

# The White Papers

## **Top Ten SQL Performance Tips**

**By Sheryl M. Larsen**

**Contents**

---

**Top Ten SQL Performance Tips ..... 3**

*Introduction* ..... 3

*Overview* ..... 3

*Tip #1: Verify that the appropriate statistics are provided* ..... 4

*Tip #2: Promote Stage 2 & Stage 1 Predicates if Possible* ..... 5

*Tip #3: SELECT only the columns needed* ..... 7

*Tip #4: SELECT only the rows needed* ..... 7

*Tip #5: Use constants and literals if the values will not change in the next 3 years (for static queries)* ..... 7

*Tip #6: Make numeric and date data types match* ..... 8

*Tip #7: Sequence filtering from most restrictive to least restrictive by table, by predicate type* ..... 8

*Tip #8: Prune SELECT lists* ..... 9

*Tip #9: Limit Result Sets with Known Ends* ..... 10

*Tip #10: Analyze and Tune Access Paths* ..... 10

*The Solution* ..... 11

*Solution for Tip #1: Verify that the appropriate statistics are provided* ..... 12

*Solution for Tip #2: Promote Stage 2 & Stage 1 Predicates if Possible* ..... 14

*Solution for Tip #3: SELECT only the columns needed* ..... 17

*Solution for Tip #4: SELECT only the rows needed* ..... 18

*Solution for Tip #5: Use constants and literals if the values will not change in the next 3 years (for static queries)* ..... 19

*Solution for Tip #6: Make numeric and date data types match* ..... 20

*Solution for Tip #7: Sequence filtering from most restrictive to least restrictive by table, by predicate type* ..... 21

*Solution for Tip #8: Prune SELECT lists* ..... 22

*Solution for Tip #9: Limit Result Sets with Known Ends* ..... 23

*Solution for Tip #10: Analyze and Tune Access Paths* ..... 24

*Summary* ..... 24

*About the Author* ..... 24

# Top Ten SQL Performance Tips

By Sheryl M. Larsen

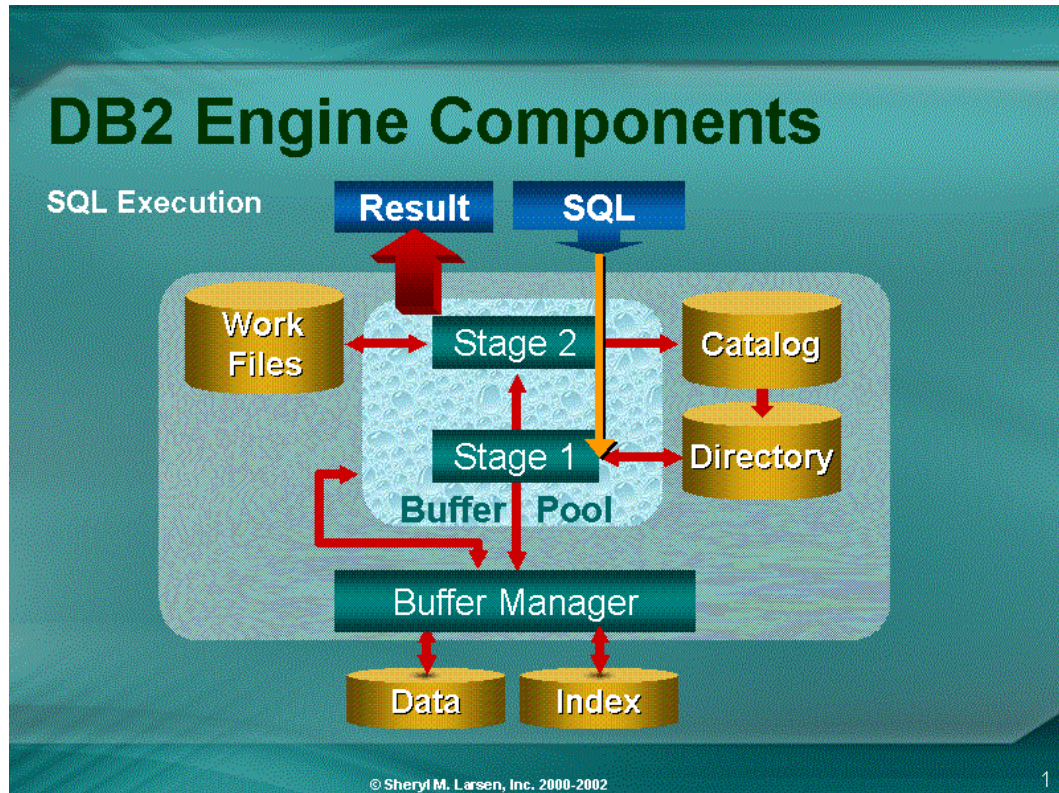
---

## Introduction

Structured Query Language (SQL) is the blessing and the curse of relational DBMSs. Since any data retrieved from a relational database requires SQL, this topic is relevant to anybody accessing a relational database; from the end user to the developer to the DBA. When efficient SQL is used, the results are highly scalable, flexible, and manageable systems. When inefficient SQL is used, response times are lengthy, programs run longer and application outages can occur. Considering that a typical database system spends 90% of the processing time just retrieving data from the database, it's easy to see how important it is to ensure your SQL is as efficient as possible. Checking for common SQL problems such as 'SELECT \* FROM' is just the tip of the iceberg. In this paper, we'll explore other common SQL problems that are just as easy to fix. Bear in mind, a SQL statement can be written with many variations and result in the same data being returned — there are no "Good" SELECT statements or "Bad" SELECT statements, just the "Appropriate for the Requirement." Each relational DBMS has its own way of optimizing and executing SELECT statements. Therefore, each DBMS has its own Top SELECT Performance Tips. This paper will focus on DB2 for OS/390 and z/OS, with examples and overview from Quest Software's Quest Central for DB2 product.

## Overview

Seventeen years ago this list of tips would have been much longer and contained antidotes to the smallest SELECT scenarios. Each new release of DB2 brings thousands of lines of new code that expand the intelligence of the optimization, query rewrite, and query execution. For example, over the years a component called Data Manager, commonly referred to as 'Stage 1 processing,' has increased its filtering capacity one hundred fold. Another component is the Relational Data Server, commonly referred to as 'Stage 2 processing,' and its main function is query rewrite and optimization. Another key component is the DB2 optimizer, which determines the access path used to retrieve the data based on the SQL presented. The DB2 optimizer improves with each release of DB2, taking into account additional statistics in the DB2 catalog, and providing new and improved access paths. These components and many more, shown in Figure 1, depict how DB2 processes requests for data or SQL. This is where the following DB2 SQL Performance tips are derived from.



**Figure 1: DB2 Engine and some related components**

In this white paper, we will review some of the more common SQL problems; however, there are many more SQL performance tips beyond what's described in this paper. Also just like all guidelines, each of these have some notable exceptions.

**Tip #1: Verify that the appropriate statistics are provided**

The most important resource to the DB2 optimizer, other than the SELECT statement itself, is the statistics found within the DB2 catalog. The optimizer uses these statistics to base many of its decisions. The main reason the DB2 optimizer may choose a non-optimal access path for a query is due to either invalid or missing the statistics. The DB2 optimizer uses the following catalog statistics:

<b>DB2 Catalog Table</b>	<b>Columns considered by optimizer</b>
<b>SYSIBM.SYSCOLDIST</b>	CARDF
	COLGROUPCOLNO
	COLVALUE
	FREQUENCYF
	NUMCOLUMNS
	TYPE
<b>SYSIBM.SYSCOLSTATS</b>	COLCARD
	HIGHKEY
	HIGH2KEY
	LOWKEY
	LOW2KEY
<b>SYSIBM.SYSCOLUMNS</b>	COLCARDF
	HIGH2KEY
	LOW2KEY
<b>SYSIBM.SYSINDEXES</b>	CLUSTERING
	FIRSTKEYCARDF
	NLEAF
	NLEVELS
	CLUSTERATIO
	CLUSTERRATIOF
<b>SYSIBM.SYSINDEXPART</b>	LIMITKEY
<b>SYSTEM.SYSTABLES</b>	CARDF
	EDPROC
	NPAGESF
	PCTPAGES
	PCTROWCOMP
<b>SYSIBM.SYSTABLESPACE</b>	NACTIVEF
<b>SYSIBM.SYSTABSTATS</b>	NPAGES

**Figure 2: Columns recognized by the DB2 optimizer and used to determine the access path**

Often, executing the RUNSTATS command (which is used to update the DB2 catalog statistics) gets overlooked, particularly in a busy production environment. To minimize the impact of executing the RUNSTATS command, consider using the sampling technique. Sampling with even as little as 10% is ample. In addition to the statistics updated by the RUNSTATS command, DB2 gives you the ability to update an additional 1,000 entries for non-uniform distribution statistics. Beware that each entry added increases BIND time for all queries referencing that column.

How do you know if you are missing the statistics? You can manually execute queries against the catalog or use tools that provide this functionality. Currently, the DB2 optimizer does not externalize warnings for missing the statistics.

## **Tip #2: Promote Stage 2 & Stage 1 Predicates if Possible**

Either the Stage 1 Data Manager or the Stage 2 Relational Data Server will process every query. There are tremendous performance benefits to be gained when your query can be processed as Stage 1 rather than Stage 2. The predicates used to qualify your query will determine whether your query can be processed in Stage 1. In addition, each predicate is

evaluated to determine whether that predicate is eligible for index access. There are some predicates that can never be processed as Stage 1 or never eligible for an index. It's important to understand if your query is indexable and can be processed as Stage 1. The following are the documented Stage 1 or Sargable predicates:

Predicate Type	Indexable	Stage 1
<b>INDEXABLE STAGE 1</b>		
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,...COLn) IN (non subq)	Y	Y
<b>NON-INDEXABLE STAGE 1</b>		
COL <> value	N	Y
COL <> noncol expr	N	Y
COL IS NOT NULL	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

**Figure 3: Table used to determine predicate eligibility**

There are a few more predicates that are not documented as Stage 1, because they are not *always* Stage 1. Join table sequence and query rewrite can also affect which stage a predicate can be filtered. Let's examine some example queries to see the effect of rewriting your SQL.

**Example 1: Value BETWEEN COL1 AND COL1**

Any predicate type that is not identified as Stage 1 is Stage 2. This predicate as written is a Stage 2 predicate. However, a rewrite can promote this query to Indexable Stage 1.

**Value >= COL1 AND value <= COL2**

This means that the optimizer may choose to use the predicates in a matching index access against multiple indexes. Without the rewrite, the predicate remains as Stage 2.

### **Example 2: COL3 NOT IN (K,S,T)**

Non-indexable Stage 1 predicates should also be rewritten, if possible. For example, the above condition is Stage 1, but not indexable. The list of values in parentheses identifies what COL3 cannot be equal to. To determine the feasibility of the rewrite, identify the list of what COL3 can be equal to. The longer and more volatile the list, the less feasible this is. If the opposite of (K, S, T) is less than 200 fairly static values, the rewrite is worth the extra typing. This promotes the Stage 1 condition to Indexable Stage 1, which provides the optimizer with another matching index choice. Even if a supporting index is not available at BIND time, the rewrite will ensure the query will be eligible for index access, should an index be created in the future. Once an index is created that incorporates COL3, a rebind of the transaction may possibly gain matching index access, where the old predicate would have no impact on rebind.

### **Tip #3: SELECT only the columns needed**

Every column that is selected has to be individually handed back to the calling program, unless there is a precise match to the entire DCLGEN definition. This may lean you towards requesting all columns, however, the real harm occurs when a sort is required. Every SELECTed column, with the sorting columns repeated, makes up the width of the sort work file wider. The wider and longer the file, the slower the sort is. For example, 100,000 four-byte rows can be sorted in approximately one second. Only 10,000 fifty-byte rows can be sorted in the same time. Actual times will vary depending on hardware.

The exception to the rule, “Disallow SELECT \*”, would be when several processes require different parts of a table’s row. By combining the transactions, the whole row is retrieved once, and then the parts are uniquely processed.

### **Tip #4: SELECT only the rows needed**

The less rows retrieved, the faster the query will run. Each qualifying row has to make it through the long journey from storage, through the buffer pool, Stage 1, Stage 2, possible sort and translations, and then deliver the result set to the calling program. The database manager should do all data filtering; it is very wasteful to retrieve a row, test that row in the program code and then filter out that row. Disallowing program filtering is a hard rule to enforce. Developers can choose to use program code to perform all or some data manipulation or they can choose SQL. Typically there is a mix. The tell tale sign that filtering can be pushed into the DB2 engine is a program code resembling:

```
IF TABLE-COL4 > :VALUE  
  GET NEXT RESULT ROW
```

### **Tip #5: Use constants and literals if the values will not change in the next 3 years (for static queries)**

The DB2 Optimizer has the full use of all the non-uniform distribution statistics, and the various domain range values for any column statistics provided when no host variables are detected in a predicate, (WHERE COL5 > ‘X’). The purpose of a host variable is to make a transaction adaptable to a changing variable; this is most often used when a user is required to enter this value. A host variable eliminates the need to rebind a program each time this variable changes. This extensibility comes at a cost of the optimizer accuracy. As soon as host variables are detected, (WHERE COL5 > :hv5), the optimizer uses the following chart to estimate the filter factors, instead of using the catalog statistics:

COLCARDF	FACTOR FOR <, <=, >, >=	FACTOR FOR LIKE AND BETWEEN
>= 100,000,000	1/10,000	3/100,000
>= 10,000,000	1/3,000	1/10,000
>= 1,000,000	1/1,000	3/10,000
>= 100,000	1/300	1/1,000
>= 10,000	1/100	3/1,000
>= 1,000	1/30	1/100
>= 100	1/10	3/100
>= 0	1/3	1/10

**Figure 4: Filter Factors**

The higher the cardinality of the column, the lower the predicated filter factor (fraction of rows predicted to remain). Most of the time the estimate leans the optimizer towards an appropriate access path. Sometimes, however, the predicated filter factor is far from reality. This is when access path tuning is usually necessary.

### Tip #6: Make numeric and date data types match

Stage 1 processing has been very strict in prior releases about processing predicate compares where the datatype lengths vary. Prior to DB2 v7, this mismatch led to the predicate being demoted to stage 2 processing. However, a new feature in DB2 v7 allows numeric datatypes to be manually cast to avoid this stage 2 demotion.

**ON DECIMAL(A.INTCOL, 7, 0) = B.DECICOL  
ON A.INTCOL = INTEGER(B.DECICOL)**

If both columns are indexed, cast the column belonging to the larger result set. If only one column is indexed, cast the partner. A rebind is necessary to receive the promotion to Stage 1.

### Tip #7: Sequence filtering from most restrictive to least restrictive by table, by predicate type

When writing a SQL statement with multiple predicates, determine the predicate that will filter out the most data from the result set and place that predicate at the start of the list. By sequencing your predicates in this manner, the subsequent predicates will have less data to filter.

The DB2 optimizer by default will categorize your predicate and process that predicate in the condition order listed below. However, if your query presents multiple predicates that fall into the same category, these predicates will be executed in the order that they are written. This is why it is important to sequence your predicates, placing the predicate with the most filtering at the top of the sequence. Eventually query rewrite will take care of this in future releases, but today this is something to be aware of when writing your queries.

Category	Condition
<b>Stage 1 and Indexable</b>	
	=, IN (Single Value)
	Range conditions
	LIKE
	Noncorrelated subqueries
<b>Stage 1 and On Index (index screening)</b>	
	=, IN (Single Value)
	Range conditions
	LIKE
<b>Stage 1 on data page rows that are ineligible for prior categories</b>	
	=, IN (Single Value)
	Range conditions
	LIKE
<b>Stage 2 on either index or data that are ineligible for prior categories</b>	
	=, IN (Single Value)
	Range conditions
	LIKE
	Noncorrelated subqueries
	Correlated subqueries

**Figure 5: Predicate Filtering Sequence**

The order of predicate filtering is mainly dependent on the join sequence, join method, and index selection. The order the predicates physically appear in the statement only come into play when there is a tie with one of the above listed categories. For example, the following statement has a tie in the category range conditions:

```
WHERE      A.COL2 = 'abracadabra'
          AND  A.COL4 > 999
          AND  A.COL3 > :hvcol3
          AND  A.COL5 LIKE '%SON'
```

The most restrictive condition should be listed first, so that extra processing of the second condition can be eliminated.

## Tip #8: Prune SELECT lists

Every column that is SELECTed consumes resources for processing. There are several areas that can be examined to determine if column selection is really necessary.

### Example 1:

```
WHERE (COL8 = 'X')
```

If a SELECT contains a predicate where a column is equal to one value, that column should not have to be retrieved for each row, the value will always be 'X'.

**Example 2: SELECT COLA,COLB ,COLC ORDER BY COLC**

DB2 no longer requires selection of a column simply to do a sort. Therefore in this example, COLC does not require selection if the end user does not need that value. Remove items from the SELECT list to prevent unnecessary processing. It is no longer required to SELECT columns used in the ORDER BY or GROUP BY clauses.

**Tip #9: Limit Result Sets with Known Ends**

The FETCH FIRST *n* ROWS ONLY clause should be used if there are a known, maximum number of rows that will be FETCHed from a result set. This clause limits the number of rows returned from a result set by invoking a *fast implicit close*. Pages are quickly released in the buffer pool when the *nth* result row has been processed. The OPTIMIZE FOR *n* ROWS clause does not invoke a *fast implicit close* and will keep locking and fetching until the cursor is implicitly or explicitly closed. In contrast, FETCH FIRST *n* ROWS ONLY will not allow the *n+1* row to be FETCHed and results in an SQLCODE = 100. Both clauses optimize the same if *n* is the same.

Existence checking should be handled using:

```
SELECT 1
        INTO :hv1
FROM TABLEX
        WHERE ..... existence check ....
        FETCH FIRST 1 ROW ONLY
```

**Tip #10: Analyze and Tune Access Paths**

Use EXPLAIN or tools that interpret EXPLAIN output, to verify that the access path is appropriate for the required processing. Check the access path of the each query by binding against production statistics in a production-like subsystem. Bufferpool, RID pool, sort pool, and LOCKMAX thresholds should also resemble the production environment. Oversized RID pools in the test environment will mask RID pool failures in production. RID pool failures can occur during List Prefetch, Multiple Index Access, and Hybrid Join Type N access paths. RID pool failures result in a full table scan.

Tune queries using a technique that will withstand future smarter optimization and query rewrite. Typical query tuning may include using one or more of the following techniques:

- OPTIMIZE FOR *n* ROWS
- FETCH FIRST *n* ROWS ONLY
- No Operation (+0, -0, /1, \*1, CONCAT ` `)
- ON 1=1
- Bogus Predicates
- Table expressions with DISTINCT
- REOPT(VARS)
- Index Optimization

All these techniques impact access path selection. Compare estimated costs of multiple scenarios to verify the success of the tuning effort.

The goal of a tuning effort should be refined access paths and optimized index design. This is an ongoing task that should be proactively initiated when any of the following occur:

- Increases in the number of DB2 objects
- Fluctuations in the size of DB2 objects
- Increases in the use of dynamic SQL
- Fluctuations of transaction rates
- Migrations

## The Solution

Quest Central for DB2 is an integrated console providing core functionality a DBA needs to perform their daily tasks of Database Administration, Space Management, SQL Tuning and Analysis, and Performance Diagnostic Monitoring. Quest Central for DB2 was written by DB2 software experts and provides rich functionality utilizing a graphical user interface. The product supports DB2 databases running on the mainframe, Unix, Linux, and Windows. No longer are DB2 customers required to maintain and utilize separate tools for their mainframe and distributed DB2 systems.

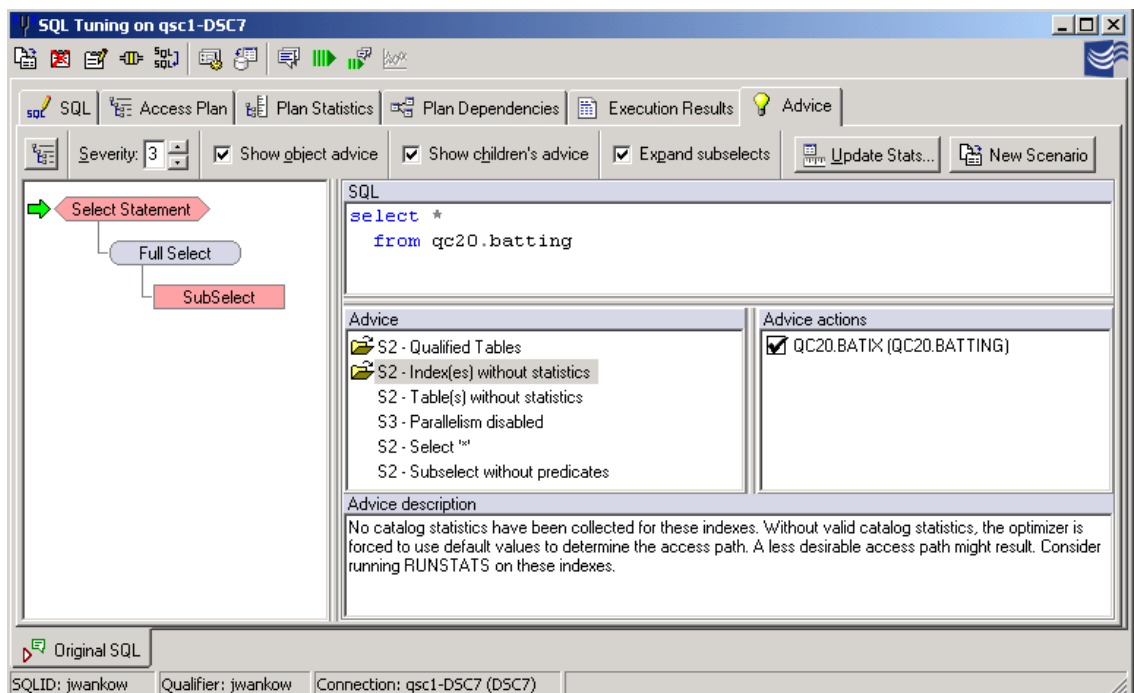
The SQL Tuning component of Quest Central provides the most complete SQL tuning environment for DB2 on the market. This environment consists of:

1. Tuning Lab – a facility where a single SQL statement can be modified multiple times, through use of scenarios. These scenarios can then be compared to immediately determine which SQL statement provided the most efficient access path.
2. Compare – immediately see the effect your modifications have on the performance of your SQL. By comparing multiple scenarios, you can see the effect on the CPU, elapsed time, I/O and many more statistics. Additionally, a compare of the data will ensure your SQL statement is returning the same subset of data.
3. Advice – the advice provided by the SQL tuning component will detect all of the conditions specified in this white paper and more. In addition, the SQL Tuning component will even rewrite the SQL if applicable into a new scenario, incorporating the advice chosen.
4. Access Path and Associated Statistics – All statistics applicable to the DB2 access path are displayed, in context to the SQL. This takes the guesswork out of trying to understand why a particular access plan was chosen.

Quest Central for DB2's robust functionality can detect the above SQL tuning tips and many more. The remainder of this white paper will demonstrate the strength and in-depth knowledge built right into Quest Central to enhance not only your SQL, but assist with overall database performance. Each tuning tip described above is contained right within Quest Central.

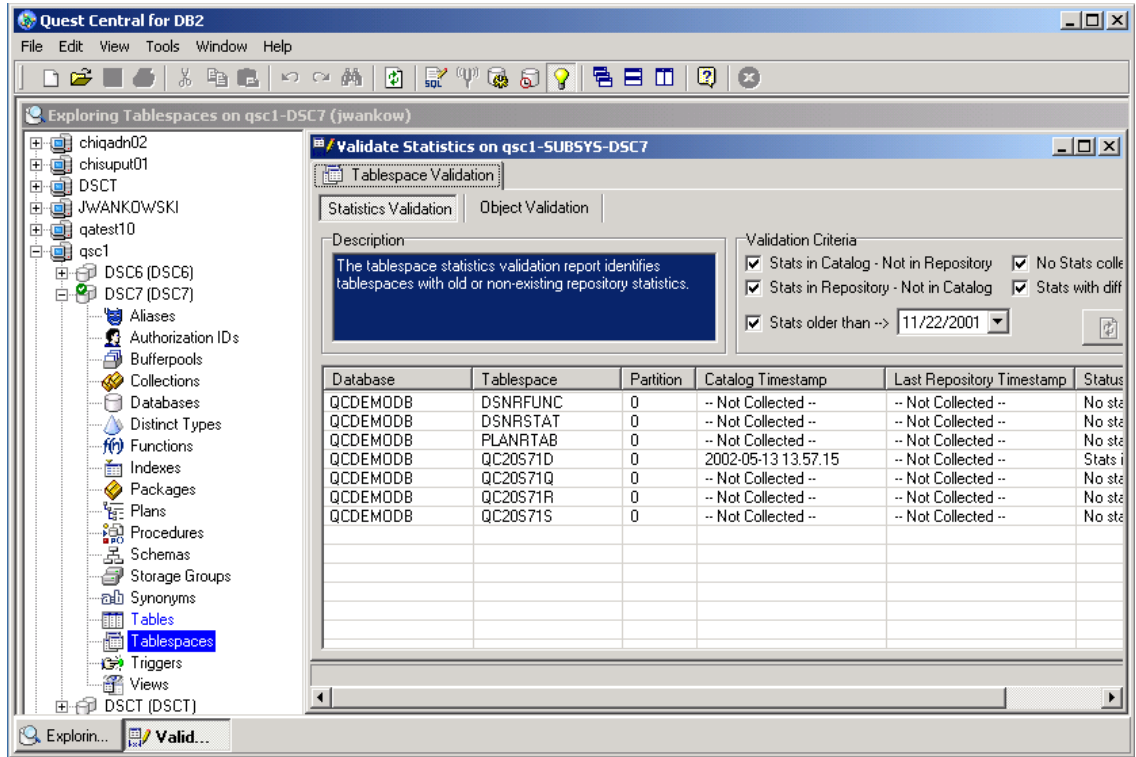
## Solution for Tip #1: Verify that the appropriate statistics are provided

Once a SQL statement has been explained within Quest Central, the advice tab provides a full set of advice, including the ability to detect when RUNSTATS are missing. Quest Central always follows this type of advice up with immediate resolution. Associated with each piece of advice is an accompanying ‘advice action.’ This advice action will look to rectify a problem detected by the advice. This will result in either a new scenario being opened with rewritten SQL or a script being generated to facilitate an object resolution. In this example, the advice indicates that statistics are missing and the accompanying advice action will build a script containing the RUNSTATS command for any objects chosen in the advice action window.



**Figure 6: The SQL Tuning component identifies all objects missing statistics and can generate the necessary command to update statistics on all objects chosen.**

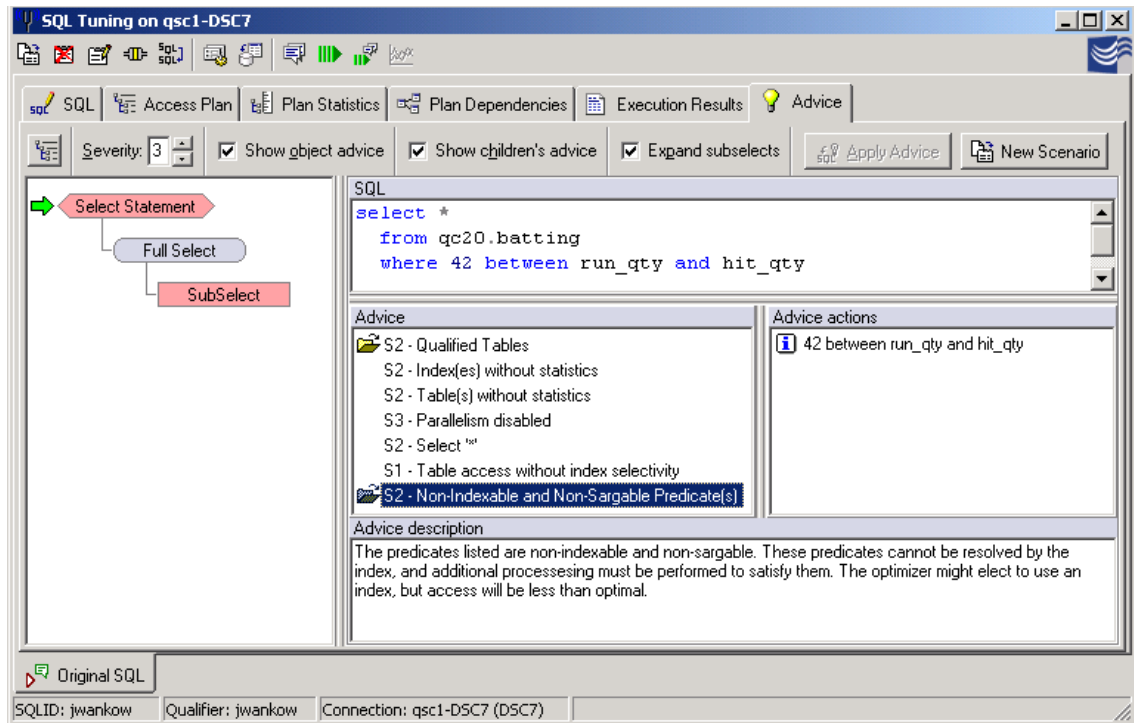
Additionally, Quest Central Space Management can automate the collection, maintenance and validation of the statistics at the tablespace, table and index levels. The following example shows the validation report for statistics of all the tablespaces in the database.



**Figure 7: Quest Central provides an easy to use graphical interface to facilitate the automation of the RUNSTATS process.**

## Solution for Tip #2: Promote Stage 2 & Stage 1 Predicates if Possible

The SQL Tuning component will list all predicates and indicate whether those predicates are ‘Sargable’ or ‘Non-Sargable’. Additionally, each predicate will be checked to determine if it is eligible for index access. This advice alone can solve response time issues and require little effort in terms of rewriting the predicate. In the examples below, a query was identified as non-sargable and non-indexable (Stage 2). This original query was written with a between predicate. A new scenario was opened and the predicate was rewritten using a greater than, less than. The compare identified the impact this query rewrite had on performance.



The screenshot shows the SQL Tuning interface for a query on qsc1-D5C7. The query is:

```
select *
from qc20.batting
where 42 between run_qty and hit_qty
```

The interface displays the following advice:

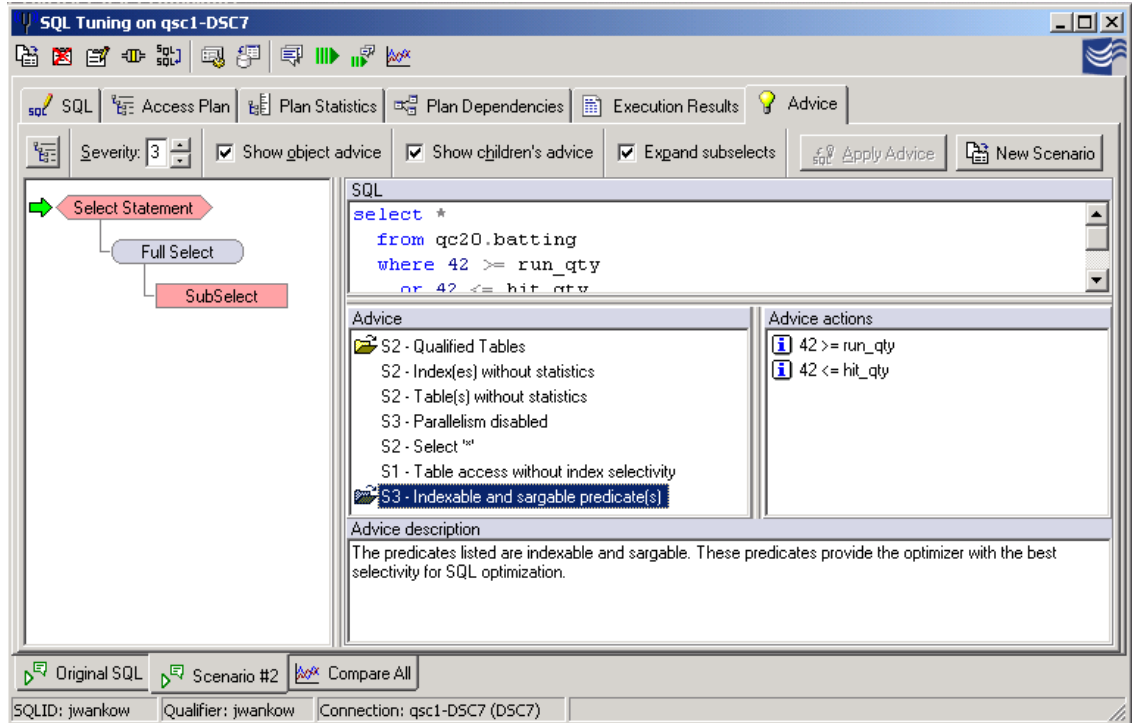
- S2 - Qualified Tables
- S2 - Index(es) without statistics
- S2 - Table(s) without statistics
- S3 - Parallelism disabled
- S2 - Select \*
- S1 - Table access without index selectivity
- S2 - Non-Indexable and Non-Sargable Predicate(s)**

The advice actions list the predicate: 42 between run\_qty and hit\_qty.

The advice description states: "The predicates listed are non-indexable and non-sargable. These predicates cannot be resolved by the index, and additional processing must be performed to satisfy them. The optimizer might elect to use an index, but access will be less than optimal."

Figure 8: Query that is non-indexable and non-sargable(stage 2)

A new scenario is created and the query is rewritten using a  $\geq$  and a  $\leq$  on the column values. Note the predicate is now indexable and sargable. Remember from the information above, the predicate will now be processed by the Data Manager (Stage 1), potentially reducing the response time of this query.



**Figure 9: Query is indexable and sargable (stage 1)**

The compare facility can then be used to compare the performance of the between vs. the  $\diamond$  to verify that indeed that the  $\diamond$  is more efficient, resulting in a dramatic reduction in elapsed time.

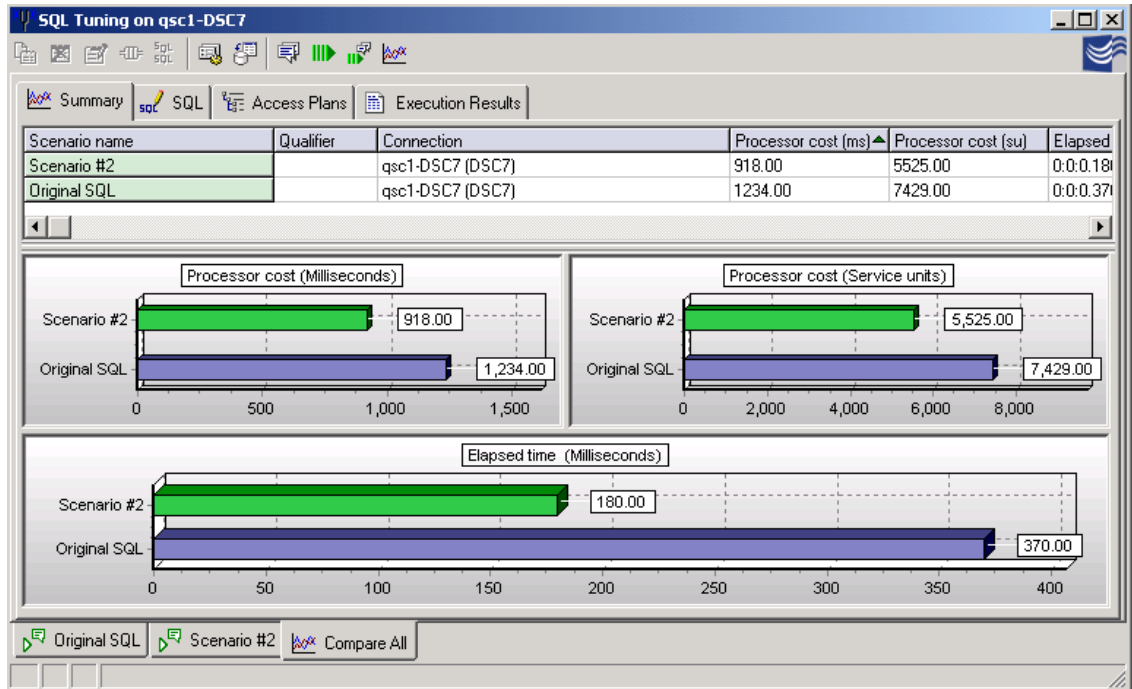
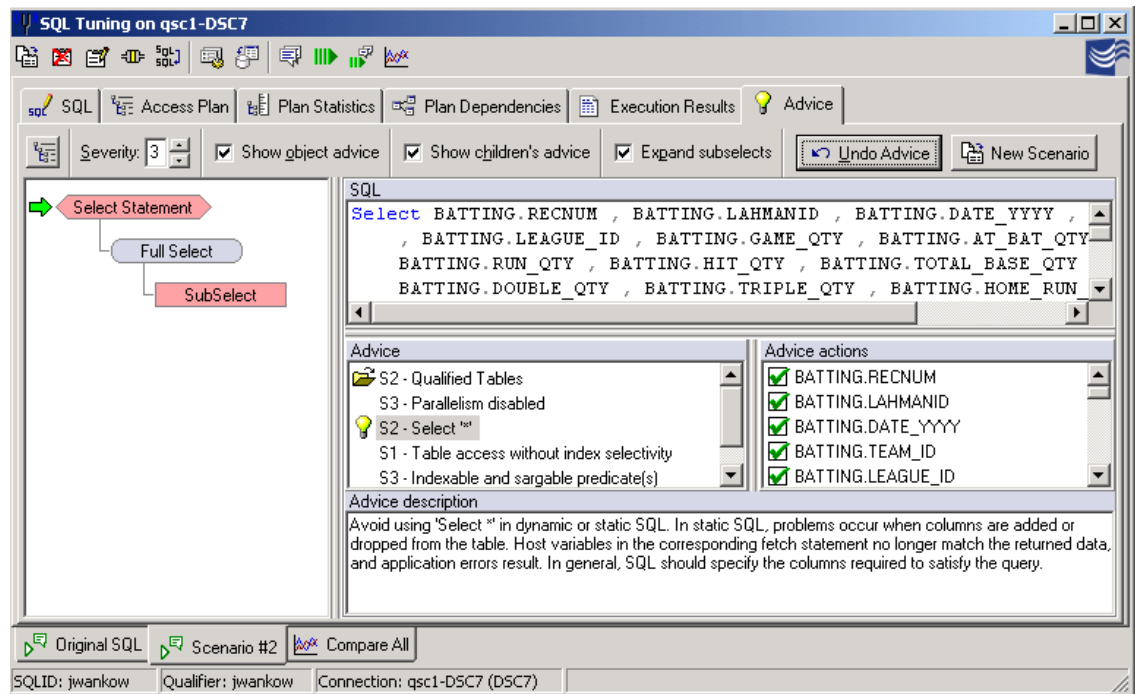


Figure 10: Elapsed time cut in half

### Solution for Tip #3: **SELECT** only the columns needed

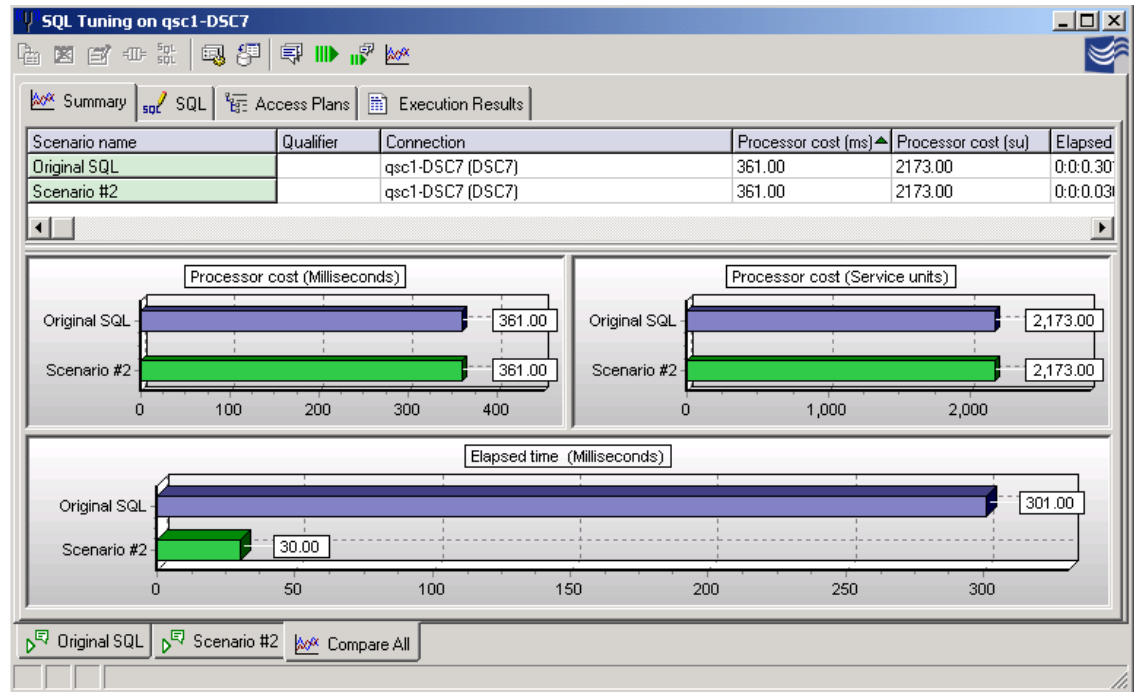
The SQL Tuning feature not only advises against using the **SELECT \***, but also provides a timesaving feature where the product can automatically rewrite your SQL. The advice and the accompanying advice actions will provide the ability to rewrite your SQL by simply checking the desired columns and selecting the ‘apply advice’ button. SQL Tuning will replace the ‘\*’ with the columns selected.



**Figure 11:** The ‘apply advice’ feature will rewrite the SQL taking into account the advice actions chosen.

## Solution for Tip #4: SELECT only the rows needed

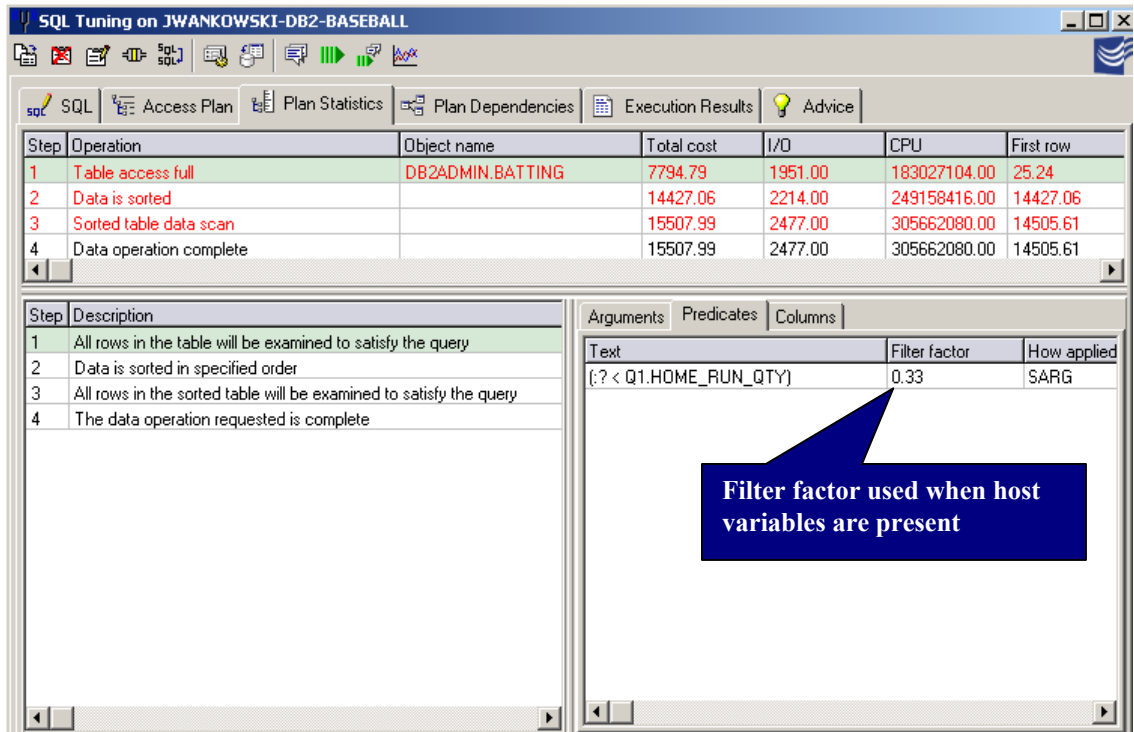
The fewer rows retrieved, the faster the query will run. With Quest Central you can compare your original SQL against the same SQL statement selecting fewer rows. Using multiple scenarios and utilizing the compare feature, comparing those scenarios immediately displays the performance impact of making this change. In the following example a join of two tables results in a significant result set. By adding a 'Fetch First 1 Row Only', the execution times were reduced significantly.



**Figure 12: A select statement was modified to reduce the number of rows, the compare identifies the performance benefits**

## Solution for Tip #5: Use constants and literals if the values will not change in the next 3 years (for static queries)

In this example let's run a test against DB2 running on a Win2K platform. When using host variables, the DB2 optimizer cannot predict the value used for the predicate filtering. Without this value, DB2 will default and use the default filter factors listed above. Quest Central SQL Tuning will always display the filter factor to help understand how many rows will be filtered.



The screenshot shows the Quest Central SQL Tuning interface for a query on DB2ADMIN.BATTING. The execution results table shows the following data:

Step	Operation	Object name	Total cost	I/O	CPU	First row
1	Table access full	DB2ADMIN.BATTING	7794.79	1951.00	183027104.00	25.24
2	Data is sorted		14427.06	2214.00	249158416.00	14427.06
3	Sorted table data scan		15507.99	2477.00	305662080.00	14505.61
4	Data operation complete		15507.99	2477.00	305662080.00	14505.61

The interface also shows a description of the steps and a detailed view of the predicate filter factor:

Step	Description
1	All rows in the table will be examined to satisfy the query
2	Data is sorted in specified order
3	All rows in the sorted table will be examined to satisfy the query
4	The data operation requested is complete

Text	Filter factor	How applied
(:? < Q1.HOME_RUN_QTY)	0.33	SARG

A blue callout box points to the filter factor value of 0.33, stating: "Filter factor used when host variables are present".

Figure 13: Quest Central displays filter factor for every predicate.

## Solution for Tip #6: Make numeric and date data types match

This particular SQL problem can be the most subtle and difficult problem to detect, particularly when host variables are used. The explain may indicate that index access will be used, but upon execution the query will resort to a tablespace scan. This is often the case when predicates are comparing the values of two items and those two items contain a mismatch in the data type. Quest Central SQL Tuning will identify this situation in the advice section. In addition, the Database Administration component can alter the column, even if that alter is not supported by native DDL (by unloading the data, dropping the table, reloading the data, and rebuilding dependencies).

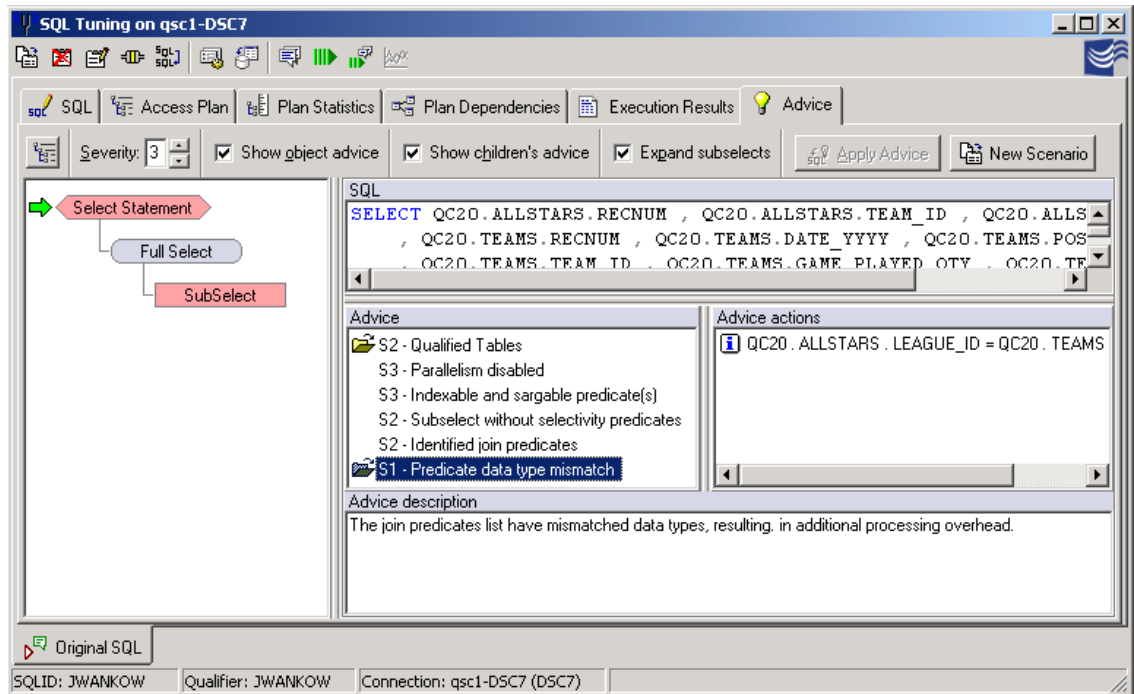


Figure 14: Quest Central will identify predicate mismatches.

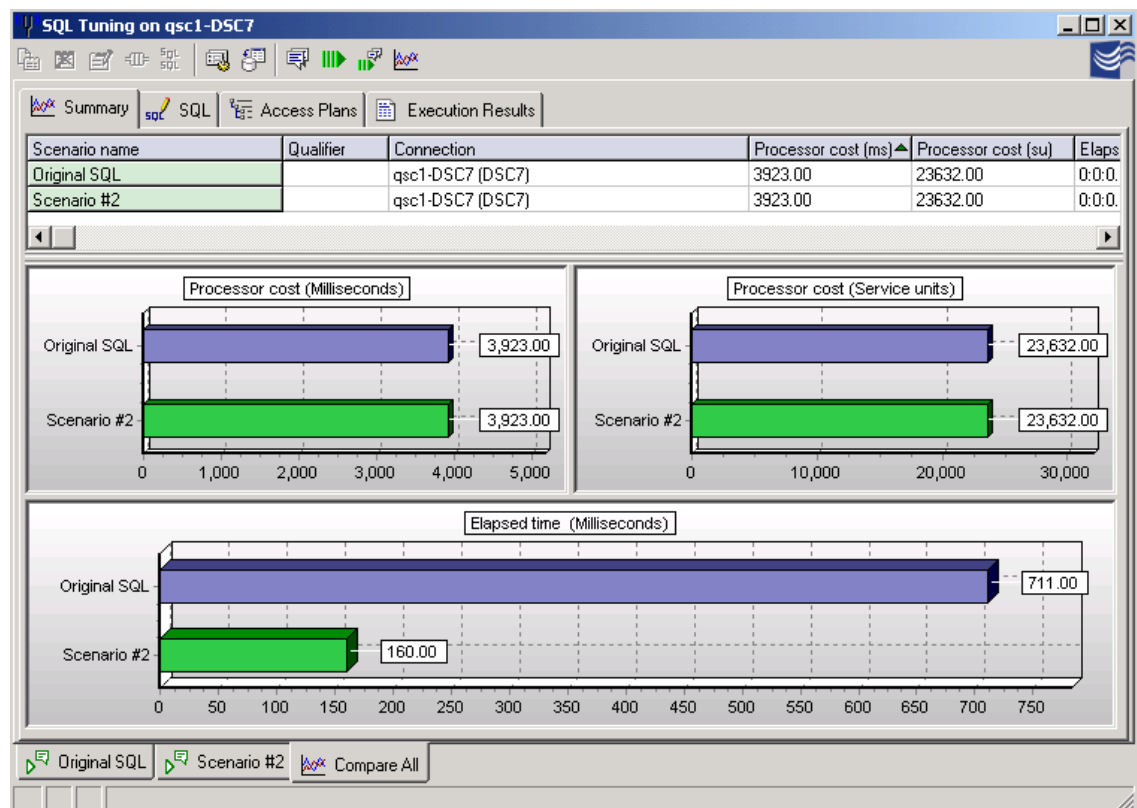
## Solution for Tip #7: Sequence filtering from most restrictive to least restrictive by table, by predicate type

SQL Tuning is designed to allow testing of these types of conditions to determine the appropriate sequence.

Example 1:

```
SELECT * FROM batting
WHERE run_qty > 2
AND hit_qty > 10
```

This SQL statement was brought into the tool and placed in the original SQL tab. The column `hit_qty` provides better filtering than the `run_qty` predicate. A new scenario was created and the predicates were sequenced with `hit_qty` predicate listed first.



**Figure 15: Comparing the different predicate orders verifies the performance improvement.**

## Solution for Tip #8: Prune SELECT lists

Selecting more columns than necessary incurs cost when returning that data back to the end user. By using the scenario feature of SQL Tuning, you can modify the original SQL statement to remove unnecessary columns and perform a cost comparison to determine the impact of removing the additional columns. In the example below, a SQL statement was modified to reduce the number of columns being returned. The savings between the original SQL statement and the modified statement was about 60%. This type of savings can have a huge impact on large databases.

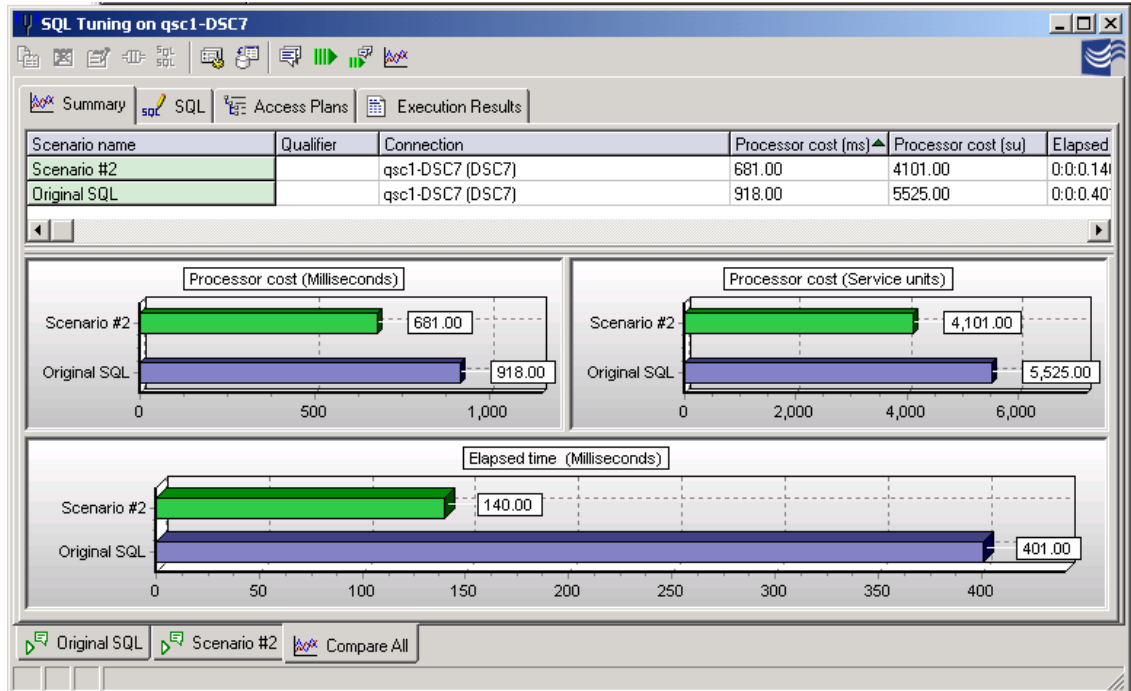
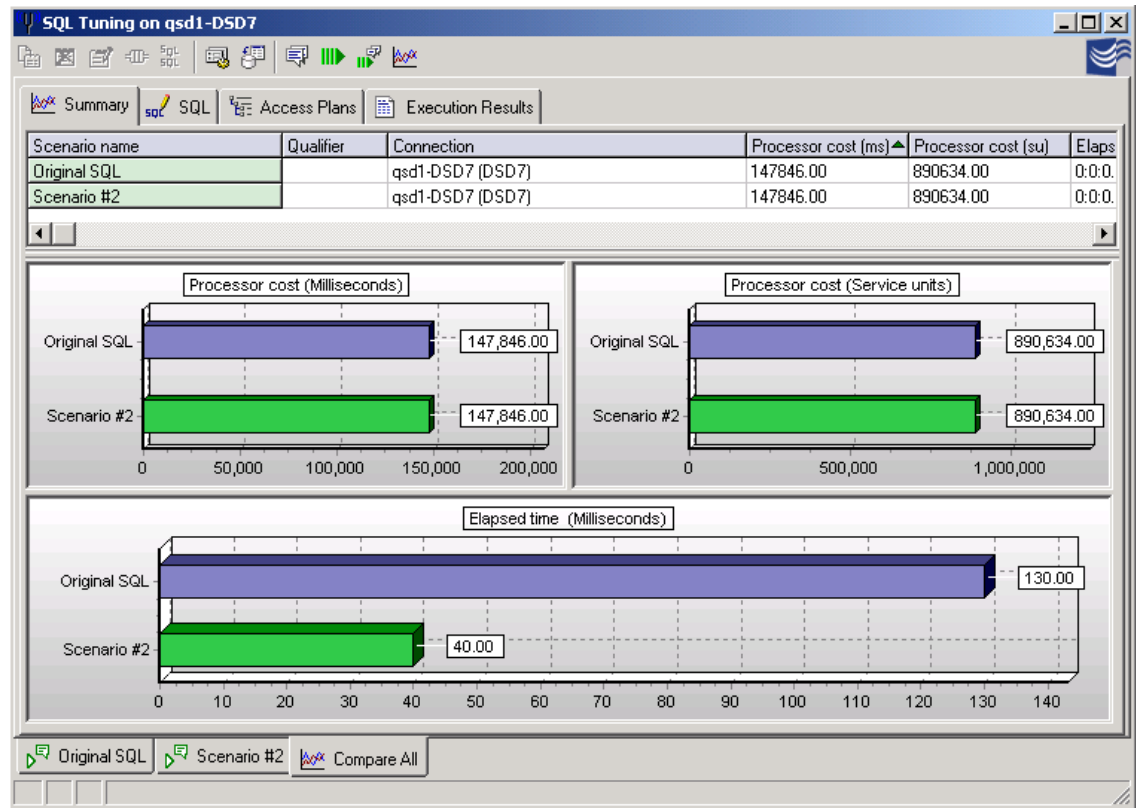


Figure 17: Compare of a *SELECT \** and a *SELECT* of specific columns

## Solution for Tip #9: Limit Result Sets with Known Ends

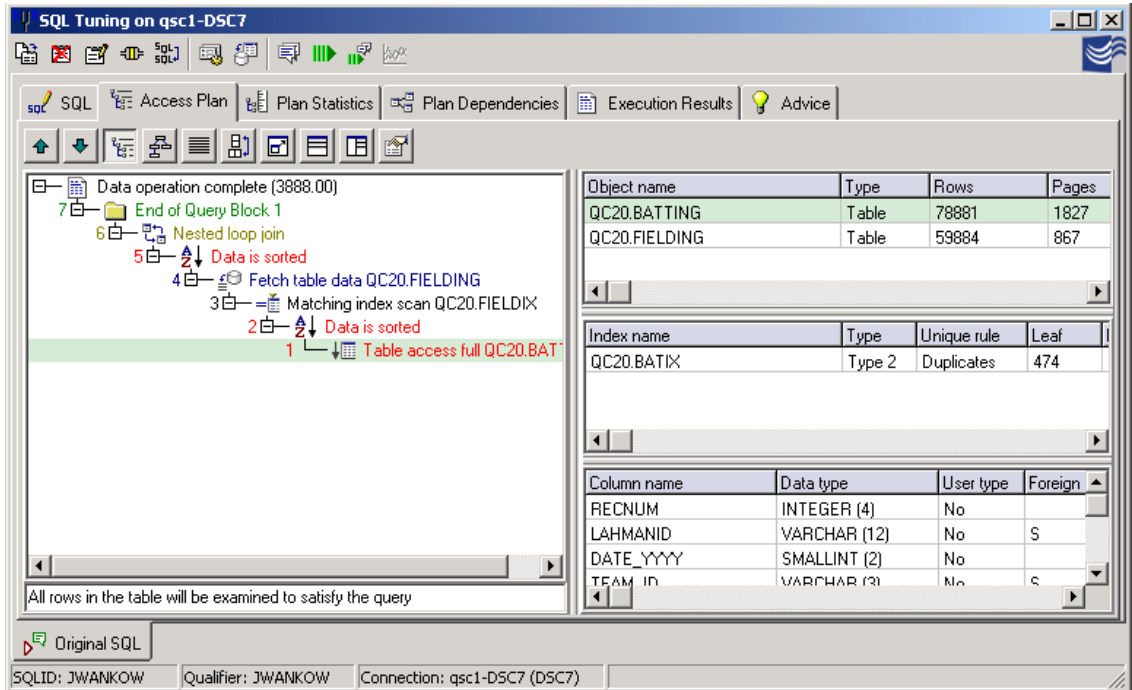
To determine the impact adding the 'FETCH FIRST n ROWS ONLY' clause will have on your SQL statement, you can bring your original SQL statement into the SQL Tuning component. Create a new scenario and include the 'FETCH FIRST n ROWS ONLY' clause. A compare will indicate the cost savings gained by adding this clause.



**Figure 17: Comparison of the same SQL statement with the 'Fetch for 1 row only' clause included**

## Solution for Tip #10: Analyze and Tune Access Paths

The Access Path tab found within SQL Tuning provides a comprehensive display of your access path. The access path automatically highlights the first step to be executed and the ‘next step’ button will highlight the next step, walking you through each step of the access plan.



The screenshot displays the 'SQL Tuning on qsc1-D5C7' window. The 'Access Plan' tab is active, showing a tree view of the query execution plan. The steps are:

1. Table access full QC20.BAT
2. Data is sorted
3. Matching index scan QC20.FIELDIX
4. Fetch table data QC20.FIELDING
5. Data is sorted
6. Nested loop join
7. End of Query Block 1

The 'Object name' table shows:

Object name	Type	Rows	Pages
QC20.BATTING	Table	78881	1827
QC20.FIELDING	Table	59884	867

The 'Index name' table shows:

Index name	Type	Unique rule	Leaf
QC20.BATIX	Type 2	Duplicates	474

The 'Column name' table shows:

Column name	Data type	User type	Foreign
RECNUM	INTEGER (4)	No	
LAHMANID	VARCHAR (12)	No	S
DATE_YYYY	SMALLINT (2)	No	
TEAM_ID	VARCHAR (3)	No	S

At the bottom, the status bar shows: 'SQLID: JWANKOW Qualifier: JWANKOW Connection: qsc1-D5C7 (D5C7)'.

**Figure 19: Quest Central's comprehensive display provides access path and associated objects highlighting tables, indexes, and columns involved in the access path step.**

## Summary

This was the minimum list of checks that a SELECT statement should go through prior to being allowed in any production DB2 for OS/390 and z/OS Version 7 environment. They are derived from current knowledge of the components that process a query within DB2. This list will change as each release of DB2 becomes more sophisticated. Tools can assist in checking adherence to many of these recommendations.

## About the Author

Sheryl Larsen is an internationally recognized researcher, consultant and lecturer, specializing in DB2 and is known for her extensive expertise in SQL. Sheryl has over 17 years experience in DB2, has published many articles, several popular DB2 posters, and co-authored a book, *DB2 Answers*, Osborne-McGraw-Hill, 1999. She was voted into the IDUG “Speaker Hall of Fame” in 2001 and was the Executive Editor of the IDUG Solutions Journal magazine 1997-2000. Currently, she is President of the Midwest Database Users Group ([www.mwdug.org](http://www.mwdug.org)), a member of IBM's DB2 Gold Consultants program, and owns Sheryl M. Larsen, Inc. ([www.smlsql.com](http://www.smlsql.com)), a firm specializing in Advanced SQL Consulting and Education.