Revival of the SQL Tuner

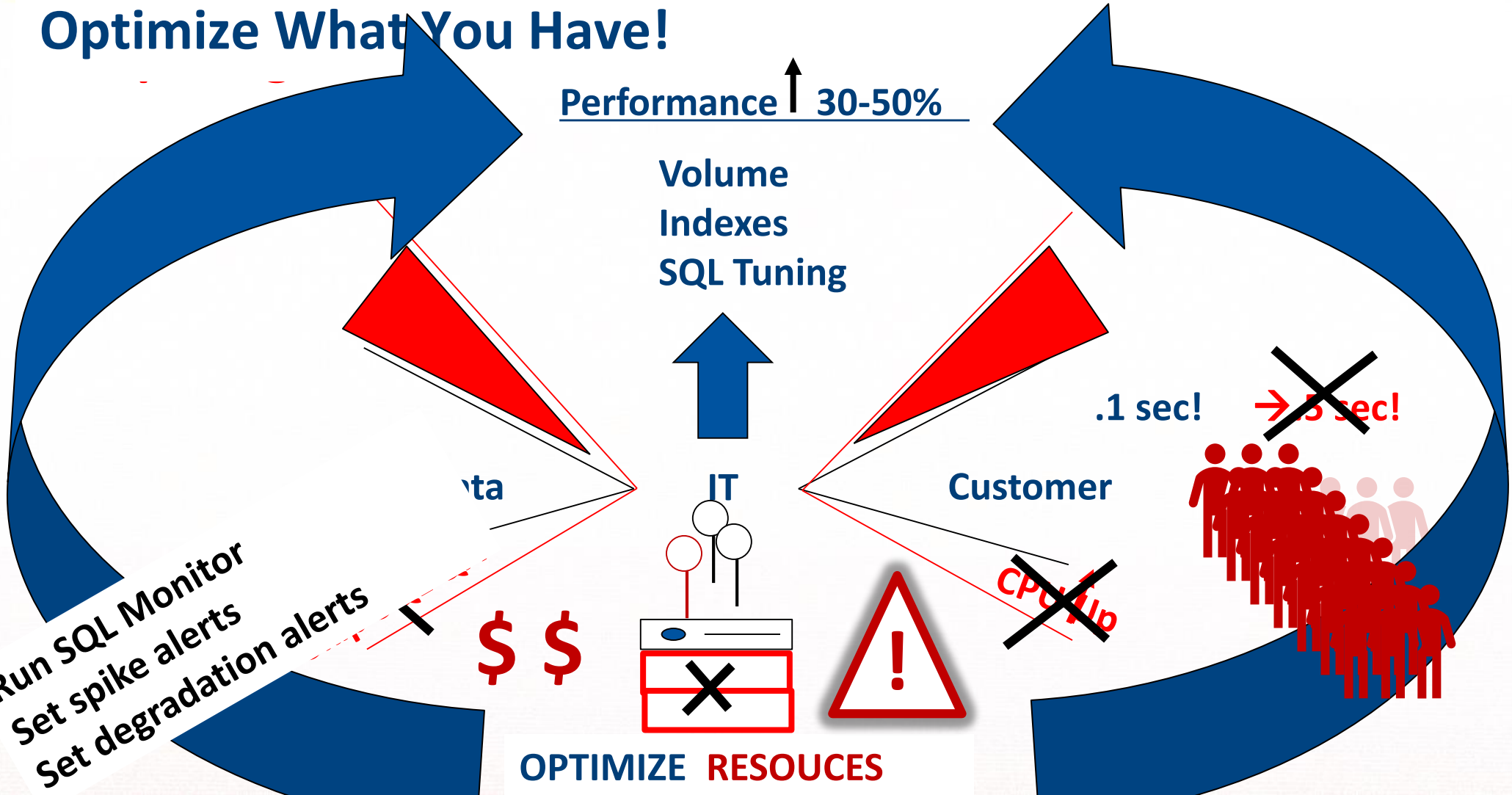
Sheryl Larsen
BMC

Session code: F16

9:20 AM Thursday, May 3, 2018

Db2 for z/OS

Optimize What You Have!



THEN WHAT?

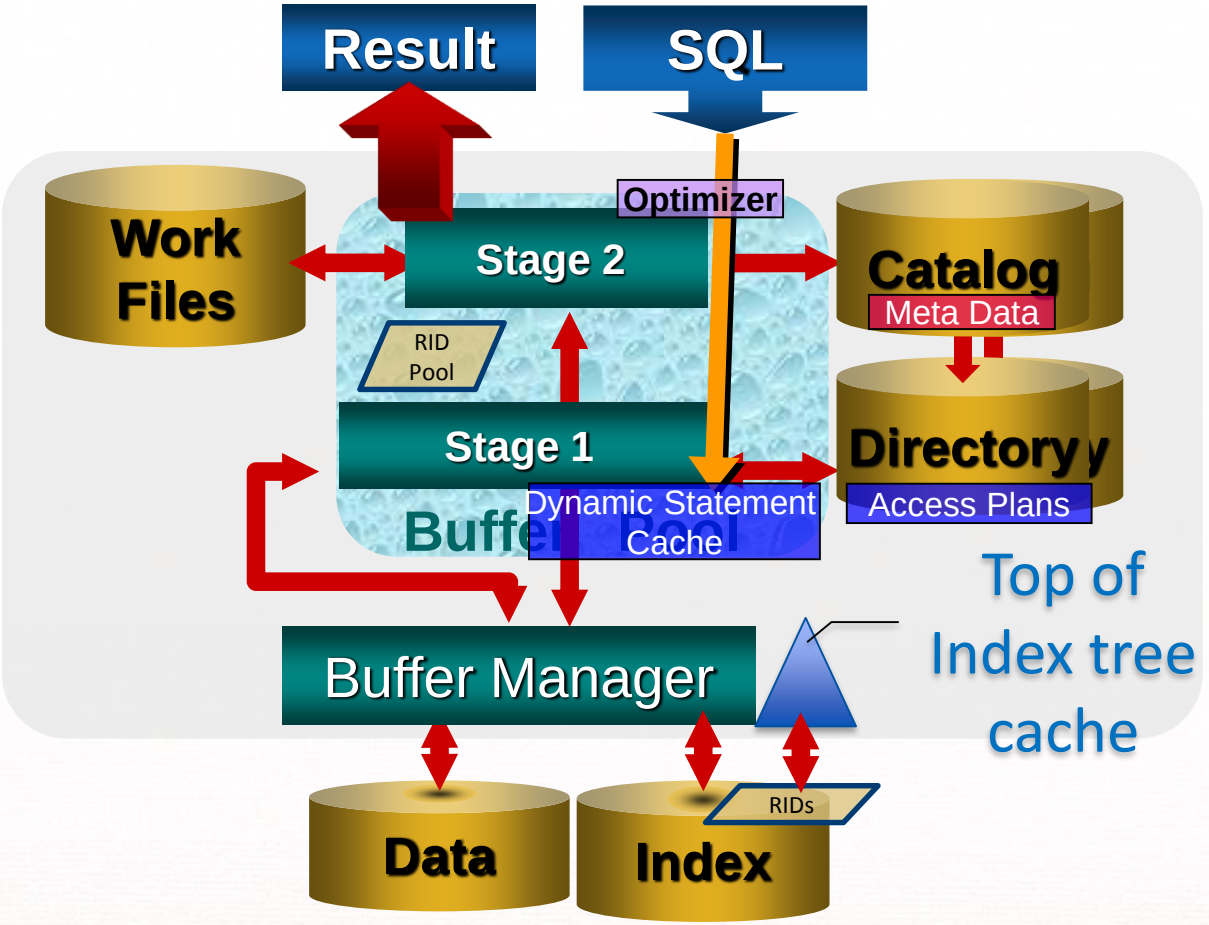
SQL Tuning Confidence Level

Table of Contents

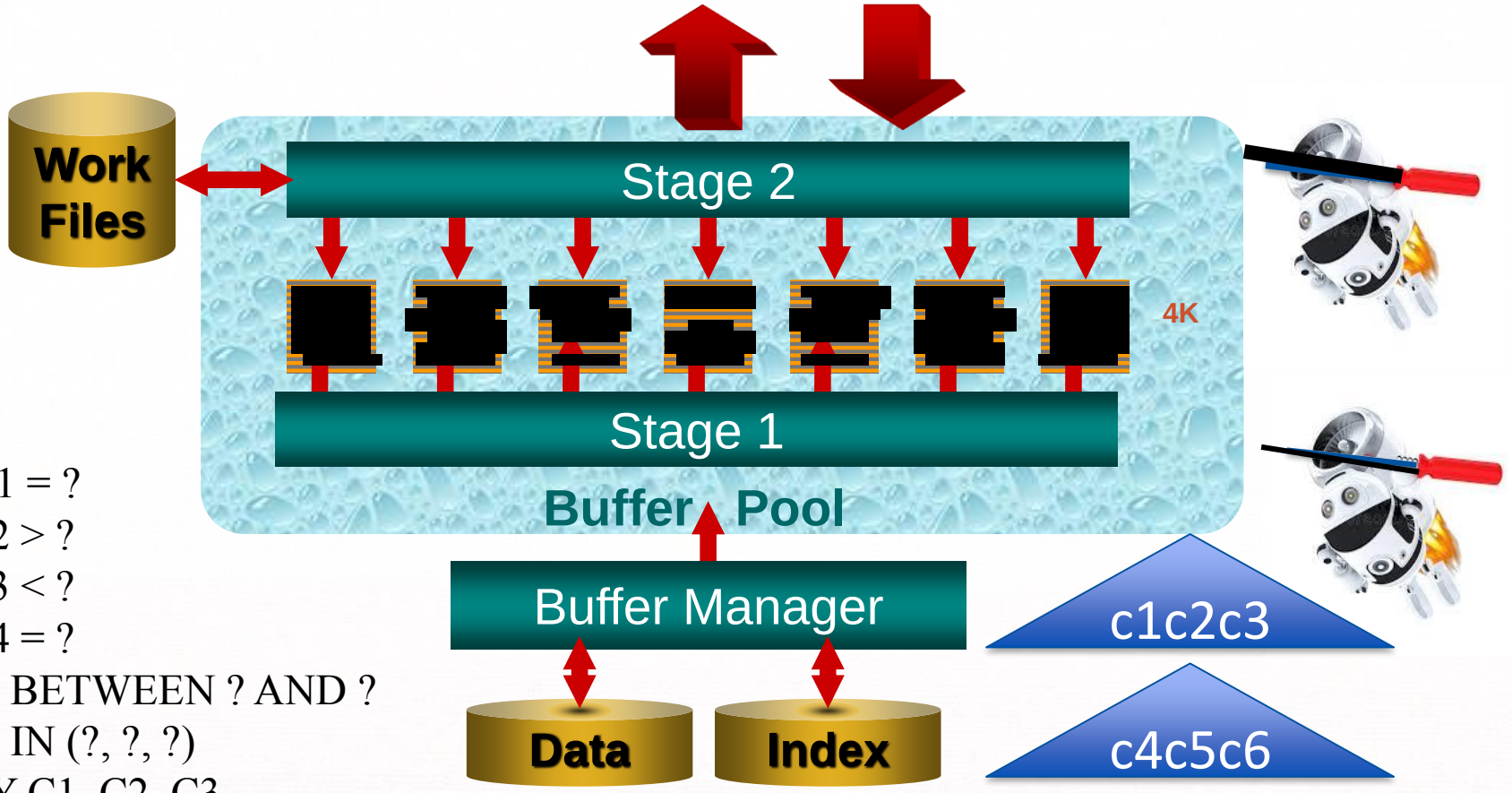
DB212 Catalog Poster Reference Guide

DB2 Engine Components

SQL Execution



Page Processing – z/OS




```
WHERE C1 = ?
AND     C2 > ?
AND     C3 < ?
AND     C4 = ?
AND     C5 BETWEEN ? AND ?
AND     C6 IN (?, ?, ?)
ORDER BY C1, C2, C3
```

Summary Of Predicate Processing

Indexable Stage 1 Predicates

Predicate Type	Indexable	Stage 1
COL = *value*	Y	Y
COL = *noncol expr*	Y	Y
COL IS NULL	Y	Y
COL *op value*	Y	Y
COL *op noncol expr*	Y	Y
COL BETWEEN *value1* AND *value2*	Y	Y
COL BETWEEN *noncol expr1* AND *noncol expr2*	Y	Y
COL LIKE '*pattern*'	Y	Y
COL IN (*list*)	Y	Y
COL LIKE *host variable*	Y	Y
T1.COL = T2.COL	Y	Y
T1.COL *op* T2.COL	Y	Y
COL=(*non subq*)	Y	Y
COL *op* (*non subq*)	Y	Y
COL *op* ANY (*non subq*)	Y	Y
COL *op* ALL (*non subq*)	Y	Y
COL IN (*non subq*)	Y	Y
COL = *expression*	Y	Y
(COL1,...COL*n*) IN (*non subq*)	Y	Y
(COL1, ...COLn) = (value1, ...valuen)	Y	Y
T1.COL = T2.colexpr	Y	Y
COL IS NOT NULL	Y	Y
COL IS NOT DISTINCT FROM value	Y	Y
COL IS NOT DISTINCT FROM noncol expression	Y	Y
COL IS NOT DISTINCT FROM col expression	Y	Y
COL IS NOT DISTINCT FROM non subq	Y	Y
T1.COL IS NOT DISTINCT FROM T2.COL	Y	Y
T1.COL IS NOT DISTINCT FROM T2.col expression	Y	Y

NOTs

Stage 1 Predicates

Predicate Type	Indexable	Stage 1
COL <> *value*	N	Y
COL <> *noncol expr*	N	Y
COL NOT BETWEEN *value1* AND *value2*	N	Y
COL NOT BETWEEN *noncol expr1* AND *noncol expr2*	N	Y
COL NOT IN (*list*)	N	Y
COL NOT LIKE ' *char*'	N	Y
COL LIKE '*%char*'	N	Y
COL LIKE '*_char*'	N	Y
T1.COL <> T2.COL	N	Y
T1.COL1 = T1.COL2	N	Y
COL <> (*non subq*)	N	Y
COL IS DISTINCT FROM	N	Y

1. Indexable = The predicate is a candidate for Matching Index access. When the optimizer chooses to use a predicate in

And partitioned filters limiting partitions are also applied.

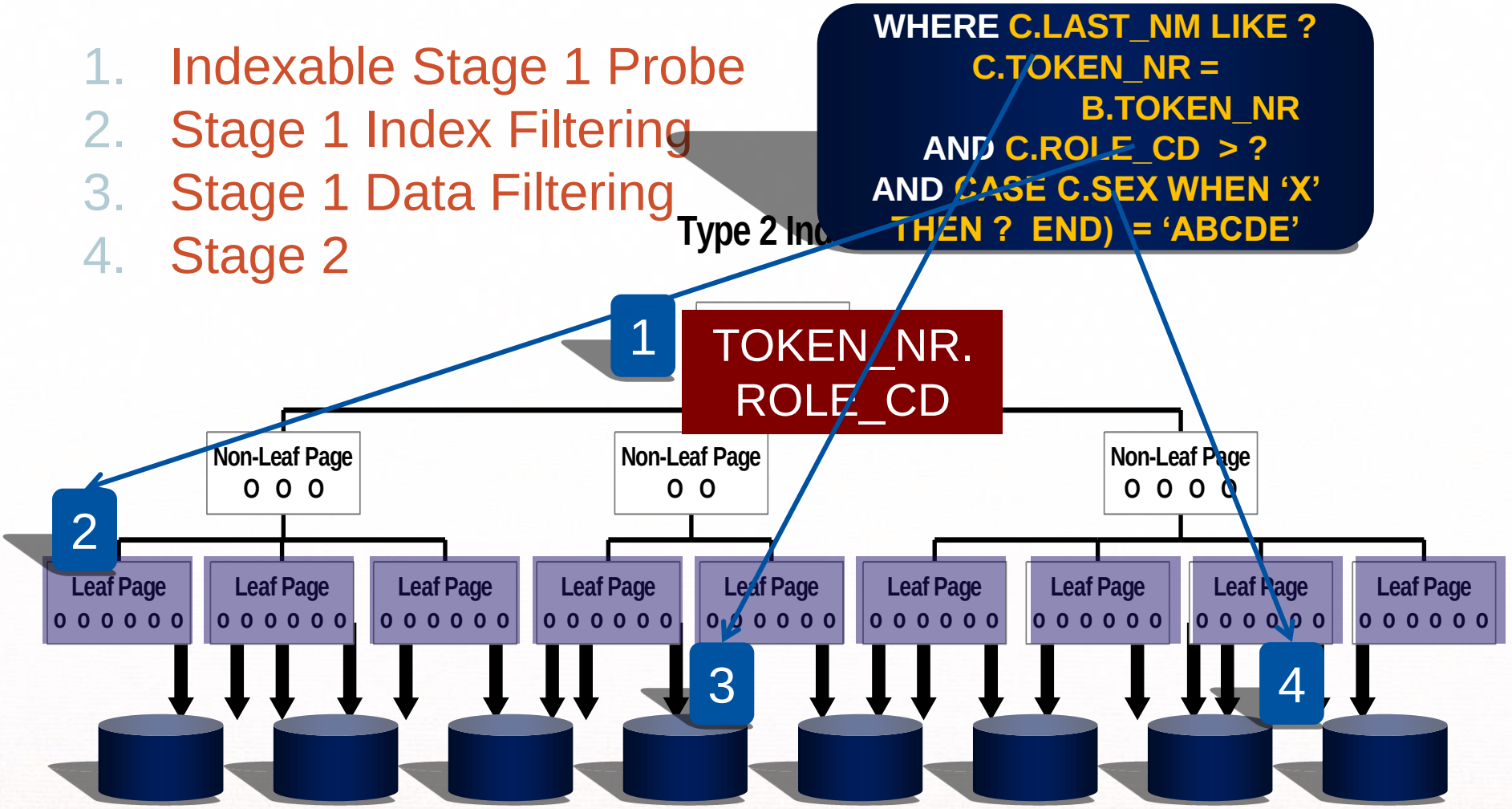
2.

3. Data Screening = The Stage 1 predicate is a candidate for filtering on the data pages. This is the third point of filtering in DB2.

4. Stage 2 = The predicate is not listed as Stage 1 and will be applied on the remaining qualifying pages from Stage 1. This is the fourth and final point of filtering in DB2.

Four Points of Filtering – DB2

1. Indexable Stage 1 Probe
2. Stage 1 Index Filtering
3. Stage 1 Data Filtering
4. Stage 2



SQL Review Checklist

1. Examine Program logic
2. Examine FROM clause
3. Verify Join conditions
4. Promote Stage 2's and Stage 1 NOTs
5. Prune SELECT lists
6. Verify local filtering sequence
7. Analyze Access Paths
8. Tune if necessary

When to Tune Queries

- Not until the query is coded the best it can be
- All predicates are the best they can be
  - Promote Stage 2's if possible
  - Promote Stage 1's if possible
  - Apply performance rules
- Check Access Paths of all Query Blocks
- Apply data knowledge and program knowledge to predict response time
- If, and only if, the predicted service levels are not met - TUNE!

How to Tune Queries

- Do not change statistics, just keep accurate
- Do not panic
- Choose a proven, low maintenance, tuning technique
- IBM's list:
  - OPTIMIZE FOR n ROWS
  - FETCH FIRST n ROWS ONLY
  - No Op (+0, CONCAT ' ')
  - TX.CX=TX.CX
  - REOPT(VARS)
  - ON 1=1

SQL Tuning Examples

```
WHERE S.SALES_ID > 44
    AND   S.MNGR = :hv-mngr
    AND   S.REGION BETWEEN
           :hvlo AND :hvhi CONCAT ' '
```

No Operation

```
SELECT  S.QTY_SOLD, S.ITEM_NO
                  , S.ITEM_NAME
              FROM       SALE S
          WHERE   S.ITEM_NO > :hv
            ORDER BY ITEM_NO
FETCH FIRST 22 ROWS ONLY
```

Limited Fetch

```
WHERE   B.BID BETWEEN
                  :hvlo AND :hvhi
    AND    B.BID = D.DID
             AND    B.SID = S.SID
AND B.COL2 >= :hv
AND B.COL3 >= :hv
AND B.COL4 >= :hv
```

Fake Filter

Tuning Tools

- Sheryl's Extended List
  - Fake Filtering
    - COL BETWEEN :hv1 AND :hv2
    - COL >= :hv
  - Table expressions with DISTINCT
    - FROM (SELECT DISTINCT COL1, COL2 …..)
  - Anti-Joins
  - Extreme Experiments
  - Index Changes
  - MQT Design

All the Possible Access Paths

Index	Table	Join
One Fetch IN(list) Index Access	Limited Partition Scan Using Non-partitioning index (NPI)	Nested Loop
Matching Index Access Sparse Index Access	Limited Partition Scan Using Partitioning Index	Hybrid Join: Type C or Type N
NonMatching Index Access	Limited Partition Scan Using Data Partitioned Secondary Index (DPSI	Star Join: Cartesian or Pair-wise
List Prefetch	Table Scan	Merge Scan
Multiple Index Access	Partitioned Table Scan	Direct Row

Db2 11

Makes Dynamic

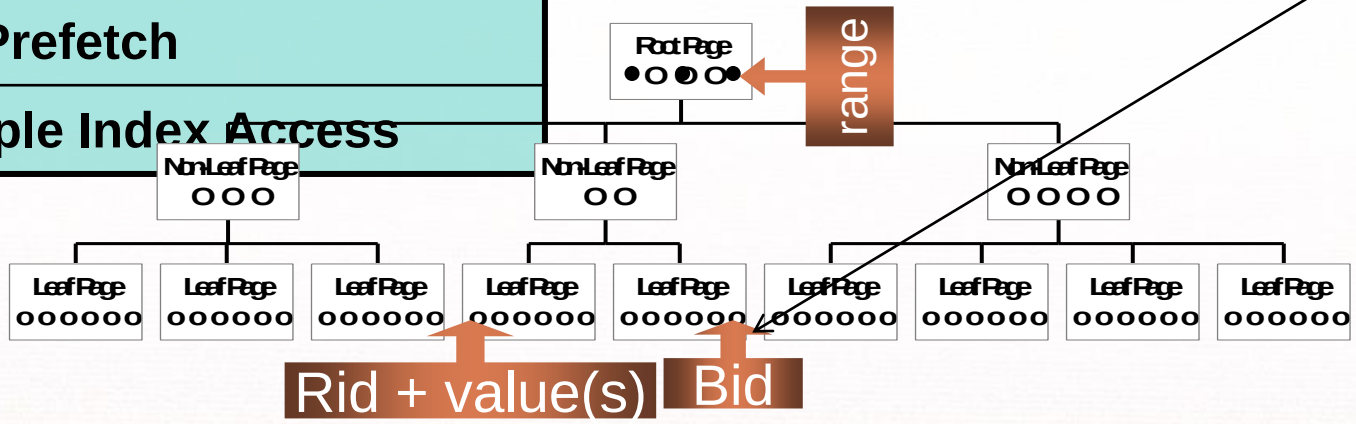
(Bold names use an Index)

Variations of Index Accesses

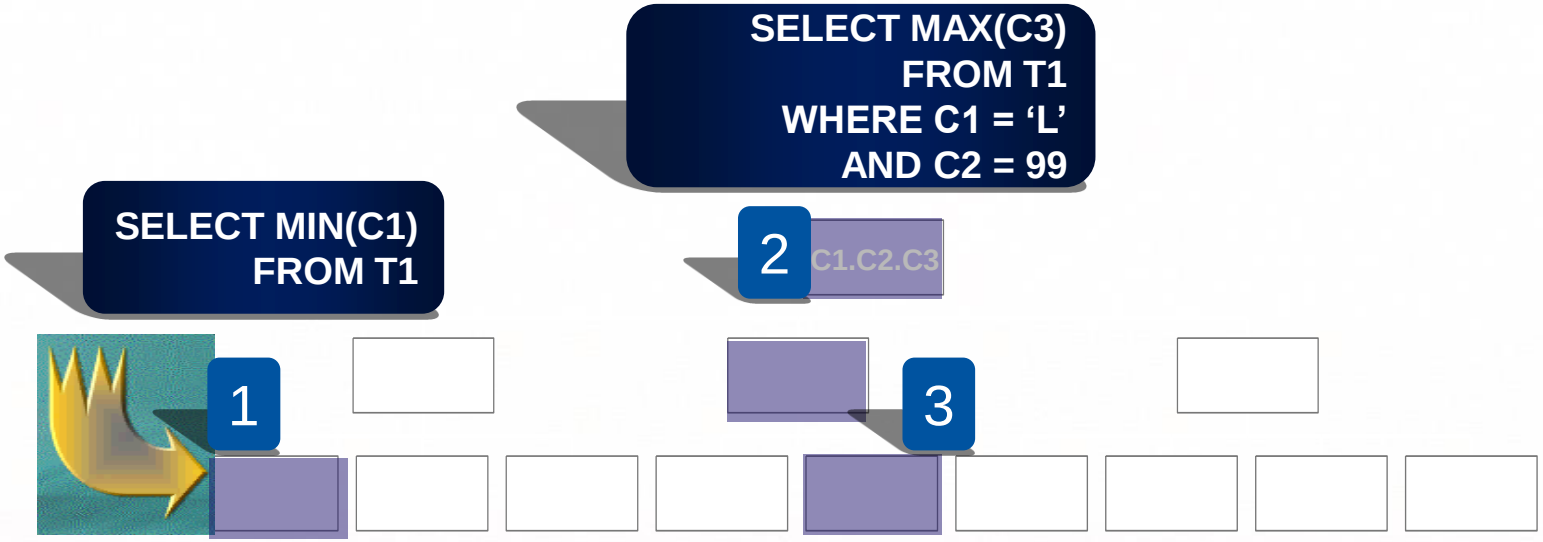
- One Fetch
- IN(list) Index Access
- Matching Index Access
- NonMatching Index Access
- Sparse Index Access
- List Prefetch
- Multiple Index Access

- All can be parallel
- All can omit data access
- Clustering index or nonclustering access

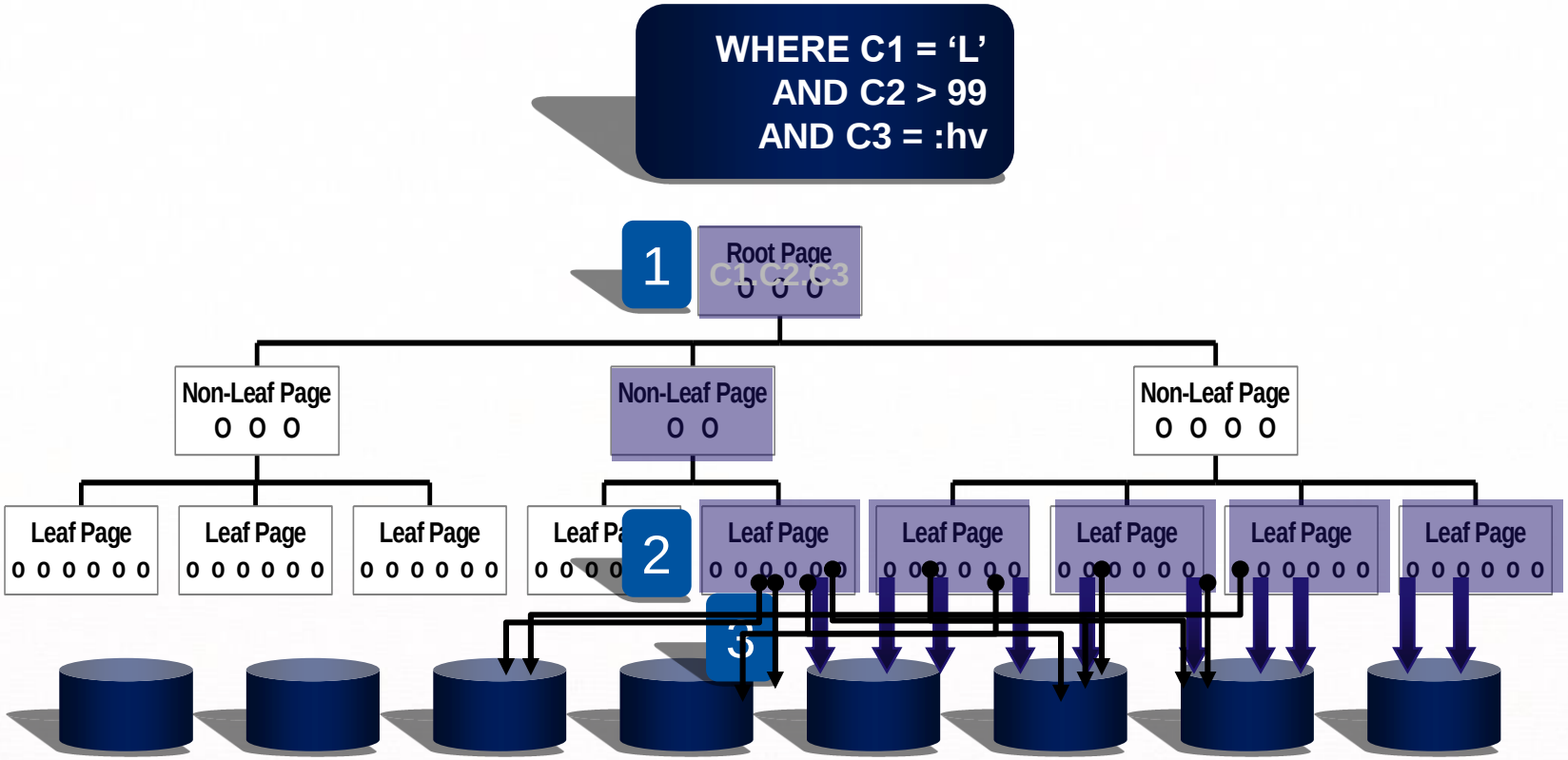
- **Limited Partition Scan With Partitioning Index**
- **Limited Partition Scan With NPI**
- **Limited Partition Scan With DPSI**
- Multidimensional Index Access Db2 for LUW



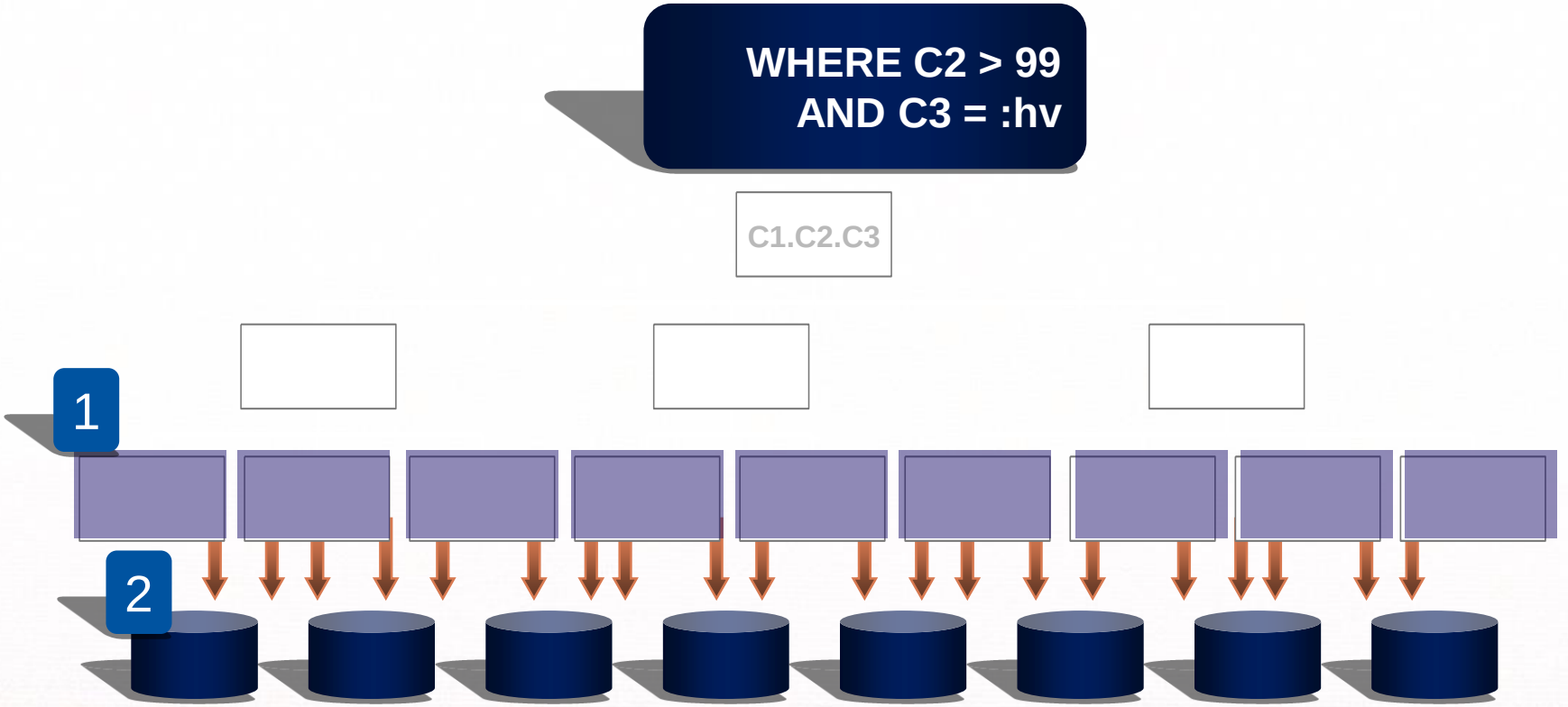
One Fetch



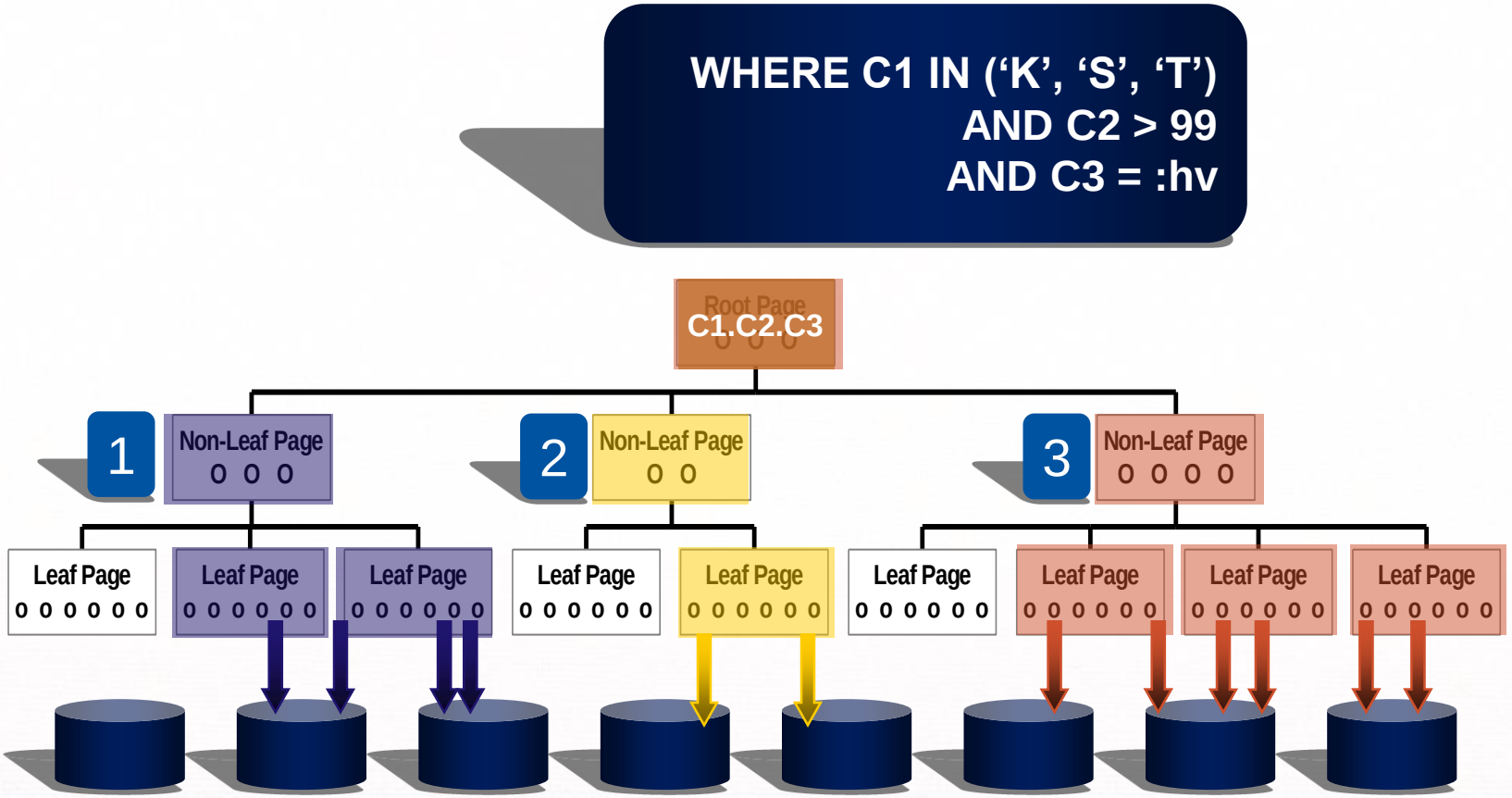
Matching Index Access



NonMatching Index Access

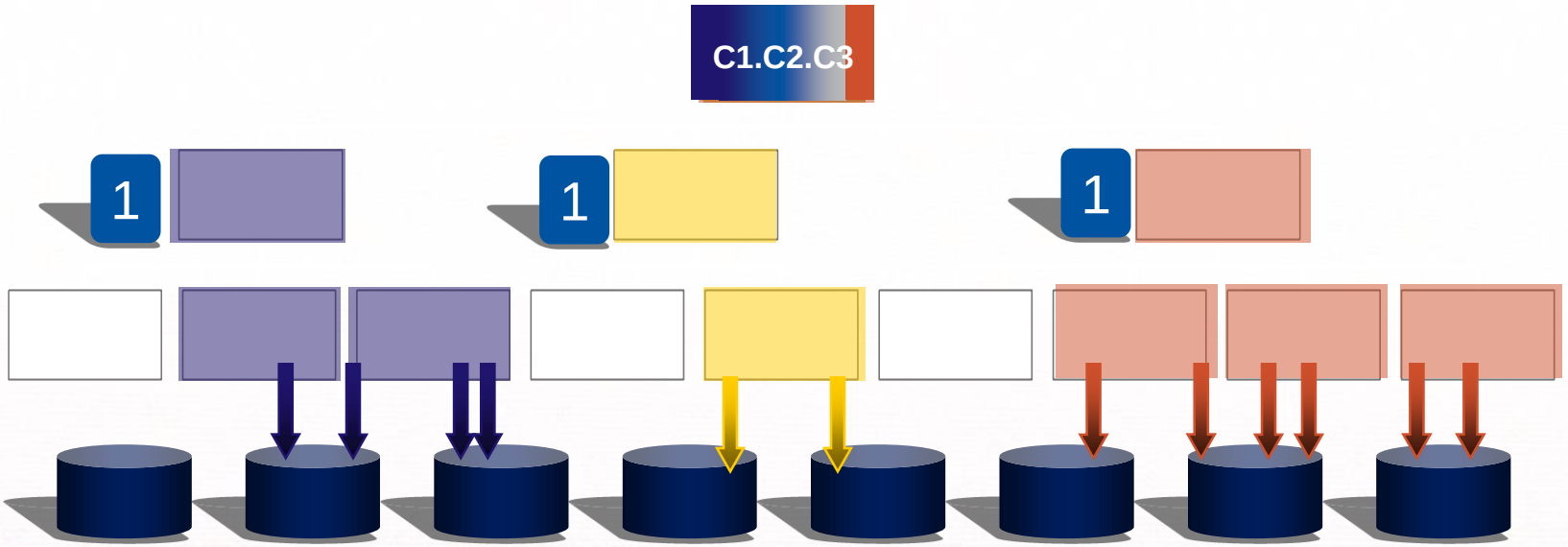


IN(list) Index Access

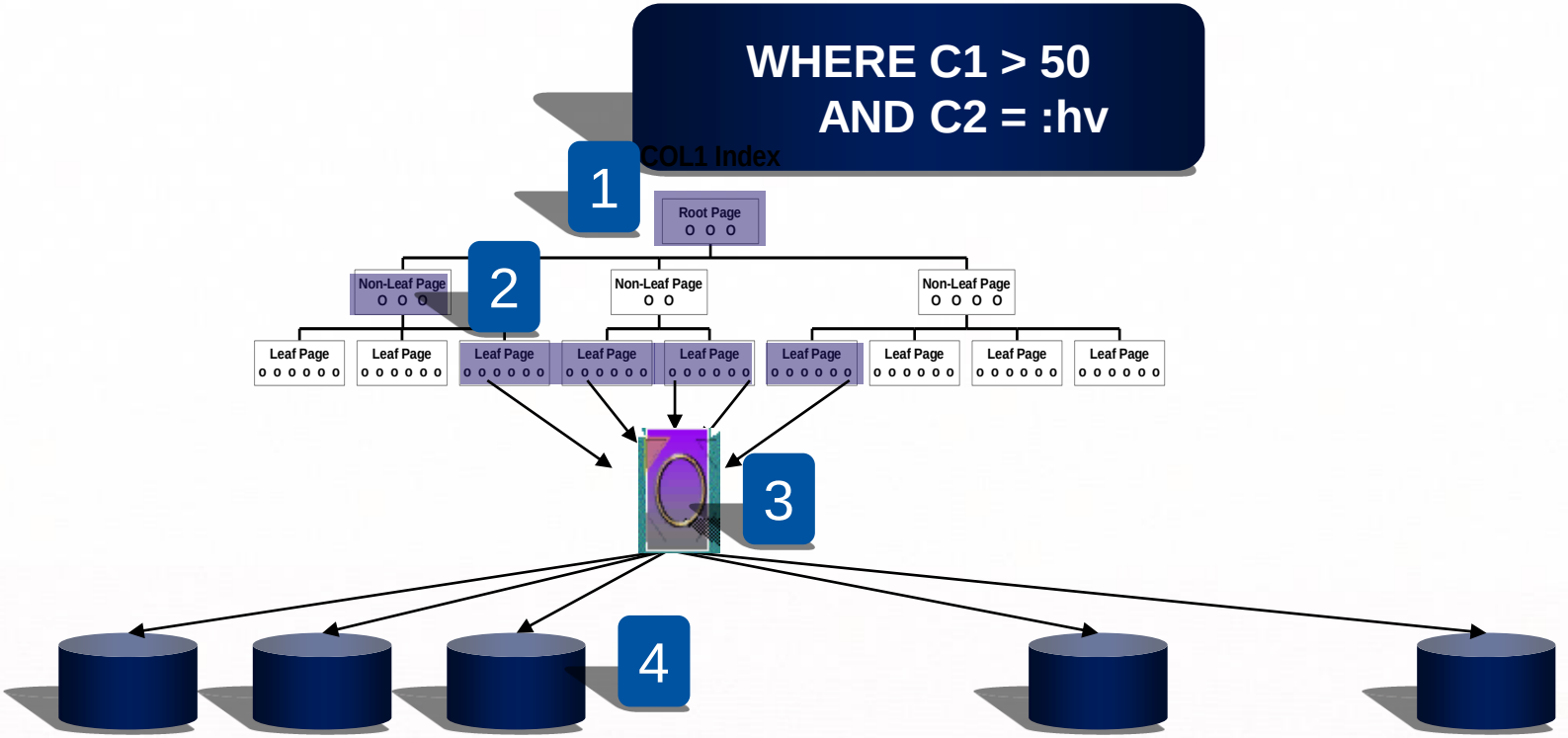


IN(list) Index Access -Parallel

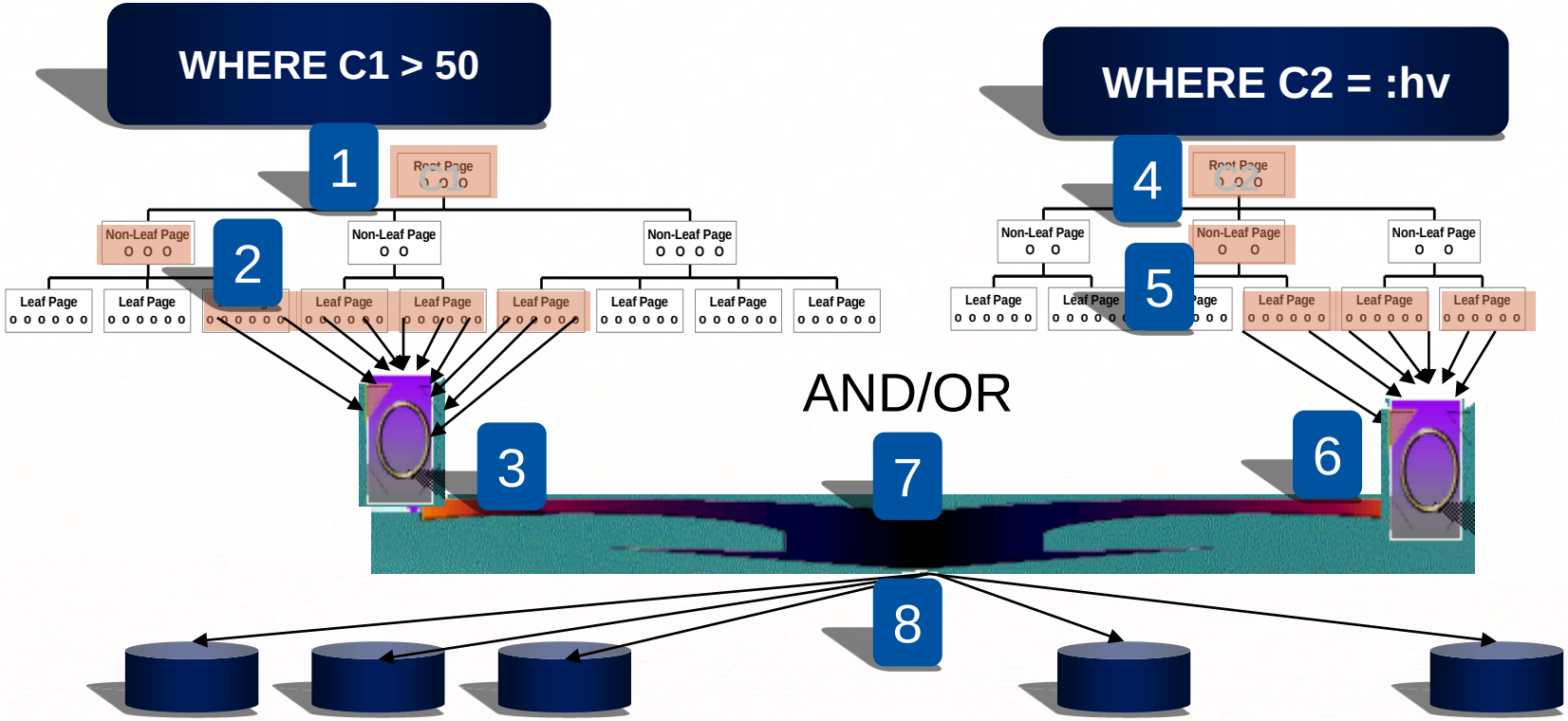
```
WHERE C1 IN ('K', 'S', 'T')
  AND C2 > 99
  AND C3 = :hv
```



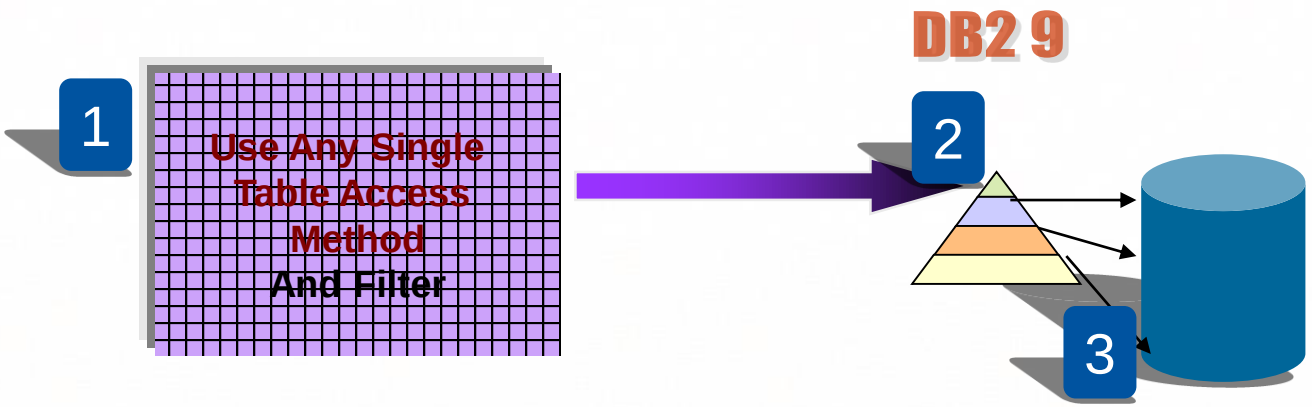
List Prefetch



Multiple Index Access

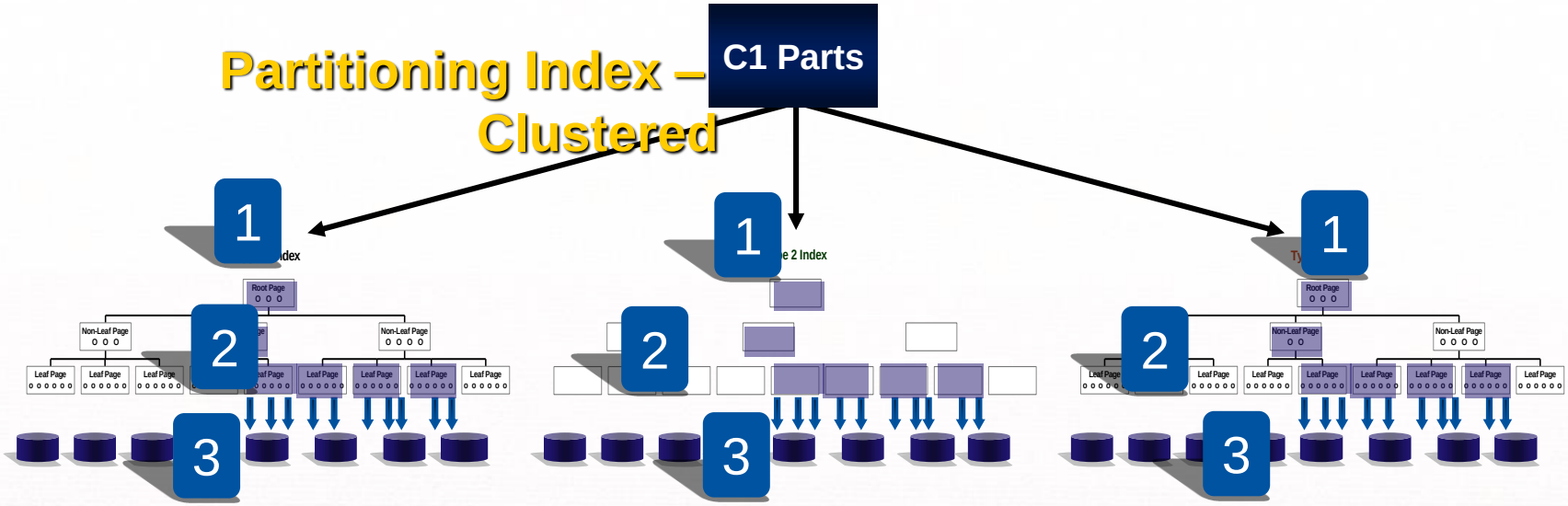


Sparse Index Access



Limited Partition Scan Using Clustered Partitioning Index

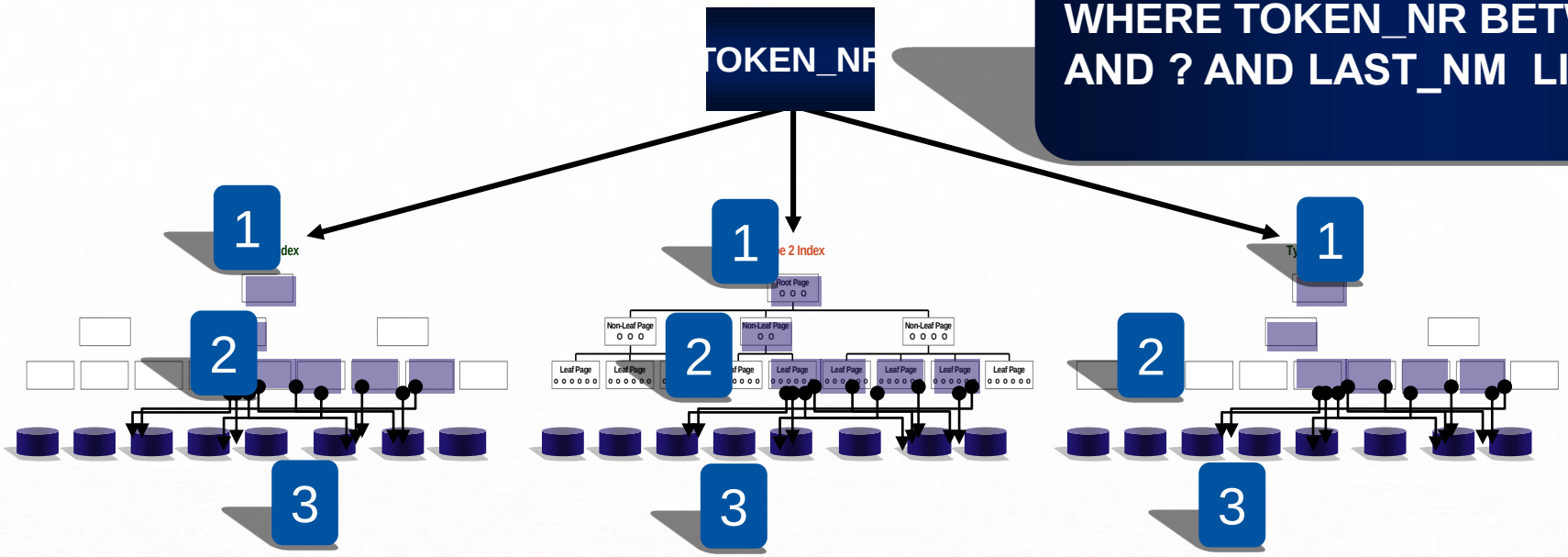
```
WHERE C1 IN ('K', 'S', 'T')
  AND C2 > 99
  AND C3 = :hv
```



Scan for Last Name Using Nonclustered Partitioning Index

Partitioning Index – Nonclustered

WHERE TOKEN_NR BETWEEN ?
AND ? AND LAST_NM LIKE 'LAR%'

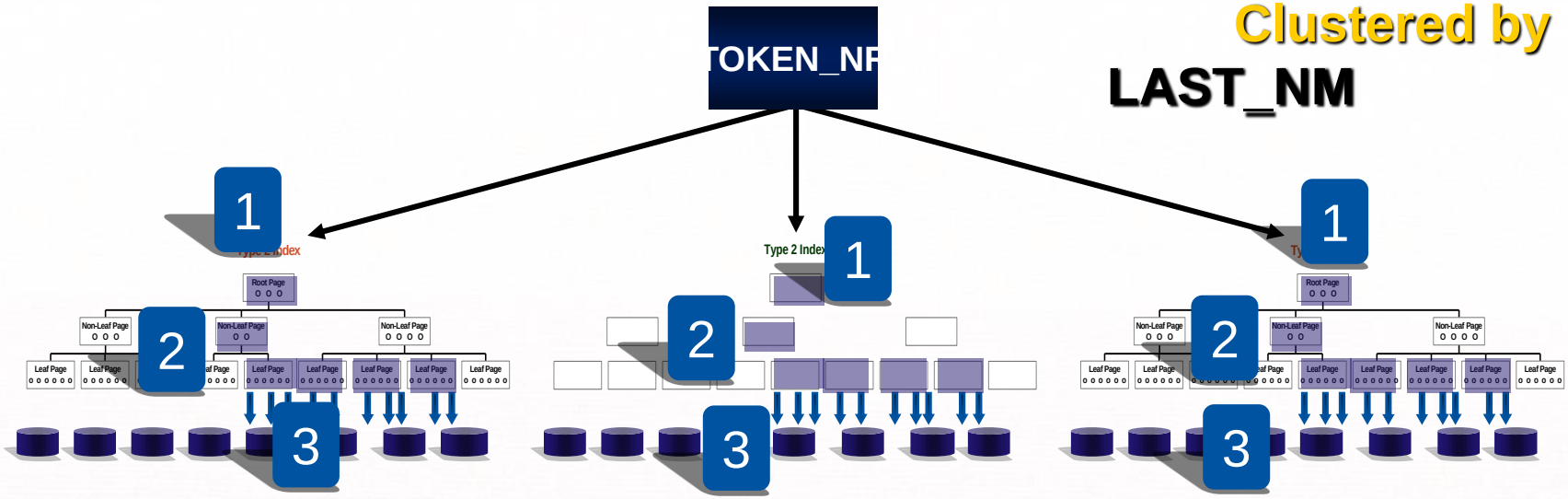


Limited Partition Scan Using DPSI

Partitioning by TOKEN_NR

```
WHERE TOKEN_NR  BETWEEN ? AND
? AND LAST_NM LIKE '%LAR
```

DPSI = Data Partitioned Secondary Index



Sample Customer Table in Optimal Condition

Table Columns

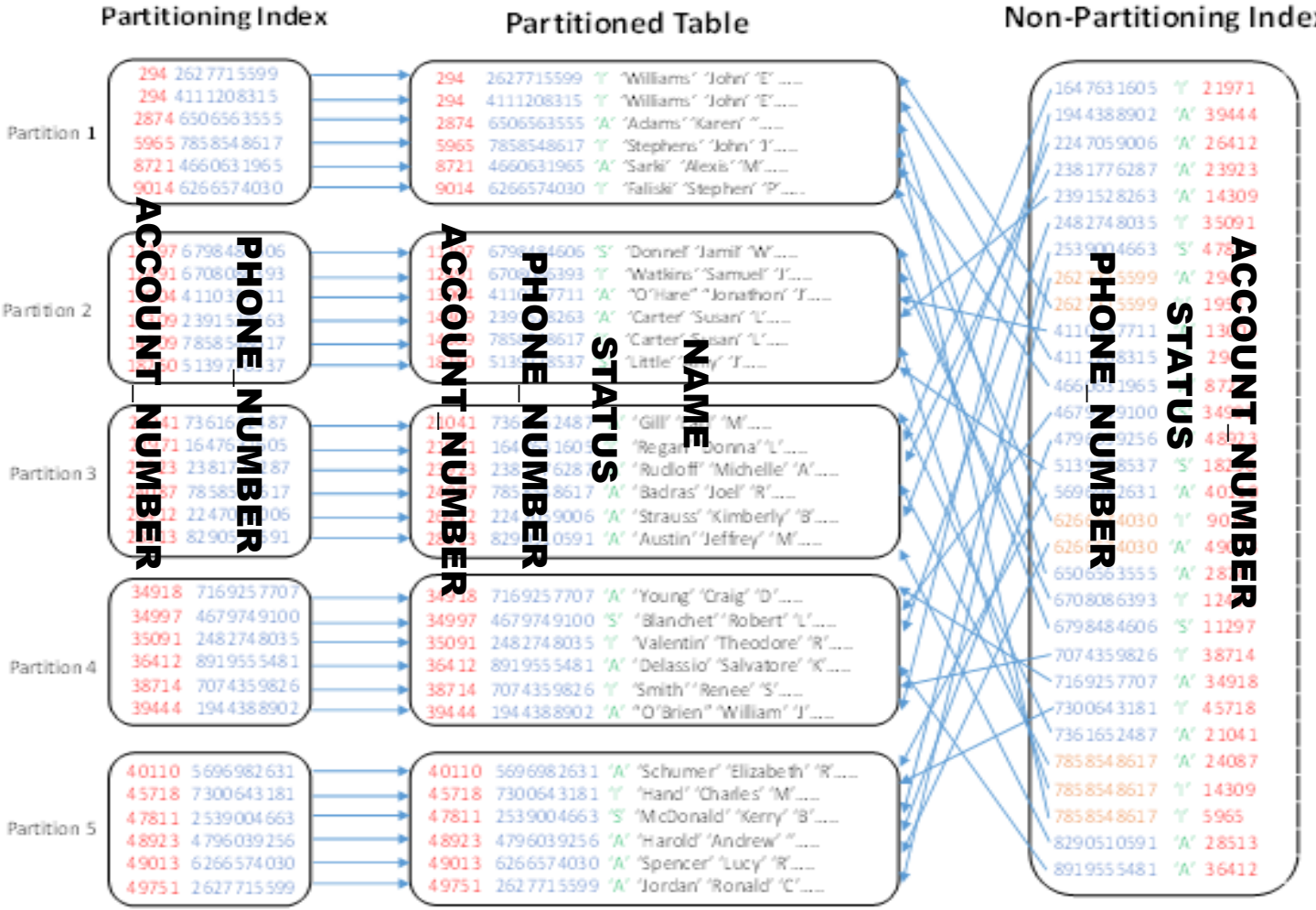
ACCOUNT_NUMBER
PHONE_NUMBER
ACCOUNT_STATUS
LAST_NAME
FIRST_NAME
MIDDLE_INIT
HOME_ADDRESS
EMAIL_ADDRESS
LAST_UPDATE_DATE

Partitioning Index Key

ACCOUNT_NUMBER
PHONE_NUMBER

Non-Partitioning Index Key

PHONE_NUMBER
ACCOUNT_STATUS
ACCOUNT_NUMBER



Thanks Tom Beck!

Variations of Table Access

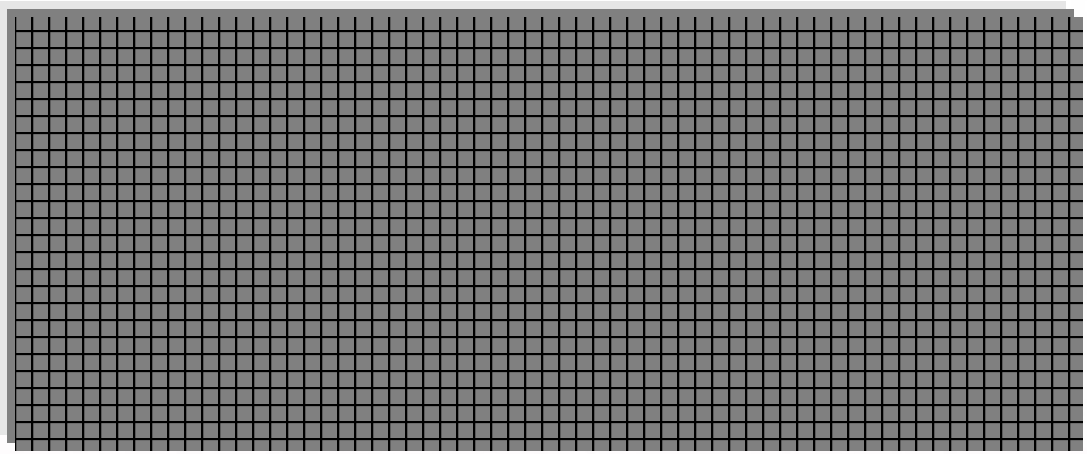
- All can be parallel
- If not enough room in memory, at run time create sparse index instead

Segmented
Partitioned
Limited Partitioned
In Memory Data Cache

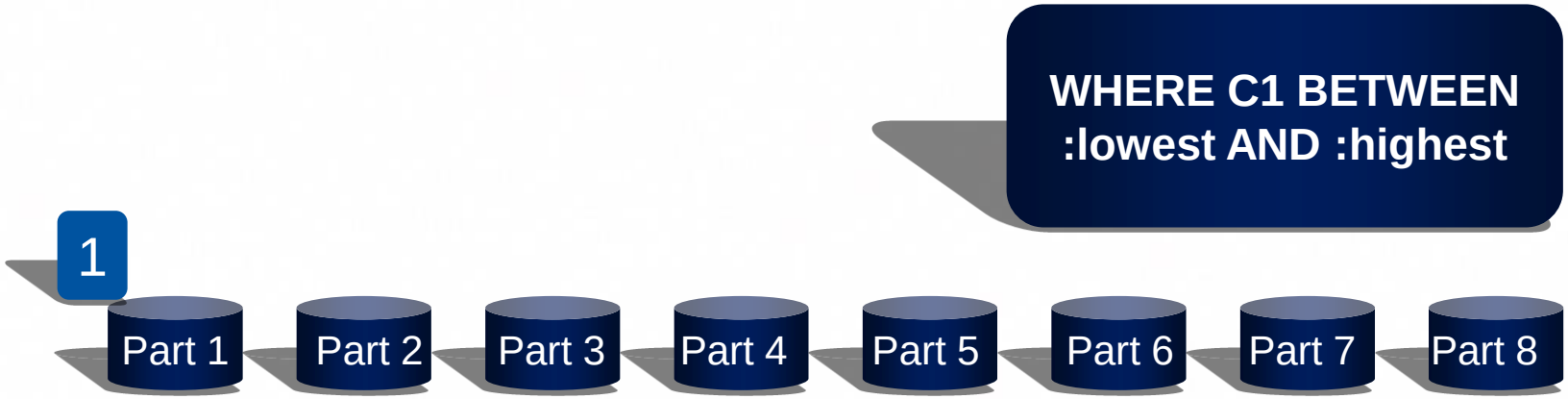
Table Scan

WHERE C1 BETWEEN :lowest AND :highest

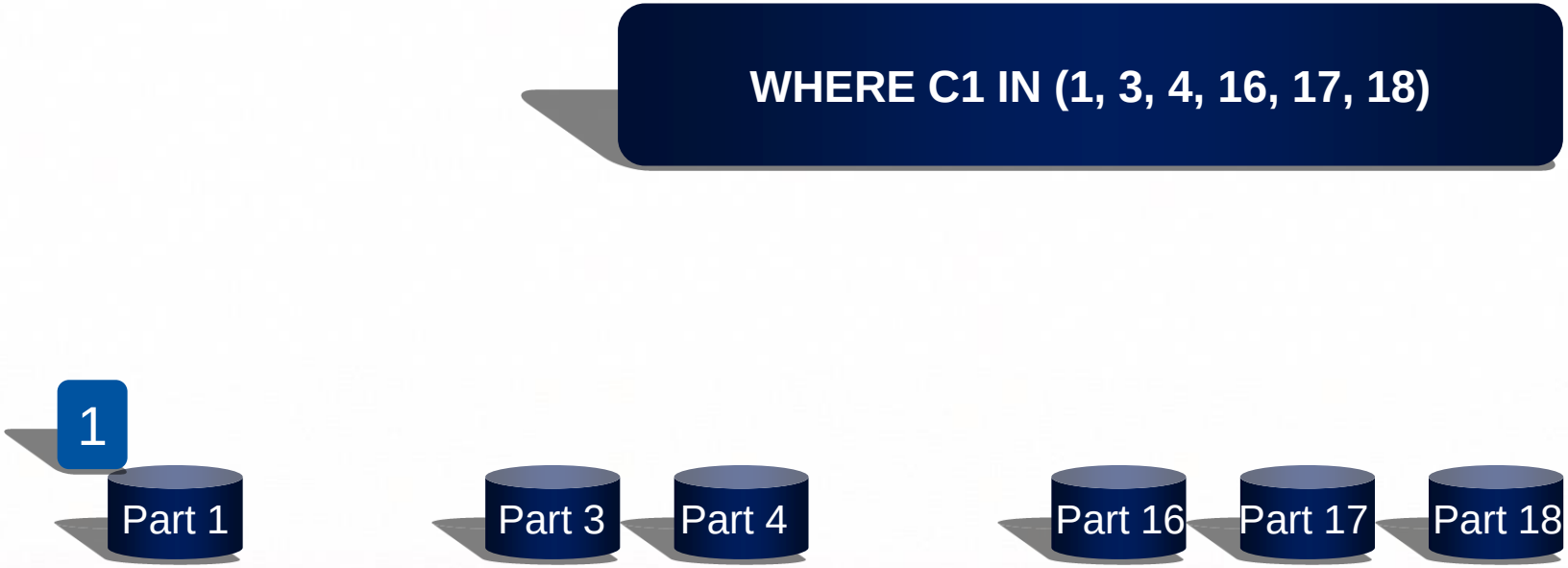
1



Partitioned Table Scan



Limited Partitioned Table Scan



Variations of Join Methods

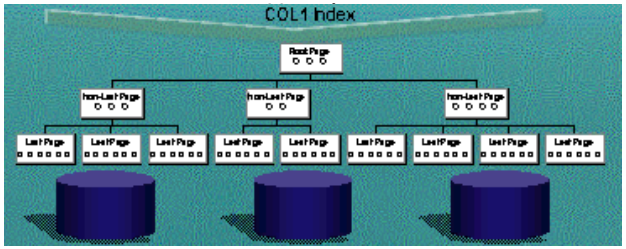
- All choose outer table and filter first
- All can be parallel (Star CPU only)
- Worry about join table sequence instead of join method

Nested Loop
Hybrid Join Type C
Hybrid Join Type N
Merge Scan Join
Star Join – Cartesian
Star Join – Pair Wise

Nested Loop Join

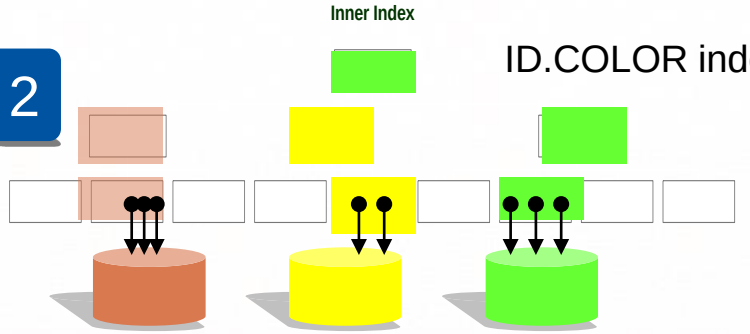
```
WHERE
T1.ORDER >
222
AND
T1.ID = T2.ID
AND
T2.COLOR IN
('BLUE',
'YELLOW',
'GREEN')
```

1

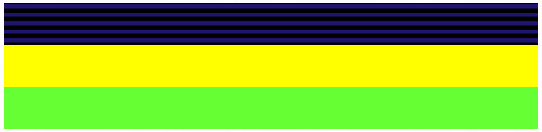


2

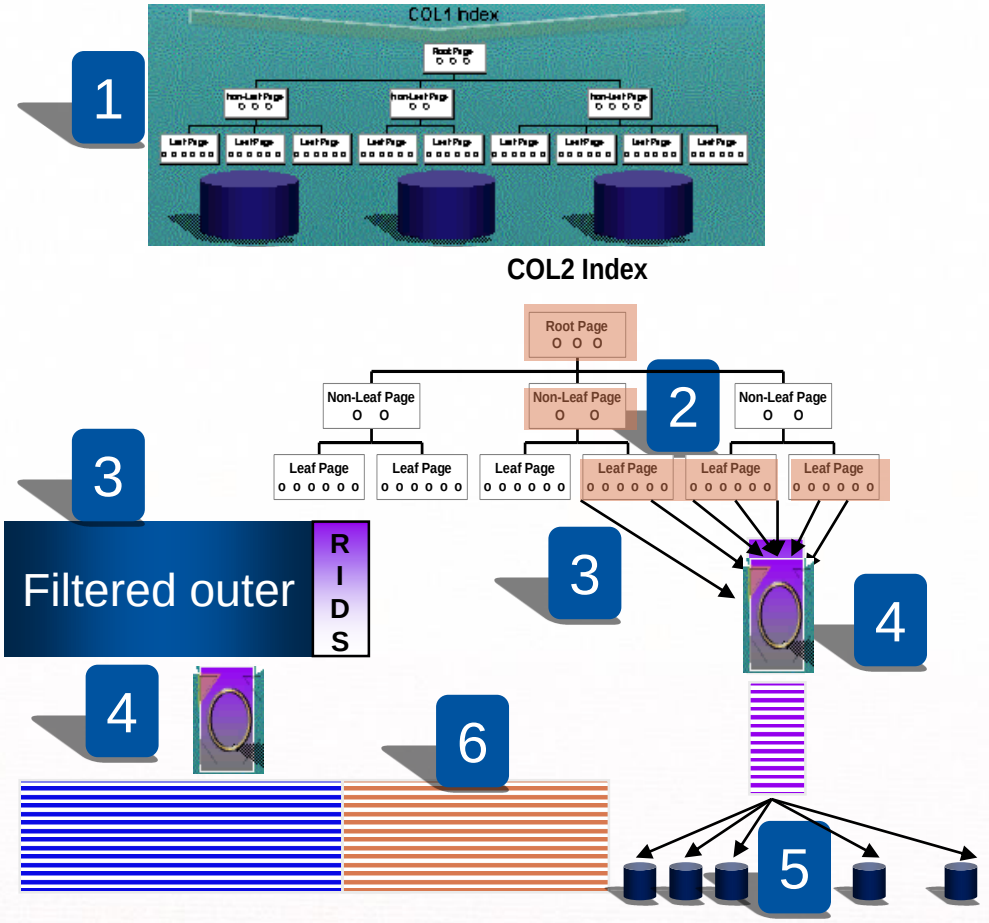
3



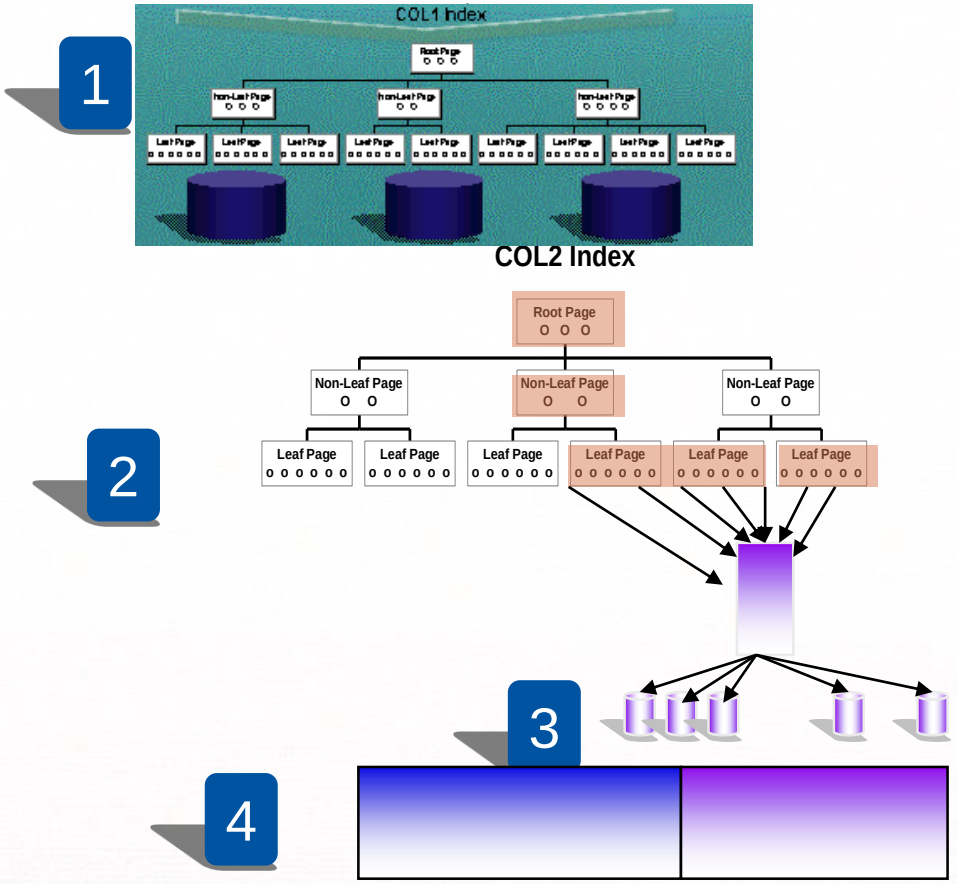
4



Hybrid Join – Type N

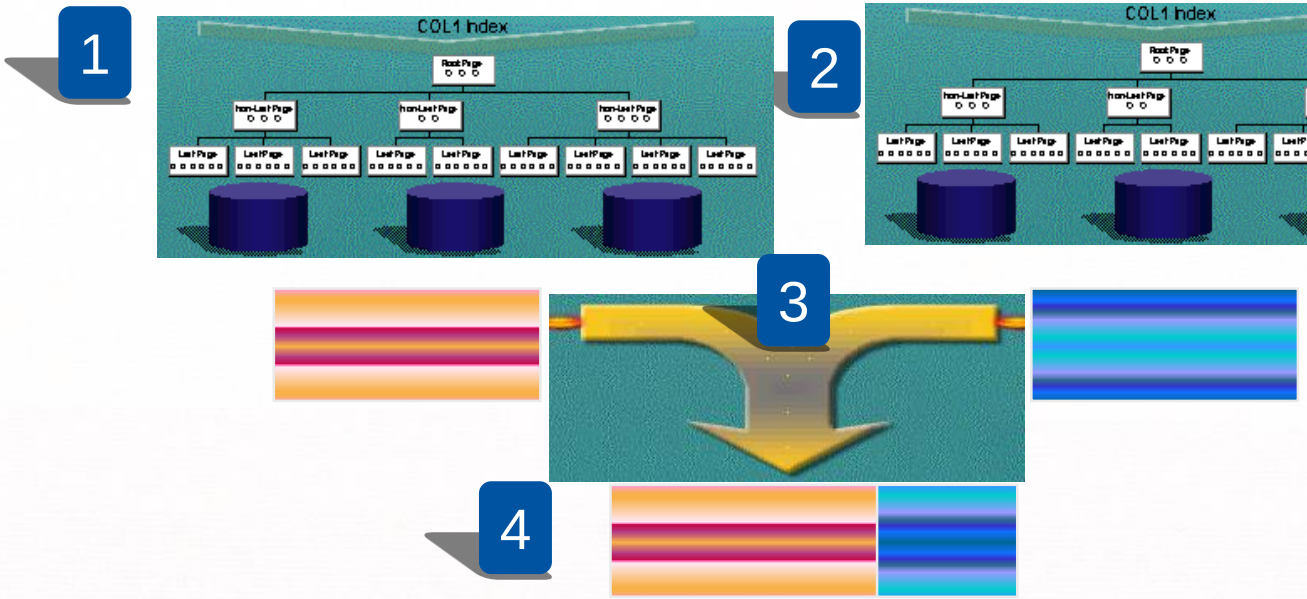


Hybrid Join – Type C

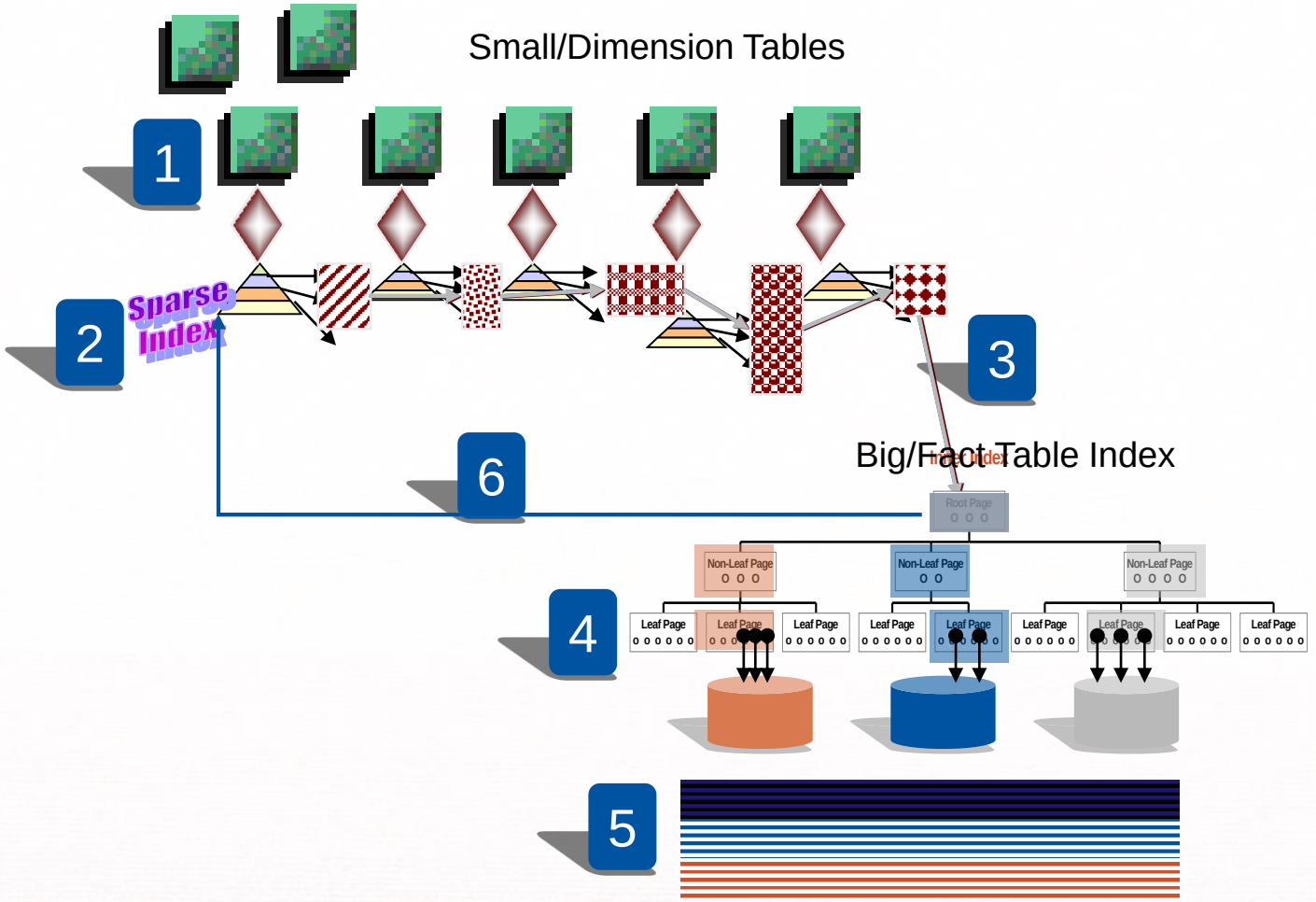


Merge Scan Join

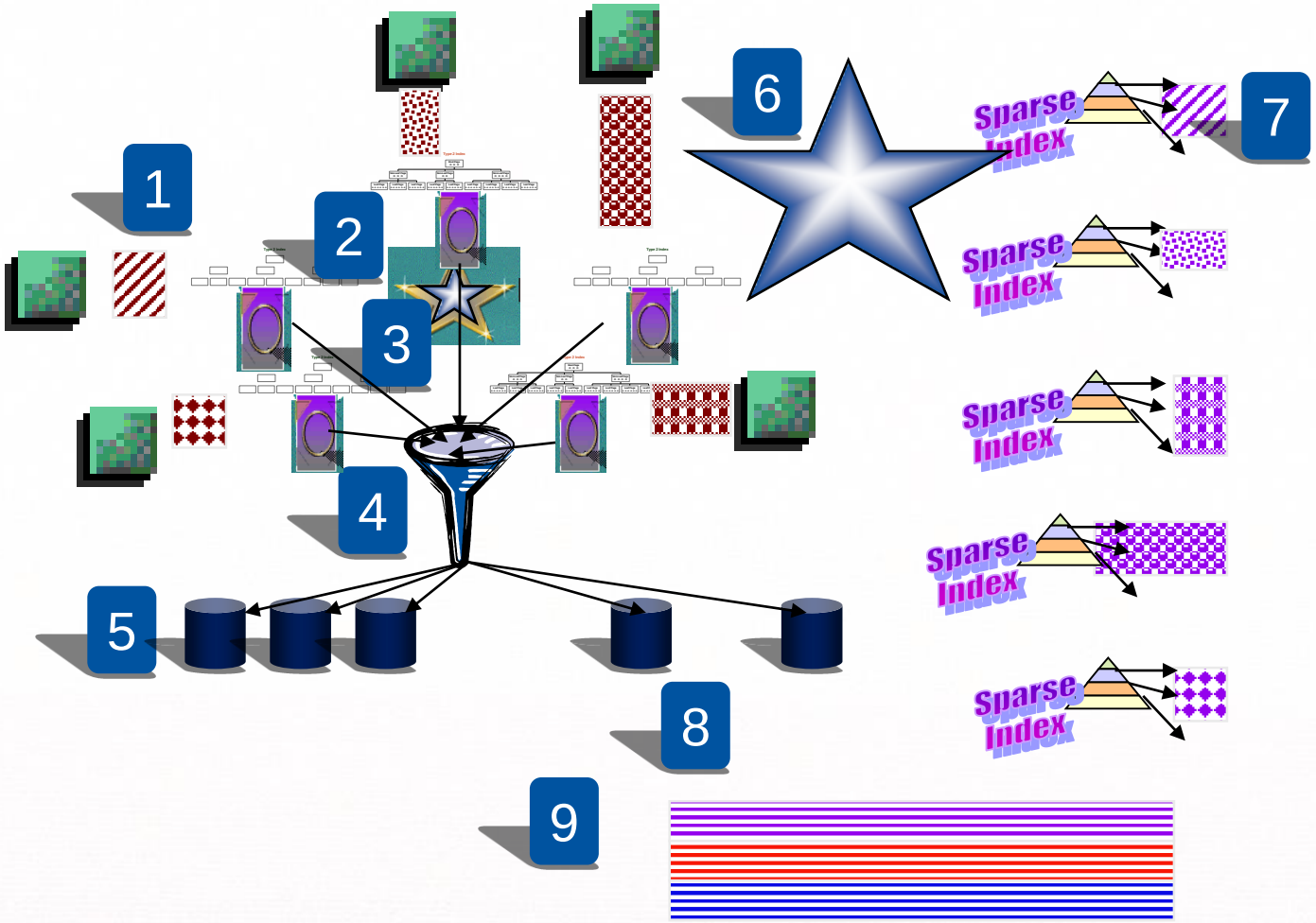
```
WHERE
T1.C1 = T2.CA AND
T1.C2 = T2.CB AND
T1.C3 = T2.CC
```



Star Join - Cartesian

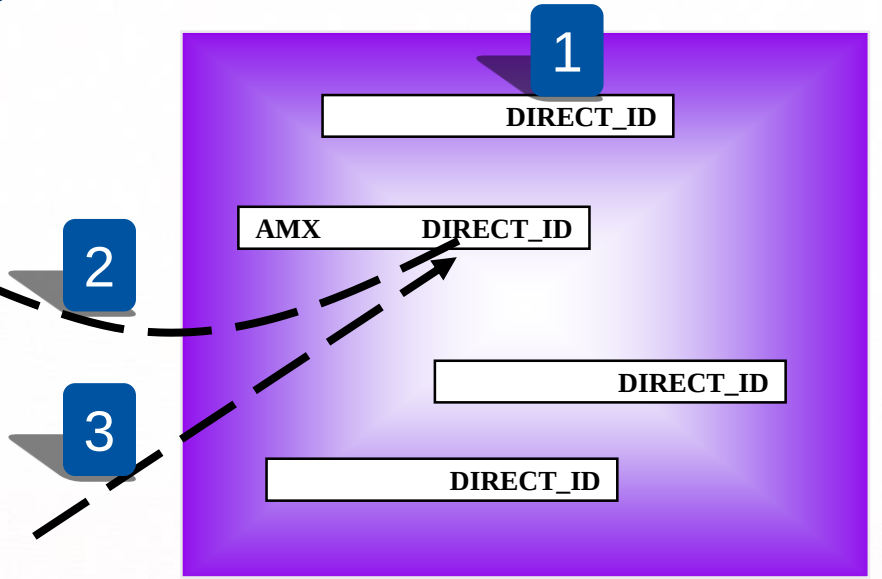


Pair-Wise Star Join

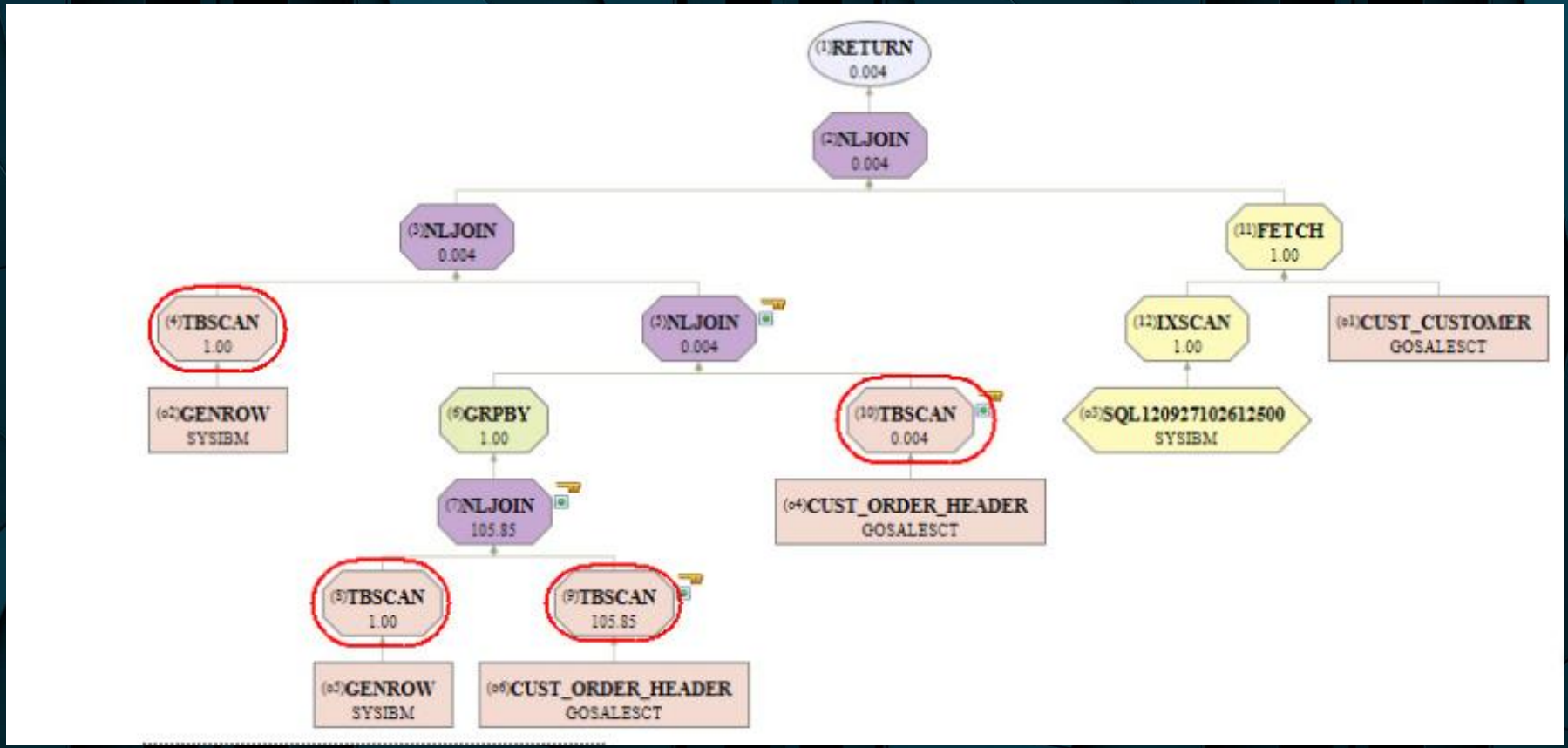


Read Direct Access

1. Create table with ROWID type column (DIRECT_ID)
2. SELECT DIRECT_ID
   INTO :direct-id
   FROM TAB12
   WHERE UKEY = 'AMX'
3. UPDATE TAB12
   WHERE
     DIRECT_ID = :direct-id



Access Path Analysis



The larger the graph and the more rows involved, the more costly it is.

Tuning SQL

- FIND ALL Expensive Queries

```
---------+----------+---
PROGNAME         PROCSU
---------+----------+---
EXPNPROG    121,059,664
EXPNPROG     21,059,664
ONESECPG         79,664
SUBSECPG          9,664
CHEEPPRG             64
FREEPROG              4
```

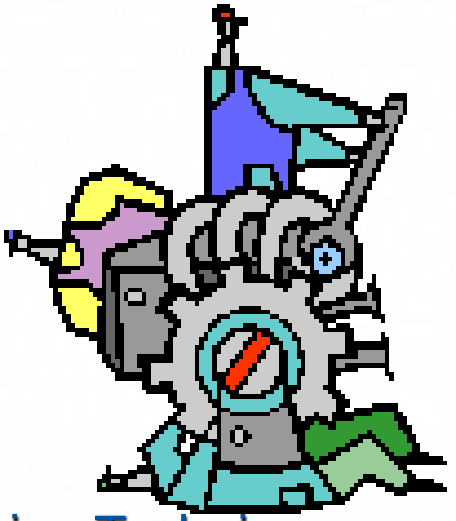
PROCSU is Too Expensive to Calculate!

2,147,483,647

Tuning Techniques to Apply When Necessary

Learn Traditional Tuning Techniques

- OPTIMIZE FOR n ROWS
- No Ops
- Fake Filtering
- ON 1 = 1
- Index & MQT Design



Experiment with Extreme Tuning Techniques

- DISTINCT Table Expressions
- Odd/old Techniques
- Anti-Joins
- Manual Query Rewrite

OPTIMIZE FOR n ROWS
FETCH FIRST n ROWS

- Both clauses influence the Optimizer
  - To encourage index access and nested loop join
  - To discourage list prefetch, sequential prefetch, and access paths with Rid processing
  - Use FETCH n = total rows required for set
  - Use OPTIMIZE n = number of rows to send across network for distributed applications
  - Works at the statement level

Fetch First Example

Query #1

```
SELECT  S.QTY_SOLD
      , S.ITEM_NO
      , S.ITEM_NAME
  FROM        SALE S
 WHERE   S.ITEM_NO > :hv
 ORDER BY ITEM_NO
```

- Optimizer choose List Prefetch Index Access + sort for ORDER BY for 50,000 rows
- All qualifying rows processed (materialized) before first row returned = .81 sec
- <.1sec response time required

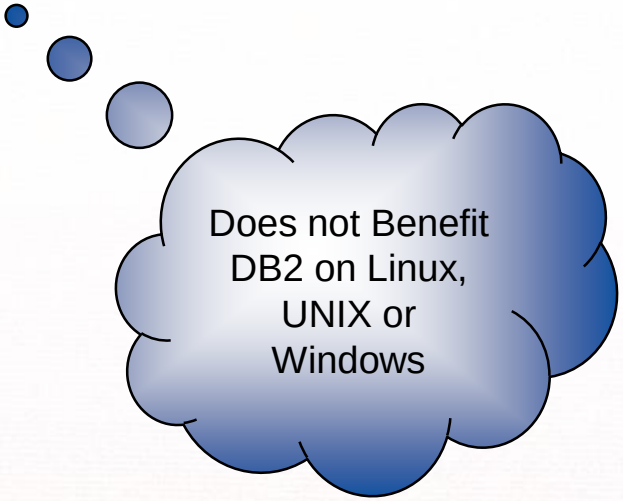
Query #1 Tuned

```
SELECT  S.QTY_SOLD, S.ITEM_NO
      , S.ITEM_NAME
  FROM        SALE S
 WHERE   S.ITEM_NO > :hv
 ORDER BY ITEM_NO
FETCH FIRST 22 ROWS ONLY
```

- Optimizer now chooses Matching Index Access (first probe .004 sec)
- No materialization
- Cursor closed after 22 items displayed (22 * .0008 repetitive access)
- **.004 + .017 = .021 sec**

No Operation (No Op)

- +0, CONCAT ‘ ‘ also –0, *1, /1
  - Place no op next to predicate
  - Use as many as needed
  - Discourages index access, however, preserves Stage 1
  - Can Alter table join sequence
  - Can fine tune a given access path
  - Can request a table scan
  - Works at the predicate level



No Op Example CONCAT ' '

SALES_ID.MNGR.REGION Index | MNGR Index | REGION Index

```
SELECT  S.QTY_SOLD
      , S.ITEM_NO
      , S.ITEM_NAME
  FROM        SALE S
WHERE   S.SALES_ID > 44
 AND  S.MNGR = :hv-mngr
 AND S.REGION BETWEEN
      :hvlo AND :hvhi
 ORDER BY S.REGION
```

```
.....
      FROM        SALE S
 WHERE S.SALES_ID > 44
  AND     S.MNGR = :hv-mngr
  AND     S.REGION BETWEEN
   :hvlo AND :hvhi CONCAT ' '
      ORDER BY R.REGION
```

- Optimizer chooses Multiple Index Access
- The table contains 100,000 rows and there are only 6 regions
- Region range qualifies 2/3 of table
  - <.1sec response time required
  - No Op allows Multiple Index Access to continue on first 2 indexes
- Two Matching index accesses, two small Rid sorts, & Rid intersection

No Op Example - Scan

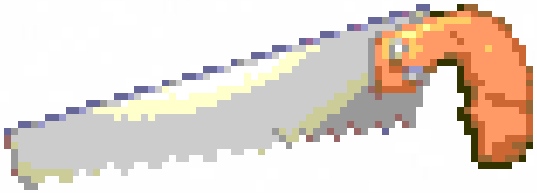
SALES_ID.MNGR.REGION Index | **MNGR Index** | **REGION Index**

```
SELECT  S.QTY_SOLD
      , S.ITEM_NO
      , S.ITEM_NAME
FROM        SALE S
WHERE   S.SALES_ID > 44 +0
AND   S.MNGR = :hv-mngr CONCAT ' '
AND     S.REGION BETWEEN
:hvlo AND :hvhi CONCAT ' '
ORDER BY S.REGION
FOR FETCH ONLY
WITH U
```

- If you know the predicates do very little filtering, force a table scan
- Use a No Op on ***every*** predicate
  - This forces a table scan
- FOR FETCH ONLY encourages parallelism
- WITH UR for read only tables to reduce CPU

Should this be Documented?

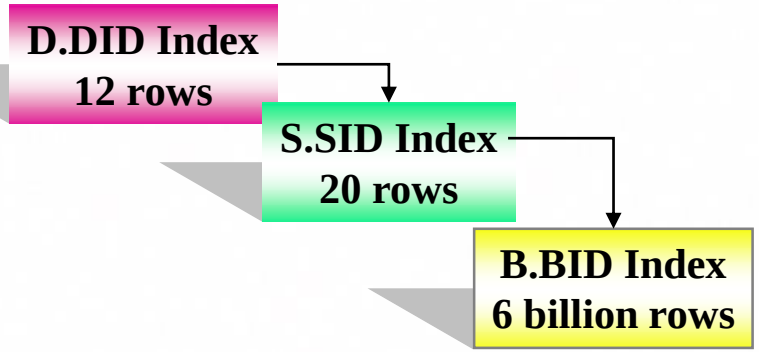
Fake Filtering



- Fake Predicates
  - To encourage index access
  - To alter table join sequence when nothing else works
  - Works by decreasing filter factor on a certain table
  - The filtering is fake and negligible cost
  - Not effective for dynamic queries if the filter contains :host variables

Fake Filtering Example

```
SELECT      B.BID, D.DID, S.SID,
                        ,D.DESC,
                        , S.DESC
            FROM        BONDS B
                , DENOM D, SERIAL S
WHERE   B.BID BETWEEN
                        :hvlo AND :hvhi
    AND     B.DID = D.DID
                AND     B.SID = S.SID
                        ORDER BY B.BID
```



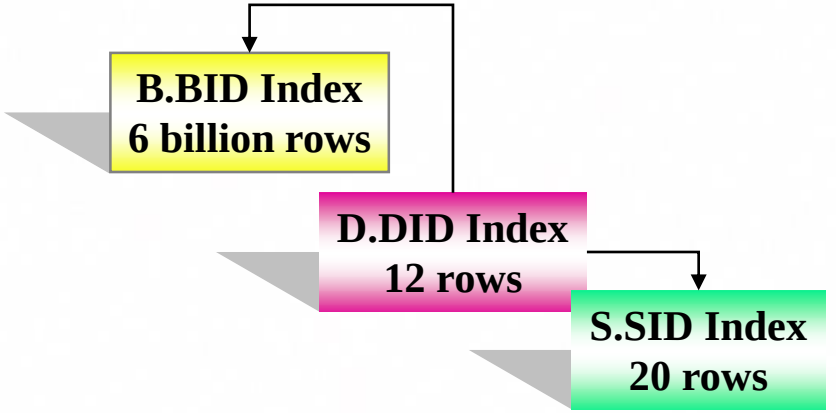
Nested Loop Joins

- Large report query with average of 400,000 row range of BID table
- Need to start nested loop with big table
- Tools required

Fake Filtering Example

```
SELECT    B.BID, D.DID, S.SID,
                      ,D.DESC,
                      , S.DESC
          FROM        BONDS B
          , DENOM D, SERIAL S
WHERE     B.BID BETWEEN
                    :hvlo AND :hvhi
     AND    B.BID = D.DID
          AND    B.SID = S.SID
  AND B.COL2 >= :hv  = B.COL2
  AND B.COL3 >= :hv  = B.COL3
  AND B.COL4 >= :hv  = B.COL4
  AND B.COL5 >= :hv  = B.COL5
  AND B.COL6 >= :hv  = B.COL6
                ORDER BY B.BI
```

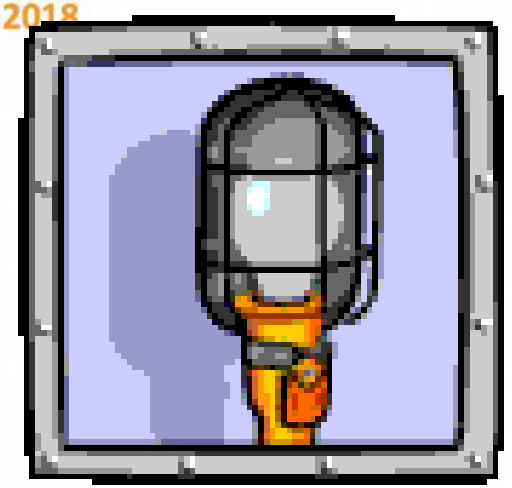
For Dynamic



Nested Loop Joins

- Keep adding filters until table join sequence changes
- Start with index columns
- To preserve index-only access
- No limit!

ON 1 = 1



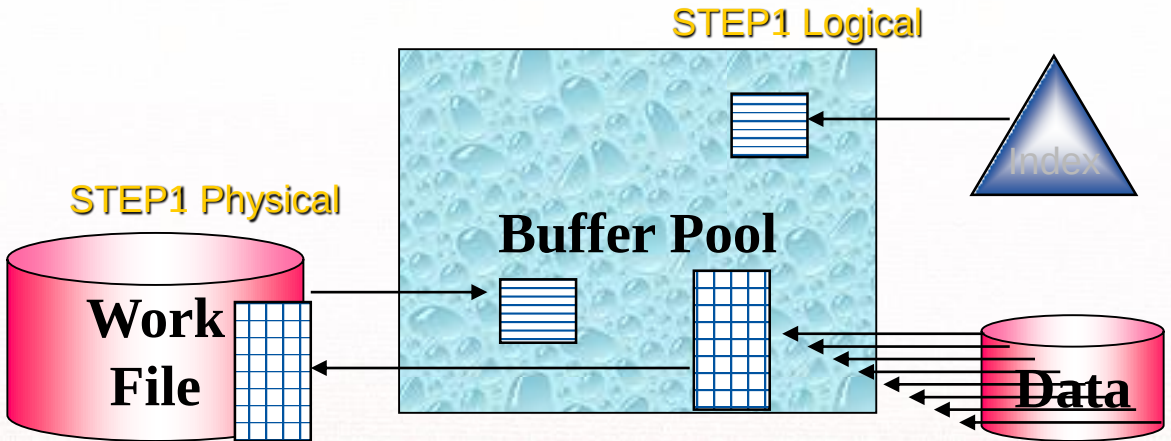
- ON 1=1
  - To fill in a required join field
  - To request a star join
  - When table ratios are under the system specified number (starts at 1:25)
  - Can benefit when large table has high selectivity

Experiment with Extreme Techniques

After Traditional Techniques Fail

DISTINCT Table Expressions

- Table expressions with DISTINCT
  - FROM (SELECT DISTINCT COL1 FROM T1 …..) AS STEP1 JOIN T2 ON … JOIN T3 ON ….
- Used for forcing creation of logical set of data
  - No physical materialization if an index satisfies DISTINCT
- Can encourage sequential detection
- Can encourage a Merge Scan join



DISTINCT Table Expressions Example

- SELECT Columns
  FROM TABX, TABY,
  (SELECT DISTINCT COL1, COL2 …..
  FROM **BIG_TABLE Z**
  WHERE local conditions) AS BIGZ
  WHERE join conditions

- Optimizer is forced to analyze the table expression prior to joining TABX & TABY

Typical Join Problem

```
SELECT COL1, COL2 .....
   FROM ADDR, NAME, TAB3, TAB4, TAB5, TAB6, TAB7 WHERE
   join conditions
         AND TAB6.CODE = :hv
```

Cardinality 1

- Result is only 1,000 rows
- ADDR and NAME first two tables in join
- Index scan on TAB6 table
  - Not good because zero filter

Tuning Technique

```
SELECT COL1, COL2 .....
  FROM ADDR, NAME,
       (SELECT DISTINCT columns
       FROM TAB3, TAB4, TAB5, TAB6, TAB7
       WHERE join conditions
             AND (TAB6.CODE = :hv OR 0 = 1))
       AS TEMP
WHERE join conditions
```

Keeps large tables joined last

Gets rid of Index Scan

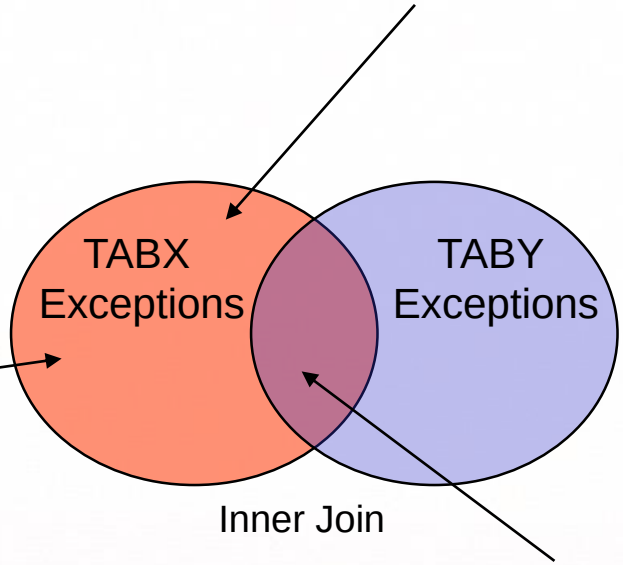
Anti-Join

Faster?

```
SELECT Columns
FROM TABY Y
LEFT JOIN TABX X
ON X.COL1 = Y.COL1
WHERE X.COL1 IS NULL
```

Slower?

```
SELECT Columns
FROM TABX X
WHERE NOT EXISTS
   (SELECT *
   FROM TABY Y
   WHERE X.COL1 = Y.COL1)
```




```
SELECT Columns
FROM TABX X, TABY Y
WHERE X.COL1 = Y.COL1
```

Anti-Join

```
SELECT Columns
FROM TABX X
WHERE NOT EXISTS
   (SELECT *
   FROM TABY Y
   WHERE X.COL1 = Y.COL1)
```

Stage 2 when correlated

Even faster when few inner join rows

Indexable Stage 1

```
SELECT Columns
FROM TABX X
LEFT JOIN TABY Y
ON X.COL1 = Y.COL1
WHERE Y.COL1 IS NULL
```

Does not Benefit LUW

SQL Tuning Confidence Level

0% ---------- 100%

Sheryl Larsen

BMC

Sheryl_larsen@bmc.com

Session code: F16

Please fill out your session evaluation before leaving!