# Revival of the SQL Tuner

## Tuning techniques and more!

Sheryl M. Larsen, BMC

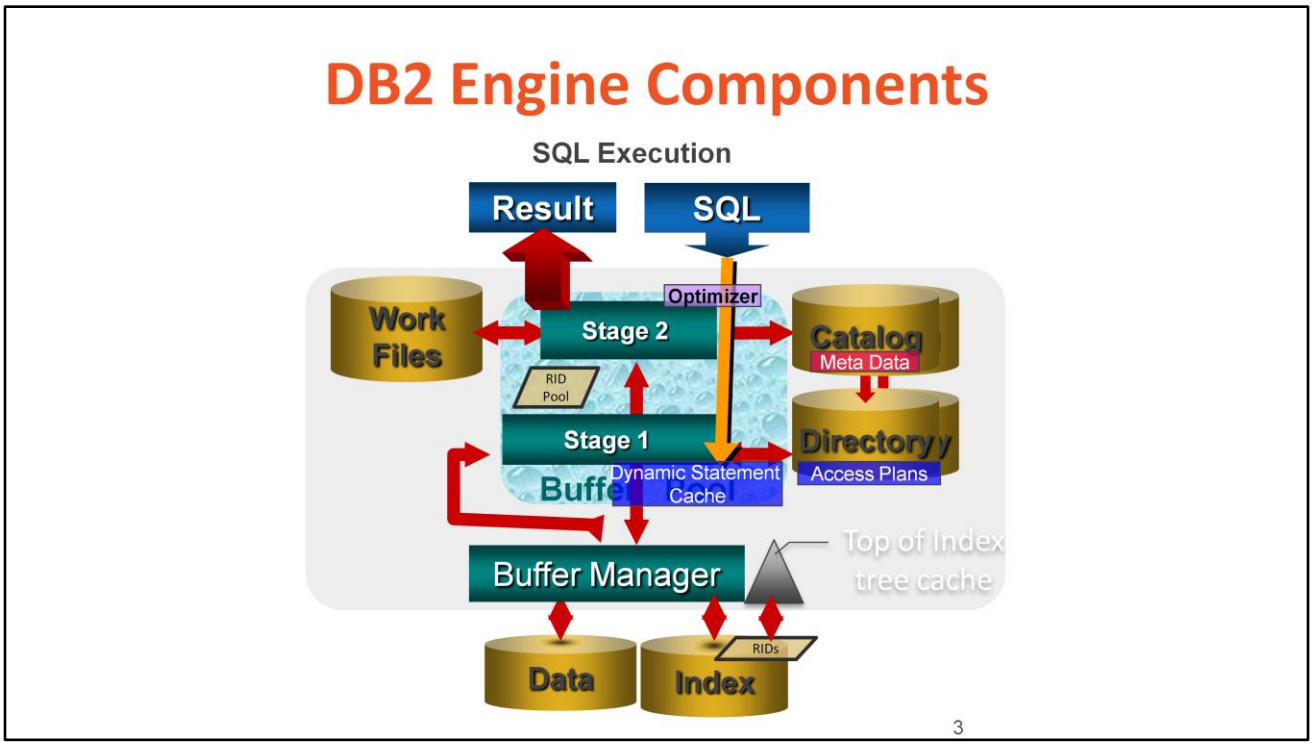
# Table of Contents



http://www.bmcsoftware.uk/forms/MCO_DB211CatalogPosterRefGuide_Collateral_BMCcom_V2.html

2

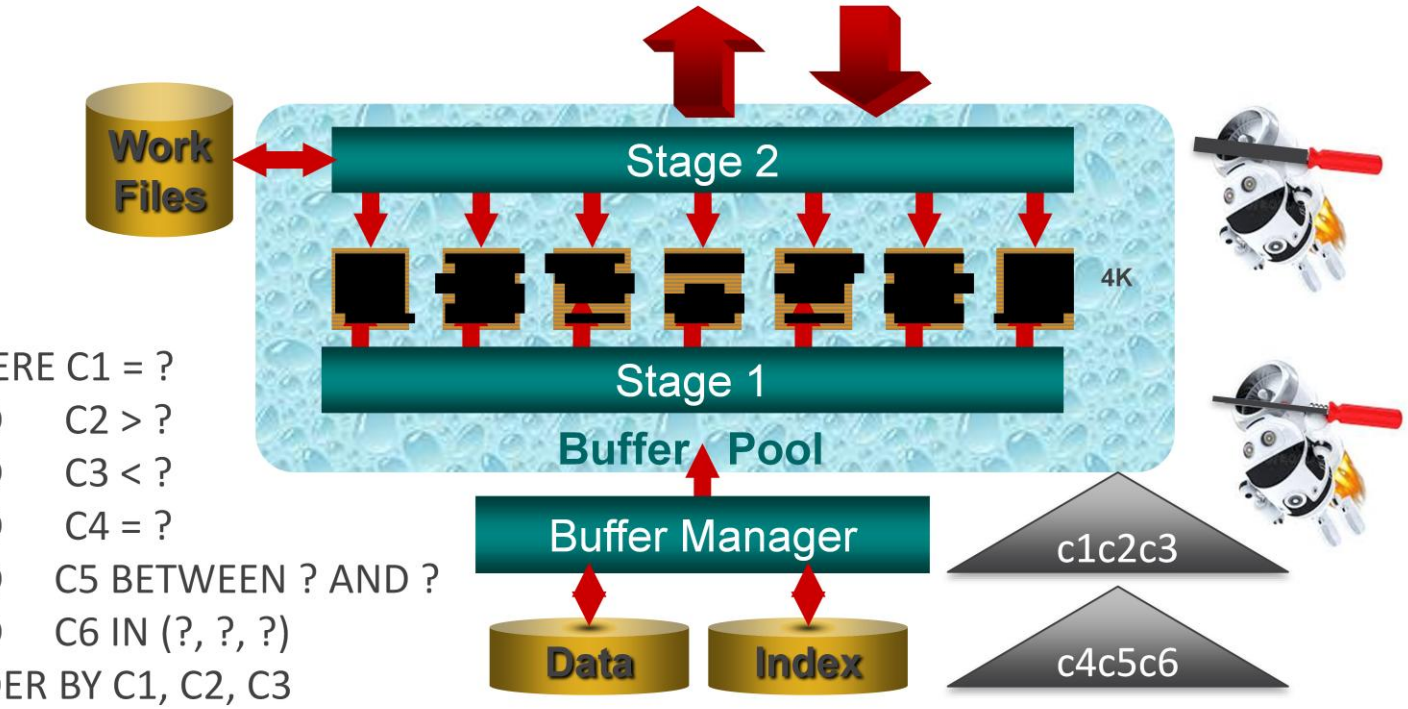
Description:
This class is all about increasing the attendee's ability to identify and fix access path problems
Skills Taught:
•Learn how DB2 executes index, table and join access paths
•Learn when each access path is optimal and non-optimal
•Learn recommended SQL tuning techniques for changing the DB2 optimizer's mind
•Learn how to identify potential access path problems




For static SQL ,DB2 will use the stored access path in the Directory.

REOPT (ONCE), REOPT(AUTO) * DB2 9
For dynamic SQL ,DB2 will check the Dynamic Statement Cache for an exact match of the statement.
If found, the cached associated access path will be used.
REOPT(AUTO) *DB2 9
For dynamic SQL, DB2 will check the Dynamic Statement Cache for an exact match of the statement.
If found, the cached associated access path will be used.
If found but the parameter marker values are not significantly different, the cached associated access path will be used.
If not found, the Optimizer costs out a new access path for use and stores it in the cache with the new statement
REOPT(Always)
For each execution, the Optimizer costs out a new access path for use ,

# Page Processing – z/OS




4

Stage 1 filtering is done first against the 4K pages brought into the Buffer Pool.

Stage 2 only examines the rows that qualify after Stage 1 filtering, however, the entire row or index entry is still on the 4k page sitting in the Buffer Pool.

Once Stage 2 is complete, data transformations requested on the SELECT clause are performed prior to returning one result *value* at a time to the calling program.

Query response time is dependent on:
- The number of I/O's to pull data and/or index pages in the Buffer Pool
- The number of rows left after Stage 1 filtering
- The number of rows left after Stage 2 filtering
- The sequence the rows are in the Buffer Pool
- The amount of translations preformed on the result values

The less rows requested, the less columns requested, the less transformations, the faster the query goes.

Summary Of Predicate Processing

## Indexable Stage 1 Predicates

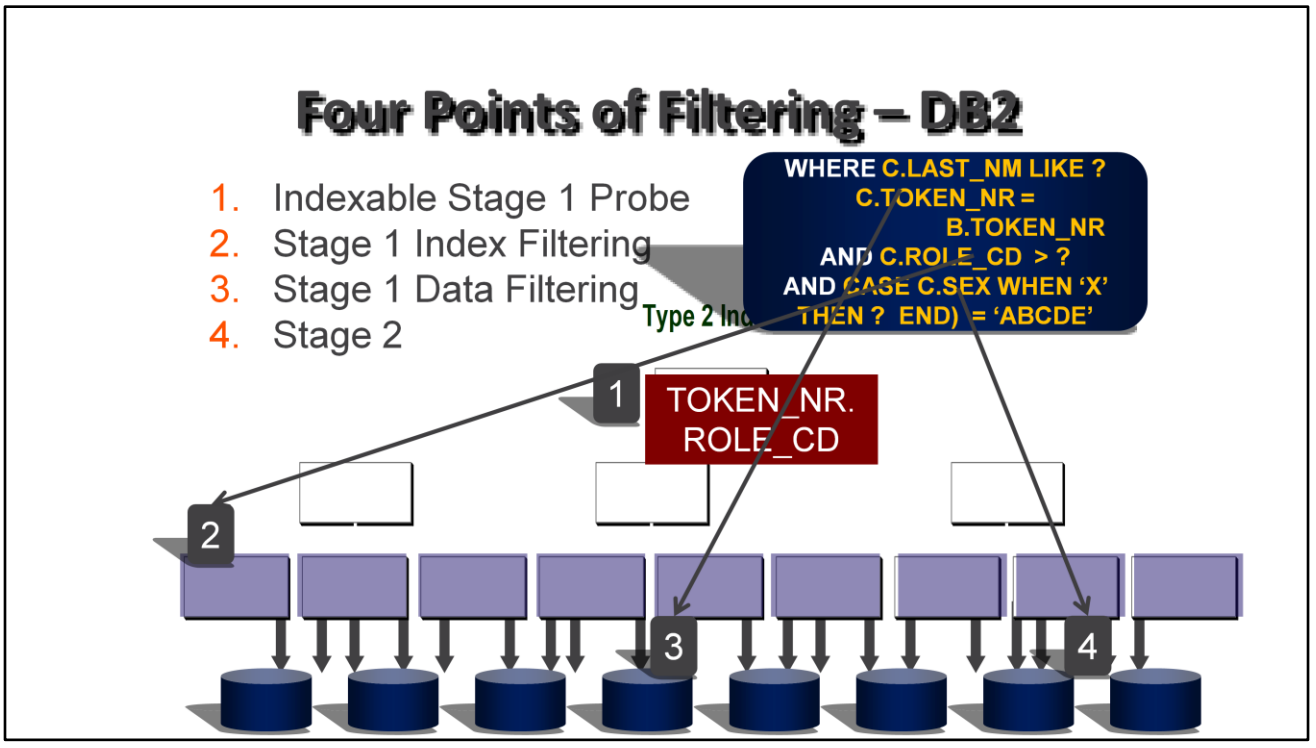
| Predicate Type | Indexable | Stage 1 |
|---|---|---|
| COL = *value* | Y | Y |
| COL = *noncol expr* | Y | Y |
| COL IS NULL | Y | Y |
| COL *op value* | Y | Y |
| COL *op noncol expr* | Y | Y |
| COL BETWEEN *value1* AND *value2* | Y | Y |
| COL BETWEEN *noncol expr1* AND *noncol expr2* | Y | Y |
| 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 |

## 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 the probe of the index, the condition is named Matching (matching the index). This is the first point that filtering is possible in DB2.
2. Index Screening = The Stage 1 predicate is a candidate for filtering on the index leaf pages. This is the second point of filtering in DB2. If partitioned filters limiting partitions are also applied
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.

https://www.ibm.com/support/knowledgecenter/en/SSEPEK_12.0.0/perf/src/tpc/db2z_summarypredicateprocessing.html




1. Indexable Stage 1 Probe - Only 28 for DB2 9, can be applied at this point. The ones that will be applied are dependent on which index was chosen, the conditions on the columns belonging in the index, and the sequence of those columns. If the first column of the index is used in a "=" predicate, the column is used to navigate the tree along with the next column (2 matching). If the next column is used in a "=" predicate, the column is used to navigate the tree along with the previous two columns (3 matching). If the next column is not an "=" predicate, the matching stops with this condition (4 matching) unless it is nonindexable or Stage 2 (3 matching). If the first column is not and "=" predicate, only the first column is used to navigate the tree (1 matching). The number of matching columns usually = one more than the last matching condition. Data types are required to match until V8.
2. Stage 1 Index Filtering - If there is no predicate involving the first column of the index, tree navigation is not allowed (0 matching). Any Stage 1 predicate (all 40) can be applied on the leaf page. This point of filtering is called index screening. Stage 2 conditions can also be applied after the Stage 1 conditions are applied (if this is index only access *and* the Stage 2 column is included in the index - like COL9 above). Data types are required to match until V8.
3. Stage 1 Data Filtering - Any Stage 1 condition that has not been applied in the index entries is applied when the data page is accessed (because all columns live there). Data types are required to match until V8.
4. Stage 2 Data Filtering - Any condition that is not Stage 1 will be applied at this point (an infinite number of possible predicates). This filtering is still better than program filtering which occurs after each element on the result row is transferred to the calling program (one at a time). Any data type mismatches were filtered here until V8.

# QL 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



1. Examine Program logic – check for program filtering and joining. Move work into the query.
2. Examine FROM clause – order of tables insignificant unless > 9 table joins. List preferred join sequence for this and OUTER JOINs
3. Verify Join conditions – make sure every table is hooked up correctly to avoid cartesian joins
4. Promote Stage 2's/Residuals and Stage 1's if possible – promotions can change access paths
5. Verify data type matches – mismatched numeric and date/time will cause delays in filtering and alter the access path
6. Prune SELECT lists – remove columns with values determined to be static by WHERE clause filtering. Remove columns used in the ORDER BY or GROUP BY sequencing but not needed for the display.
7. Verify local filtering sequence – If host variables are used, add parenthesis to override the predetermined filtering sequence when necessary. This reduces the CPU required to disqualify rows
8. Analyze Access Paths – Only check the access path of the FINAL query, after query rewrite, bound with production statistics in a subsystem that resembles the production thresholds as closely as possible.
9. Tune if necessary – A topic for today!

# When to Tune Queries

8

- 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

9

- **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

10

```
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
```

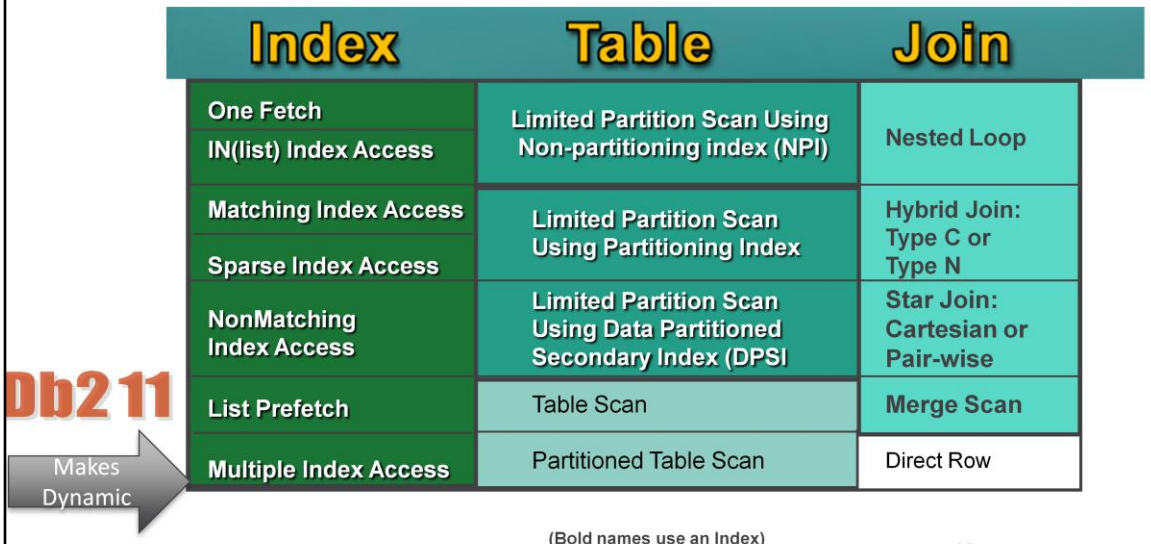
Fake Filter

# 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

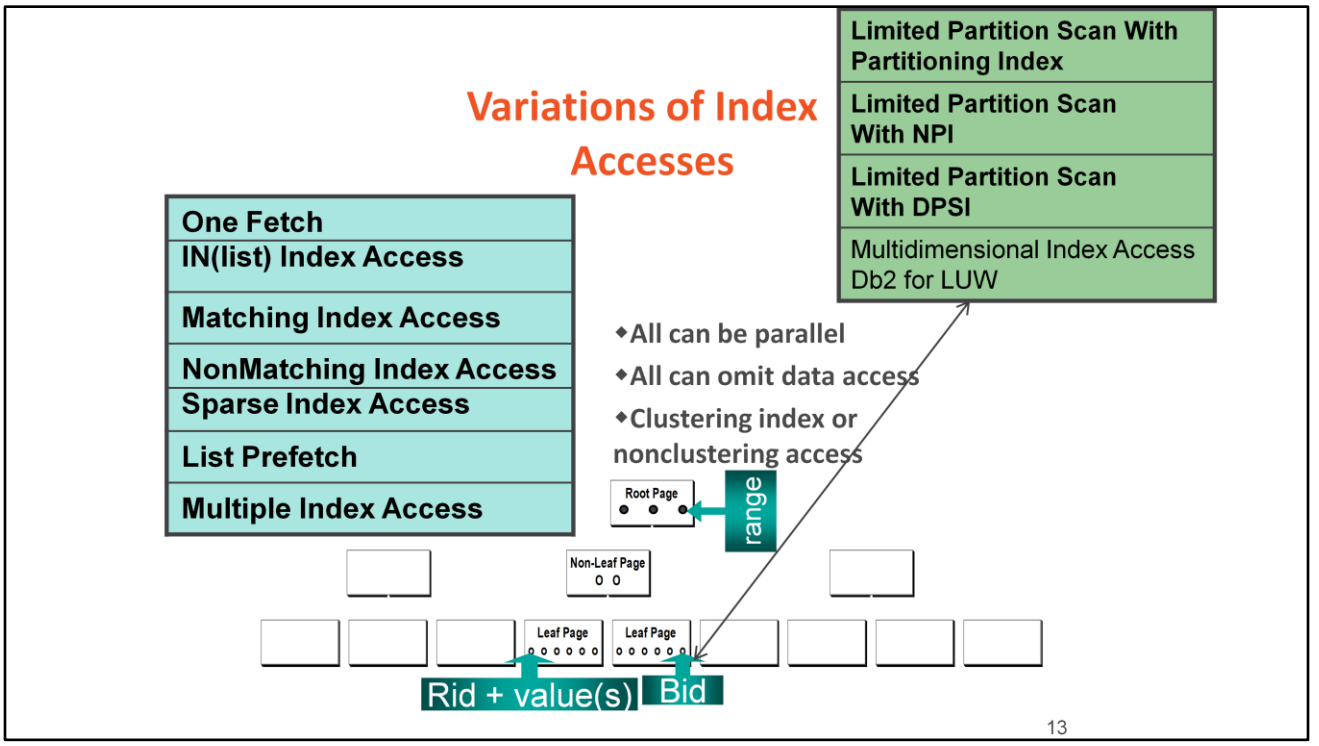
11

# All the Possible Access Paths



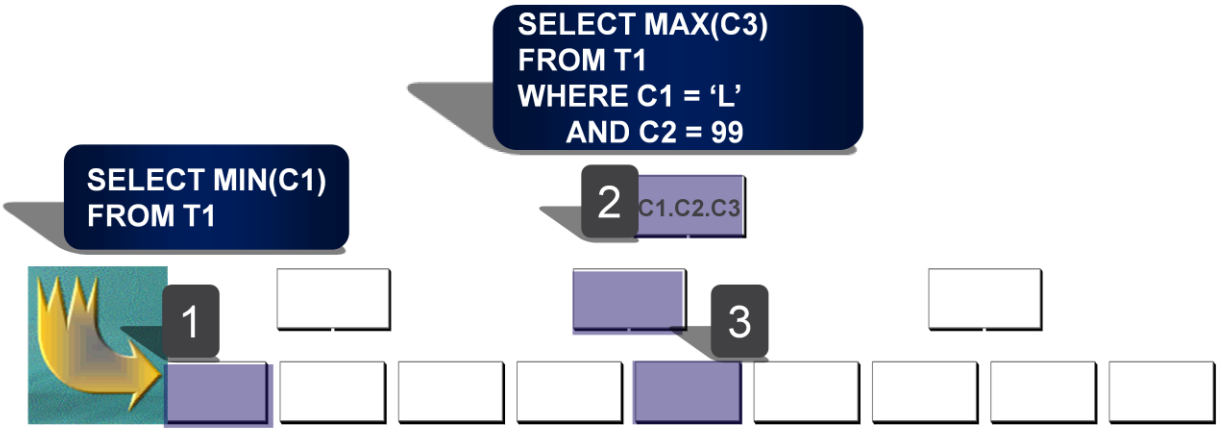

(Bold names use an Index)

12




RID = Row ID, a single pointer/address to a single row
BID = Block ID, a single pointer/address to a block of rows

# One Fetch

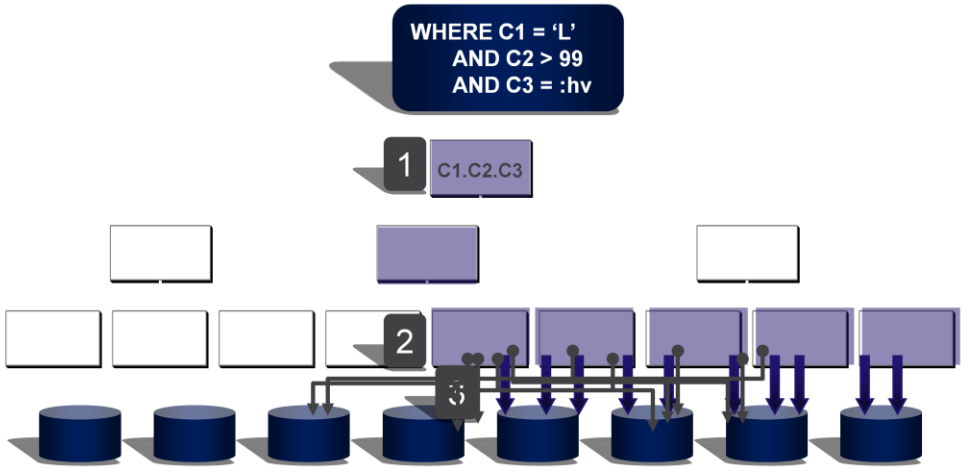





1. For MIN or MAX on the first column of the index, retrieve the first or last leaf page of the index only
2. For MIN or MAX on columns past the first column of the index, and equal predicates on previous index columns, start at the root page and probe through the nonleaf page to the leaf page, applying all matching predicates
3. Proceed forward or backward on the leaf pages to satisfy the MIN or MAX

Note: All indexes ALLOW REVERSE SCANS for One Fetch, ORDER BY, GROUP BY, and DISTINCT.

# Matching Index Access

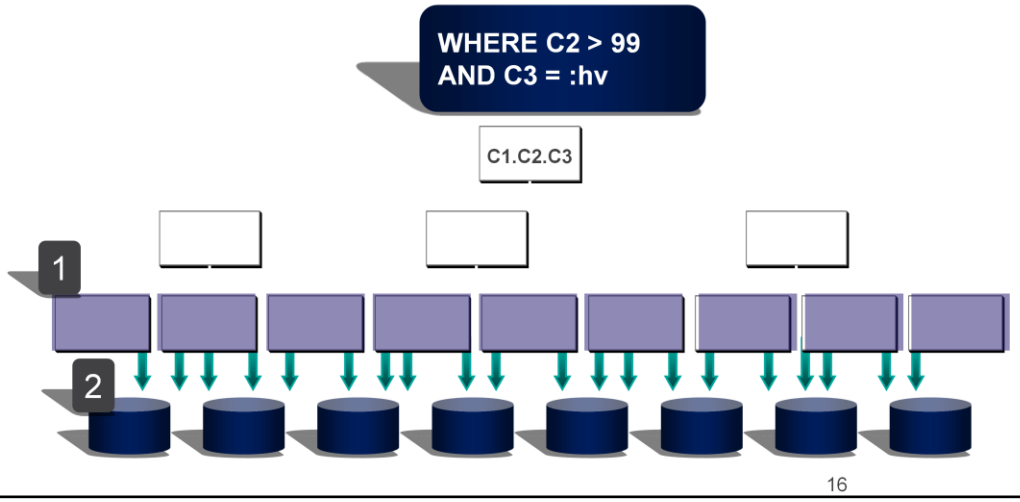



15

1. Start at the root page of the index and probe through the nonleaf page to the leaf page, applying all matching predicates
2. Perform index screening, applying all nonmatching predicates to leaf pages
3. Follow qualifying row-ids to retrieve qualifying data pages, apply remaining Stage 1 predicates and then remaining Stage 2 predicates

Note: A predicate becomes matching when a column is located in the first position of the index and is referenced by an indexable predicate. If the column is not in the first position of the index, the preceding columns are included in the matching when they have consecutive = predicates. The total number of matching columns includes all consecutive = predicate columns, in the order of the index columns, plus one past the last = predicate. The higher the percent matching, i.e. 4 out of 5 columns are 80% matching, the closer the probe will be to the first qualifying row.

# NonMatching Index Access

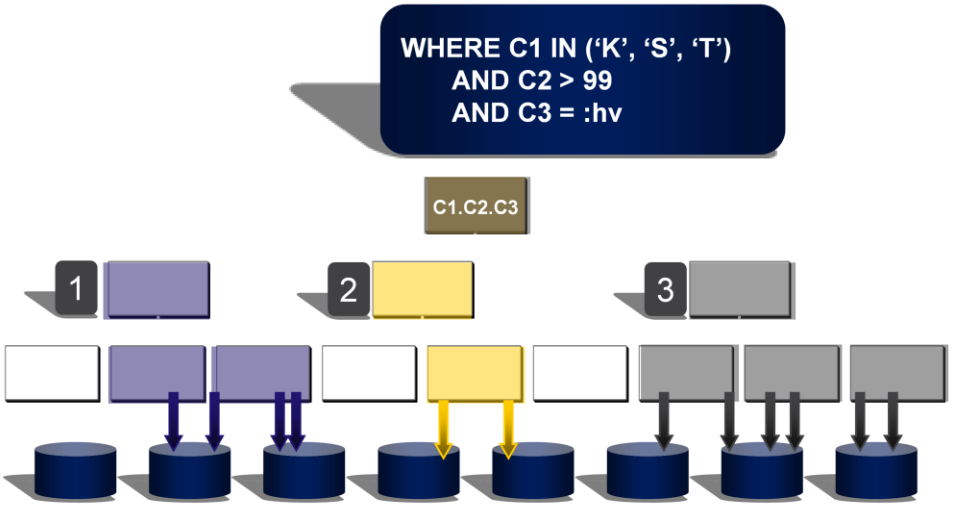



16

1. Start at the first or last leaf page of the index and perform index screening going forward or backward using sequential prefetch, applying all nonmatching predicates to leaf pages
2. Follow qualifying row-ids to retrieve qualifying data pages, apply remaining Stage 1 predicates and then remaining Stage 2 predicates

Note: A nonmatching index scan is chosen when the column located in the first position of the index is not referenced by an indexable predicate but remaining index column(s) are.

# IN(list) Index Access

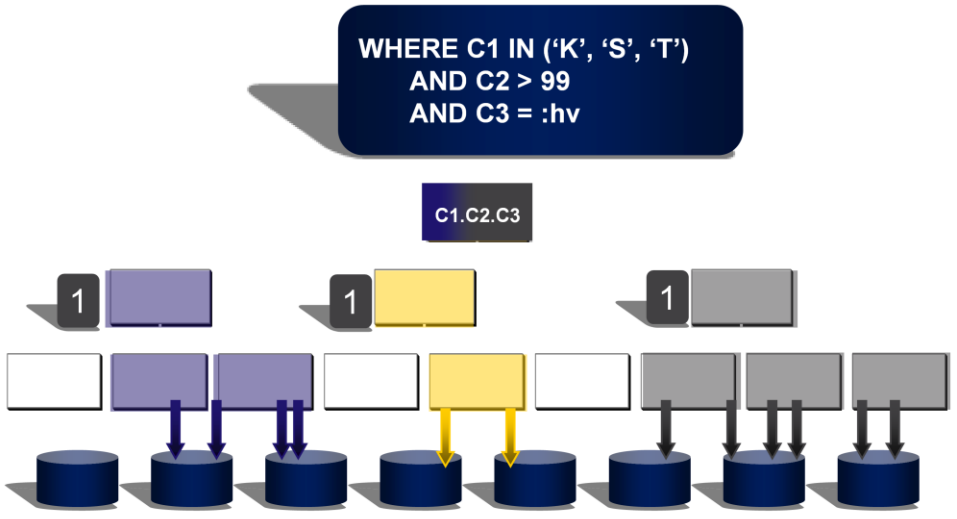



17

1. Start at the root page of the index and probe through the nonleaf page to the leaf page, applying one IN(list) value filter plus all matching predicates, perform index screening, applying all nonmatching predicates to leaf pages, Follow qualifying row-ids to retrieve qualifying data pages, apply remaining Stage 1 predicates and then remaining Stage 2 predicates
2. Repeat Step 1 for the next value in the IN(list)
3. Repeat Step 1 for the next value in the IN(list)

Note: This is essentially multiple Matching Index Accesses done sequentially. This access path is beneficial when qualifying values are spread out. The more spread out the qualifying entries are, the higher the benefit. This access path can be used on the inner or outer table of most join methods.

# IN(list) Index Access -Parallel

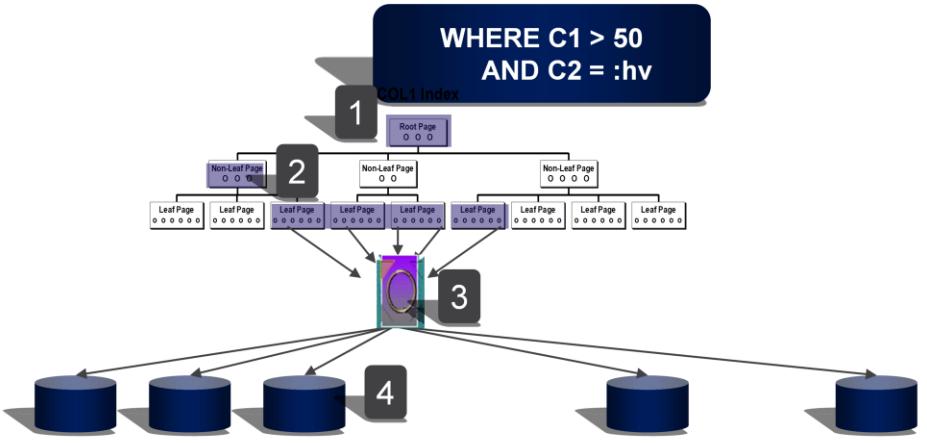



18

1. Start at the root page of the index and probe through the nonleaf page to the leaf page, applying one IN(list) value filter per concurrent probe plus all matching predicates, perform index screening, applying all nonmatching predicates to leaf pages, retrieve qualifying data pages, apply remaining Stage 1 predicates and then remaining Stage 2 predicates

Note: This is essentially multiple Matching Index Accesses done concurrently. This access path is beneficial when qualifying values are spread out. The more spread out the qualifying entries are, the higher the benefit. This access path can be used on the inner or outer table of most join methods.

# List Prefetch

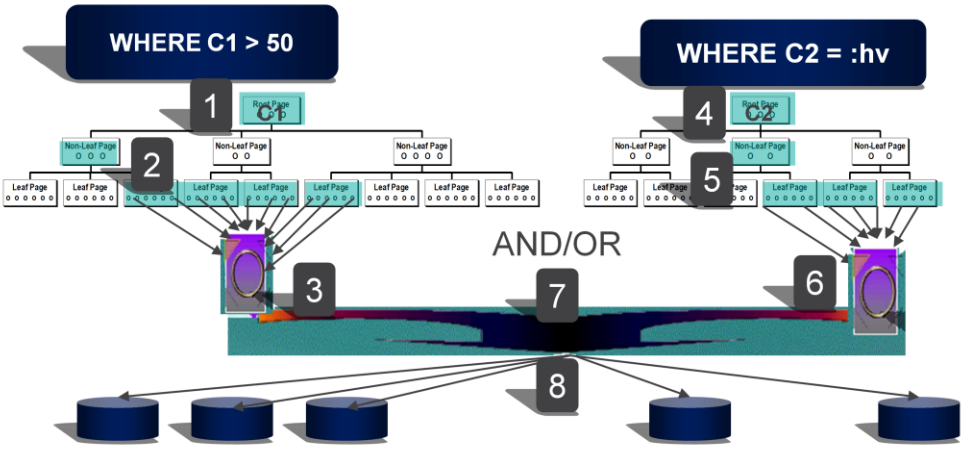



19

1. Start at the root page of the index and probe through the nonleaf page to the leaf page, applying all matching predicates
2. Perform index screening, applying all nonmatching predicates to leaf pages
3. Place qualifying row-ids in the RID Pool and sort by page number to remove duplicate pages
4. Use skip sequential prefetch (each I/O retrieves 32 noncontiguous qualifying data pages) to retrieve data pages identified in Step 3, apply remaining Stage 1 predicates and then remaining Stage 2 predicates

Note: This access path is very beneficial when all the result rows are required and the index is poorly clustered, due to the elimination of random I/O to retrieve data pages. If a sort was performed in Step 3, an additional sort may be required to satisfy an optional ORDER BY, GROUP BY or DISTINCT.

# Multiple Index Access



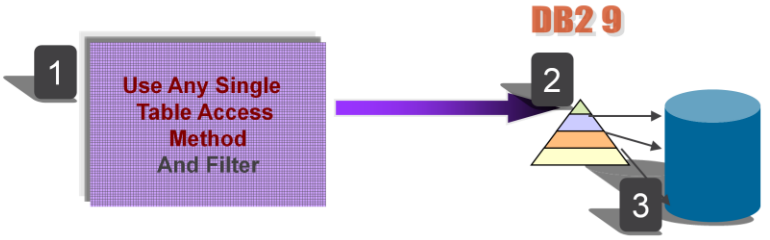

20

1. Start at the root page of one index and probe through the non-leaf page to the leaf page, applying all matching predicates for that index
2. Perform index screening, applying all nonmatching predicates to leaf pages
3. Place qualifying row-ids in the RID Pool and sort by page number to remove duplicate pages
4. Start at the root page of another index and probe through the nonleaf page to the leaf page, applying all matching predicates for that index
5. Perform index screening, applying all nonmatching predicates to leaf pages
6. Place qualifying row-ids in the RID Pool and sort by page number to remove duplicate pages
7. For ORed predicates, combine the page numbers and remove duplicates (referred to Index ORing). For ANDed predicates, intersect the page numbers and remove duplicates (referred to Index ANDing).
8. Use skip sequential prefetch (each I/O retrieves 32 noncontiguous qualifying data pages) to retrieve data pages identified in Step 7, apply remaining Stage 1 predicates and then remaining Stage 2 predicates

Note: This access path is very beneficial when all the result rows are required and the index is poorly clustered, due to the elimination of random I/O to retrieve data pages. An additional sort may be required to satisfy an optional ORDER BY, GROUP BY or DISTINCT.

# Sparse Index Access

21



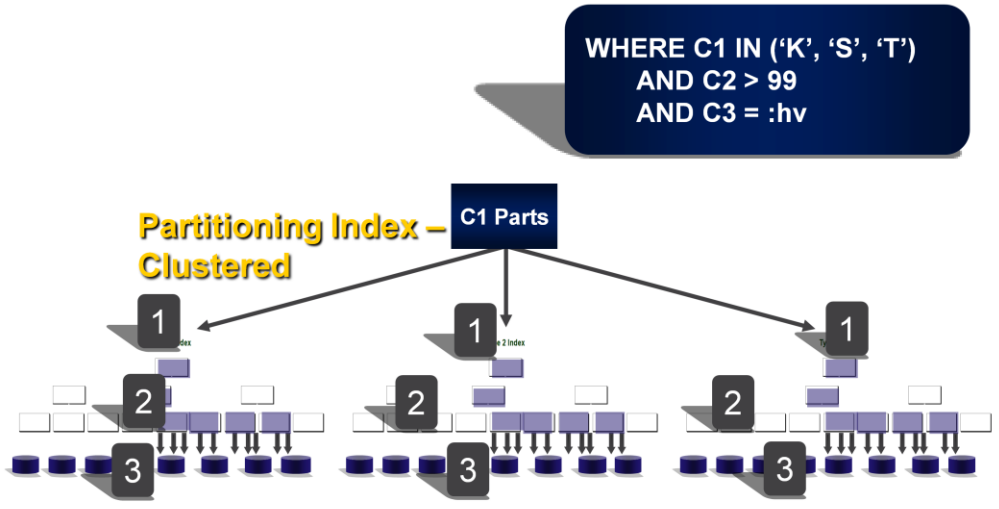



1. Start at the first page of the table and scan using sequential prefetch or any viable single table access method, applying Stage 1 predicates and then remaining Stage 2 predicates
2. Create a Sparse Index, contains pointers to values in the filtered table work file
3. Follow Sparse Index pointers to work file to retrieve rows

Note: This access path can be used for inner tables in Nested Loop Join, materialized table expressions, views, global temporary tables and small tables participating in Star Join - Cartesian.

# Limited Partition Scan
# Using Clustered Partitioning Index

22






0. At optimization time, determine the target partitions using matching predicates without host variables or parameter markers. If REOPT options are used, target partitions will be determined at run time when host variables or parameter marker values are known

1. Start at the root page of each target partition and probe through the nonleaf page to the leaf page, applying all matching predicates
2. Perform index screening on each target, applying all nonmatching predicates to leaf pages
3. Follow qualifying row-ids to retrieve qualifying data pages, apply remaining Stage 1 predicates and then remaining Stage 2 predicates

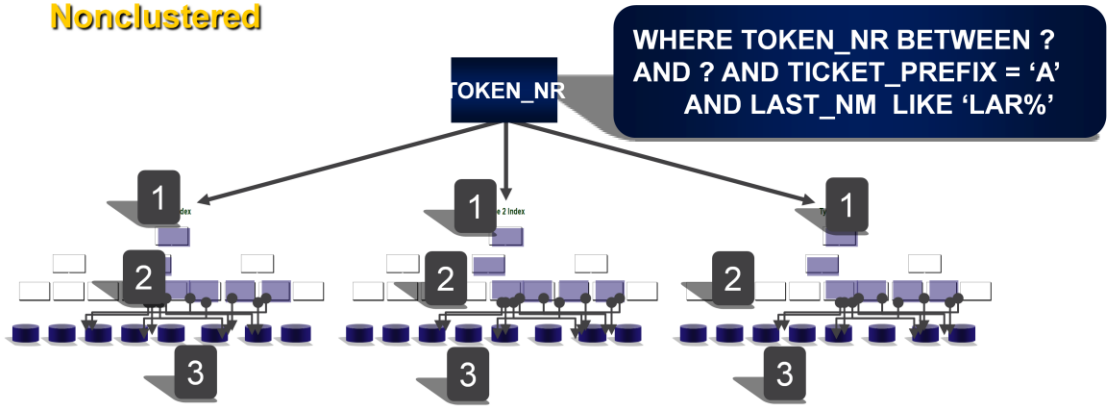
Note: There will not be any random I/O to the data pages within each target partition

# Scan for Last Name

## Using Nonclustered Partitioning Index

23

### Partitioning Index – Nonclustered

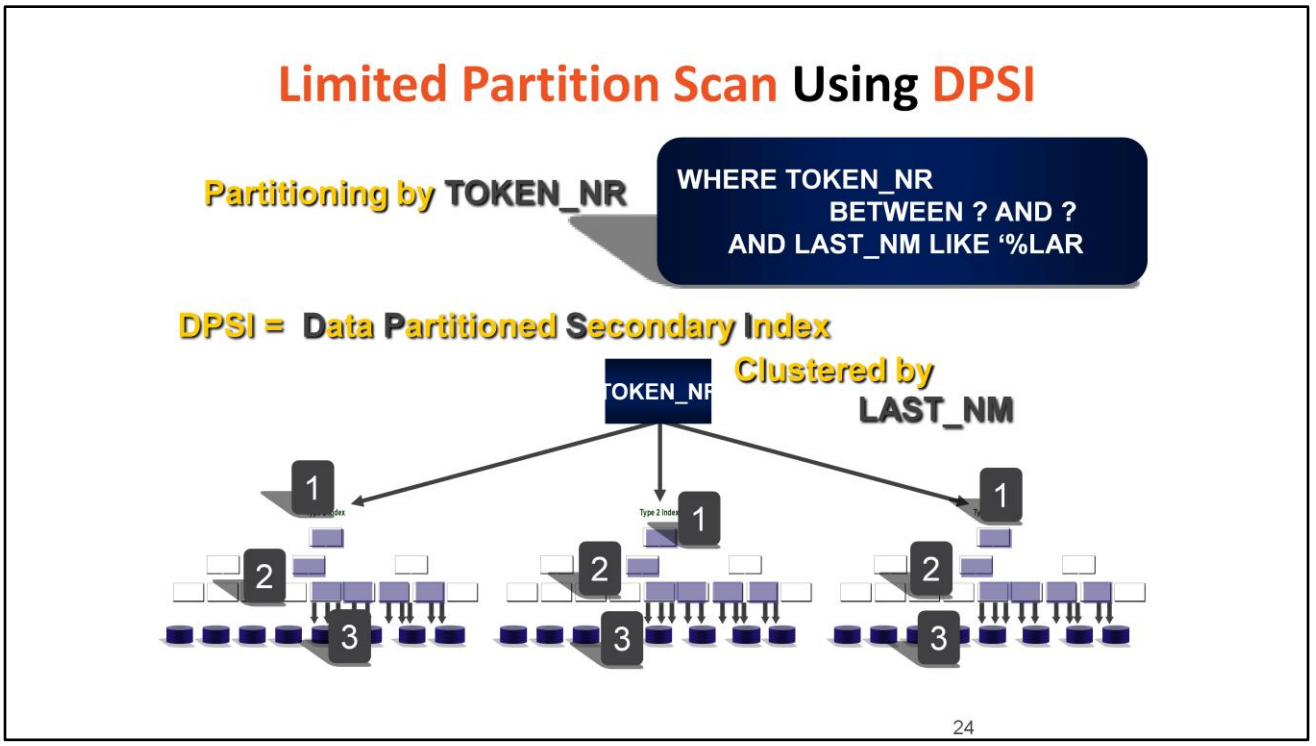





0. At optimization time, determine the target partitions using matching predicates without host variables or parameter markers. If REOPT options are used, target partitions will be determined at run time when host variables or parameter marker values are known

1. Start at the root page of each target partition and probe through the nonleaf page to the leaf page, applying all matching predicates
2. Perform index screening on each target, applying all nonmatching predicates to leaf pages
3. Follow qualifying row-ids to retrieve qualifying data pages, apply remaining Stage 1 predicates and then remaining Stage 2 predicates

Note: There will be random I/O to the data pages within each target partition.




0. At optimization time, determine the target partitions using predicates matching the partitioning index without host variables or parameter markers. If REOPT options are used, target partitions will be determined at run time when host variables or parameter marker values are known

1. Start at the root page of each target partition of the DPSI index and probe through the nonleaf page to the leaf page, applying all matching predicates
2. Perform index screening on each target, applying all nonmatching predicates to leaf pages
3. Follow qualifying row-ids to retrieve qualifying data pages, apply remaining Stage 1 predicates and then remaining Stage 2 predicates

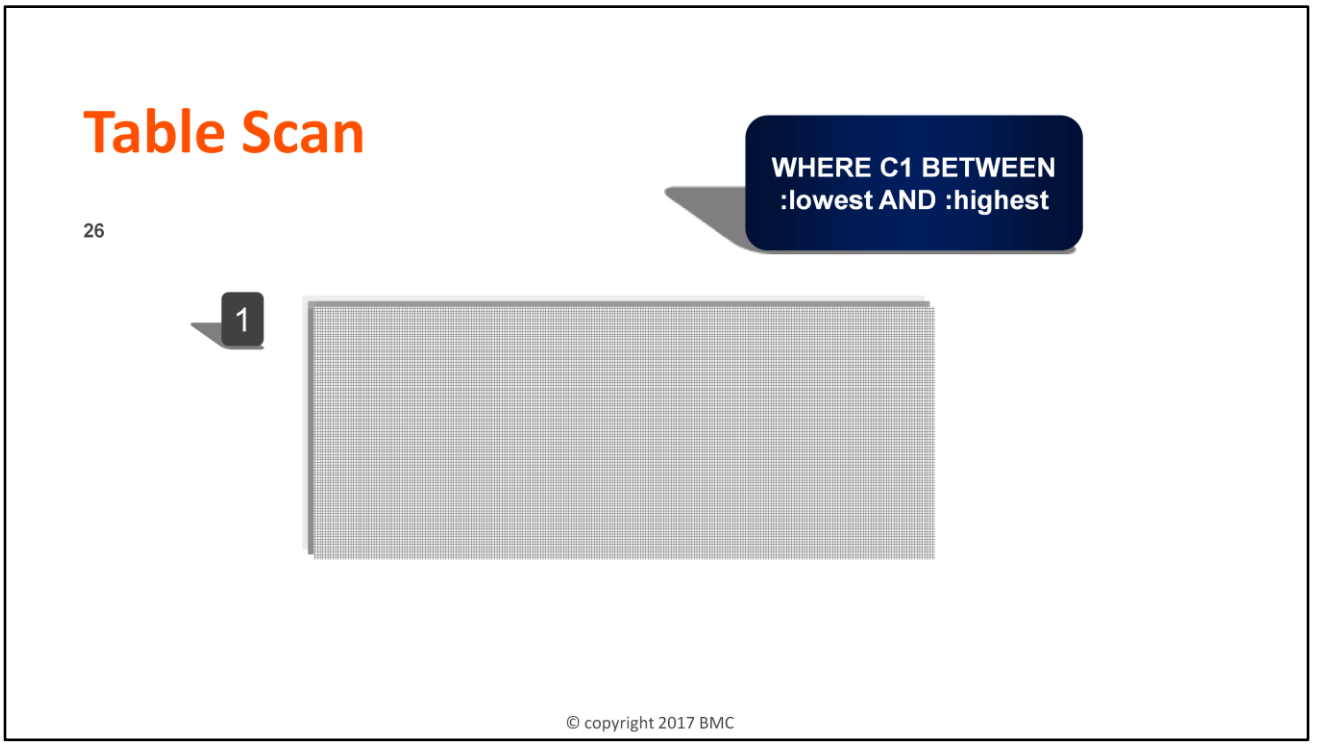
Note: There will not be any random I/O to the data pages within each target partition

# Variations of Table Access

25

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

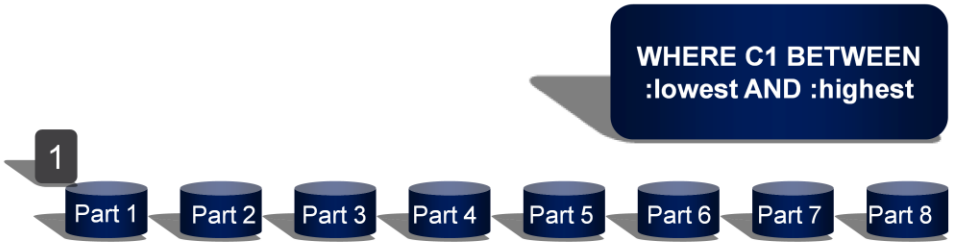
| Segmented |
| --- |
| Partitioned |
| Limited Partitioned |
| In Memory Data Cache |




1. Start at the first data page of the table and perform data screening going forward using sequential prefetch, applying all Stage 1 predicates and then remaining Stage 2 predicates

# Partitioned Table Scan

27

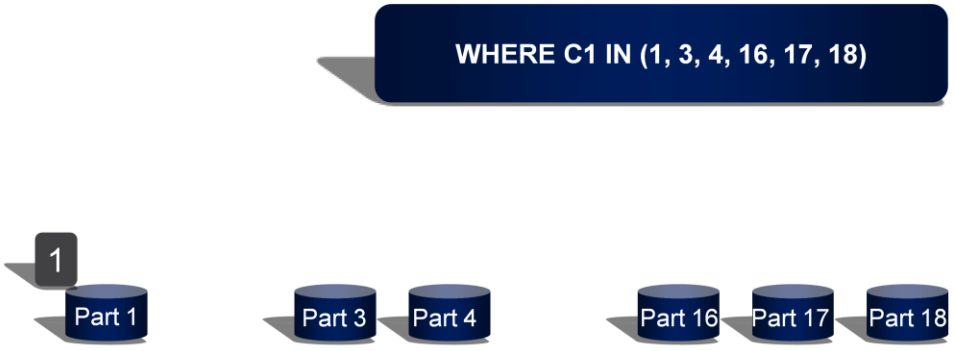





0. At optimization time, determine the target partitions using predicates matching the partitioning index without host variables or parameter markers. If REOPT options are used, target partitions will be determined at run time when host variables or parameter marker values are known

1. Start at the first data page of each target partition and perform data screening going forward using sequential prefetch, applying all Stage 1 predicates and then remaining Stage 2 predicates

# Limited Partitioned Table Scan





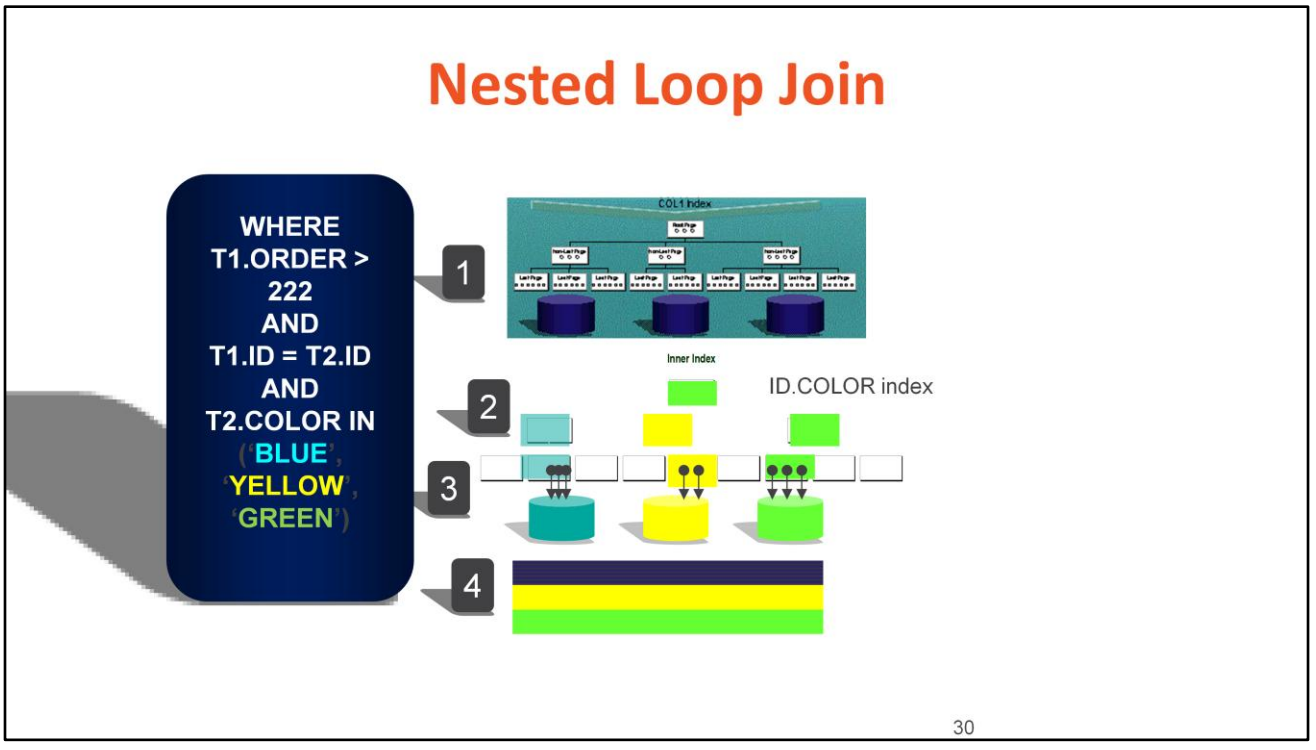

1. Start at the first data page of each partition and perform data screening going forward using sequential prefetch, applying all Stage 1 predicates and then remaining Stage 2 predicates

# Variations of Join Methods

29

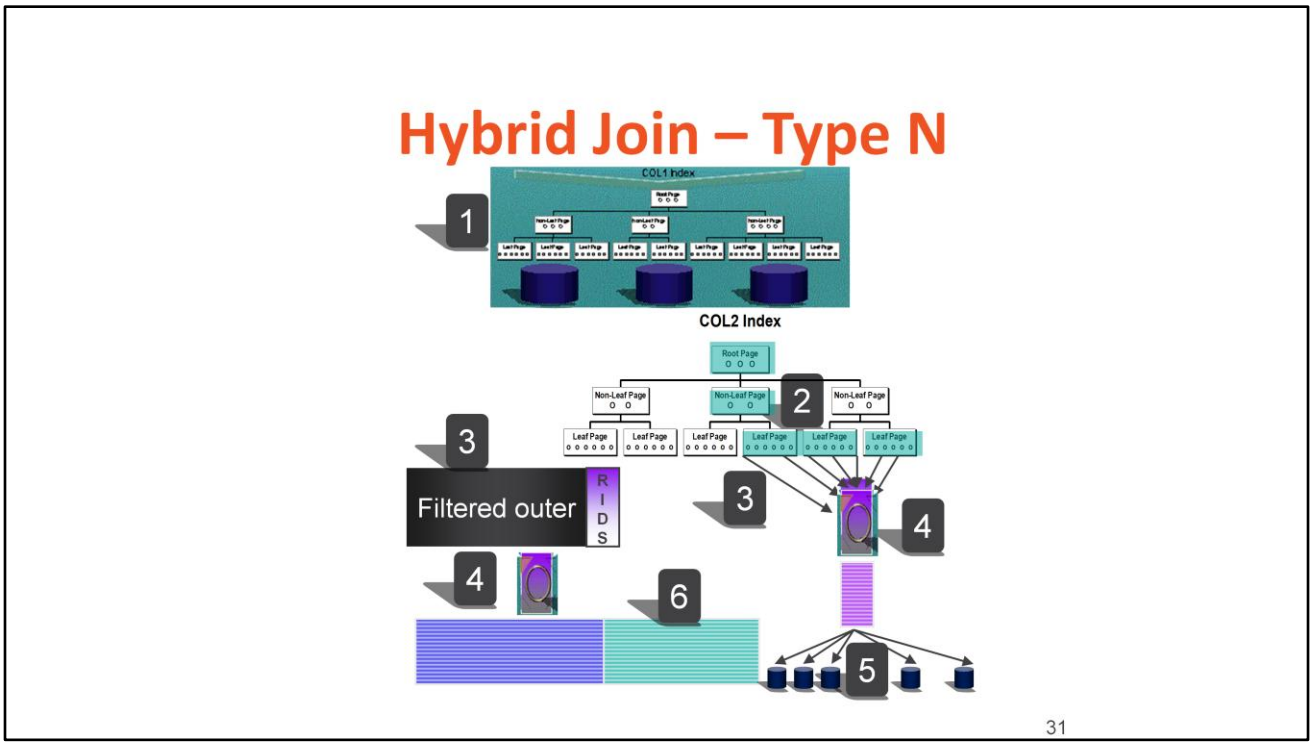
- All choose outer table and filter first
- All can be parallel (Star CPU only )
- Worry about join table sequence instead of join method

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




1. Access outer table using the most efficient single table access path for applying all outer table filters, as soon as the first outer table qualifying row is determined, add the join column values to the join predicates and merge with inner table predicates
2. Apply matching index filters to root page of inner table index and probe through nonleaf to leaf pages and perform index screening
3. Perform index screening, applying all nonmatching predicates to leaf pages, follow qualifying row-ids to retrieve qualifying data pages, apply remaining Stage 1 predicates and then remaining Stage 2 predicates
4. Place filtered outer row with joining filter inner row in the result, if LEFT JOIN, keep all filtered outer rows and NULL missing filter inner row values

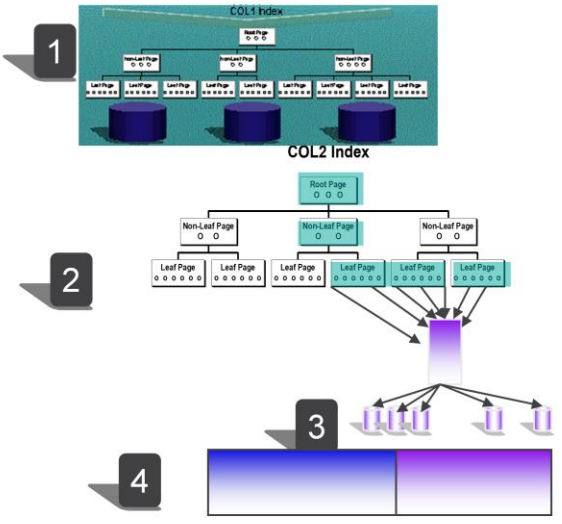
   Note: Step 1 does not have to complete prior to starting the remaining steps. This access path is optimal when only the first part of the result set is needed.




1. Access outer table using the most efficient single table access path for applying all outer table filters, a sort of these rows may be required to match the inner table index sequence
2. Apply matching index filters to root page of inner table index and probe through nonleaf to leaf pages and perform index screening
3. Place qualifying row-ids in the RID Pool and attach to filtered outer row to form an intermediate table
4. Sort row-ids and intermediate table by page number
5. Use skip sequential prefetch (each I/O retrieves 32 noncontiguous qualifying data pages) to retrieve inner table data pages, apply remaining Stage 1 predicates and then remaining Stage 2 predicates
6. Replace inner table row-ids in intermediate table with qualifying inner table rows to form result rows

Note: This access path is very beneficial when all the result rows are required and the index is poorly clustered, due to the elimination of random I/O to retrieve data pages. An additional sort may be required to satisfy an optional ORDER BY, GROUP BY or DISTINCT.

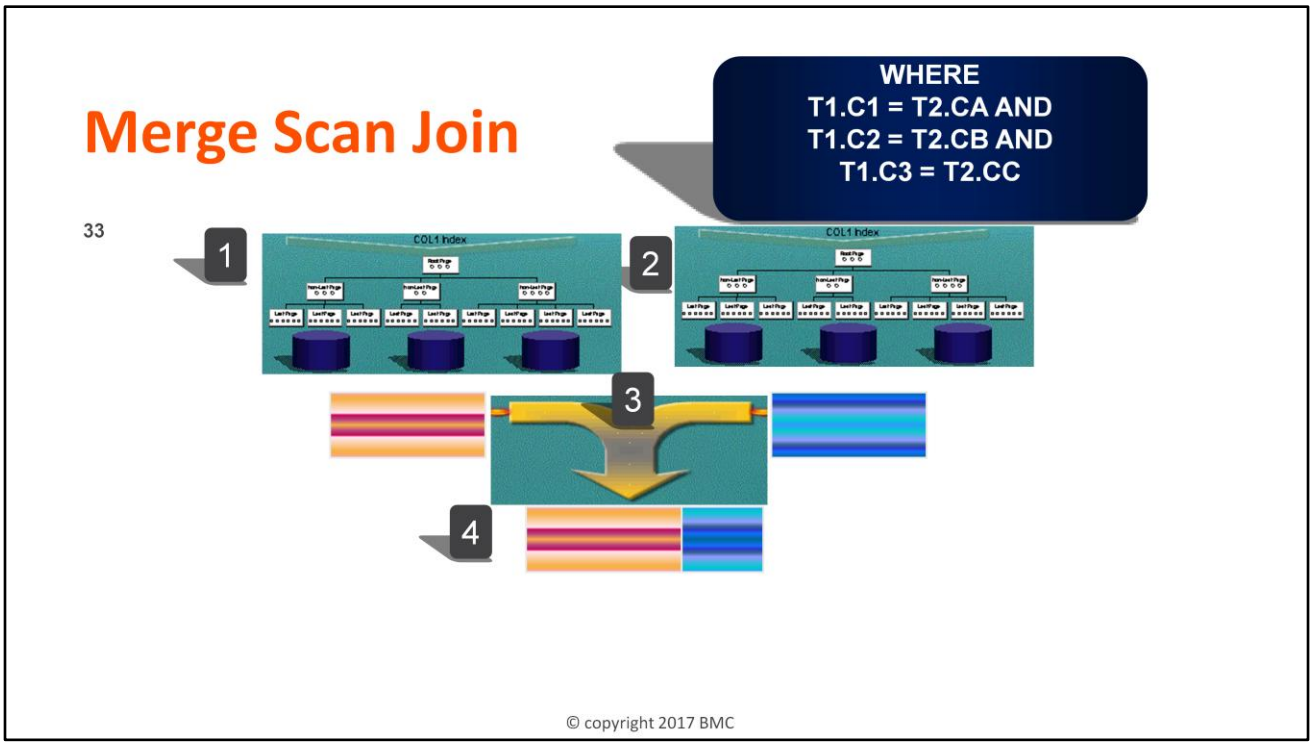
# Hybrid Join – Type C




32

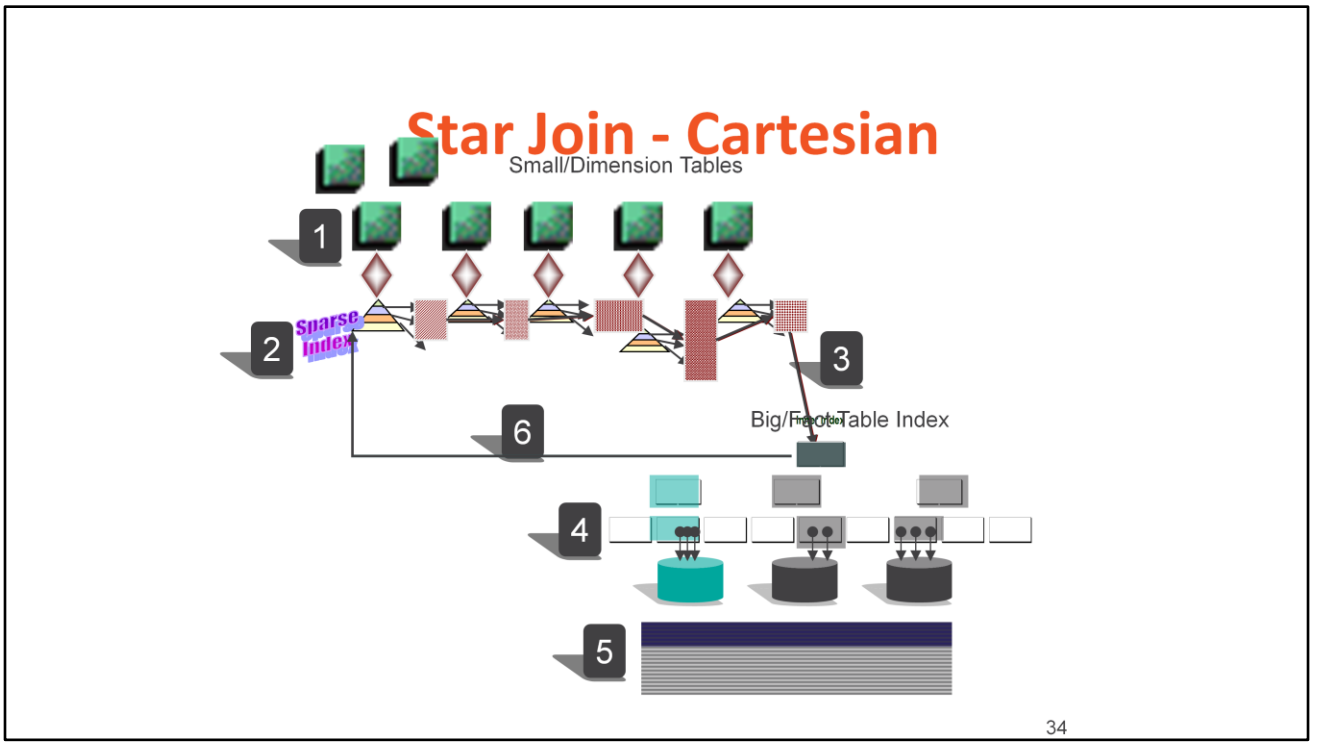
1. Access outer table using the most efficient single table access path for applying all outer table filters, a sort of these rows may be required to match the inner table index sequence
2. Apply matching index filters to root page of inner table index and probe through nonleaf to leaf pages and perform index screening
3. Gather first set of page numbers and use skip sequential prefetch (each I/O retrieves 32 noncontiguous qualifying data pages) to retrieve inner table data pages, apply remaining Stage 1 predicates and then remaining Stage 2 predicates
4. Place filtered outer row with joining filter inner row in the result

   Note: This access path is optimal when only the first part of the result set is needed and the sort for the filtered outer table is not extensive.



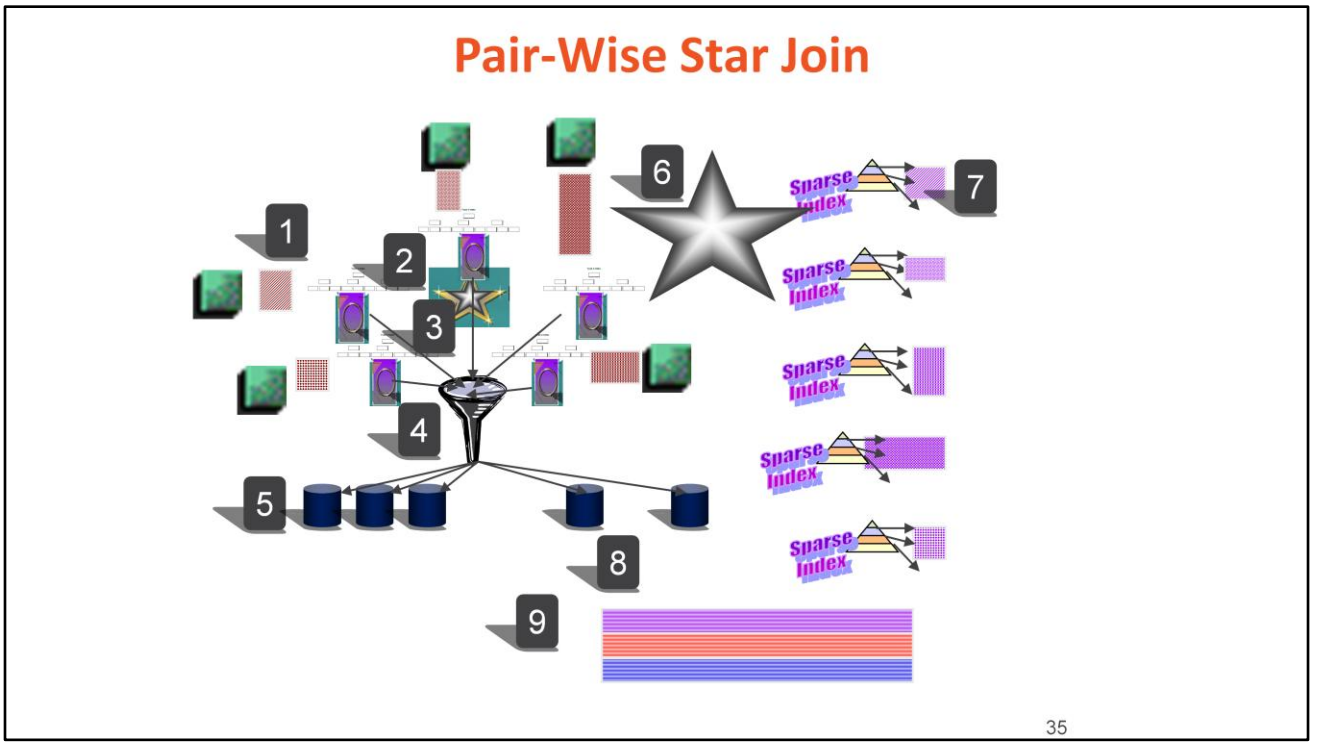

1. Access outer table using the most efficient single table access path for applying all outer table filters, a sort of these rows may be required to match the join column(s) sequence
2. Access inner table using the most efficient single table access path for applying all inner table filters, a sort of these rows may be required to match the join column(s) sequence
3. Perform match-merge check to join outer and inner table rows
4. Place filtered outer row with joining filter inner row in the result, if FULL JOIN keep all filtered outer rows and NULL missing filter inner row values, and keep all filtered inner rows and NULL missing filter outer row values

   Note: This access path is optimal when the whole result set is needed and the sort for the filtered outer and inner tables are not needed or extensive.




1. Scan small/dimension tables, merge snowflakes, applying filters, sort filtered dimension tables, and create sparse index for each (execution time may promote this and place data in-memory instead of creating sparse index)
2. *Emulate* building a gigantic Cartesian product using entries from the small/dimension data pointed to by the sparse indexes (or in-memory) avoiding entry combinations when possible
3. Probe the big/fact table index once for every calculated combination of small/dimension table join values
4. Perform index screening, applying all nonmatching predicates to leaf pages
5. Place qualifying big/fact table values with qualifying small/dimension table values and in the result
6. Use sophisticated feedback loop technology to omit unnecessary big/fact table index probes by passing back the next possible qualifying entry combination

   Note: This access path is optimal when there is high selectivity on the big/fact table index and good selectivity on the first few dimensions accessed.

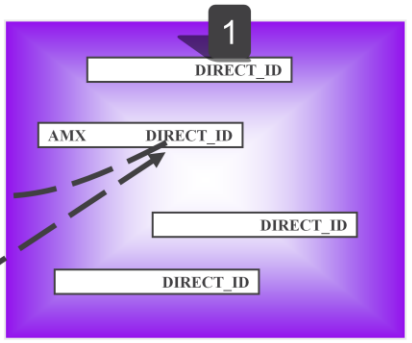



1. Scan the most filtering small/dimension tables, merge snowflakes, applying filters, and create sparse index for each (execution time may promote this and place data in-memory instead of creating sparse index)
2. Join filtered dimension to big/fact table applying matching index filters in parallel to root page of big/fact table join column indexes, probe through nonleaf to leaf pages and perform index screening
3. Sort row-ids in parallel
4. Perform dynamic index row-id ANDing
5. Gather the first 32 noncontiguous qualifying data pages in RID-list
6. Use skip sequential prefetch to retrieve 32 big/fact data pages identified in Step 4 each time, apply remaining Stage 1 predicates and then remaining Stage 2 predicates
7. Use filtered big/fact table rows
8. If SELECT columns are needed, join back to small/dimension tables sequentially through sparse indexes (execution time may promote to in-memory) if materialized in Step 1, otherwise scan the dimension
9. Place big/fact table join rows with small/dimension rows in the result

Note: This access path requires one single column index per join column on fact table, is optimal when there is high selectivity on the big/fact table index and unpredictable selectivity on the dimensions accessed. This access path is also beneficial when there is no optimal multi-column index on the fact table for Star Join – Cartesian.

# 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

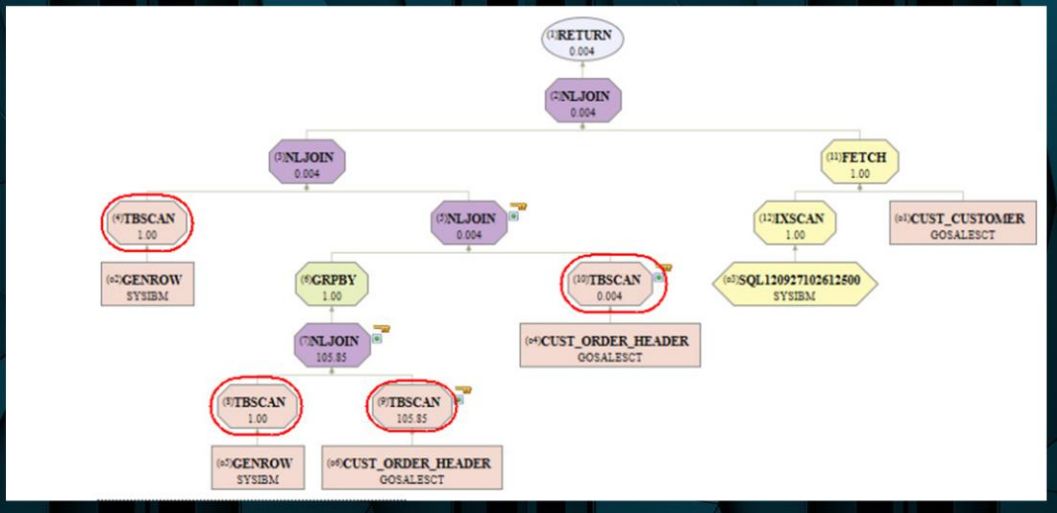



36

1. Create/alter 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**

   Note: This access path is optimal when there is high volume access to LOBs, CLOBs and DBCLOBs or high volume updates to columns.

# 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**


39

# 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 (X2QBOpt) covered in Extreme

40

# 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

41

# 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

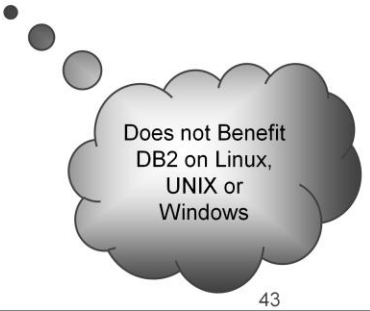
## Query #1 Tuned

```
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




43

# No Op Example CONCAT ' '

44

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

45

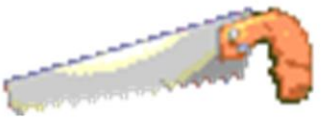
SALES_ID.MNGR.REGION Index | MNGR Index | REGION Index

```
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 UR
```

- 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



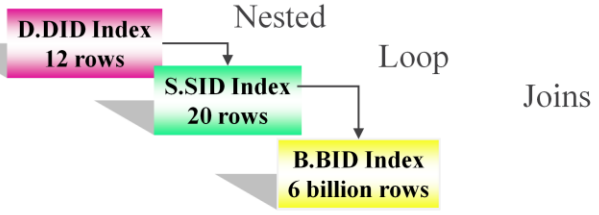
- **Fake Predicates**
  - To encourage index access
  - To alter table join sequence when nothing else works
  - Works by decreasing filter factor on a certain table
  - The filtering is fake and negligible cost
  - Not effective for dynamic queries if the filter contains :host variables

46

# Fake Filtering Example

47

```
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

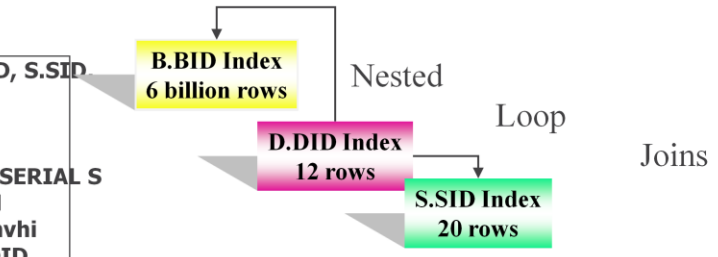
- Large report query with average of 400,000 row range of BID table
- Need to start nested loop with big table
- Tools required

# Fake Filtering Example

48

```
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 >= B.COL2
  AND B.COL3 >= B.COL3
  AND B.COL4 >= B.COL4
  AND B.COL5 >= B.COL5
  AND B.COL6 >= B.COL6
ORDER BY B.BID
```

For Dynamic




- Keep adding filters until table join sequence changes
- Start with index columns
  - To preserve index-only access

# 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

49

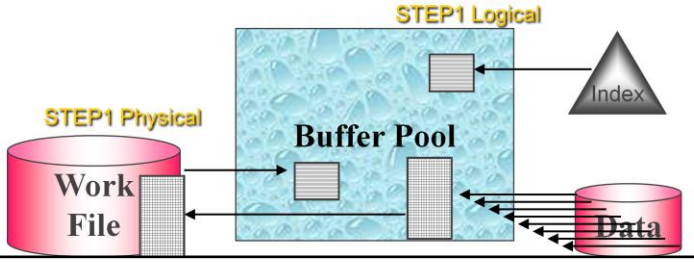
# Experiment with Extreme Techniques

## After Traditional Techniques Fail

50

# 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




51

# DISTINCT Table Expressions Example

52

- 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



BIG_TABLE is access first

Possibly results in materialized and sorted BIGZ workfile if DISTINCT cannot be satisfied using an index

Great for tuning dynamic queries!

# Typical Join Problem

53

```
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

54

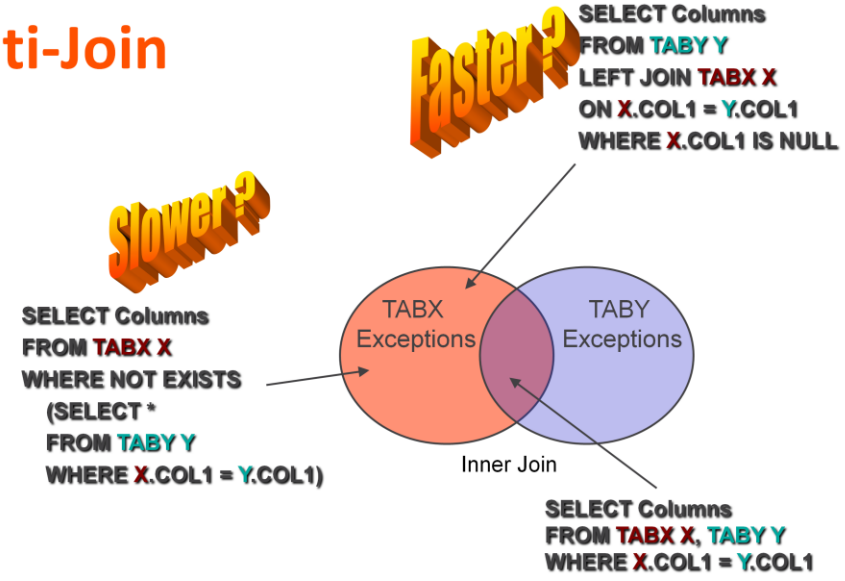
```
SELECT COL1, COL2 .....
FROM ADDR, NAME,
        (SELECT DISTINCT columns
        FROM TAB3, TAB4, TAB5, TAB6, TAB7
        WHERE join conditions
            AND (TAB6.CODE = :hv OR 0 = 1))
    AS TEMP
WHERE join conditions
```

Keeps large tables joined last

Gets rid of Index Scan

# Anti-Join

55

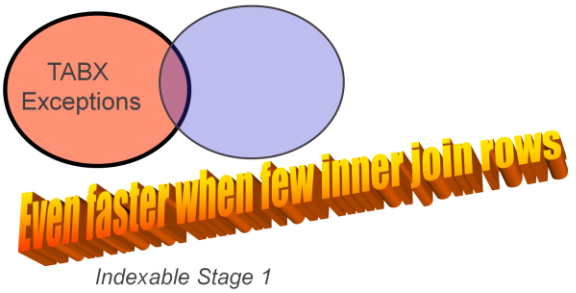


# Anti-Join

56

```
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

Sheryl Larsen
Sr. DB2 Product Specialist
President, Chicago DB2 Users Group
IBM Z Champion
(224) 343-5427
sheryl_larsen@bmc.com