To: Reporting Initiative developers

RE: OPTIMAL SQL CODE

Date: 10/28/93

On Thursday at 1:30 in room 1879 we'll have a demo of how to use showplan, to test how good your SQL code is. Please RSVP, and bring any questions related to this document, or SQL speed. Most C/S projects have some problems with SQL efficiency. You may find this helpful. In future, we'll publish a document on PB optimization. Here are some SQL tips

Joins

A join condition is used to notify the SQL server of the relationship between two tables. Every table involved in a join must have at least one join condition. If three tables are used in a join, at least two join conditions must exist. If four tables are involved three join conditions must exist etc.

For Example:

```
select a.col1, b.col2, c.col3
  from a,b,c
 where a.col1 = b.col2
   and b.col3 = c.col3
```

Each table is involved in at least one join condition. This is the minimum number of join conditions which is required.

The join should always use an indexed = for a where condition to provide the join criteria.

The order of the join conditions is irrelevant as is the order of the tables in the from clause.

If there are more than four tables involved in a join, the SYBASE optimize may not optimize the query correctly (eight table joins are allowed in version 10). If possible, break your query into multiple queries using temporary tables.

For Example:

```
select a.col1, f.col7
  from a,b,c,d,e,f
 where a.col2 = b.col2
   and b.col3 = c.col3
   and c.col4 = d.col4
   and d.col5 = e.col5
   and e.col6 = f.col6
```

Performance will probably improve by dividing the query as follows:

```
select a.col1, c.col4
  into #temp
  from a,b,c
 where a.col2 = b.col2
   and b.col3 = c.col3

select #temp.col1, f.col7
  from #temp,d,e,f
```

```
              where #temp.col4 = d.col4
                and d.col5     = e.col5
                and e.col6     = f.col6
```

Also, if specify all possible choices to allow the optimizer to evaluate options for performing the join:

```
      EX:   select name from table1, table2
            where table1.a = table2.b and
            table1.a =100
      should include table2.b = 100 to give the optmizer all possible
choices.
```

Notify the database modeler if joins frequently exceed four tables. Table changes may be required. This should be resolved during the database design process.

Network Traffic

The network is often the bottleneck of fast running systems. Even the fastest network has a fixed capacity so network load must be minimized. The following techniques may be used:

- Used stored procedures if the code/command is executed/used frequently. This avoids a recompile and re optimization code.
- Select only the columns required to perform the query. Avoid using ***select **** unless all columns are required.
- Isolate data that is required to resolve a query. If only the first 10 characters of a varchar(50) are required, then use a substring to retrieve only the first 10 characters.
- If a query will be executed multiple times, execute it once and save the results on the client. For example, frequently used pop-ups or validations can be stored to C: or RAM. (This may require version control flags).
- Make full use of SYBASE aggregate functions (MIN, MAX, SUM, AVG, COUNT, GROUP BY). These functions execute on the server, thus reducing the amount of data to be moved, and are faster performing the calculations on the client.

Indexes

Indexes supply SYBASE with query paths which can be used to access data. Without an index, SYBASE must scan the entire table to retrieve information. When the test data is small, no significant delay will be realized. However, delays can be come extreme in large production data environments.

A demo on doing this will be performed on Desktop DBA. (Lea - Sybase). However, you may choose to use ISQL.

What Indexes Are Available?

To find out what indexes are available for a table, SYBASE provides a system procedure called sp_helpindex. It is used as follows:

```
sp_helpindex tablename
```

What Indexes Am I using?

For a given select statement, SYBASE's query plan can be documented by using the showplan option. Tables with at least 100 rows and good data distribution are essential to getting accurate output. Query plans work as follows:

```
            set showplan on  /* show how the plan will be executed */
            set noexec   on  /* Do not actually execute the query    */
            go
            select *
            from a,b
            where a.col1 = b.col2
            go
            set noexec   off /* Allow execution again             */
            set showplan off /* Do not show query plans */
            go
```

The output from this will look like this:

(See attached)

By using **set statistics on**, the number of IO operations and the time required to perform a query may be obtained.

Two things to keep in mind when examining the output from a showplan:

- The first table listed in the showplan will usually require a full table scan. If that table is very large, the query will be slow.
- If more than one table is full table scanned, then there is a serious problem.

Notify the data modeler of large data loads or index changes so that stored procedures and statistics may be recompiled.

Why Is The Query Not Using Indexes?

In order to use an index, the query must have search arguments (SARGs). The SARGs are located in the where clause and the optimize uses SARGs to pick a query plan. For example:

```
           where type = 50   /* is a SARG */
           and   cost < 500  /* is a SARG */
```

If an index is created on cost or type, SYBASE will probably use it. If there are indexes on both type and cost, SYBASE will use the index that the optimizer selects as best for resolving the query. (More examples are available in the Performance & Tuning section of APC'S Database Administration for SQL Server guide).

Operators which may compose a SARG are =,<,>,<=,>=,BETWEEN, and possibly LIKE. Operators which may not be used in a SARG are IN, NOT, and possibly LIKE. The LIKE operator can be used as a SARG when it compares to a value which starts as a constant. Refer to the chart below for examples of when expressions can be used as SARGs:

Type Expression	**Example Expression**	**Sargable**
Equality	columnA = 'Larry'	yes
Comparison	columnB > 12	yes
List of Values	columnC in (911, 914, 916)	yes
Inequality	dept != 10	no
Another Column	columnB = columnC	no
Aggregate	avg(columnA) = 50	no
Computation	columnB * 2 > 30	no

Value Manipulation	substring(columnA, 1, 2) = no "BE"

Please refer to the *Sybase Performance & Tuning Class Notes, Chapter 7, Query Optimization* for more information.

All SARGs should be connected using ANDs not ORs. SYBASE handles OR conditions in a more complicated manner, therefore ORs should be avoided wherever possible.

A query may avoid using an index for any number of reasons. The following series of steps should help identify the problem:

- Does an index exist for what the query is trying to do? If not, talk to the modeler about creating one.
- Does the query have a SARG for the index?
- Still doesn't work? Ask the DBA to update the statistics on the table.
- Call the query doctor (Get help).

Transaction Processing

Always isolate transactions atomicaly, and never return the control to the client during an open transaction.

Bad EX:

```
Begin tran
select ...
update ...
commit
```

Good EX:

```
select ...
commit
Begin tran
update ...
commit
```

Note: the commit following the select in the above example is required in PowerBuilder dynamic calls because PowerBuilder automatically issues a hold lock. Not issuing this commit could cause locking and impede performance. This commit would not be required in a stored procedure.

Miscellaneous Issues :

Calculations on columns, such as column * 2 should be performed on the client. Calculations involving aggregation (Group by) should be performed on the server. Each calculation should be examined on an individual basis to determine which machine would help the query to perform more efficiently. While a client machine has a dedicated CPU, the server may be a larger machine that can more efficiently handle the processing required to resolve the query. Always include network traffic in any decisions about query placement. A good example to remember is aggregate at the server, compute at the client.

Sorting of data should be done on the server. The side effect is that it may create a temporary table.

If certain columns are consistently examined in an ordered fashion, consider requesting covered indexes to boost performance. This should be included as a step within the design process.

Use RPC instead of connect for remote servers.

Avoid using ODBC or other middleware.

OPTIMIZER

Table Queries:

- only data manipulation queries are optimized

- only one index is used per table

- for a query to use an index, it must have search arguments (SARG's), such as...
  ...must have conjunctions only (and...no or's)
  ...column - operator - constant
  ...no 'NOT' operands
  ...no column - operand - constant - operand - constant (ie. salary * 12 > 30000)
  ...no 'BETWEEN' operand (use > <)
  ...no 'LIKE %constant' (could use 'LIKE constant%')

- the more SARG's on the index, the more selective the index (exception: the optimizer will know that equality on a unique index will always return one row)

- if composite index (A,B,C) must have a SARG on A or index is not useful

- SARG's should establish lower bound, upper bound, or both (=)

- if datatypes do not match or there are unknown values, optimizer may not be as accurate in deciding whether or not to use an index. A good index may be overlooked.

Table Joins:

- *order of tables in from clause and order of where clause makes absolutely no difference in how tables are joined!!*

- needs SARG's as described above

- only one index is used per table

- most subqueries are equivalent to a join. can't assume a subquery is done first. the optimizer will unwind the subquery and treat it like a join.

- exception: 'where exists'...in this case the subquery is done first

- when no index exists or if index is not useful, a work table is created in tempdb, the data is reformatted in the work table, and if deemed necessary, an index is created...can be costly!

- if many queries done as mentioned in bullet above, space in tempdb could become bottleneck (log activity as well as amount of space required)

- joining more than four tables is costly

Aggragates:

- select count(*)...would use a nonclustered index if you provided a SARG (where col > 0) and would be faster than no index. the nonclustered has a row in the index for each data row. since index row is smaller than data row, would be more efficient. clustered index would not be helpful because it would still have to go to the data pages to count the rows (the index points to a page, not a row)

- max(col)/min(col)...if used in separate selects will use nonclustered index on col if there is one. can follow the tree down one side. if max/min used together, won't use index. to optimize, use separately.

- order by...no where clause needed for clustered index to be used. must have where clause for nonclustered index to be used.

- group by...if no index, work table is created using a table scan. if clustered index, work table is created using the clustered index. nonclustered index does not help much.

Or Strategy:

- IN is converted to OR

- selects both sides of the OR using the indexes if possible. uses a work table and dynamic indexes. sorts and removes duplicates. if it does a table scan on part of the OR it will not use the dynamic index...only the table scan.

Deferred Updates:

- used when updating multiple rows

- writes the rows to be changed to the log, reads the log and makes the changes, loging the changes (writes twice as much to the log...must make two passes to make the update)

- costly, but allows for tricky updates

- twice as many log records for deferred vs direct update

- example: unique key = number = 1,2,3,4,5,6; number = number + 1; as soon as you add one to the first number you create a non unique key and update would fail

- another example: update table set number = 10 where number > 0; would be an endless loop without SQL Servers deferred updates)

Direct Updates:

- data and log page are changed in cache then do a normal write-ahead log

- requirements are...
  ...must be only one row being affected and server knows it
  ...must have unique index and server knows it
  ...column(s) being updated can not be part of the index
  ...column must be fixed length, no varchars and no nulls
  ...data and log page are changed in cache then do a normal write-ahead log

- next best thing, after an update in place

Updates in Place:

- no indexes have to be adjusted. row does not move, new row goes in position of old row...**fastest method of updating!!**

- requirements are same as direct update, in addition to...
  ...no triggers

- column must be fixed length, no varchars or nulls

Update Performance Issues:

- when row moves to new page, nonclustered indexes must be updated

- the update to the nonclustered index is also done as an update (delete/insert)

- all deletes and updates will affect the nonclustered indexes

- also, on table with clustered and nonclustered indexes, all inserts, updates and deletes will affect the nonclustered indexes

Clustered Index:

Facts:

Note: If no clustered index, data is in a heap. Data is always added to bottom of table!!

- Clustered index is always faster than a table scan or a nonclustered index.

- Faster for retrievals. Index tree is generally one level smaller than nonclustered. Index points to page rather than row.

- Especially faster when there is duplicate data. Less I/O required because the data is more likely to be on the same page.

- Data is put in proper order on insert. If not enough room on page, page splits...performance penalty. (page splits 50%/50%, monotonic page splits 100%, 0%, average page is 75% full)

- Requires more overhead than nonclustered for deletes and inserts (data *and* index rows must move).

When to use:

- primary key columns (if used in where clause)
- columns accessed by range (may do partial table scan in between)
- columns frequently accessed sequentially
- columns used with "order by" or "group by"

- columns *not* frequently modified
- columns used in joins
- most commonly used physical order...faster than sorting data outside of database
- avoid always inserting to end of table (heap...causes bottleneck)

Nonclustered Index:

Facts:

- Index has to be updated when inserting rows. Leaf level has one entry per row...guaranteed to have to change leaf level and possibly other levels (if page split occurs).

- Data is stored in heap. Data is always added to bottom of table. SQL Server does not go back and fill in gaps left from deletes.

- Data is not sorted. Provides a set of pointers which are in sorted order.

- Index points to rows rather than page.

When to use...when you already have a clustered index and...:

- retrieving small portion of table (less than 20%)
- avoiding table access (index covering)
- join, order by, group by
- aggragrate columns (if where clause access index)
- useful when there is a low level of duplicates

Note: If the number of rows that qualify in the select is greater than the number of pages in the table, the nonclustered index will be *slower* than a table scan!! Also, if the table is small (approx. 2 pages or less), a table scan will most likely be done.

Index Size:

- Keep index small! If all index levels fit in cache, 100% hit ratio!!

- The more rows that can be read into cache per page, the faster the I/O.

- The fewer levels of indexing required (index tree), the faster the I/O.

- The larger the index, the less number of rows per index page, requiring more index levels which equals slower I/O.

- If index pages are full and splitting frequently, adds more indexing levels. Drop indexes and recreate them (when using a fillfactor).

- If a top level index splits, it may cascade down, splitting other levels. It is much worse to have index levels splitting than data pages splitting.

Building a Clustered Index:

- Clustered index is always on same segment as data. If you put table on one segment and clustered index on another, the whole table will move to where the clustered index is. Exception: with very large tables, could split table containing clustered index across two segments. Must place first piece of table on first segment and create the clustered index, then place second piece of table on second segment. If you wait until placing both pieces of table to create the clustered index, the second piece of the table will move to where the first piece of the table is.

- To build a clustered index on table that already contains data, must have space equal to 120% of table size (sort the data and create the index).

Building a Nonclustered Index:

NOTE: For each nonclustered index you add to a table, less index levels remain in cache, reads & writes go up up up...!!! (only use them when you need them)

- If you drop and rebuild a clustered index, all nonclustered indexes get rebuilt (because data may rearrange itself). Build clustered index before nonclustered indexes.

- Could improve performance (less contention) by putting the table on one device and the index on another device. Must use segments to do this.

- Greatest performance improvement ...2 separate disks and 2 controllers and channels.

Statistics:

- UPDATE STATISTICS command provides SQL Server with info about the distribution of the key values in the indexes. SQL Server uses this info to make decisions about which indexes to use when it processes a query.

- Statistics are gathered automatically when an index is created/recreated on a table that already has data.

- Stats are only kept on first column in a composite index. Should choose the most selective column to go first in a composite index so the distribution will be accurate.
- Very Improtant to rerun update statistics when skew of data changes.

- Could severly impact performance if stats are not correct.

- CLUE: In the sysindexes table, if distribution is 0, no statistics are kept for the table. (sp_helpindex tablename)

- If stats are not available, server will assume them. (max(index), min(index) and # of rows)

- UPDATE STATISTICS REGULARLY!!

Deletes:

- space is always left at the end of the page, not in the middle

- empty pages are always deleted, but only released back to the os in extents of 8 pages

- when one row is left on an index page, it is moved to an adjoining page and the index page is deleted

- when a data row is deleted, all nonclustered leaf rows which point to it also must be deleted

- could end up with many data pages, one row each

- only way to get full pages again...bcp data out and back in OR create and destroy a clustered index

Nested Iteration of Joins

- **Evaluates Cost of Each Table Search**
- **Estimates Number of Logical and Physical I/Os**
- **Finds Optimal Join Order (Outer Table to Inner Table)**
- **Only the Outer Table Search Can Be Limited**

If there is a join clause in the query, the optimizer evaluates the number of tables, indexes, and joins to determine the optimal order for the nested iteration.

Strategy

- Chooses processing order. Evaluates four tables at a time.
- Projects value from next qualifying row in outer table into inner query.
- Outer table search argument can use an index because it limits the search.
- Processes single table queries according to previously described tactics.

Performance

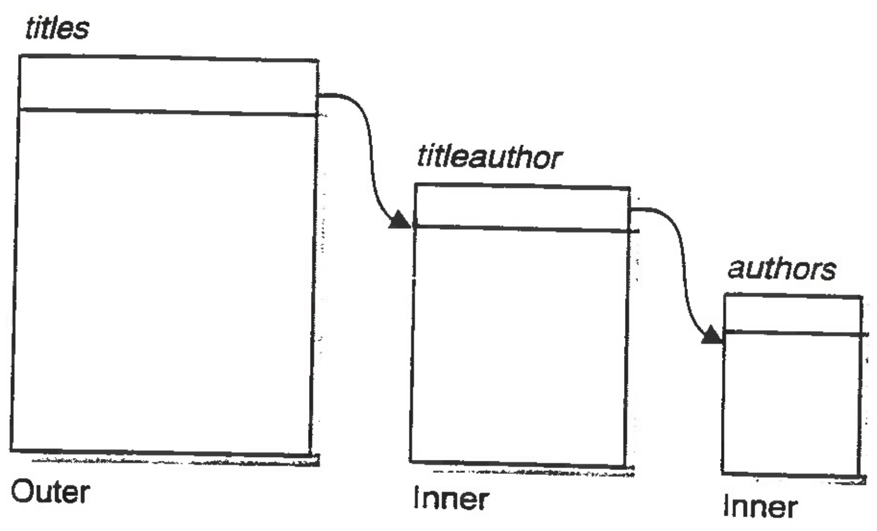
- Performance is based on how many times the inner query is done.
- Processing order is affected by restrictions that can be applied to each table (search arguments and join clauses contained in the WHERE clause).
- Choices are made using the statistics about the indexes to estimate how many rows are to be returned.
- When evaluating the cost of join order, the optimizer also checks to see if the table will fit in cache. If so, physical I/O is not included in the cost of the plan.

Note The number of I/Os can vary, based on the processing order of the tables that the optimizer chooses.

Guidelines

- Add more join clauses to allow the optimizer more choices in query plans.
- Add redundant clauses.

Nested Iteration of Joins: Example



The nested iteration of joins strategy constructs a set of nested loops by finding a row from the first table and then using that row to scan the next table and so on until the result that matches is used to scan the last table. The results set is narrowed down as it progresses from table to table with each iteration.

The query plan specifies the ordered set of nested tables to use. The number of different possible plans is related to the number of tables, indexes, and joins.

A search argument on the outer table limits the search if an index is present. However, all the inner tables are searched completely for matching rows as part of the join.

Execution

The following steps outline how the query optimizer executes the nested iteration of joins strategy.

```
For each qualifying row in the outermost table
   Find every matching row in the next outermost table
   Determine whether the matching row meets the WHERE criteria
   For each qualifying row
      ...
      For each qualifying row in the innermost table
         Solve constant query
```