
**Session: F08**

# Performance Topics for Application Tuners

Sheryl M. Larsen
*Sheryl M. Larsen, Inc.*

**Tuesday 9 October 2009 •15:45 – 16:45**
**Platform: DB2 for z/OS**

IDUG
The Worldwide DB2 User Community

***Notes:***

# Objectives

- Objective 1: Demonstrate when to change the DB2 Optimizer's Mind
- Objective 2: Demonstrate how to change the DB2 Optimizer's Mind
- Objective 3: Demonstrate how to design indexes for a Workload
- Objective 4: Demonstrate when manual query re-write is necessary
- Objective 5: Demonstrate when layered table expressions are not effective


© Sheryl M. Larsen, Inc. 2008-2009

***Notes:*** Outline:

# Table of Contents





© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

# Table of Contents (Continued)

## Step 4 & 5 – Learn Tuning Techniques & Apply When Necessary

Learn Traditional Tuning Techniques
- Using OPTIMIZE FOR n ROWS
- Using No Ops
- Using Fake Filtering
- ON 1 = 1

Experiment with Extreme Tuning Techniques
- Using Table Expressions
- Using Anti-Joins
- Odd/old Techniques are Still Needed
- Keep Query "Thin" Technique
- MQT Optimization

## Step 6 – Optimize Index Design for the Workload


© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

## 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.

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.

***Notes:***

# SQL Performance Rules …..

- Promote Stage 2's/Residuals, if possible
  - Value BETWEEN COL1 AND COL2
  - Rewrite to: (Value >= COL1 AND value <= COL2)


  - COL NOT IN (K, S, T) = COL IN (known values)
- SELECT only the columns needed
  - Disallow SELECT *
- SELECT only the rows needed
  - Disallow program filtering
- Use constants and literals if the values will not change in the next 3 years
  - Increase optimizer attribute for static values


© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

These are the top performance rules for coding high performing SQL statements. These rules are created with intimate knowledge of the current release of DB2.

The first rule is the most critical.

# …… SQL Performance Rules

- Make data types match – until V8 z/OS
  - **Must be exact match or demoted to 4th point of filtering**
- Do not SELECT columns with known static values
  - **Disallow COLn from SELECT list if COLn = value**
- Do not place any local filtering in the ON clause
  - **Disallow "during join" filtering, encourage "before join"**
- Sequence filtering from most restrictive to least restrictive by table, by predicate type

```
WHERE   A.COL2 = 'abracadabra'
  AND   A.COL1 IN (:hv1, :hv2, :hv3)
  AND   A.COL4 > :hvcol4
  AND   A.COL3 > :hvcol3
  AND   A.COL5 LIKE '%SON'
```

***Ties Matter!***

**For high volume only**



© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

Predicates should be separated by table, by predicate type. Within each table, sequence the filtering from most restrictive to least within a predicate type.

**Predicate Types**: =, IN (equality),

>,>=, <, <=, Between (range)

LIKE

non correlated subsqueries

correlated subqueries (cost should also be taken into consideration)

The true order of filtering is dependent on the join sequence, join method, and index selection. These optimizer decisions determine which predicates are in which class.

**Classes of filtering:**

Matching Indexable

Nonmatching Stage 1 (index screening)

Stage 1 (data screening)

Stage 2

FOR HIGH VOLUME – tons of data or tons of transactions

Predicates are evaluated within a class, as the tables are accessed, predetermined by the access path.

# SQL Review Checklist

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



© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

1. Examine Program logic – check for program filtering and joining. Move work into the query. Check for misuse of SQL features, i.e. no DISTINCT Joins, and many others that are presented in the Advanced SQL class.
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, see SQL Performance Rules

-- Verify data type matches – mismatched numeric and date/time will cause delays due to the translations. As of V8, conversions are tolerated by the Index Manager but cost CPU.

5. 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.
6. 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
7. 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.
8. Tune if necessary – Change the .Optimizer's mind when access path is non-optimal

# Step 3

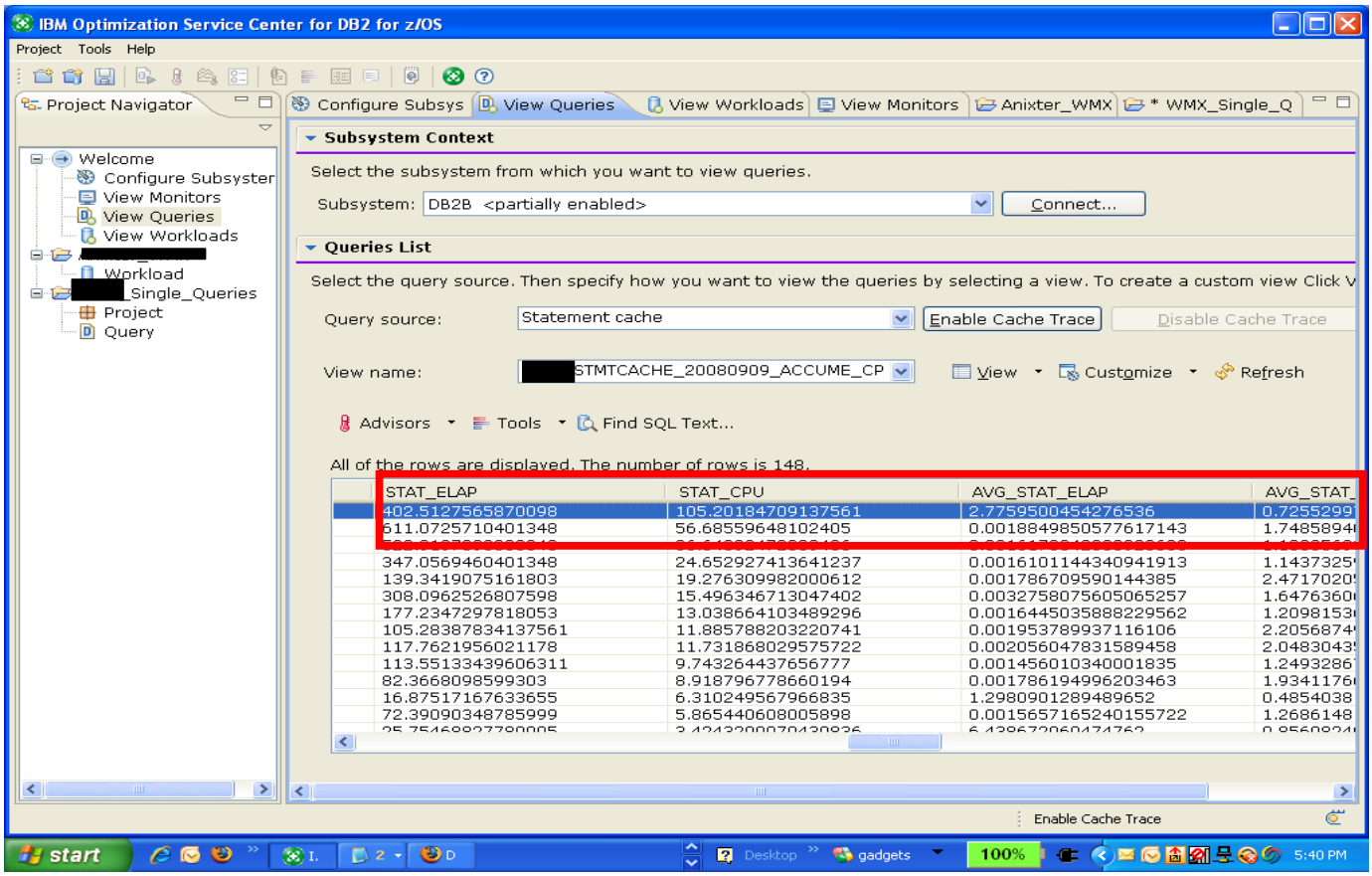
## Fetch Top CPU Consumers

*Client screen shots were removed from cd/web*



***Notes:***

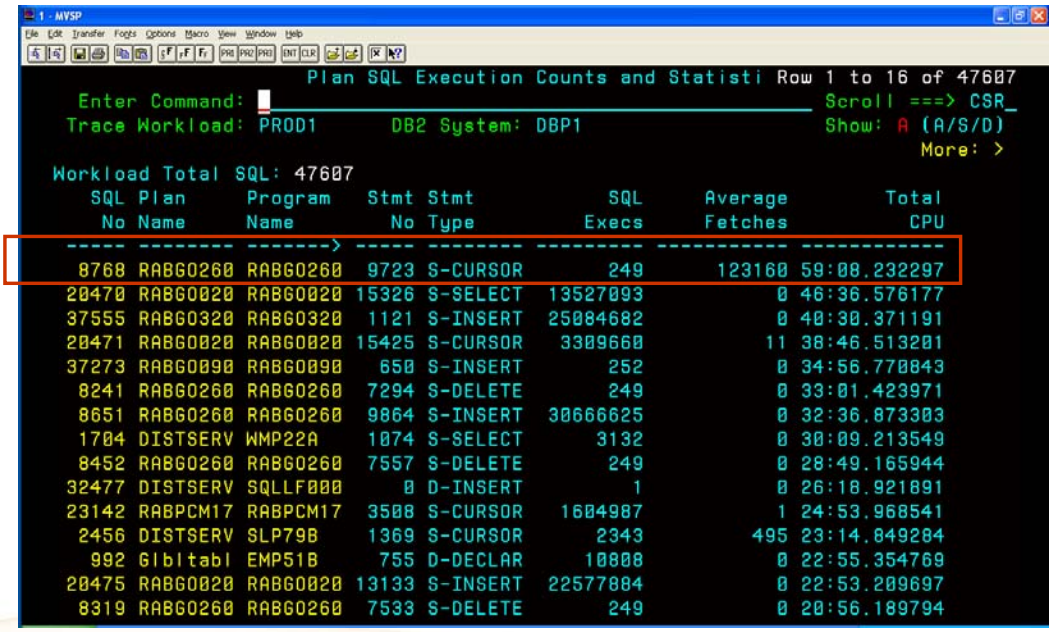
# OSC – Top CPU Consumers




***Notes:***

# EZ-DB2 – Top CPU Consumers

Top CPU consumer



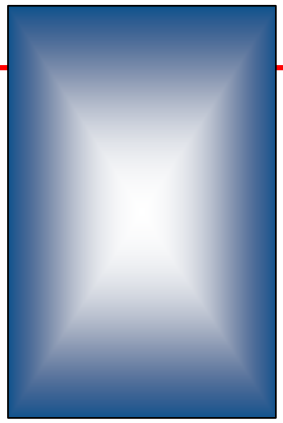
```
                    Plan SQL Execution Counts and Statisti Row 1 to 16 of 47607
 Enter Command:                                                Scroll ===> CSR_
 Trace Workload: PROD1      DB2 System: DBP1                   Show: A (A/S/D)
                                                                      More: >
Workload Total SQL: 47607
  SQL Plan     Program   Stmt Stmt          SQL      Average        Total
   No Name     Name        No Type         Execs     Fetches          CPU
 ----- -------- --------> ----- --------- --------- ----------- ------------
 8768 RABGO260 RABGO260  9723 S-CURSOR        249       123160 59:08.232297
20470 RABGO020 RABGO020 15326 S-SELECT   13527093            0 46:36.576177
37555 RABGO320 RABGO320  1121 S-INSERT   25084682            0 40:30.371191
20471 RABGO020 RABGO020 15425 S-CURSOR    3309660           11 38:46.513201
37273 RABGO090 RABGO090   650 S-INSERT        252            0 34:56.770843
 8241 RABGO260 RABGO260  7294 S-DELETE        249            0 33:01.423971
 8651 RABGO260 RABGO260  9864 S-INSERT   30666625            0 32:36.873303
 1704 DISTSERV WMP22A    1074 S-SELECT       3132            0 30:09.213549
 8452 RABGO260 RABGO260  7557 S-DELETE        249            0 28:49.165944
32477 DISTSERV SQLLF000    0 D-INSERT          1            0 26:18.921891
23142 RABPCM17 RABPCM17  3508 S-CURSOR    1604987            1 24:53.968541
 2456 DISTSERV SLP79B    1369 S-CURSOR       2343          495 23:14.849284
  992 GIbltabl EMP51B     755 D-DECLAR      10808            0 22:55.354769
20475 RABGO020 RABGO020 13133 S-INSERT   22577084            0 22:53.209697
 8319 RABGO260 RABGO260  7533 S-DELETE        249            0 20:56.189794
```



© Sheryl M. Larsen, Inc. 2000-2009

***Notes:***

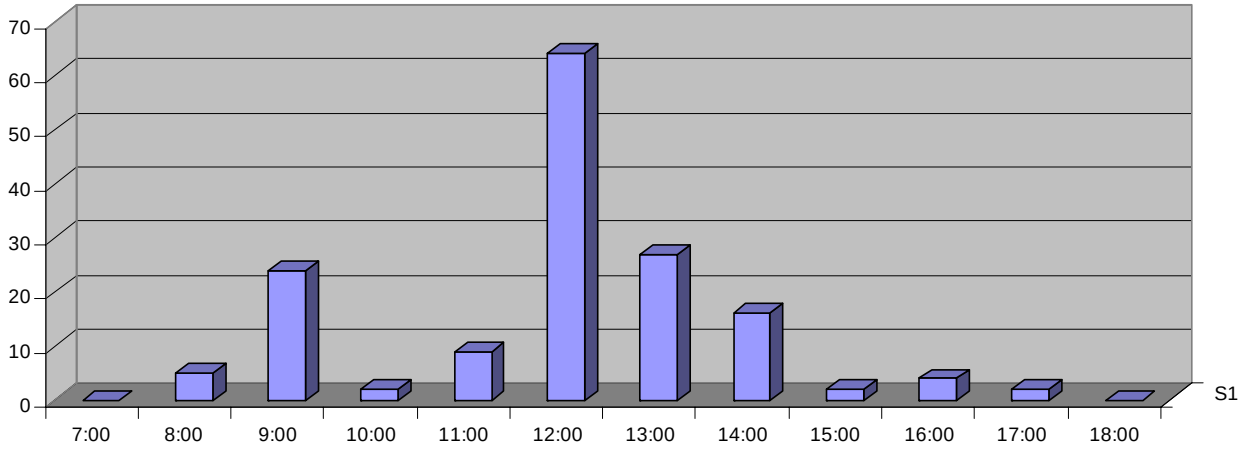
# Detector - RID Pool Heavy Users

| PLANNAME | RID_USED | RID_FLIM |
|---|---|---|
| | 327911 | 155 |
|  | 151 | 13 |
| | 48895 | 0 |
| | 4105 | 0 |
| | 58 | 0 |
| | 1813 | 0 |


© Sheryl M. Larsen, Inc. 2000-2009

***Notes:***

# Detector - RID Pool Failures by Hour for One Program





© Sheryl M. Larsen, Inc. 2000-2009

***Notes:***

# SQL Performance for DB2 – Top Consumers

- Identifies the most expensive SQL statements
- Makes tuning recommendations that improve SQL performance
  - Without costly SQL traces
- Provides access path analysis


© Sheryl M. Larsen, Inc. 2000-2009

***Notes:***

- Uses intelligent analysis and continuous monitoring
- Identifies SQL statements that can hinder application performance.


© Sheryl M. Larsen, Inc. 2000-2009

***Notes:***

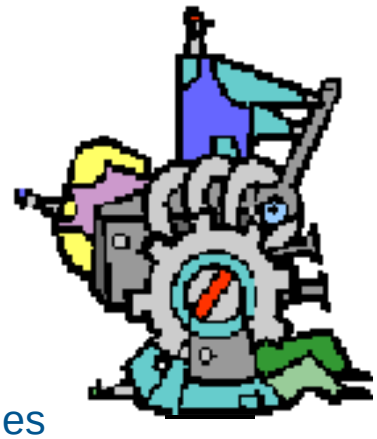
# Step 4 & 5

## Learn Traditional and Extreme Tuning Techniques and Apply When Necessary



***Notes:***

# Tuning Techniques to Apply When Necessary



Learn Traditional Tuning Techniques

- OPTIMIZE FOR n ROWS
- No Ops
- REOPT(ONCE), REOPT(ALWAYS)
- Fake Filtering
- ON 1 = 1
- Index & MQT Design

Experiment with Extreme Tuning Techniques

- DISTINCT Table Expressions
- Correlated Table Expressions
- Odd/old Techniques
- Anti-Joins
- Manual Query Rewrite (X2QBOpt)



© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

# 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


© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

# 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**

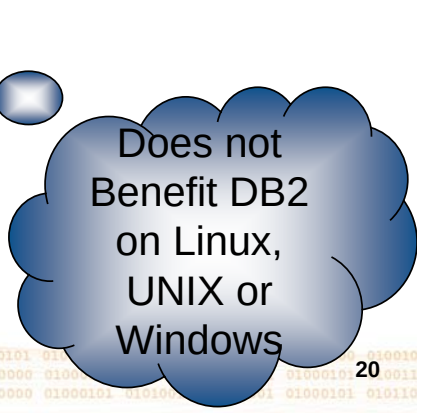


© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

# 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





© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

# 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
- ____unique Rids/16 * .000375 sec = _____ + sort for Region = _____


© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

# 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 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

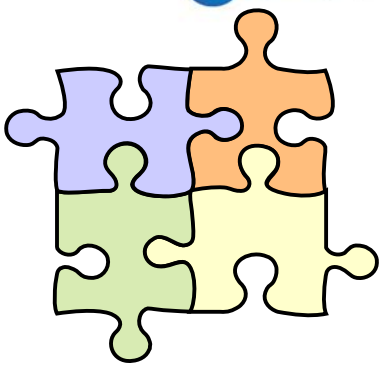
**Should this be Documented?**



© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

# REOPT ONCE/ALWAYS

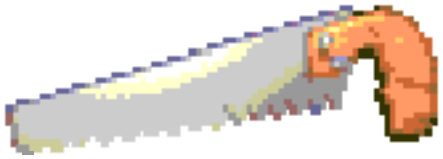


- Package/Plan level option
  - Explain snapshot information for each reoptimizable incremental bind SQL statement is placed in the explain tables at run time. In addition, explain snapshot information is gathered for reoptimizable dynamic SQL statements at run time, even if the CURRENT EXPLAIN SNAPSHOT register is set to NO.
  - QUERYOPT-- *optimization-level* - REOPT ONCE OR REOPT ALWAYS
  - Every query's access path gets re-optimized with host variable contents *known*
    - *ONCE or ALWAYS for every execution*
  - Long running queries/programs easily benefit
  - Web-type transactions will reflect the cost of the re-bind (90% of a full BIND)


© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

# Fake Filtering



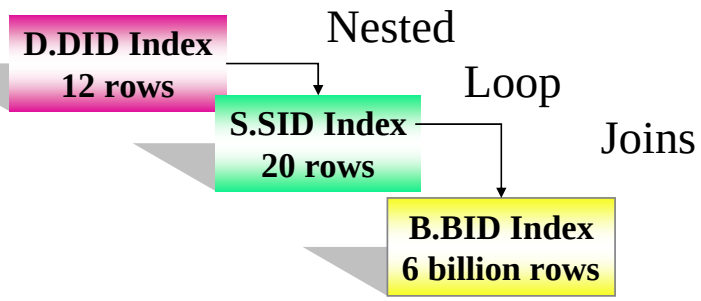
- Fake Predicates
  - To encourage index access
  - To alter table join sequence when nothing else works
  - Works by decreasing filter factor on a certain table
  - The filtering is fake and negligible cost
  - Not effective for dynamic queries if the filter contains :host variables


© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

# 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
```



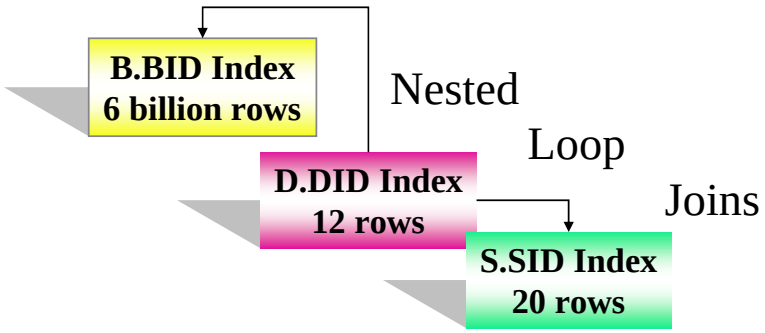

- Large report query with average of 400,000 row range of BID table
- Need to start nested loop with big table
- Tools required


© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

# 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 >= 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
```



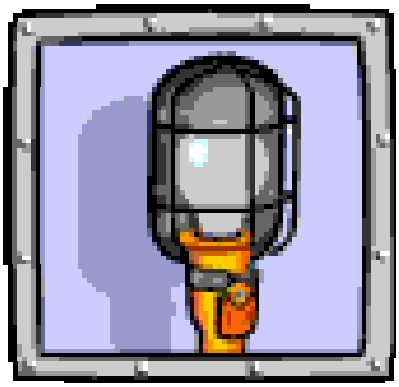

- Keep adding filters until table join sequence changes
- Start with index columns
  - To preserve index-only access
- No limit!

For Dynamic


© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

# 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


© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

# Experiment with Extreme Techniques

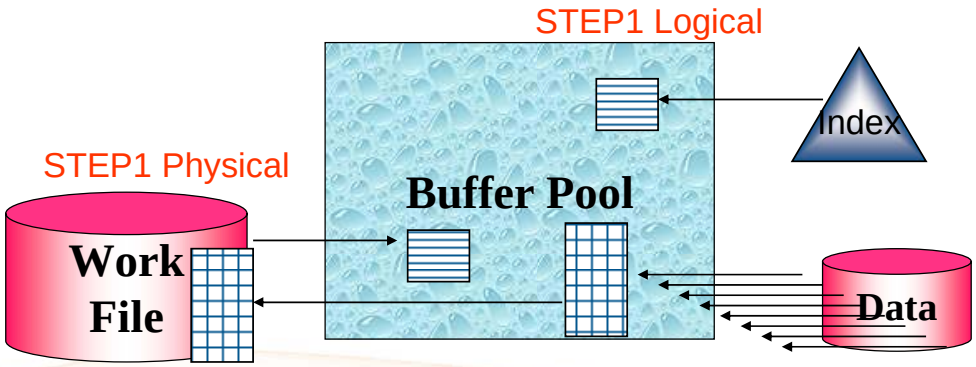
## After Traditional Techniques Fail



***Notes:***

# 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






***Notes:***

# 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
  - 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!


© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

# Typical Join Problem

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

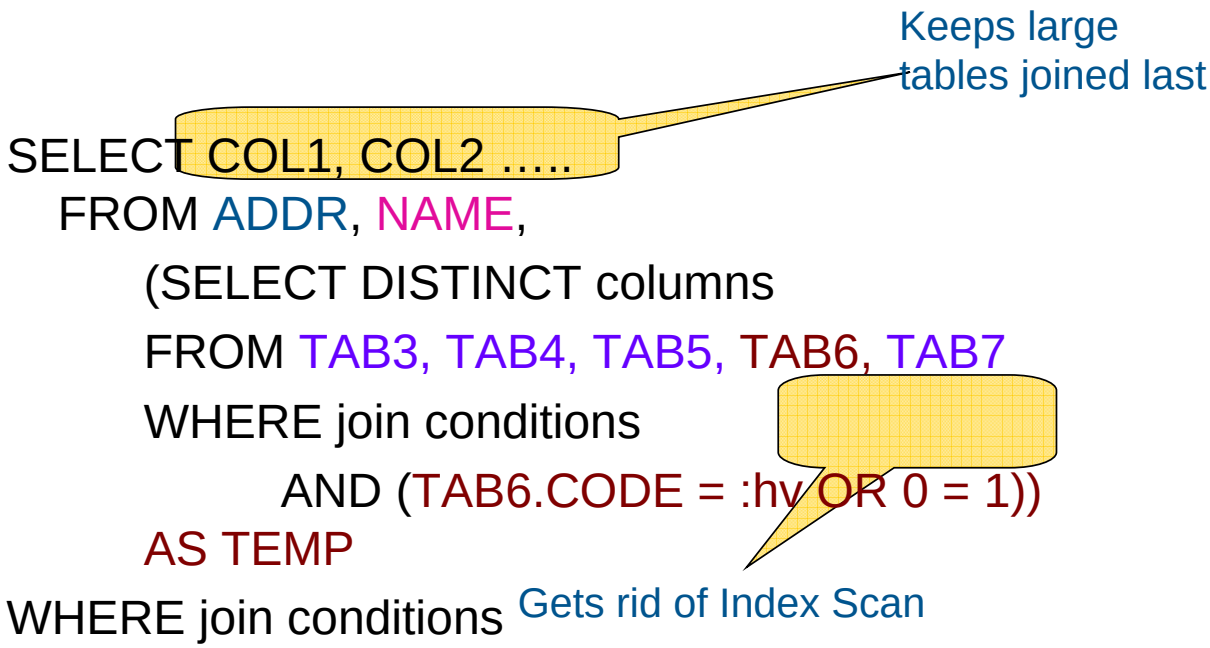
Cardinality 1

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


© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

# Tuning Technique



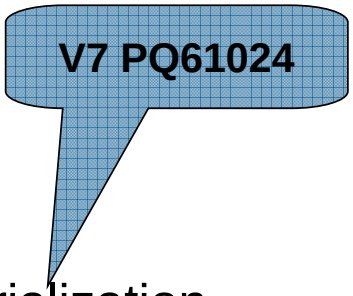


© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

# Correlated Table Expressions

- Advantage over correlated subquery
  - Access both sets of data
- Performance gains
  - Access path changes
  - Encourages Nested Loop Join
  - Reduced processing and/or materialization




```
SELECT  Columns from either set of data
FROM    table1 T1
      ,TABLE(SELECT  T2.STUDENT_YEAR, SUM(C8) AS SUM8, MAX(C9) AS MAX9
                     FROM    table2 T2
                     WHERE  T1.column = T2.column
                     GROUP BY T2.STUDENT_YEAR) AS T2
```



© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

Correlated subqueries that are used to compare detail row information against a unique summary value, have the restriction of top SELECT data access only.

The bottom subquery is used to generate the summary value, however, that summary value is not available to the outer SELECT for display.

One way to make the summary value available for the display is to add a Table Expression to the top SELECT.

Another, more efficient way, is to add a correlated Table Expression and remove the subquery all together.

The TABLE keyword is required for the table expressions that are correlated and the expression must follow (not precede) the referenced table in the FROM clause.

For more examples of Correlated Table Expressions and details on access path changes on nonOS/390, refer to the IDUG North America 2001 presentation given by Daniel L. Luksetich, YL&A titled, “Really CooL SQL: Replacing Programs with Table Expressions” . Session C3 on the CD. Check out his web site at www.db2expert.com

# Correlated Table Expressions

- Get a student's class load, and average & max class load of their peers

```
SELECT  SR.NAME, SS.CLASS_LOAD, P.AVG_CLASS_LOAD, P.MAX_CLASS_LOAD
FROM    SMLU_STUDENT_ROSTER SR
        ,   TABLE(SELECT  SR2.STUDENT_YEAR
                  ,AVG(SR2.NUM_OF_CLASSES) AS AVG_CLASS_LOAD
                  ,MAX(SR2.NUM_OF_CLASSES) AS MAX_CLASS_LOAD
                  FROM    SMLU_STUDENT_ROSTER SR2
                  WHERE  SR.STUDENT_YEAR = SR2.STUDENT_YEAR
                  GROUP BY SR2.STUDENT_YEAR) AS P
LEFT JOIN   TABLE(SELECT SS.SID, COUNT(*) AS  SS.CLASS_LOAD
                  FROM    SMLU_STUDENT_SCHED SS
                  WHERE  SS.SID = SR.SID
                  GROUP BY SS.SID) as SS
ON 1=1
```


© Sheryl M. Larsen, Inc. 2000-2008

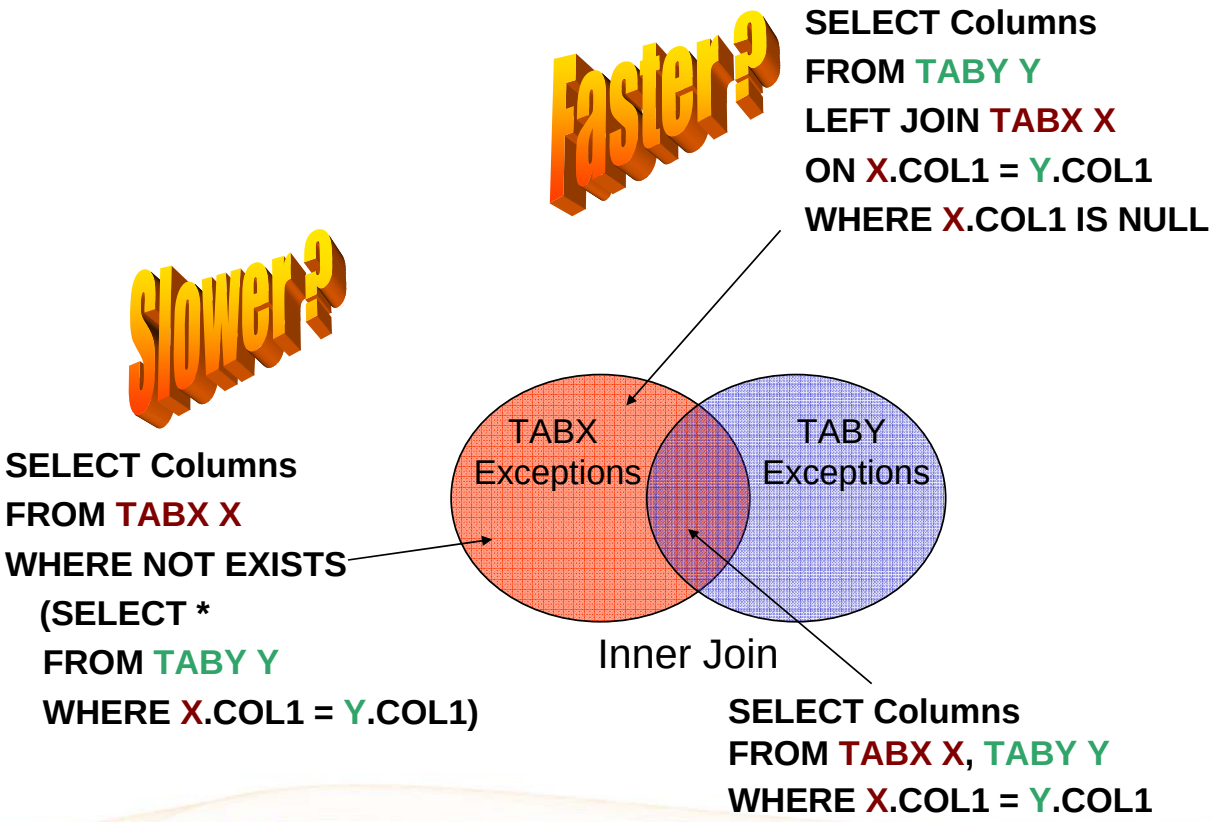
***Notes:***

This example shows one table SR, correlated to two Table Expressions, SS and P.

Each Table expression is calculating summary values that are available for the display.

**V7 PQ61024** this APAR makes correlated table expressions Stage 1 for z/OS

# Anti-Join

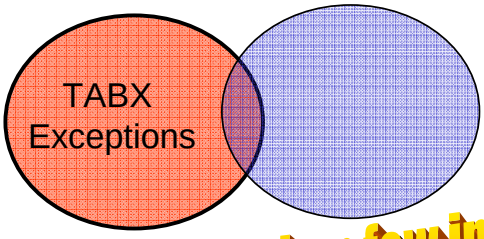





© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

# Anti-Join




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

Even faster when few inner join rows

*Indexable Stage 1*

*Stage 2 when correlated*

Does not Benefit LUW

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



© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

# Manual Query Rewrite

For Extreme Cases
(used on all platforms)



***Notes:***

# A Typical Data Warehouse Query

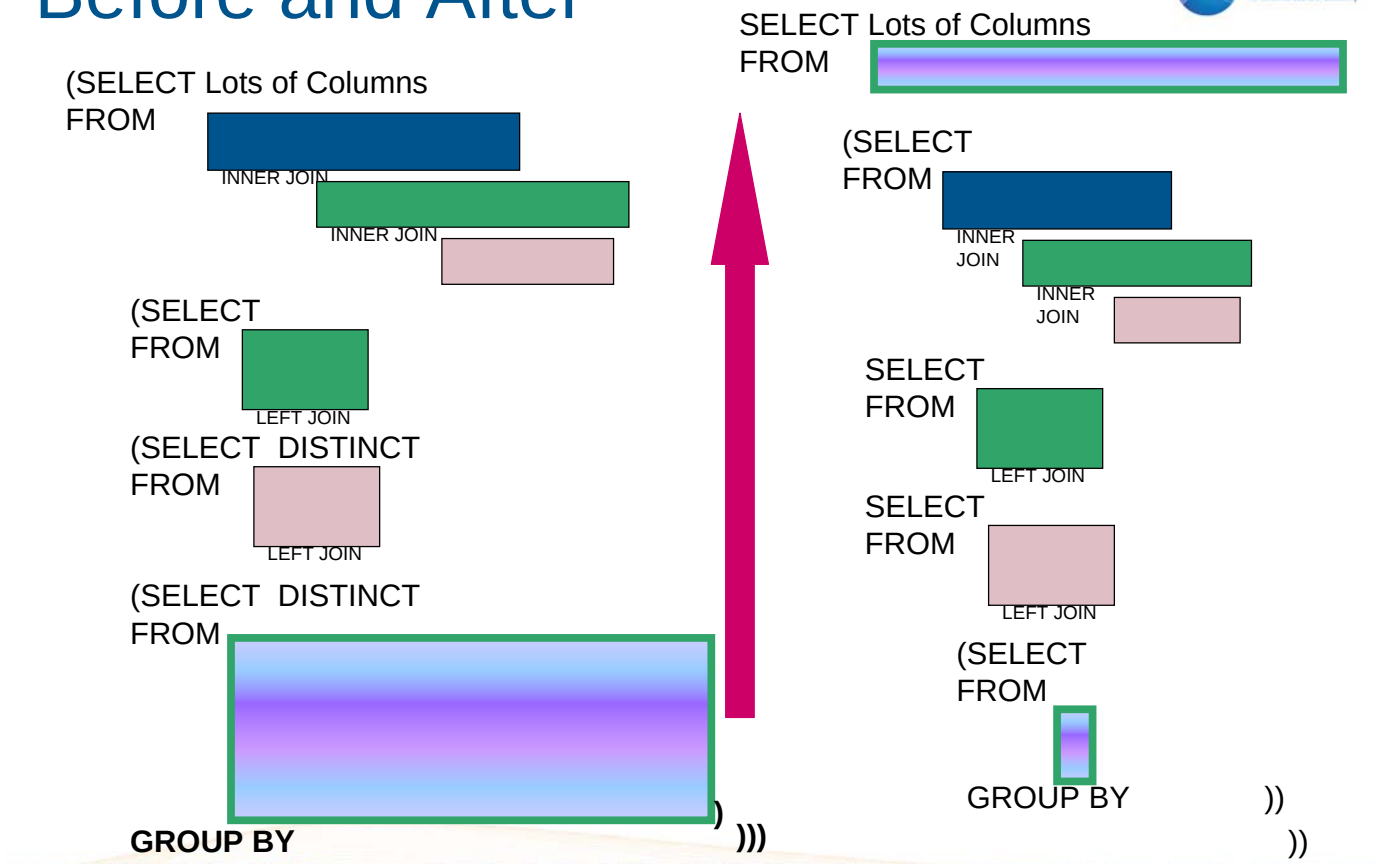
- Initial cost of 16 million timerons
  - WOULD NOT FINISH!
- Multiple DISTINCT table expressions
- Initial join involved all columns and all rows
- The very wide and very deep set was dragged through many more query steps


© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

As queries get more complex, intra query optimization becomes necessary. Cross queryblock knowledge can greatly assist the optimizer in query rewrite. Right now this is a manual rewrite process. One example is a statement that contained multiple UNION ALL subselects with the initial subselect involving and join requesting most rows all columns. This very wide and very deep set was dragged through many query steps.

# Before and After






© Sheryl M. Larsen, Inc. 2000-2008

***Notes:***

The outermost query block, the last step, requested a GROUP BY. This query would not even finish and the timeron value was over 16 million. To manually rewrite this query, the largest block was analyzed for the columns required for GROUP BY, and remaining LEFT JOINs. The initial SELECT list for that really wide table expression was then manually pruned down to SELECT only critical columns, but all rows. This essentially put the subselect on a diet so that the next 5 join steps were much narrower. The final step was then rewritten to join back to the tables to get the remaining SELECT list columns. This did increase the number of times that main set was accessed but the savings from the wide joins more than offset the cost.

Further analysis was done on the GROUP BY columns. It was determined that the only columns needed in the GROUP BY calculation were from the main set. The GROUP BY operation was moved from the outer most step and pushed into the first table expression. This greatly reduced the cost of the GROUP BY operation since it did not involve many columns.

# Tuning Technique – X2QBOpt

- Extreme Cross Query Block Optimization
- Identify and pre-qualify the core set of data and only select the keys early on
- Once all the steps are complete, go back and get the remaining columns
- Referred to as “Group By Push Down”
  - Keeping it thin through the DB2 engine
- Brought cost down to 270,000 timerons
  - Query now finishes in 4 minutes!


© Sheryl M. Larsen, Inc. 2000-2008

***Notes:*** The query now finishes and the timerons were reduced to .27 million. This technique of keeping the query thin through the DB2 engine has to be accomplished through manual query rewrite for now. Start by identifying the core set of data and only select the keys and grouping columns early on. Once all the step are complete, go back and get the remaining columns.

# Summary

- Many Things Impact Query Speed
- It is Easy to Change the Optimizer's Mind
- The Trick … is to Know WHEN to use:

Traditional Tuning Techniques

- OPTIMIZE FOR n ROWS
- No Ops
- REOPT(VARS)
- Fake Filtering
- Index & MQT Redesign

Extreme Tuning Techniques

- DISTINCT Table Expressions
- Correlated Table Expressions
- OR 0 = 1
- Anti-Join Technique
- X2QBOpt

Redesign Performance Structures

- Indexes
- MQTs



© Sheryl M. Larsen, Inc. 2000-2008

***Notes:*** Investments in Education & Proactive Tuning Payoff through:

- Increased processing throughput
- Reduced administration costs
- Improved agility

**Performance Topics for Application Tuners**

Sheryl M. Larsen
Sheryl M. Larsen, Inc.
smlsql@comcast.net
smlsql.com
(630) 399-3330


***Notes:***