Because this material can be relatively complex and because many of the concepts discussed here build apond one another, I am going to ask that you hold all questions until the end. We will also be available after the presentation to answer questions as we are available at anytime during the conference.

The optimizer plays a very important role in a relational database management system. Although SQL is used to specify what data to retrieve, the language itself does not explain how to retrieve the data. Usually there are a variety of ways to retrieve data and its the responsibility of the optimizer to determine the best execution strategy. This presentation will explain the new features available in SQLBase 6.0 to understanding the SQLBase optimizer.

This presentation is divided into five major sections.

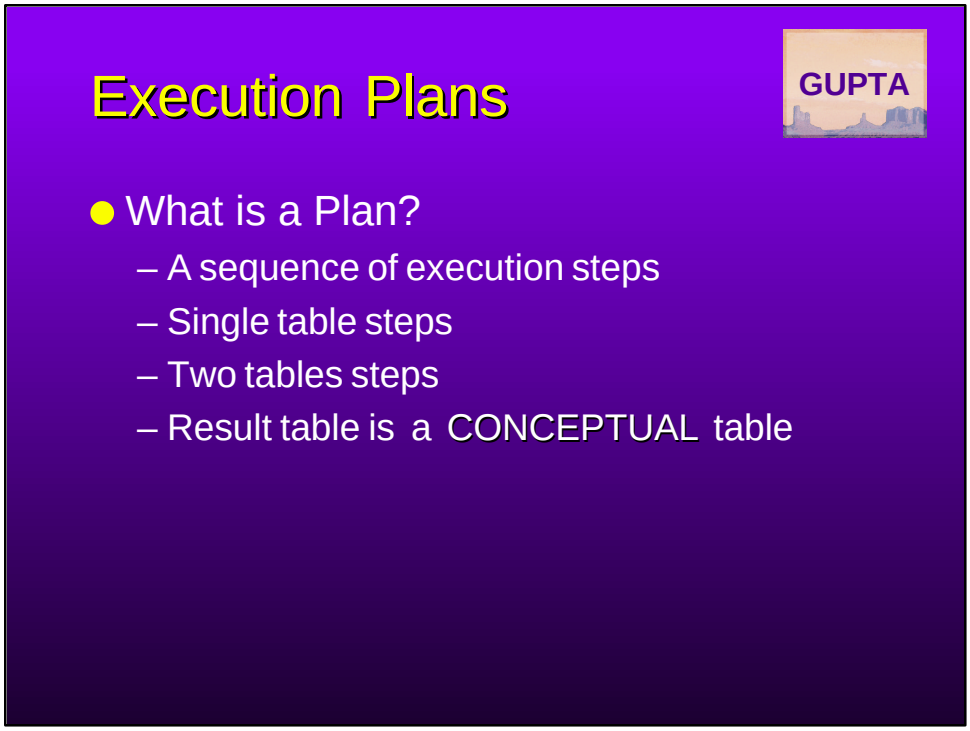
In the first section, I will explain what is an execution plan and by way of example show you how to read the new SQLBase 6.0 execution plans. This should give you a clear understanding of what the optimizer is doing with your SQL statements.

Secondly, I will discuss the SQLBase optimizer costing model. This will help you understand why a certain plan was chosen.

Next, I will discuss the new optimizer statistical columns added to the system catalog for SQLBase 6.0.

Then, I'll explain how you can modify these statistics for your own devious purposes.

And finally, I discuss how you can influence a new execution plan by modifying the optimizer statistics.

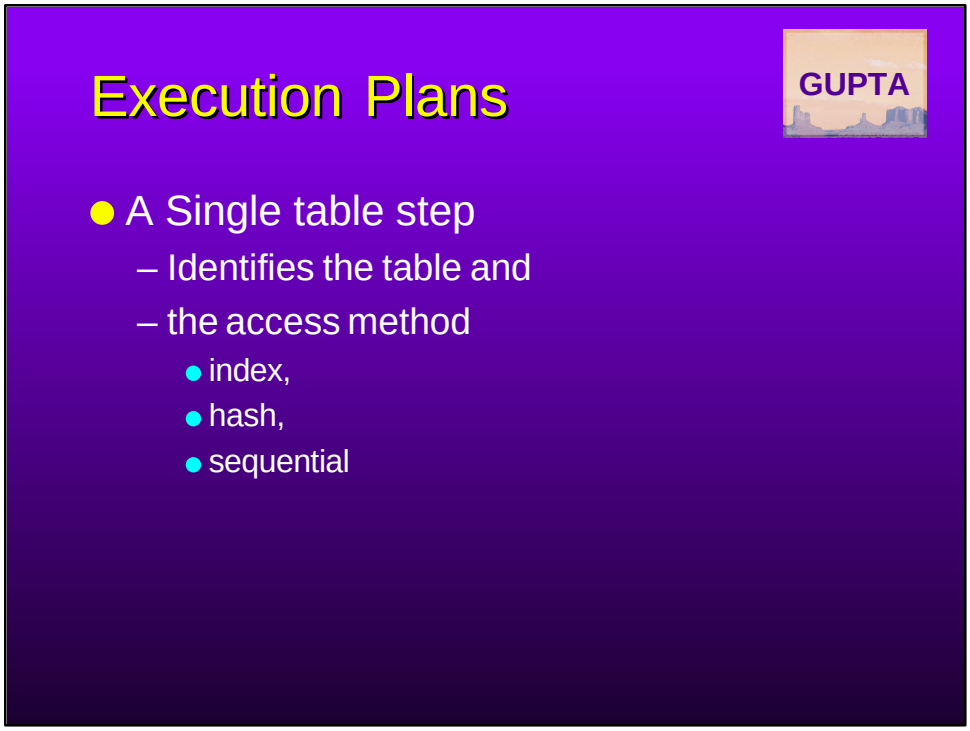


So, let's begin.

What is an execution plan? What does it look like?

Simply speaking, an execution plan is a sequence of steps that tells SQLBase how to execute a query.

Each execution plan can only be one of two types. A execution plan can be a "single table execution step" or it can be a "two table execution step" . Each step produces a result table. The result table may not be a phycical table but most often it is simply a conceptual table. Keep in mind that a definition of a table may be a user defined table, a temporary table, or a conceptual table.



The single step table identifies an input table, the access method to retrieve rows from that table, and the name of result table produced.

The SQLBase access methods currently supported are:

The index access method which can use an index in two different ways. In the first way, a key value is known and the index can be used to position itself and retrieve rows from that point. Or the second way, where the index is scanned to find the key value. If your query processing requires columns other than the key column values then an extra read will be necessary to retrieve the associated data otherwise the request can be satisfied by reading the index. The index access method is only a alternative if the appropriate index or indexes exist.

The hash access method uses the key value to directly retrieve the data containing the rows matching the key value. The technique of converting a key into an direct address where the row exists is known as hashing. Obviously, hashing could only be used if a hash index existed.

And finally, the most simple access method is the sequential scan. This technique causes all data pages of the table to be read. All data pages for a given table are chained together therefore only the data pages of the table need to be read and we don't have to read the entire database file.

Execution Plans

GUPTA

- Two tables Step
  - Identifies two input tables and one output table
  - Outer and inner tables
  - Join method
    - Nested Loop, or
    - Index Loop, or
    - Index Merge, or
    - Hybrid Hash Join

The two table step execution plan identifies the two tables, the access method to retrieve the rows from the two tables, and the join method chosen to join the rows and the result table.

One table is called the outer table and the other is called the inner table.

The two table step execution plan supports the following join methods: nested loop, index loop, index merge, and the hybrid hash join.

And, although the current execution plan output is limited to showing the details of a few major operations such as the access and join methods, the real execution plan stored in the database contains other information things like grouping, Unions, projection, data conversion, and expression evaluation.

Execution Plans

GUPTA

- Example Schema
  - Tables
    - S (S#, SName, Status, City)
    - SP (S#, P#, QTY)
    - P (P#, PName, Color, Weight, City)

To demonstrate some execution plans I will use the following database schema. This is very similiar to the database schema that Date uses in his text called “A Guide to DB2”. This database schema consists of three tables.

The S table is the suppliers table consisting of supplier number, name, status, and city. The SP table is the shipments table consisting of shipment supplier number, part number, and quantity. The P table is the parts table consisting of part number, name, color, weight, and city.

Execution Plans

- Example Schema
  - Indexes
    - SXCITY on S(City)
    - SPXS#P# on SP(S#,P#)
    - PXP# on P(P#)
    - SXS# on S(S#)
    - PXCOLOR on P(Color)
    - PXWEIGHT on P(Weight)

The available indexes include an index called SXCITY on the suppliers city, a composite or concatenated index on the shipment's supplier number and part number, an index on the part number of the part's table, an index on the supplier number of the Supplier table, and indexes on the color and weight of the Parts table.

Execution Plans

GUPTA

- Example Index Access
  - Select * from P where color = ‘Red’ and weight = 19;
  - Execution Plan:

Outer Tbl	Ind_Used-O	Inner Tbl	Ind _Used-I	Result-Tbl	Join Method
		P	PXWEIGHT	RESULT	

Ok, so let’s look at the resulting execution plan for this first query. The optimizer has three choices (1) a sequential scan; (2) index access using the index PXCOLOR; or (3) index access using the index PXWEIGHT. Here the optimizer chose the index on weight. Therefore, SQLBase will use the weight to retrieve all rows that match the criteria “weight=19” and filter out all rows that are NOT ‘Red’ in color. The optimizer chose the weight index over the color index because it thinks that there are fewer row that match the weight criteria. In other words color is a poor choice and would lead to more I/O and CPU processing.

Let me explain this one other way. The optimizer feels there are less parts to deal with that have a weight of 19 as opposed to dealing with all of parts that have a color of ‘Red’.

The query result has been given a conceptual name of RESULT table.

Execution Plan

GUPTA

- Example Index Access
  - Select * from P where color >= ‘Red’ and weight >= 19;
  - Execution Plan:

Outer Tbl	Ind_Used-O	Inner Tbl	Ind _Used-I	Result-Tbl	Join Method
		P	PXCOLOR	RESULT	

This second query is very much like the first except greater than or equal to signs are used instead of equal signs. Its interesting that the optimizer now thinks that the index on color is better to use than the index on weight. Why? Well, its because of the distribution of the data. Since the color ‘Red’ occurs near the end of the alphabet, the predicate “‘color >= ‘Red’” selects a smaller set of rows than the predicate “weight >= 19”. The choice of index is clearly dependent on the target value being compared.

The SQLBase optimizer is smart enough to determine the percentage of data selected based on the key value used in a predicate and the actual index. The optimizer actually performs an index scan to determine the selectivity of a range predicate.

Once again the query result has been given the conceptual name of RESULT table..

Execution Plan

- Example Sequential Access:
  - Select * from P where city = ‘London’;
  - Execution Plan:

Outer Tbl	Ind_Used-O	Inner Tbl	Ind _Used-I	Result-Tbl	Join Method
		P		RESULT	

This particular query leaves no alternative other than to choose a sequential scan access method. There is no available index on the city in the Parts table. SQLBase will scan all of the data pages in the Part table P.

Execution Plan

GUPTA

- Example Index-Merge:
  - Select S.SName,SP.P#, SP.Qty from S,SP where S.S# = SP.S# and S.S# = ‘S1’;
  - Execution Plan:

Outer Tbl	Ind_Used-O	Inner Tbl	Ind _Used-I	Result-Tbl	Join Method
S	SXS#	SP	SXS#P#	RESULT	INDEX-MERGE

This query is a little more complex. It uses a join-method of index merge. In this method, SQLBase scans the indexes of both the inner and outer tables taking advantage of the ordering of the keys. In this query, it uses the predicate S.S# = ‘S1’ to position the start of the scan on the outer index SXS#. The same predicate will also decide the end of the scan on the outer index. The index SXS#P# is used to find the matching rows from the shipment table, SP.

Execution Plan

GUPTA

- Example Three-Table Join:
  - Select S.* from S,SP,P where S.S# = SP.S# and SP.P# = P.P# and P.color = 'Red';
  - Execution Plan:

Outer Tbl	Ind_Used-O	Inner Tbl	Ind _Used-I	Result-Tbl	Join Method
P	PXCOLOR	SP	SPXS#P#	TMP11	INDEX-LOOP
TMP11	SPXS#P#	S	SXS#	RESULT	INDEX-MERGE

The query results in a two step execution plan. First, the rows of tables P and SP are joined together to produce the conceptual table called TMP11. Then the tables TMP11 and S are joined together to produce the conceptual table RESULT.

Order of the operations should not be taken literally as no physical tables are being created. As soon as a result row is available from step 1 it will participate in the join operation in step 2. When a row is available from the step 2 it will be made available to the application.

This query used the index on color to retrieve qualified rows from the Parts table, P. For each row retrieved from P, the index SPXS#P# is scanned to find the qualifying supplier rows in table SP. Then, the SPXS#P# index will be index-merged with the index SXS# to find the matching supplier table rows in table S.

Execution Plans

GUPTA

- A View Query:
  - Create view SV1 as select * from S where Status > 20;
  - Select SV1.* from SV1,SP,P where SV1.S# = SP.S# and SP.P# = P.P# and P.color = 'Green';
  - Execution Plan:

Outer Tbl	Ind_Used-O	Inner Tbl	Ind _Used-I	Result-Tbl	Join Method
		SV1		TMP1	
P	PXCOLOR	S		TMP21	INDEX-LOOP
TMP21		SP	SXS# P#	RESULT	INDEX-LOOP

This query is very similiar to an earlier query but table S is being replaced by a view called SV1.

The first row of the execution plan shows a dummy step that does not correspond to any execution but it does indicate that the view has been fully collapsed into the rest of the query. The base table of view SV1, S is joined with table P using the index loop join method. No index on the inner supplier table S could be used.

The result table TMP21 join with shipment table SP to produce the final result set.

The main purpose of this slide is to show the use of a view in the execution plan.

Execution Plans

GUPTA

- An Union View :
  - Create view SPV1(S#,P#, Qty) as
    select * from SP where Qty > 200
    Union
    select * from SP where P# in
    (select P# from P where Color = ‘Red’);

This slide shows the creation of view based on a query doing a UNION of two selects.

Execution Plans

GUPTA

- An Union View Query:
  - Select SPV1.*, S.SName from SPV1,S where SPV1.S# = S.S# and S.Status > 10;
  - Execution Plan:

Outer Tbl	Ind_Used-O	Inner Tbl	Ind _Used-I	Result-Tbl	Join Method
		P	PXCOLOR	TMP1	
		SP		TMP2	
		SP		TMP4	
S		TMP3		RESULT	NESTED LOOP

This execution plan is interesting because the first three rows of the exeuction plan outline the steps to execute the view. Since you can see the steps being outlined, the view has not been collapsed or folded into the main query. The result of the subquery in the view is stored in the table TMP1. Since TMP1 does not participate in the join operation with any other table, it gives us a clue that it must be used in a WHERE clause.

Results of two selects of the UNION clause are stored in conceptual tables TMP2 and TMP4.

The result table TMP3 is not indicated in the plan as a result table. This is a current limitation in the show plan implementation because we don't currently have a way to indicate a UNION operation.

Finallly, the supplier table S is joined with the table TMP3 using the NESTED LOOP join method.

Costing Model

GUPTA

- Cost based
  - CPU cost in milliseconds
  - I/O cost converted into milliseconds
- All possible table access methods
- All possible join methods
- Every possible permutation
- Least expensive plan

The SQLBase optimizer uses a cost based technique to determine the best execution plan for any given query as opposed to inferior syntax or rule-based techniques. The optimizer computes CPU and I/O costs with each different access method and operation. The CPU cost is estimated in milliseconds and I/O costs which are the number of I/Os per second are also converted to milliseconds. This way a trade off can be made between I/O versus CPU time. Performance of access methods and join operations is affected by the size and distribution of the data. Data characteristics are captured in the table and index statistics. A query is broken down into a number of smaller individual steps. The smallest step is access of a single table which is analyzed for all possible access methods. All permutations of table ordering as well as all possible joing methods between any two tables are tried. Eventually, the optimizer picks the least expensive plan.

The following slides try to show how costs are computed for the varous access and join methods. The topic is very complex so we've made some attempt to simplify in the slides. For example, CPU costs have been eliminated in many places but it doesn't mean that CPU cost is not important.

Costing Model

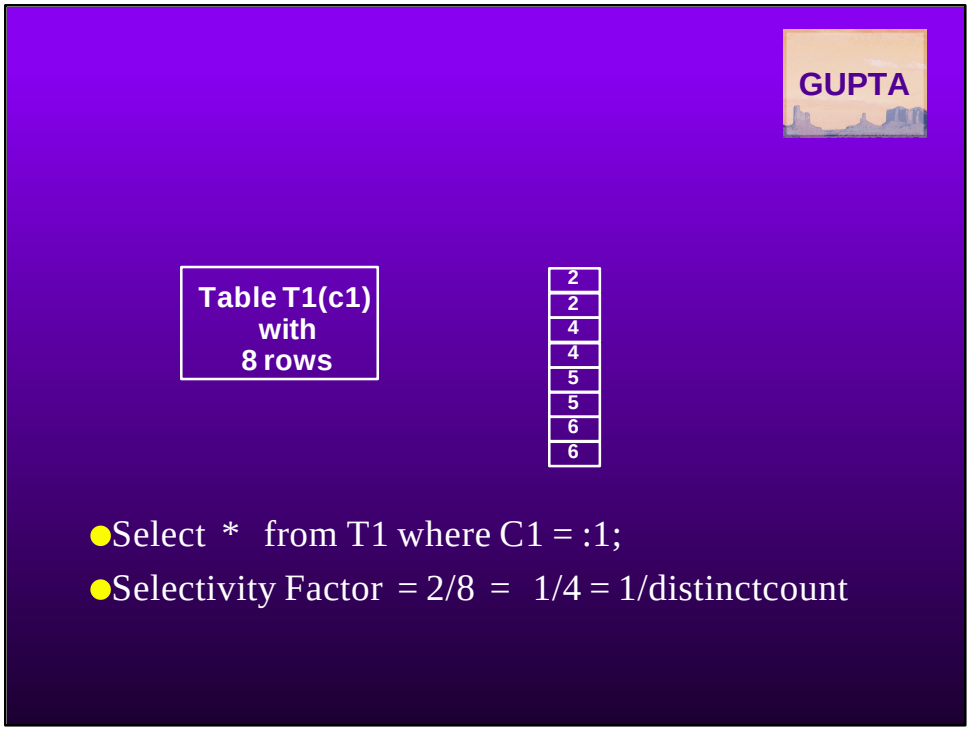
GUPTA

- Sequential (Table) Access
  - I/O Cost = Number of data pages
  - Number of rows and selectivity determines number of rows returned

The sequential access method of a table results in the table being scanned from one end to the other.

The associated cost is determined by the number data pages in the table to scanned; plus the number of rows in the table to be checked for acceptance; plus the number of rows selected. The number of rows to be selected can be computed by multiplying the selectivity factor of the predicate by the total number of rows in the table.

The selectivitiy factor is a number that determines what percentage of rows will be returned for a given predicate. For more information on how to determine the selectivity factor for different predicates you should read the new "SQLBase Advanced Topics Guide" coming out with SQLBase 6.0.



Here's an example of what we mean by a selectivity factor. Let's suppose there is a table called T1 with one column called C1 and it has 8 rows of data.

What would the selectivity factor be for the query SELECT * FROM T1 WHERE C1 = :1 be?

Well, there are eight rows with two values for each distinct value present in the table. That is, there are two 2's, two 4's, two 5's, and two 6's. So, the odds of one particular value coming up is 2 out of 8 or 1 over 4.

Another way to compute this is to just use the statistical value DISTINCTCOUNT in the SYSKEYS catalog table. The selectivity factor is alway equal to one over the DISTINCTCOUNT.

Costing Model

GUPTA

- Index Access Parameters
  - Height of the B-Tree
  - Number of leaf pages
  - Number of distinct keys
  - Cluster count
  - Number of rows
  - Number of data pages
  - Selectivity of the predicate

The index access method provides quick access to data when a key value is known. It can also be used to retrieve rows in the key order. The cost of index access is determined by the following parameters:

Height of the B-Tree

Number of leaf pages

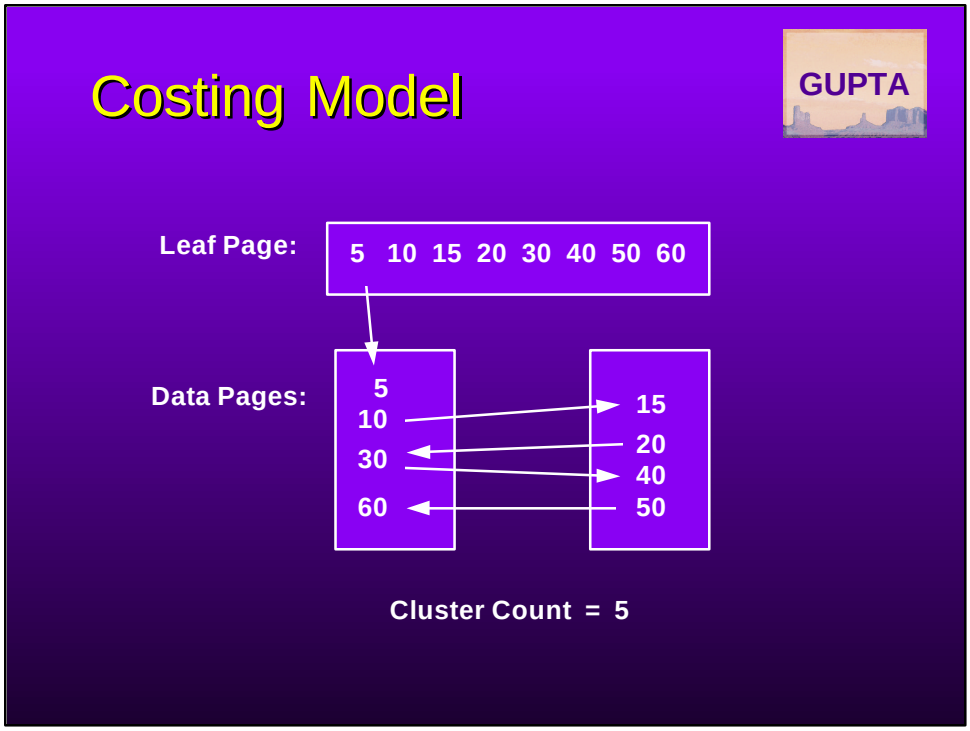
Number of distinct keys

Cluster count

Number of rows

Number of data pages

Selectivity of the predicate



The clustercount is the total number of page changes that would occur if the entire table was read thru the index. For a perfectly clustered index, this column is equal to the number of base data pages in the table. At the other extreme, the highest value for a totally unclustered index is equal to the number of rows in the table.

The example in this slide shows a clustercount of five. Each arrow shows the jumping from one page to another. To scan the entire index, you would be required to move from one page to another five times.

Costing Model

GUPTA

- Nested Loops Join
  - A simplest join method
  - An outer table (say TO) is joined with an inner table (say TI)
  - Think of cross-product and filtering

This is one of the most basic join methods. The nested loop join reads a row from the first table, called the outer table, and then each row in the second table, called the inner table, is read as a candidate for the join. Then the second tow in the outer table is read, and again each of the rows in the inner table is read again, and so on until all row in the outer table are exhausted. If the outer table has X number of rows and the inner table has Y number of rows the number of rows to be read are a product of X times Y.

GUPTA

- Nested Loops Algorithm

```
while (more rows in TO)
        fetch a row from  TO
        while there is a row
            fetch matching rows from TI
            output the joined rows
        end of inner loop
end of outer loop
```

GUPTA

- Nested Loops I/O Cost (*APPROXIMATE*)
- Table Scan = TO_pages + TO_rows * TI_pages
- Hash Index = TO_pages + TO_rows * 1
- Index Scan = TO_pages + TO_rows * (height_of_tree)

This slide shows the simplified formulas for I/O cost calculations for a nested loop or index loop join operation.

For a table scan the I/O cost is the number of outer pages plus the number of outer rows times the number of inner pages.

For a hash index the I/O cost is the number of outer pages pluse the number of outer rows. The times one only implies that it takes one access to get to the inner row.

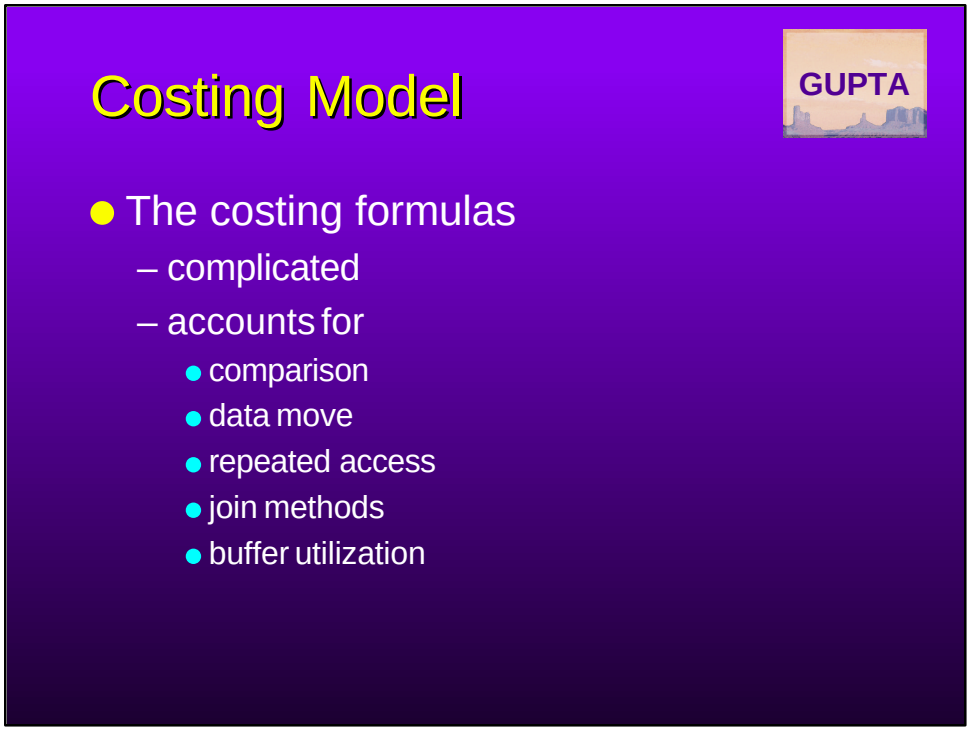
The index scan I/O cost is the number of outer pages plus the number of outer rows times the height of the B-Tree.

Costing Model

GUPTA

- Hash Access Parameters
  - Number of primary pages
  - Number of overflow pages

The hash access method provides very quick access to a desired row when the key value is known. That is, it works only with equal predicates. In normal cases, the I/O cost is 1 page but it becomes complicated in the presence of overflow pages. Therefore, the I/O cost for the hash access method is a function of the number of primary pages and the number of overflow pages.



In conclusion, the costing formulas that make up the SQLBase optimizer cost model can be very complicated. The costing of an execution plan also takes into consideration the comparison oprations, data movement (like the staging of data in memory), repeated accesses, join methods, cache buffer utilization, and the selectivity factors of available predicates.

New Statistics Columns

GUPTA

- New columns added to
  - SYSTABLES
  - SYSINDEXES
  - SYSCOLUMNS
  - SYSKEYS
- Optimizer specific statistics can be forced

By now you may be wondering, “How do I know what the statistics are?”. The statistics have been added in SQLBase 6.0 to existing SQLBase system catalog tables. These statistics have been added to the SYSTABLES, SYSCOLUMNS, SYSINDEXES, and SYSKEYS tables.

We added these new columns so the statistics will be available to you. And, for the first time, in SQLBase 6.0, you will be allowed to force the statistics to any values you determine to influence the behavior of the optimizer. This modification of statistics can also be a very useful learning tool.

New Statistics Columns

GUPTA

- What statistics are available
  - SYSTABLES
    - ROWCOUNT, PAGECOUNT, ROWPAGECOUNT, LONGPAGECOUNT , EXTENTPAGECOUNT, FREESLOTS, USEDSPACE, FREESPACE, AVGROWLEN, AVGLONGLEN, GROUPNUM
  - SYSCOLUMNS
    - AVGCOLLEN, AVGCOLLONGLEN

All of the columns on this slide are available in the SYSTABLES and SYSCOLUMNS table. But, the ROWCOUNT, PAGECOUNT, ROWPAGECOUNT, and LONGPAGECOUNT are optimizer staticistics that you are allowed to modify.

New Statistics Columns

GUPTA

- What statistics are available
  - SYSINDEXES
    - HEIGHT, INDEXPAGECOUNT, LEAFCOUNT, CLUSTERCOUNT, PRIMPAGECOUNT, OVFLPAGECOUNT , AVGKEYLEN, GROUPNUM
  - SYSKEYS
    - DISTINCTCOUNT
      - select count (distinct LNAME) from EMPLOYEE;

You are now allowed to modify the HEIGHT, INDEXPAGECOUNT, LEAFCOUNT, CLUSTERCOUNT, PRIMPAGECOUNT and OVFLPAGECOUNT optimizer statistical columns in the SYSINDEXES table.

And, you are allowed to modify the DISTINCTCOUNT in the SYSKEYS tables to influence your selectivity factors. This sample query is an example of how you could determine the distinctcount for a particular column, in this case LASTNAME, from some table like EMPLOYEE.

Forcing Statistics

GUPTA

- Forcing table statistics
  - Syntax:
    - Update Statistics on Table *table-name* Set *stat-column* = *expression* [, ...] ;
  - Example:
    - update statistics on table ORDER set ROWCOUNT= 10000, PAGECOUNT = 100;

The optimizer statistics that you can modify can be forced with the new, improved UPDATE STATISTICS command. In SQLBase 5, you could update the current optimizer statitics of a table or an index with the UPDATE STATISTICS command. Now, you can actually modify certain important statictical columns. The complete syntax for this new command is described in the SQLBase 6.0 SQL Language Reference Manual but breifly on this slide you can see that we have added the SET clause. So, now you can set a particular statistical column to your desired value.

The example shows the changing of the optimizer statistics for the table called ORDER. We are changing the number of rows to ten thousand and the number of page to one hundred.

Forcing Statistics

GUPTA

- Forcing index statistics
  - Syntax:
    - Update Statistics on Index *index-name* Set *stat-column* = *expression* [, ...] ;
  - Example:
    - update statistics on index ORDER_IX1 set HEIGHT = 2, CLUSTERCOUNT = 10000, LEAFCOUNT = 10000/150;

This slide shows the syntax for updating the statistics of an index. And again, as you can see, we have added the SET clause. This allows the modification of the statistical columns for an index.

The example shows the modification of statistics for the index called ORDER_IX1. In this example, we are modifying the B-Tree height, the clustercount, and the leafcount.

Forcing Execution Plans

GUPTA

- Designed For
  - Database Modeling
    - Execution Behavior
    - Database Size
- Other Benefits
  - Able to Influence Execution Plans
  - Not A substitute for A Force Plan Command

The ‘forcing of execution plans’ is really a bad term. The modification of the optimizer statistics gives you the ability to ‘influence’ the execution but it doesn’t necessarily cause you to force the plan.

This feature was implemented to provide you with a tool to study the behavior of applications during development where you do not have access to the production size database. And, to help you model your database using a smaller database.

As I mentioned, this tool is for influencing the execution plan. It is not a a substitute for a FORCE PLAN command. We still plan to implement a FORCE PLAN facility in the future. This will provide you with a way to force a particular access plan.

Forcing Execution Plans

GUPTA

- Table P
  - update statistics on table P set rowcount=1000, pagecount=100, rowpagecount=100;
- Index PXCOLOR
  - update statistics on index PXCOLOR set distinctcount(color)=900, leafcount=5, height=2, clustercount=100;

This slide shows how we can influence execution plans by modifying the statistics of the Parts table, P and the index PXCOLOR.

In the Parts table, we will increase the number of rows to 1000, the number of pages to 100, and the rows per page to 100. We will increase the distinct number of colors to 900, the leafcount to 5, the B-Tree height to 2, and the clustercount to 100.

Now, let's see how this effects the query execution plans.

Forcing Execution Plans

GUPTA

- Query: Select * from P where color = ‘Red’ and weight = 19;

Plan before :

Outer Tbl	Ind_Used-O	Inner Tbl	Ind _Used-I	Result-Tbl	Join Method
		P	PXWEIGHT	RESULT	

Plan after:

Outer Tbl	Ind_Used-O	Inner Tbl	Ind _Used-I	Result-Tbl	Join Method
		P	PXCOLOR	RESULT	

The query to select parts where the color is Red and weight is 19 previously resulted in the use of the PXWEIGHT index. Now, that we’ve adjusted the optimizer statistics, it chose the PXCOLOR index. How come?

Well, it chose PXWEIGHT because there were more distinct weights than number of distinct colors but now we’ve increased the number of distinct colors to 900 which now exceed the number of distinct weights. So PXCOLOR is better choice.

To summarize, by increasing the number of rows in the parts tables and making PXCOLOR more attractive by increasing the number of distinct keys in the column color we have made PXCOLOR a more attracitve index to use than PXWEIGHT.

Forcing Execution Plans

GUPTA

- Query: Select * from P where color >= ‘Red’ and weight >= 19;

Plan before:

Outer Tbl	Ind_Used-O	Inner Tbl	Ind _Used-I	Result-Tbl	Join Method
		P	PXCOLOR	RESULT	

Plan after:

Outer Tbl	Ind_Used-O	Inner Tbl	Ind _Used-I	Result-Tbl	Join Method
		P	PXWEIGHT	RESULT	

The UPDATE STATISTICS command does not effect the selectivity factor for range predicates. This query changed indexes from PXCOLOR to PXWEIGHT because PXCOLOR is now a bigger index with a higher HEIGHT value (2 as opposed to 1) and a greater number of leaf pages. PXWEIGHT is an index with a leafpage count of 1 and height of 1. So computation wise, PXWEIGHT is producing a lesser number of total I/Os. Therefore, PXWEIGHT has become preferable to PXCOLOR.

Forcing Execution Plans

GUPTA

- Query: Select S.S#, SP.P# from S,SP where S.S# = SP.S#;

Execution Plan before:

Outer Tbl	Ind_Used-O	Inner Tbl	Ind _Used-I	Result-Tbl	Join Method
SP		S	SXS#	RESULT	INDEX LOOP

One way to influence an execution plan is to change the statistics to make the current index choice by the optimizer a bad choice and force the optimizer to try the second best plan instead.

The query on this slide normally would use the SXS# index with the index loop join method.

Forcing Execution Plans

GUPTA

- Force Statistics:
  - update statistics on table SP set rowcount = 100, pagecount=2, rowpagecount=2 ;
  - update statistics on index SXS# set distinctcount(s#) = 100, leafcount=2, height=1, clustercount=100;

The Plan:

Outer Tbl	Ind_Used-O	Inner Tbl	Ind _Used-I	Result-Tbl	Join Method
S		SP	SPXS#P#	RESULT	INDEX-LOOP

To make the optimizer chose another plan, let's increase the numbers on the shipment table, SP. And, increase the numbers on the index SXS# that was being used so the optimizer will chose some other alternative.

Let's change the rowcount to 100, the pagecount to 2, and the number of rows per page to 2. Additionally, let's increase the number on the SXS# index.

The new plan shows the usage of a new index, SPXS#P# and the inner and outer tables have been reversed.

This is example a placing a negative influence on one table and index to make another table and index look more attractive.

What We Learned

GUPTA

- 6.0 Execution Plans
- SQLBase Optimizer Costing Model
- New Statistics Columns
- Modifying the Optimizer Statistics
- Influencing the Execution Plans

You should have learned that SQLBase is a cost based optimizer. We reviewed a few SQLBase 6.0 execution plans and the mechanics of the SQLBase optimizer costing model. You should have also learned about the new statistical columns available in the SQLBase system catalog of SQLBase 6.0 and how you can modify these optimizer statistics. And finally, you have seen how you can influence the execution plans by modifying the optimizer statistics.