IBM System z Technology Summit

What's new for SQL optimization in DB2 9 for z/OS

Disclaimer

Agenda

- **Plan Stability**
- **Indexing Enhancements**
- **General Query Performance Enhancements**
- **Histogram Statistics**
- **Global Query Optimization**
- **Generalized sparse index and in-memory data cache**
- **Dynamic Index ANDing**
- **REOPT AUTO**

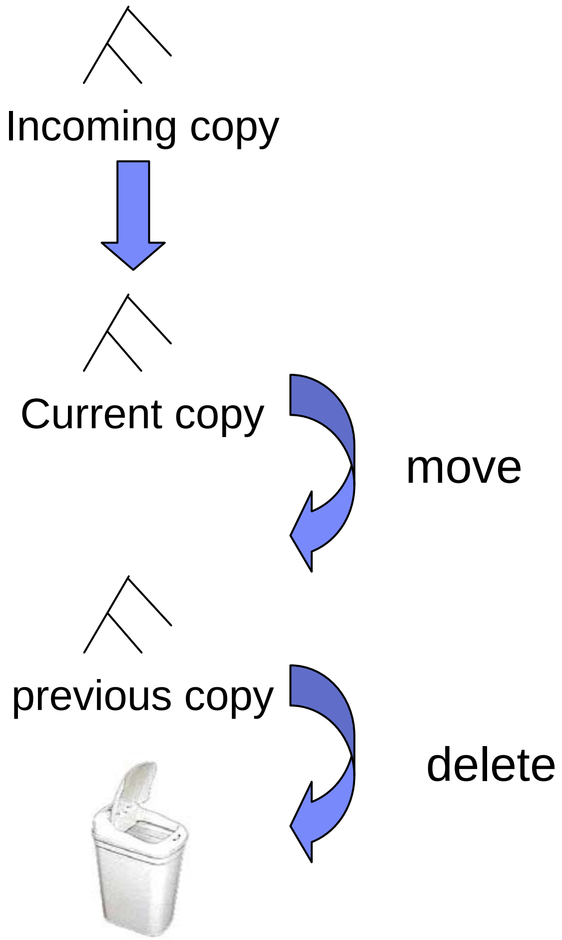
Plan Stability Overview

- **Ability to backup your static SQL packages**

- **At REBIND**
  - Save old copies of packages in Catalog/Directory
  - Switch back to previous or original version
- **Two flavors**
  - BASIC
    - 2 copies: Current and Previous
  - EXTENDED
    - 3 copies: Current, Previous, Original
  - Default controlled by a ZPARM
  - Also supported as REBIND options

Plan Stability - BASIC support

REBIND … PLANMGMT(BASIC)



REBIND … SWITCH(PREVIOUS)

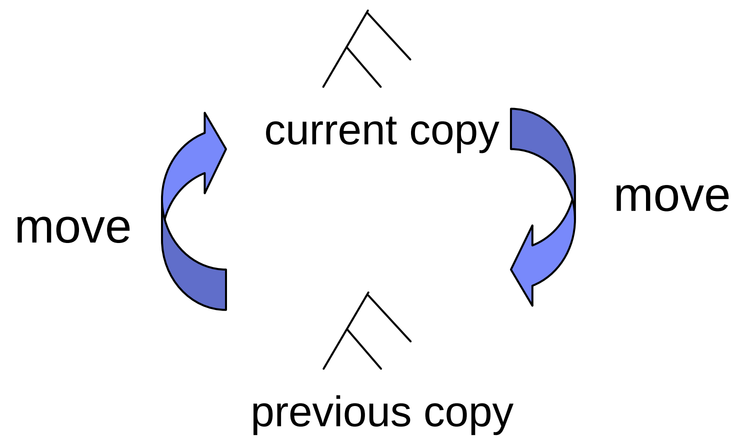
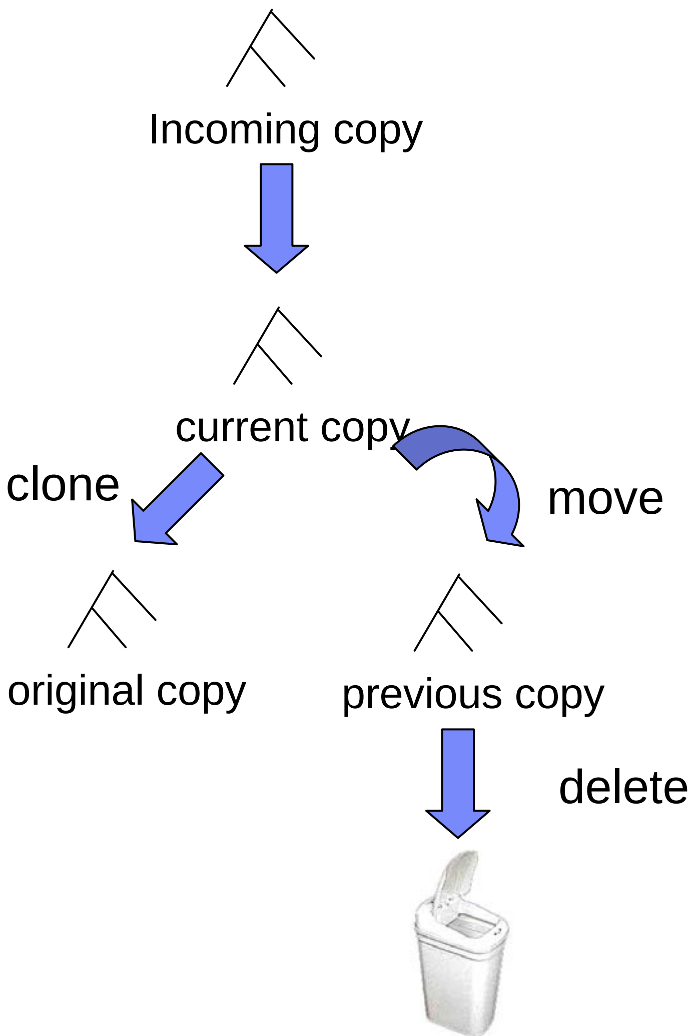


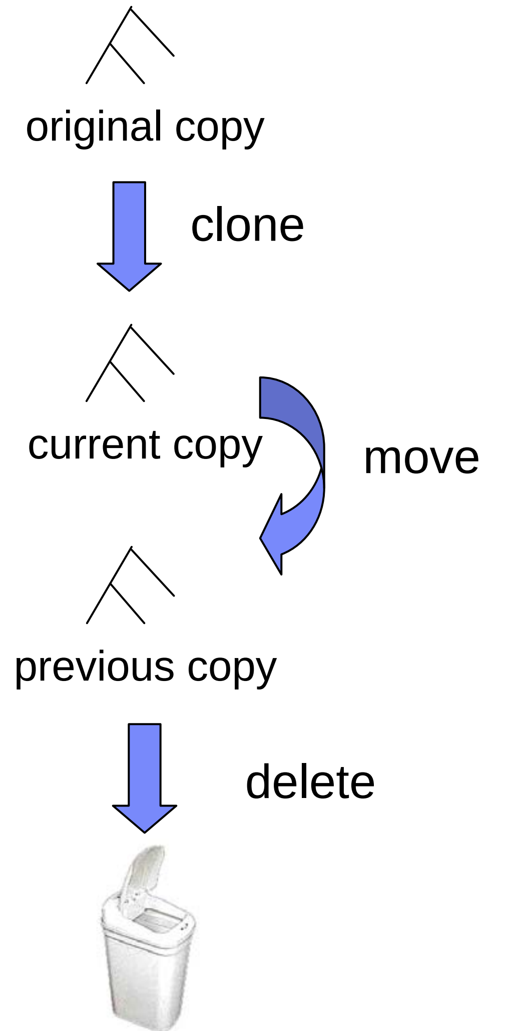
Chart is to be read from bottom to top

Plan Stability - EXTENDED support

REBIND … PLANMGMT(EXTENDED)



REBIND … SWITCH(ORIGINAL)



Access Plan Stability Notes

- **REBIND PACKAGE …**
  - PLANMGMT (BASIC)

  2 copies: Current and Previous
  - PLANMGMT (EXTENDED)

  3 copies: Current, Previous, Original

- **REBIND PACKAGE …**
  - SWITCH(PREVIOUS)

  Switch between current & previous
  - SWITCH(ORIGINAL)

  Switch between current & original

- **Most bind options can be changed at REBIND**
  - *But a few must be the same …*

3 important updates:

1. APAR PK80375 – SPT01 Compression
2. APAR PM09354 – Support DBPROTOCOL change
3. Article – Search for "Escaping the REBIND blues in DB2 9 for z/OS"

- **FREE PACKAGE …**
  - PLANMGMTSCOPE(ALL) – Free package completely
  - PLANMGMTSCOPE(INACTIVE) – Free old copies

- **Catalog support**
  - SYSPACKAGE reflects active copy
  - SYSPACKDEP reflects dependencies of all copies
  - Other catalogs (SYSPKSYSTEM, …) reflect metadata for all copies

- **Invalidation and Auto Bind**
  - Each copy invalidated separately

Agenda

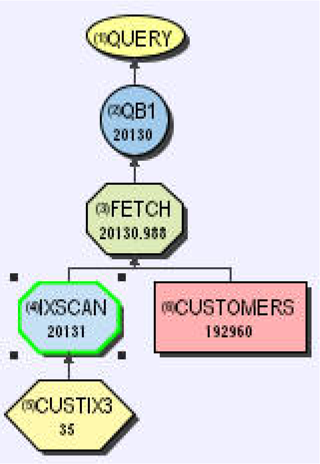
- **Plan Stability**
- **Indexing Enhancements**
- **General Query Performance Enhancements**
- **Histogram Statistics**
- **Global Query Optimization**
- **Generalized sparse index and in-memory data cache**
- **Dynamic Index ANDing**
- **REOPT AUTO**

Index on Expression

- **DB2 9 supports “index on expression”**
  - Can turn a stage 2 predicate into indexable

```
SELECT *
FROM CUSTOMERS
WHERE YEAR(BIRTHDATE) = 1971

CREATE INDEX ADMF001.CUSTIX3
     ON ADMF001.CUSTOMERS
(YEAR(BIRTHDATE) ASC)
```



Previous FF = 1/25
Now, RUNSTATS collects frequencies. Improved FF accuracy

Name	Value
Input RIDs	192960
Index Leaf Pages	241
Matching Predicates	Filter Factor
ADMF001.CUSTOMERS.= CAST(1971 AS INTEGER)	0.1043
Scanned Leaf Pages	26
Output RIDs	20131
Total Filter Factor	0.1043
Matching Columns	1

Index Enhancement - Tracking Usage

- **Additional indexes require overhead for**
  - Utilities
    - REORG, RUNSTATS, LOAD etc
  - Data maintenance
    - INSERT, UPDATE, DELETE
  - Disk storage
  - Optimization time
    - Increases optimizer's choices

- **But identifying unused indexes is a difficult task**
  - Especially in a dynamic SQL environment

Tracking Index Usage

- **RTS records the index last used date.**
  - SYSINDEXSPACESTATS.LASTUSED
    - Updated once in a 24 hour period
      - RTS service task updates at 1st externalization interval (set by STATSINT) after 12PM.
    - if the index is used by DB2, update occurs.
    - If the index was not used, no update.

- **"Used", as defined by DB2 as:**
  - As an access path for query or fetch.
  - For searched UPDATE / DELETE SQL statement.
  - As a primary index for referential integrity.
  - To support foreign key access

Tracking Index Usage Implications

- **What can you do with this information?**
  - LAST_USED only shows when the index was last used
    - Cannot predict future use
  - Assume you decide to DROP an index due to lack of usage
    - Is the index UNIQUE?
      - Is there another index that can guarantee that UNIQUEness?
    - Related statistics will be dropped
      - Same issue as “What If?” Optimization
      - For index on C1, C2, C3
        - > RUNSTATS options to collect statistics

```
COLGROUP (C1) FREQVAL COUNT 10
COLGROUP (C1, C2, C3)
```

Agenda

- **Plan Stability**
- **Indexing Enhancements**
- **General Query Performance Enhancements**
- **Histogram Statistics**
- **Global Query Optimization**
- **Generalized sparse index and in-memory data cache**
- **Dynamic Index ANDing**
- **REOPT AUTO**

GROUP BY Sort Avoidance

- **Improved sort avoidance for GROUP BY**
  – Reorder GROUP BY columns to match available index

```
SELECT … FROM T1
GROUP BY C2, C1     ←GROUP BY in C2, C1 sequence

Index 1 (C1, C2)    ←Index in C1, C2 sequence
```

  – Remove 'constants' from GROUP BY ordering requirement

```
SELECT … FROM T1
WHERE C2 = 5        ←C2 Constant
GROUP BY C2, C1
```

    • ordering requirement reduced to just C1

GROUP BY Sort Avoidance

- **Continued….**
  - Allow swapping of ordering columns using transitive closure

```
SELECT … FROM T1, T2
WHERE T1.C1 = T2.C1
GROUP BY T1.C1, T2.C3  ←Contains T1 & T2
```

    - ordering requirement changed to T2.C1, T2.C3

  - Improvement for 'partially ordered' cases with unique index

```
SELECT C1, C2+C3, C4 FROM T1
GROUP BY 1, 2, 3
```

    - if we have unique index on C4, C1
      - Sort can be avoided

GROUP BY Sort Avoidance Implications

- **Implications of improved sort avoidance for GROUP BY**
  - May improve query performance!!!
  - Data may be returned in a different order
    - Always been true in any DB2 release
      - Also true in other DBMSs
    - Relational theory states that order is NOT guaranteed without ORDER BY

Sort Improvements

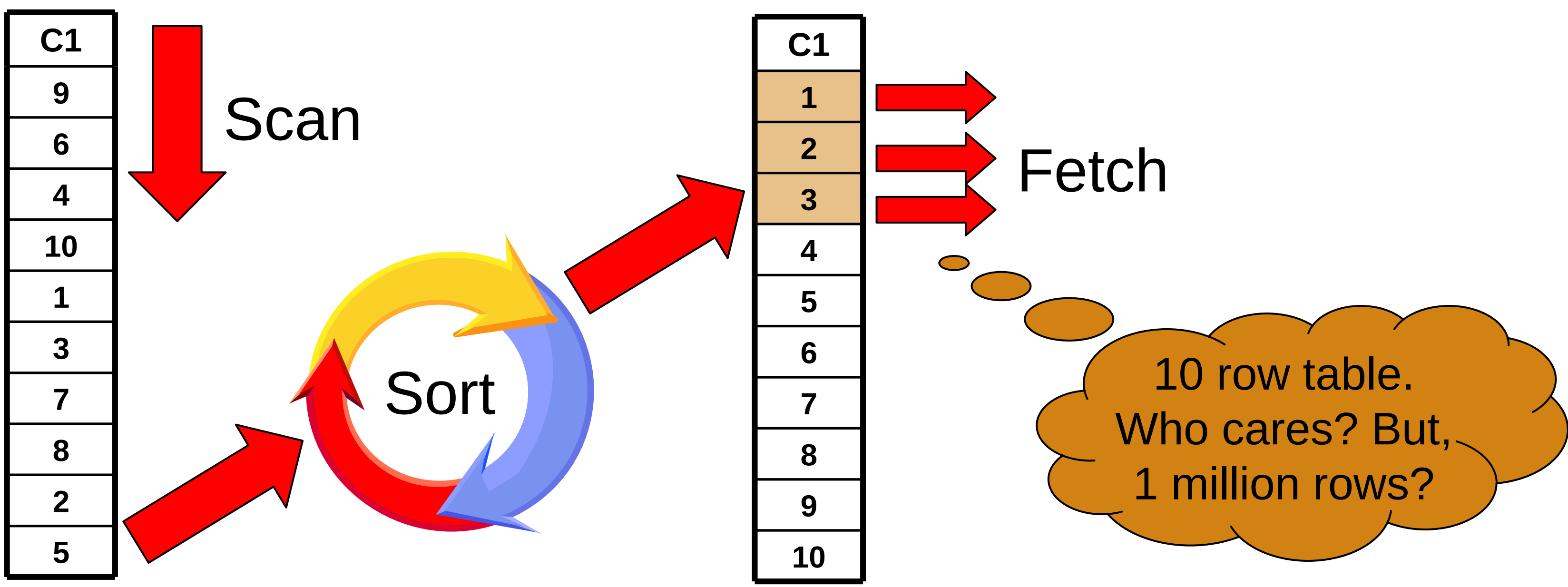
- **Reduced workfile usage for very small sorts**
  - Final sort step requiring 1 page will NOT allocate workfile

- **More efficient sort with FETCH FIRST clause**
  - V8 and prior,
    - Sort would continue to completion
    - Then return only the requested ‘n’ rows
  - From V9,
    - If the requested ‘n’ rows will fit into a 32K page,
      - As the data is scanned,
        - > Only the top ‘n’ rows are kept in memory
        - > Order of the rows is tracked
        - > No requirement for final sort

FETCH FIRST V8 Example

- **Sort is not avoided via index**
  - – Must sort all qualified rows

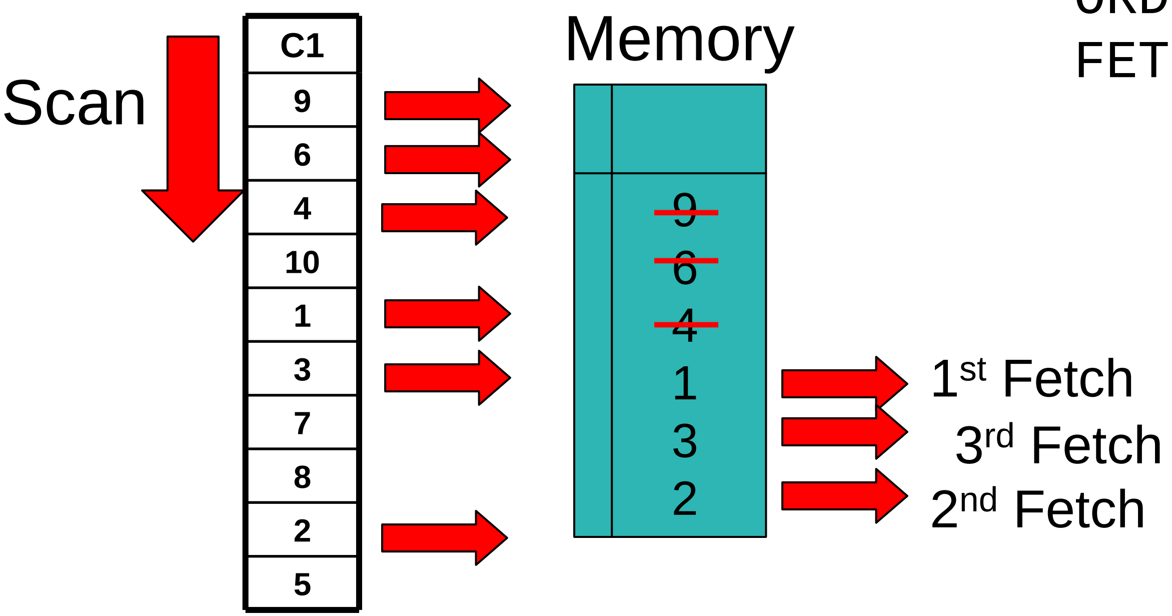
```
SELECT C1
FROM  T
ORDER BY C1
FETCH FIRST 3 ROWS ONLY
```



FETCH FIRST DB2 9 Example

- **Sort is not avoided via index**
  - But in-memory swap avoids sort
    - Pointers maintain order

```
SELECT C1
FROM   T
ORDER BY C1
FETCH FIRST 3 ROWS ONLY
```



Suggestion: Add FETCH FIRST when subset is required.

Dynamic Prefetch Enhancements

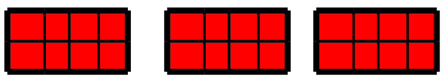
Sequential Prefetch	Dynamic Prefetch
Chosen at bind/prepare time	**Detected at runtime**
Requires hit to a triggering page	**Tracks sequential access pattern**
Only prefetch in one direction	**Prefetch forward or backward**
Used for tablespace scan & LOBs	**Used for index & index+data access**

- **Seq. Pref. cannot fall back to Dyn. Pref. at run time**
- **Plan table may still show 'S' for IX + Data access**

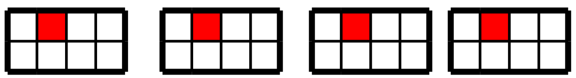
- ET reductions between 5-50% measured at SVL
- 10-75% reduction in synchronous I/O's

Clusterratio Enhancement

- **New Clusterratio formula in DB2 9**
  - Including new DATAREPEATFACTOR statistic
    - Differentiates density and sequential



Dense (and sequential)



Sequential (not dense)

- **Controlled by zparm STATCLUS**
  - ENHANCED is default
  - STANDARD disables, and is NOT recommended

- **Recommend RUNSTATS before mass REBIND in DB2 9**

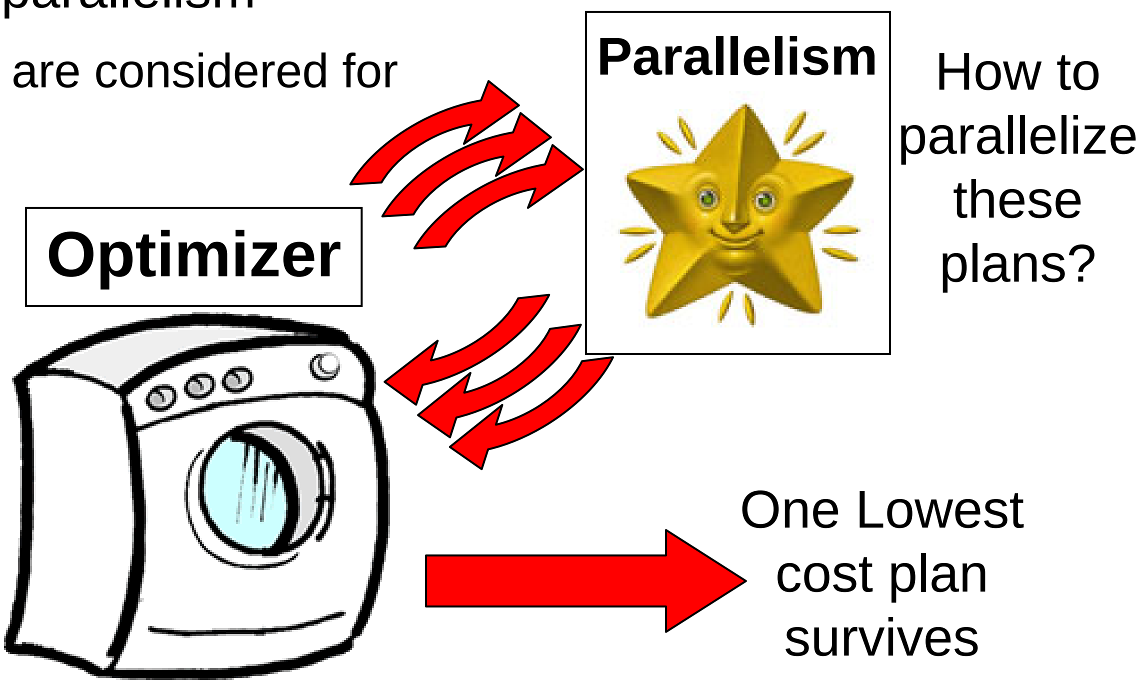
Clusterratio Impacts

- **Clusterratio may be**
  - Higher for indexes
    - With many duplicates (lower colcardf)
      - In recognition of sequential RIDs
    - On smaller tables
      - Less clusterratio degradation from random inserts
    - Indexes that are reverse sequential
  - Lower for random indexes
    - No benefit from dynamic prefetch
- **Clusterratio(CR)/DataRepeatfactor (DRF) patterns**

	High DRF	**Low DRF**
High CR	**Sequential but not dense**	**Density matching clustering or small table**
Low CR	**Random index**	**Unlikely**

Parallelism Enhancements

- **In V8**
  - Lowest cost is BEFORE parallelism
- **In DB2 9**
  - Lowest cost is AFTER parallelism
    - Only a subset of plans are considered for parallelism



Additional Parallelism Enhancements

- **In V8**
  - –Degree cut on leading table (exception star join)

- **In DB2 9**
  - –Degree can cut on non-leading table
    - Benefit for leading workfile, 1-row table etc.
  - –Histogram statistics exploited for more even distribution
    - For index access with NPI
  - –CPU bound query degree <= # of CPUs * 4
    - <= # of CPUs in V8

Agenda

- Plan Stability
- Indexing Enhancements
- General Query Performance Enhancements
- Histogram Statistics
- Global Query Optimization
- Generalized sparse index and in-memory data cache
- Dynamic Index ANDing
- REOPT AUTO

RUNSTATS Histogram Statistics

- RUNSTATS **will produce equal-depth histogram**
  - Each quantile (range) will have approx same number of rows
    - Not same number of values
  - Another term is range frequency

- **Example**
    - 1, 3, 3, 4, 4, 6, 7, 8, 9, 10, 12, 15 (sequenced)
  - Lets cut that into 3 quantiles.
    - 1, 3, 3, 4 ,4 6,7,8,9 10,12,15

Seq No	Low Value	High Value	Cardinality	Frequency
1	**1**	**4**	**3**	**5/12**
2	**6**	**9**	**4**	**4/12**
3	**10**	**15**	**3**	**3/12**

RUNSTATS Histogram Statistics Notes

- **RUNSTATS**
  - Maximum 100 quantiles for a column
  - Same value columns WILL be in the same quantile
  - Quantiles will be similar size but:
    - Will try to avoid big gaps inside quantiles
    - Highvalue and lowvalue may have separate quantiles
    - Null WILL have a separate quantile

- **Supports column groups as well as single columns**

- **Think “frequencies” for high cardinality columns**

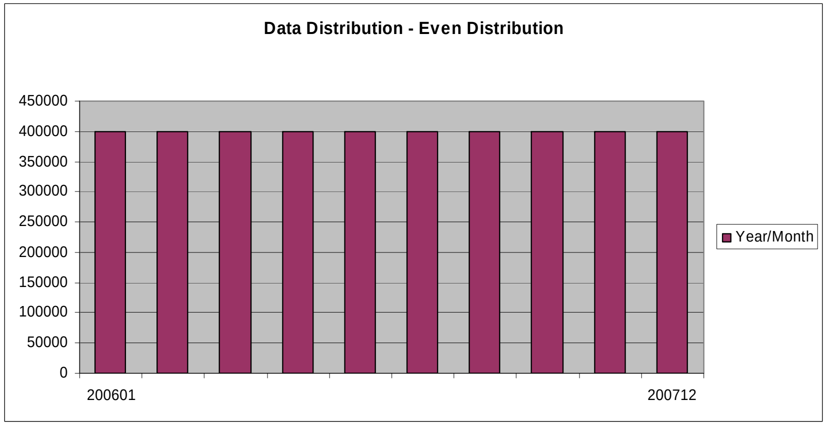
Histogram Statistics Example

- **SAP uses INTEGER (or VARCHAR) for YEAR-MONTH**

WHERE YEARMONTH BETWEEN 200601 AND 200612

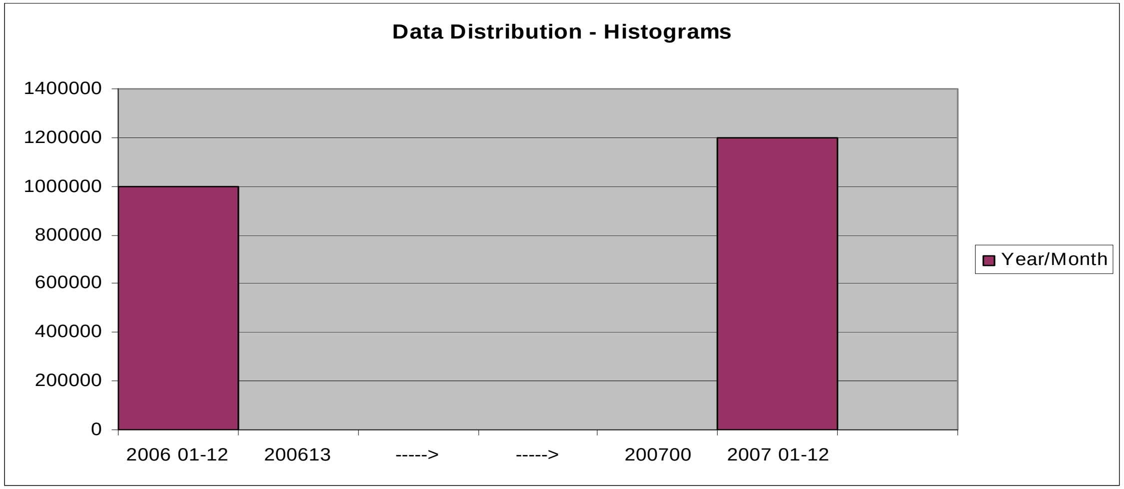
- Assuming data for 2006 & 2007
  - FF = (high-value – low-value) / (high2key – low2key)
  - FF = (200612 – 200601) / (200711 – 200602)
  - **10% of rows estimated to return**

Data assumed as evenly distributed between low and high range



Histogram Statistics Example

- Example (cont.)
  - Data only exists in ranges 200601-12 & 200701-12
    - Collect via histograms
      - 45% of rows estimated to return



No data between 200613 & 200700

WHERE YEARMONTH BETWEEN 200601 AND 200612

Agenda

- **Plan Stability**
- **Indexing Enhancements**
- **General Query Performance Enhancements**
- **Histogram Statistics**
- **Global Query Optimization**
- **Generalized sparse index and in-memory data cache**
- **Dynamic Index ANDing**
- **REOPT AUTO**

Problem Scenario 1

- **V8, Large Non-correlated subquery is materialized***

```
SELECT * FROM SMALL_TABLE A
WHERE A.C1 IN
        (SELECT B.C1 FROM BIG_TABLE B)
```

* Assumes subquery is not transformed to join

  - "BIG_TABLE" is scanned and put into workfile
  - "SMALL_TABLE" is joined with the workfile

- **V9 may rewrite non-correlated subquery to correlated**
  - Much more efficient if scan / materialisation of BIG_TABLE was avoided
  - Allows matching index access on BIG_TABLE

```
SELECT * FROM SMALL_TABLE A
WHERE EXISTS
  (SELECT 1 FROM BIG_TABLE B WHERE B.C1 = A.C1)
```

Problem Scenario 2

- **V8, Large outer table scanned rather than using matching index access***

```
SELECT * FROM BIG_TABLE A
WHERE EXISTS
  (SELECT 1 FROM SMALL_TABLE B WHERE A.C1 = B.C1)
```

  - "BIG_TABLE" is scanned to obtain A.C1 value
  - "SMALL_TABLE" gets matching index access

* Assumes subquery is not transformed to join

- **V9 may rewrite correlated subquery to non-correlated**

```
SELECT * FROM BIG_TABLE A
WHERE A.C1 IN
  (SELECT B.C1 FROM SMALL_TABLE B)
```

  - "SMALL_TABLE" scanned and put in workfile
  - Allows more efficient matching index access on BIG_TABLE

Virtual Tables

- **A new way to internally represent subqueries**
  – Represented as a Virtual table
    - Allows subquery to be considered in different join sequences
    - May or may not represent a workfile

    - Apply only to subqueries that cannot be transformed to joins

Correlated or non-correlated?......I shouldn't have to care!

Agenda

- **Plan Stability**
- **Indexing Enhancements**
- **General Query Performance Enhancements**
- **Histogram Statistics**
- **Global Query Optimization**
- **Generalized sparse index and in-memory data cache**
- **Dynamic Index ANDing**
- **REOPT AUTO**

Pre-V9 Sparse Index & in-memory data cache

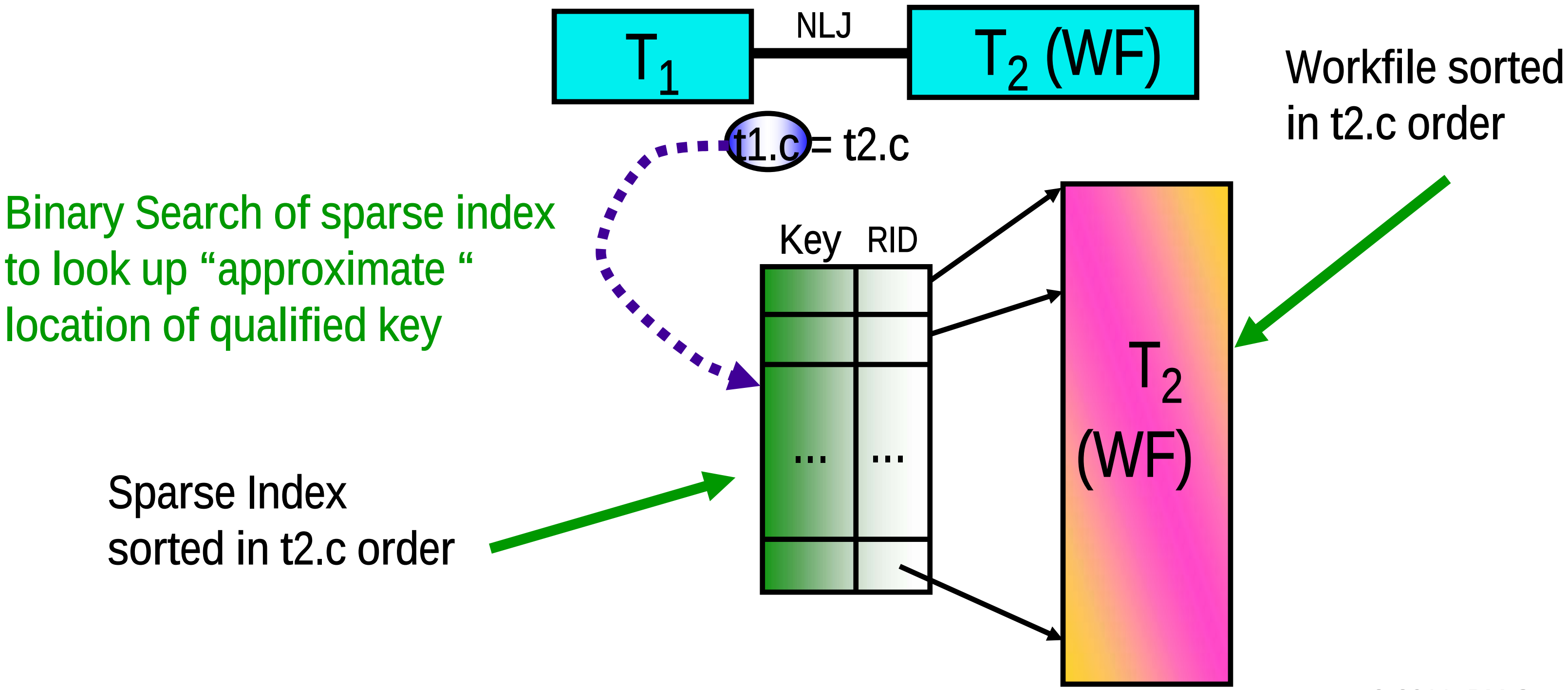
- **V4 introduced sparse index**
  – for non-correlated subquery workfiles

- **V7 extended sparse index**
  – for the materialized work files within star join

- **V8 replaced sparse index**
  – with in-memory data caching for star join
    • Runtime fallback to sparse index when memory is insufficient

How does Sparse Index work?

- Sparse index may be a subset of workfile (WF)
  - Example, WF may have 10,000 entries
    - Sparse index may have enough space for 1,000 entries
    - Sparse index is "binary searched" to find target location of search key
    - At most 10 WF entries are scanned



Data Caching vs Sparse Index

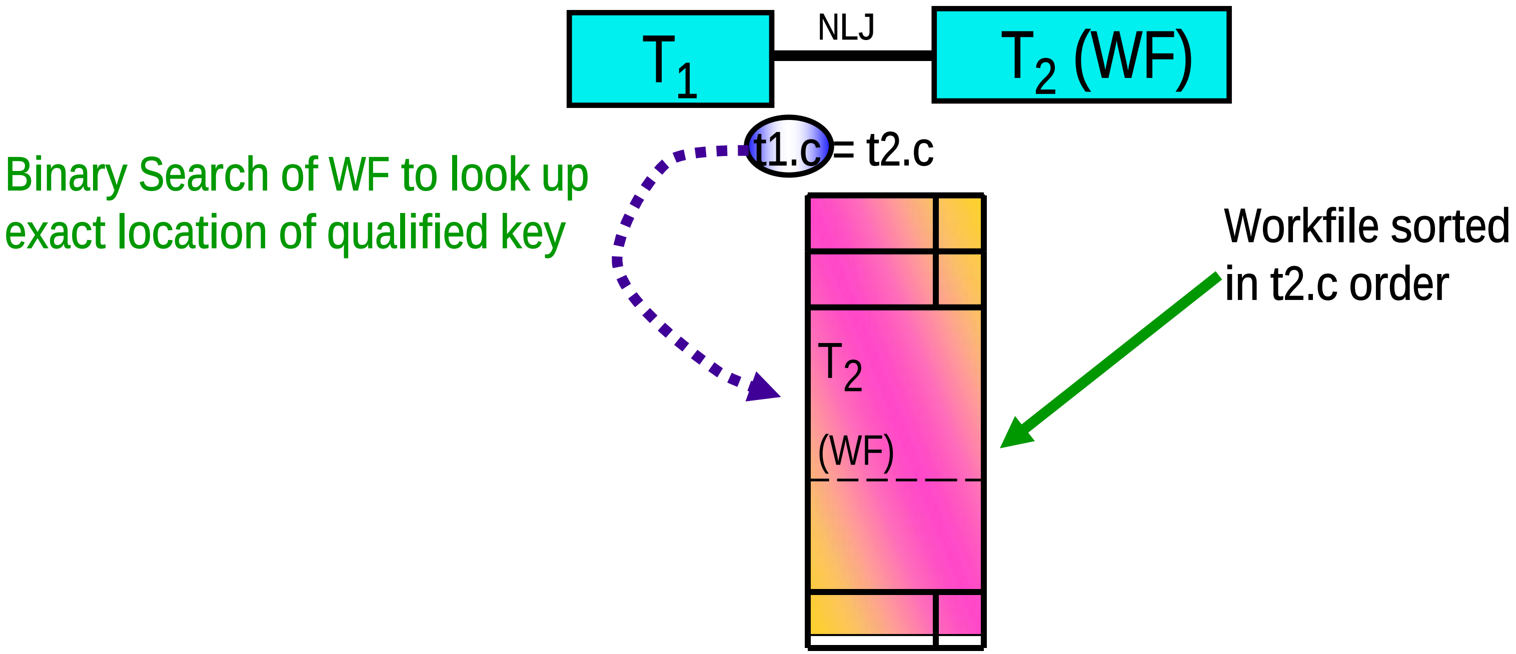
- **Data Caching**
  - Also known as In-Memory WF
  - Is a runtime enhancement to sparse index

- **Sparse Index/In-Memory WF**
  - Extended to non-star join in DB2 9

- **New ZPARM MXDTCACH**
  - Maximum extent in MB, for data caching per thread
  - If memory is insufficient
    - Fall-back to sparse index at runtime

How does In-Memory WF work?

- Whereas sparse index may be a subset of WF
  - IMWF contains the full result (not sparse)
  - Example, WF may have 10,000 entries
    - IMWF is "binary searched" to find target location of search key



Benefit of Data Caching

- **All tables lacking an index on join column(s):**
  - Temporary tables
  - Subqueries converted to joins
  - …..any table

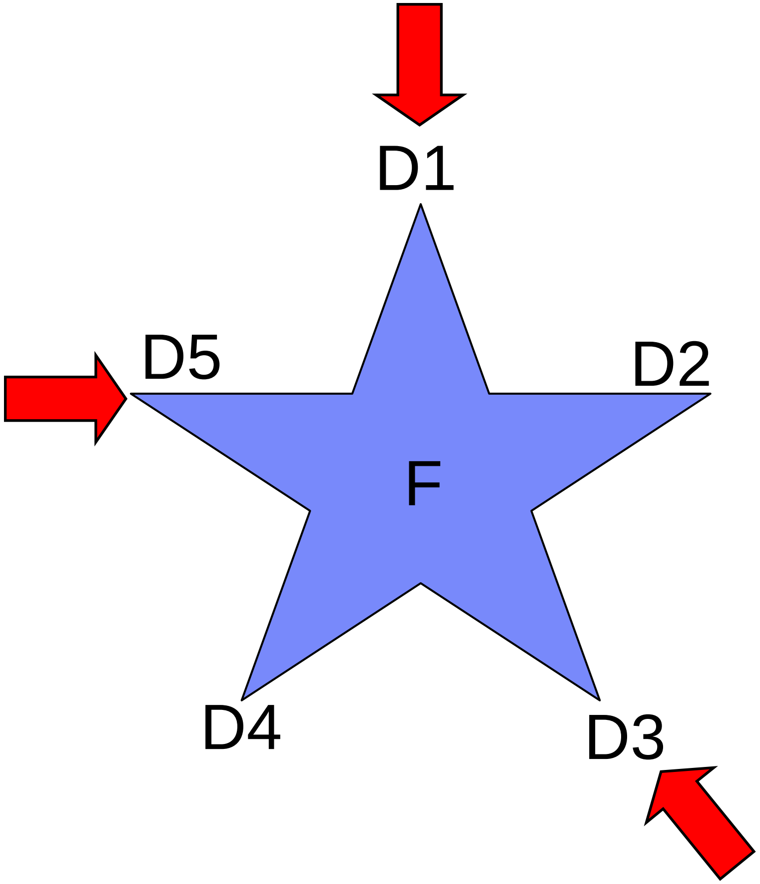
- **DB2 9 also supports multi-column sparse index**

Agenda

- **Plan Stability**
- **Indexing Enhancements**
- **General Query Performance Enhancements**
- **Histogram Statistics**
- **Global Query Optimization**
- **Generalized sparse index and in-memory data cache**
- **Dynamic Index ANDing**
- **REOPT AUTO**

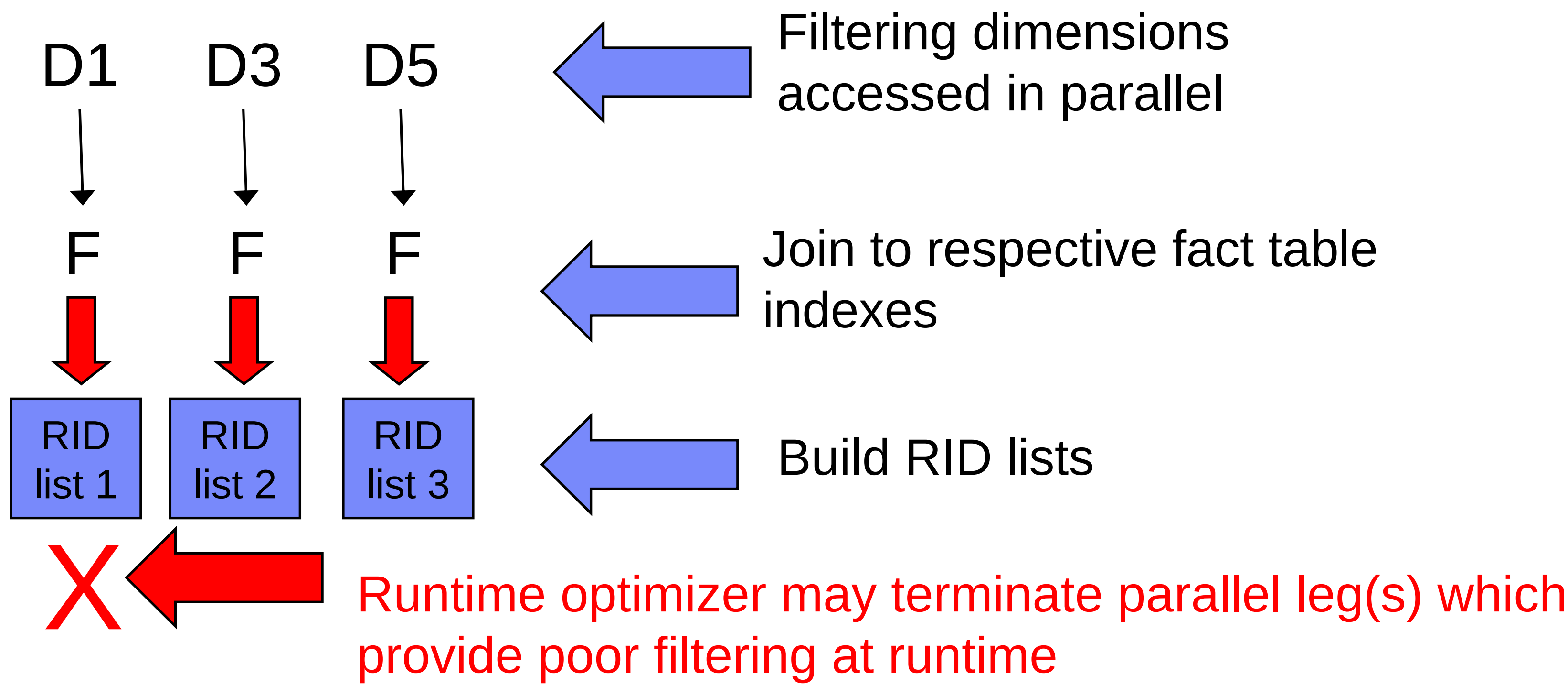
Dynamic Index ANDing Challenge

- **Filtering may come from multiple dimensions**
  - Creating multi-column indexes to support the best combinations is difficult



Index ANDing – Pre-Fact

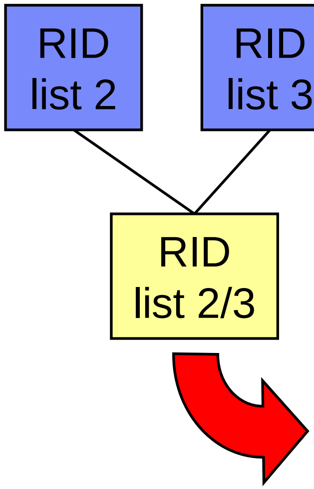
- **Pre-fact table access**
  - –Filtering may not be (truly) known until runtime



Index ANDing – Fact and Post-Fact

- **Fact table access**
  - –Intersect filtering RID lists
  - –Access fact table
    - From RID list
- **Post fact table**
  - –Join back to dimension tables

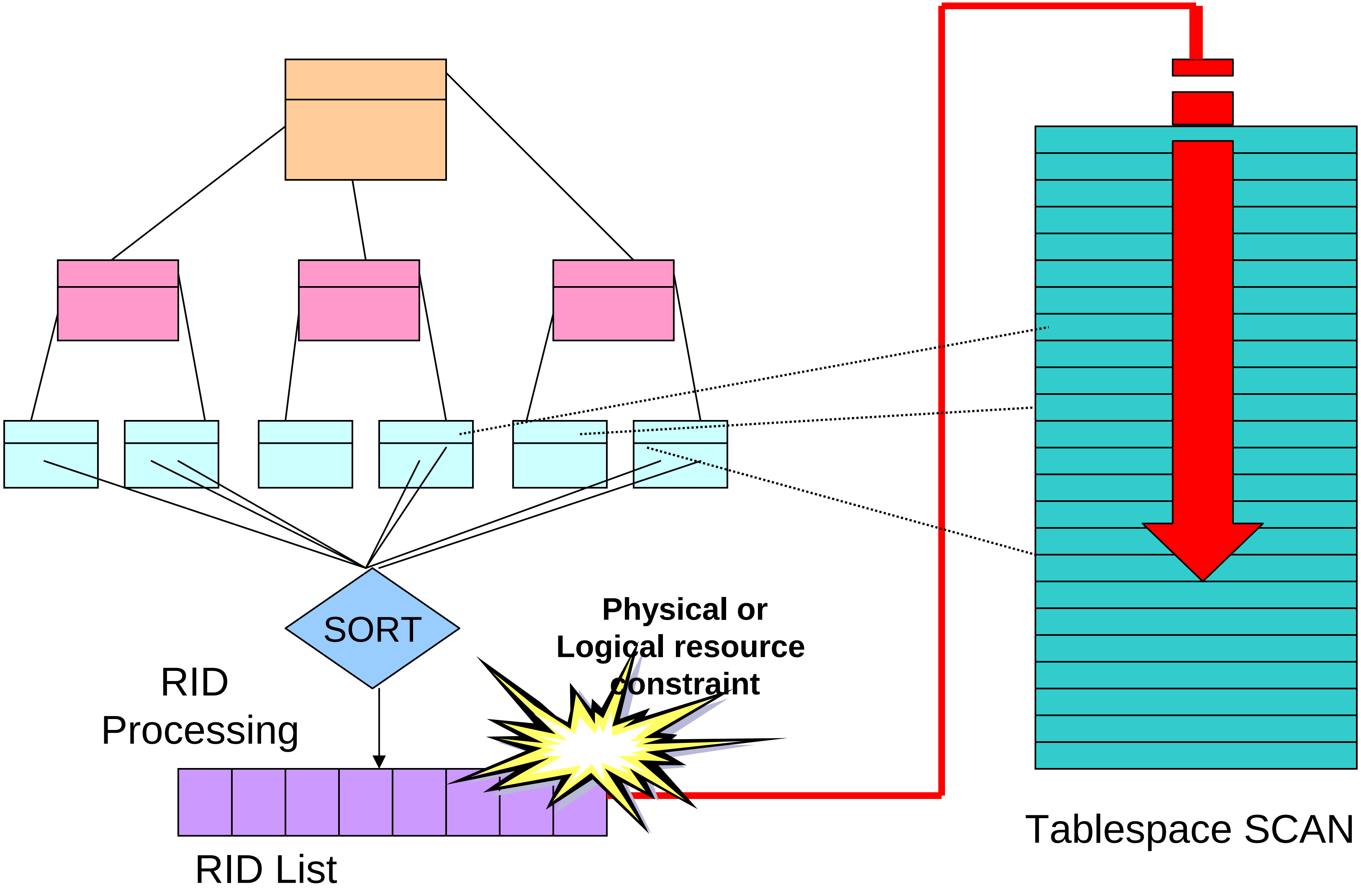
Using parallelism



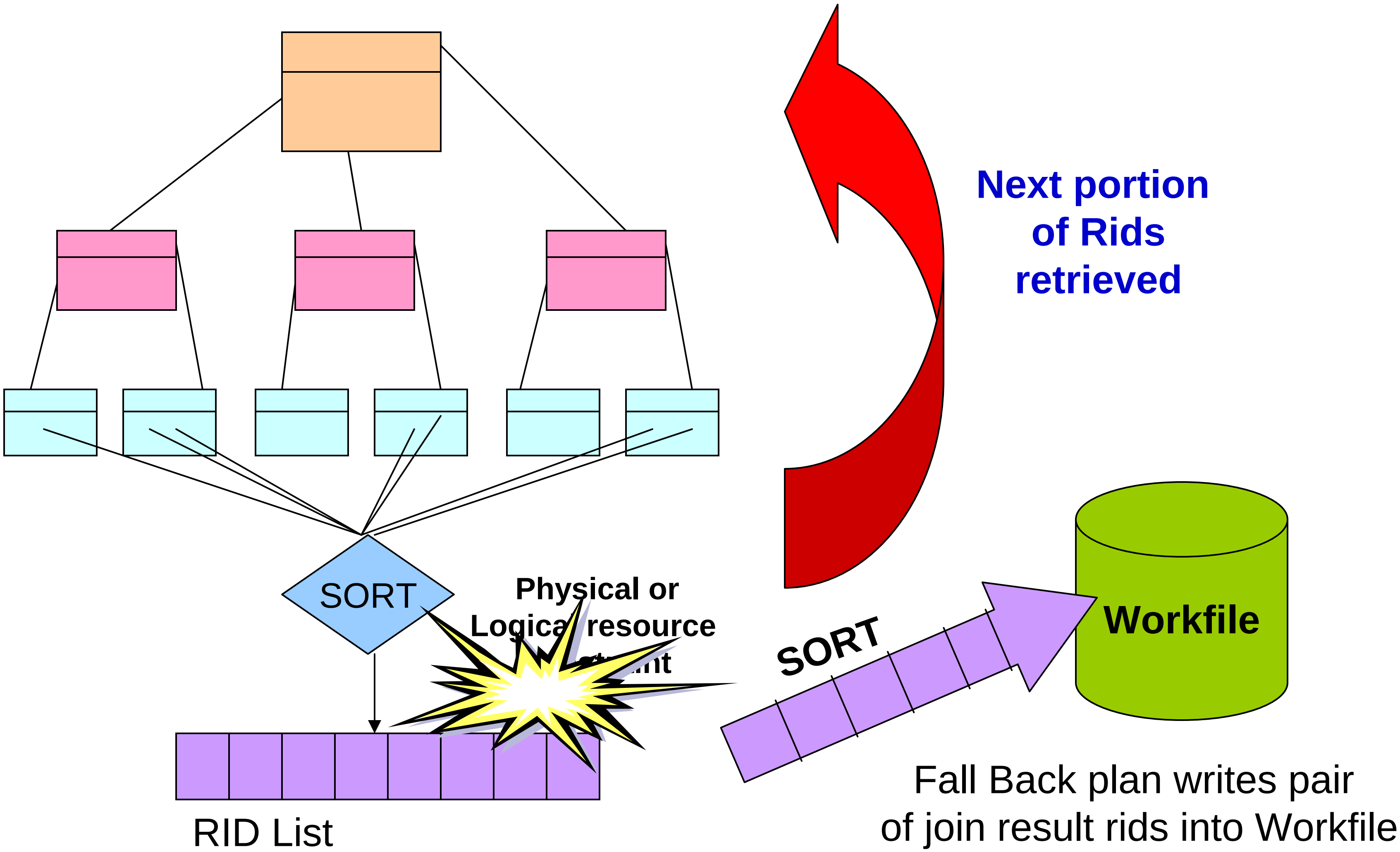
Remaining RID lists are "ANDed" (intersected)

Final RID list used for parallel fact table access

V8 RID Pool failure = TS Scan



V9 RID Pool Fallback Plan



Dynamic Index Anding Highlights

- **Pre-fact table filtering**
  - Filtering dimensions accessed concurrently
- **Runtime optimization**
  - Terminate poorly filtering legs at runtime
- **More aggressive parallelism**
- **Fallback to workfile for RID pool failure**
  - Instead of r-scan

APAR PK76100 – zparm to enable EN_PJSJ

Agenda

- **Plan Stability**
- **Indexing Enhancements**
- **General Query Performance Enhancements**
- **Histogram Statistics**
- **Global Query Optimization**
- **Generalized sparse index and in-memory data cache**
- **Dynamic Index ANDing**
- **REOPT AUTO**

REOPT enhancement for dynamic SQL

- **V8 REOPT options**
  – Dynamic SQL
    - REOPT(NONE, ONCE, ALWAYS)
  – Static SQL
    - REOPT(NONE, ALWAYS)

- **V9 Addition for Dynamic SQL**
  – Bind option REOPT(AUTO)

Dynamic SQL REOPT - AUTO

- **For dynamic SQL with parameter markers**
  - DB2 will automatically reoptimize the SQL when
    - Filtering of one or more of the predicates changes dramatically
      - Such that table join sequence or index selection may change
    - Some statistics cached to improve performance of runtime check
  - Newly generated access path will replace the global statement cache copy.

- **First optimization is the same as REOPT(ONCE)**
  - Followed by analysis of the values supplied at each execution of the statement