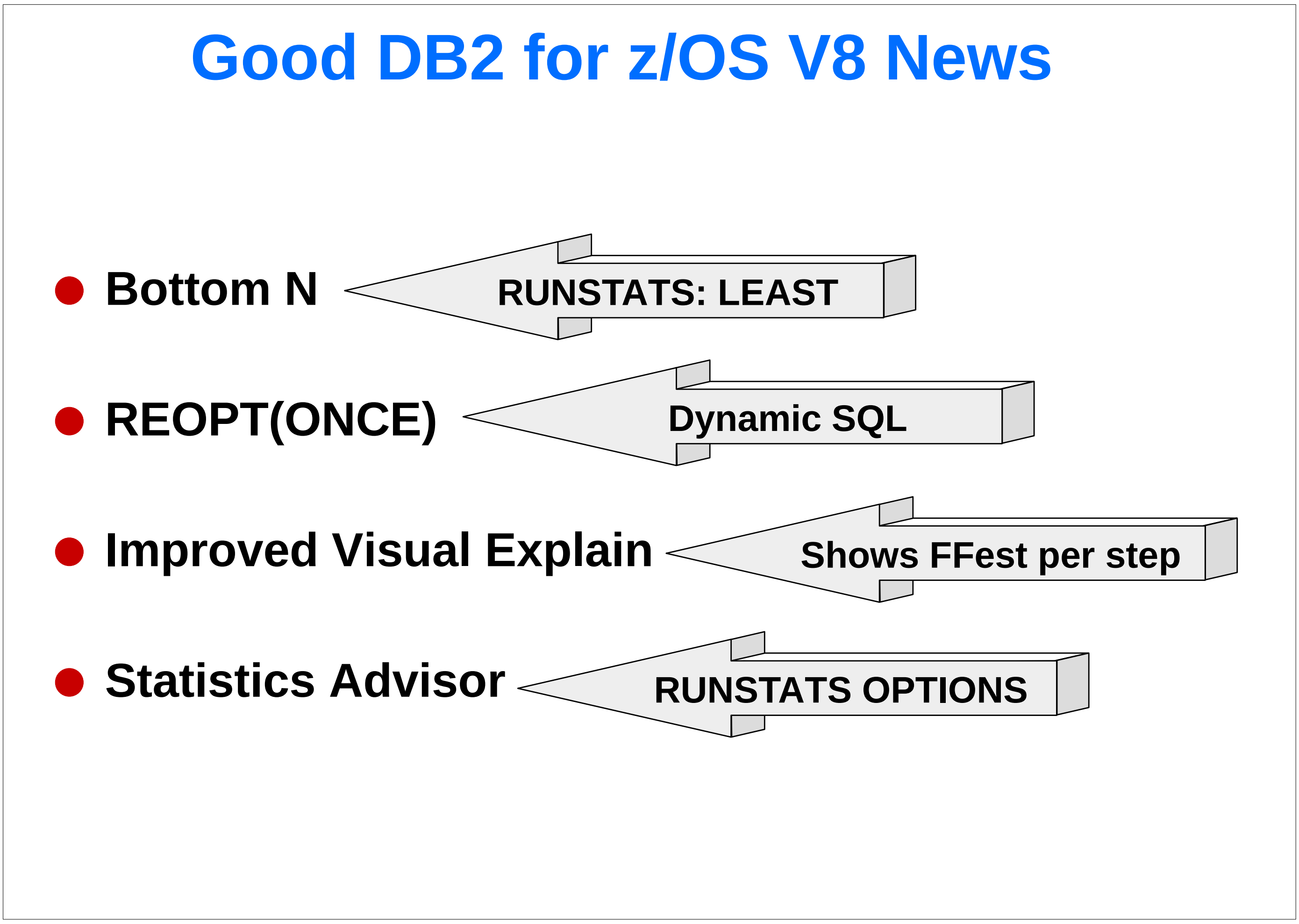
- Very strange (and slow)
- Asia huomattiin, koska eräajo alkoi V8 Compatibility Modessa kestämään ruhtinaallisesti pidempään kuin indeksipohjaisilla saantipoluilla V7:ssa. t.Jarmo
- Alkuperäinen EXPLAIN oli tuotantoympäristöstä, jossa on DB2V8 Compatibility Mode. Kokeilin samaa systeemitestiympäristöstä, jossa on DB2V8 New Function Mode ja tulos oli se, että saantipolut olivat TS Scan <> TS Scan. Eli ei muutosta näiden kahden moden välillä saantipolkujen valinnassa. t.Jarmo

V8 & BETWEEN: NL, Outer Table A



```
WHERE SUBSTR(A.TUNNI,1,5) IN('OTV01','OQV01') AND
      A.ARKPV >= '01.08.2004'  AND
      A.ARKPV <  '01.10.2004'  AND
      B.KUULLÄPV BETWEEN '01.08.2004'  AND '30.9.2004' AND
      A.HENRO = B.HENRO
```

- Essentially same as V7
- FFest(ALL) for table B now 1/10, according to Admin Guide
- Still a mystery -- case open
- Redbook DB2 for z/OS V8 Performance Topics: Better filter factor estimation in V8: 'DB2 can estimate a better filter factor by using **statistics inference derivation**. This can result in with or without the additional statistics mentioned before: Improved query selectivity estimation...Queries which contain **two or more predicates referencing a table** may get better access path...Significant performance improvement for some complex joins...Consider rebinding this type of query'
- What does statistical inference mean?
- http://www.cs.brown.edu/research/ai/dynamics/tutorial/Documents/StatisticalInference.html
- 'Statistical inference concerns the problem of inferring properties of an unknown distribution from data generated by that distribution. The most common type of inference involves approximating the unknown distribution by choosing a distribution from a restricted family of distributions. Generally the restricted family of distributions is specified parametrically.'
- Hmmm



Good DB2 for z/OS V8 News

- Bottom N
- REOPT(ONCE)
- Improved Visual Explain
- Statistics Advisor

RUNSTATS: LEAST
Dynamic SQL
Shows FFest per step
RUNSTATS OPTIONS

- Today, the optimizers are black boxes with many windows

No Pain No Gain

- Optimizer improvements make some queries slower

  Remember list prefetch in V2R2?

- Optimization hint provides quick fallback

  When old access path saved

- Additional statistics better solution

- Or more optimizer-friendly (FF-wise) predicates