Microsoft T-SQL Performance Tuning
Part 3: Query Optimization Strategies

Adapted from "Transact-SQL Programming" By Kevin Kline, Andrew Zanevsky, and Lee Gould. Published by O'Reilly & Associates. ISBN: 1565924010

By Kevin Kline, Senior Product Architect for SQL Server

Contents

Microsoft T-SQL Performance Tuning
Part 3: Query Optimization Strategies

By Kevin Kline

Introduction

This article is the third in a series that describes a variety of performance tuning techniques that you can apply to your Microsoft SQL Server Transact-SQL programs. In many cases, you could use the graphic user interface provided in the Microsoft tools to achieve the same or similar results to those described here. For example, the graphic showplan feature of SQL Query Analyzer will show you the query plan for a query. However, this series focuses on using Transact-SQL commands as the basis for our solutions. All examples and syntax are verified for Microsoft SQL Server 2000.

Other articles in this series cover topics like:

1. Datatype tuning
2. Tuning through database and table partitioning
3. Indexing strategies
4. Query optimizer strategies
5. SHOWPLAN output and analysis
6. Optimizer hints and Join techniques
7. Query tuning tips & tricks

Overview

These articles illustrate, through examples and explain plans, useful techniques for improving queries in Microsoft SQL Server 2000. There are a number of small tips and techniques applicable in narrow classes of programming tasks. Knowing them expands your resources in performance optimization. We have chosen to use Microsoft *SHOWPLAN_ALL* output in all examples in this section, because they are more compact and still show all the critical information. (Just as an FYI, Sybase's query plans are essentially the same for our sample queries though they include some additional messages).

Note: Most examples are based on either the PUBS database or on standard system tables. I have greatly expanded the size of the tables used in the PUBS database adding tens of thousands of rows to many tables.

Subqueries Optimization

As a good rule of thumb try to replace all subqueries with joins. The optimizer may sometimes automatically *flatten out* subqueries and replace them with regular or outer joins. But it doesn't always do a good job at that. Explicit joins give the optimizer more options to choose the order of tables and find the best possible plan. When you optimize a particular query, investigate if getting rid of subqueries makes a difference.

Example

The following queries select the names of all user tables in the *pubs* database and the clustered index name for each table if one exists. If there is no clustered index, then table name still appears in the list with a dash in the clustered index column. Both queries return the same result set, but the first one uses a subquery, while the second employs an outer join. Compare the query plans produced by Microsoft SQL Server.

Subquery Solution	**Join Solution**

`SELECT st.stor_name AS 'Store',`
`       ISNULL((SELECT SUM(bs.qty)`
`          FROM big_sales AS bs`
`          WHERE bs.stor_id = st.stor_id)`
`       AS 'Books Sold'`
`FROM   stores AS st`
`WHERE  st.stor_id IN`
`   (SELECT DISTINCT stor_id`
`    FROM big_sales)`
	`SELECT st.stor_name AS 'Store',`
	`   SUM(bs.qty) AS 'Books Sold'`
	`FROM   stores    AS st`
	`JOIN   big_sales AS bs`
	`   ON bs.stor_id = st.stor_id`
	`WHERE  st.stor_id IN`
	`   (SELECT DISTINCT stor_id`
	`   FROM big_sales)`
	`GROUP BY st.stor_name`

SQL Server parse and compile time: CPU time = 28 ms, elapsed time = 28 ms. *SQL Server Execution Times:* *CPU time = 145 ms,* *elapsed time = 145 ms.*	SQL Server parse and compile time: CPU time = 50 ms, elapsed time = 54 ms. *SQL Server Execution Times:* *CPU time = 109 ms,* *elapsed time = 109 ms.*
Table 'big_sales'. Scan count 14, *logical reads 1884*, physical reads 0, read-ahead reads 0. Table 'stores'. Scan count 12, logical reads 24, physical reads 0, read-ahead reads 0.	Table 'big_sales'. Scan count 14, *logical reads 966*, physical reads 0, read-ahead reads 0. Table 'stores'. Scan count 12, logical reads 24, physical reads 0, read-ahead reads 0.

Without probing deeper, we see that the join was faster in terms of both CPU and total elapsed time, requiring almost half as many logical reads as the subquery solution. Incidentally, the result sets are the same in both cases, though the sort orders are different because the join query (with its *GROUP BY* clause) has an implicit *ORDER BY*:

```
Store                                    Books Sold
---------------------------------------- -----------
Barnum's                                 154125
Bookbeat                                 518080
Doc-U-Mat: Quality Laundry and Books     581130
Eric the Read Books                      76931
Fricative Bookshop                       259060
News & Brews                             161090
```

```
(6 row(s) affected)

Store                                    Books Sold
---------------------------------------- -----------
Eric the Read Books                      76931
Barnum's                                 154125
News & Brews                             161090
Doc-U-Mat: Quality Laundry and Books     581130
Fricative Bookshop                       259060
Bookbeat                                 518080

(6 row(s) affected)
```

Examination of the query plan of the subquery approach shows:

```
  |--Compute Scalar(DEFINE:([Expr1006]=isnull([Expr1004], 0)))
       |--Nested Loops(Left Outer Join, OUTER REFERENCES:([st].[stor_id]))
            |--Nested Loops(Inner Join, OUTER REFERENCES:([big_sales].[stor_id]))
            |    |--Stream Aggregate(GROUP BY:([big_sales].[stor_id]))
            |    |    |--Clustered Index Scan(OBJECT:([pubs].[dbo].[big_sales].
                            [UPKCL_big_sales]), ORDERED FORWARD)
            |    |--Clustered Index Seek(OBJECT:([pubs].[dbo].[stores].[UPK_storeid] AS [st],
                       SEEK:([st].[stor_id]=[big_sales].[stor_id]) ORDERED FORWARD)
            |--Stream Aggregate(DEFINE:([Expr1004]=SUM([bs].[qty])))
                 |--Clustered Index Seek(OBJECT:([pubs].[dbo].[big_sales].
                    [UPKCL_big_sales] AS [bs]),
                    SEEK:([bs].[stor_id]=[st].[stor_id]) ORDERED FORWARD)
```

Whereas in the join query, we have:

```
  |--Stream Aggregate(GROUP BY:([st].[stor_name]) DEFINE:([Expr1004]=SUM([partialagg1005])))
       |--Sort(ORDER BY:([st].[stor_name] ASC))
            |--Nested Loops(Left Semi Join, OUTER REFERENCES:([st].[stor_id]))
                 |--Nested Loops(Inner Join, OUTER REFERENCES:([bs].[stor_id]))
                 |    |--Stream Aggregate(GROUP BY:([bs].[stor_id])
                          DEFINE:([partialagg1005]=SUM([bs].[qty])))
                 |    |    |--Clustered Index Scan(OBJECT:([pubs].[dbo].[big_sales].
                                [UPKCL_big_sales] AS [bs]), ORDERED FORWARD)
                 |    |--Clustered Index Seek(OBJECT:([pubs].[dbo].[stores].
                          [UPK_storeid] AS [st]),
                          SEEK:([st].[stor_id]=[bs].[stor_id]) ORDERED FORWARD)
                 |--Clustered Index Seek(OBJECT:([pubs].[dbo].[big_sales].
                    [UPKCL_big_sales]),
                    SEEK:([big_sales].[stor_id]=[st].[stor_id]) ORDERED FORWARD)
```

A solution using a join is more efficient. It does not require additional stream aggregate that sums the *big_sales.qty* column required for subquery processing.

UNION vs. UNION ALL

Whenever possible use *UNION ALL* instead of *UNION*. The difference is that *UNION* has a "side-effect" of eliminating all duplicate rows and sorting results, which *UNION ALL* doesn't do. Selecting a distinct result requires building a temporary worktable, storing all rows in it and sorting before producing the output. (Displaying the showplan on a *SELECT DISTINCT* query will reveal a *stream aggregation* is taking place, consuming as much as 30% of the resources used to process the query.) In some cases that's exactly what you need to do, then *UNION* is your friend. But if you don't expect any duplicate rows in the result set, then use *UNION ALL*. It simply selects from one table or a join, and then selects from another, attaching results to the bottom of the first result set. *UNION ALL* requires no worktable and no sorting (unless other unrelated conditions cause that). In most cases it's much more efficient. One more potential problem with *UNION* is the danger of flooding tempdb database with a huge worktable. It may happen if you expect a large result set from a *UNION* query.

Example

The following queries select ID for all stores in the `sales` table, which ships as-is with the *pubs* database, and the ID for all stores in the `big_sales` table, a version of the `sales` table that I populated with over 70,000 rows. The only difference between the two solutions is the use of *UNION* versus *UNION ALL*. But the addition of the *ALL* keyword makes a big difference in the query plan. The first solution requires stream aggregation and sorting the results before they are returned to the client. The second query is much more efficient, especially for large tables. In this example both queries return the same result set, though in a different order. In our testing we had two temporary tables at the time of execution. Your results may vary.

UNION Solution	**UNION ALL Solution**

`SELECT stor_id FROM big_sales`
`UNION`
`SELECT stor_id FROM sales`
	`SELECT stor_id FROM big_sales`
	`UNION ALL`
	`SELECT stor_id FROM sales`

`|--Merge Join(Union)`
`       |--Stream Aggregate(GROUP BY:`
`          ([big_sales].[stor_id]))`
`       |    |--Clustered Index Scan`
`               (OBJECT:([pubs].[dbo].`
`               [big_sales].`
`               [UPKCL_big_sales]),`
`               ORDERED FORWARD)`
`       |--Stream Aggregate(GROUP BY:`
`          ([sales].[stor_id]))`
`            |--Clustered Index Scan`
`               (OBJECT:([pubs].[dbo].`
`               [sales].[UPKCL_sales]),`
`               ORDERED FORWARD)`
	`|--Concatenation`
	`       |--Index Scan (OBJECT:([pubs].[dbo].`
	`          [big_sales].[ndx_sales_ttlID]))`
	`       |--Index Scan (OBJECT:([pubs].[dbo].`
	`          [sales].[titleidind]))`

`Table 'sales'. Scan count 1, logical`
`reads 2,`` physical reads 0,`
`read-ahead reads 0.`
 	
`Table 'big_sales'. Scan count 1, logical`
`reads 463,`` physical reads 0,`
`read-ahead reads 0.`
	`Table 'sales'. Scan count 1, logical`
	`reads 1,`` physical reads 0,`
	`read-ahead reads 0.`
	
	`Table 'big_sales'. Scan count 1, logical`
	`reads 224,`` physical reads 0,`
	`read-ahead reads 0.`

Although the result sets in this example are interchangeable, you can see that the *UNION ALL* statement consumed less than half of the resources that the *UNION* statement consumed. So be sure to anticipate your result sets and in those that are already distinct, use the *UNION ALL* clause.

Functions and Expressions that Suppress Indexes

When you apply built-in functions or expressions to indexed columns, the optimizer cannot use indexes on those columns. Try to rewrite these conditions in such a way that index keys are not involved in any expression.

Examples

You have to help SQL Server remove any expressions around numeric columns that form an index. The following queries select a row from the table `jobs` by a unique key that has a unique clustered index. If you apply an expression to the column, the index is suppressed. But once you change the condition 'job_id – 2 = 0' to 'job_id = 2', the optimizer performs a *seek* operation against the clustered index.

Query With Suppressed Index	Optimized Query Using Index

`SELECT  *`
`FROM    jobs`
`WHERE   (job_id-2) = 0`
	`SELECT  *`
	`FROM    jobs`
	`WHERE   job_id = 2`

`|--Clustered Index` **`Scan`**`(OBJECT:`
`([pubs].[dbo].[jobs].`
`[PK__jobs__117F9D94]),`
`WHERE:(Convert([jobs].[job_id])-2=0))`
	`|--Clustered Index` **`Seek`**`(OBJECT:`
	`([pubs].[dbo].[jobs].`
	`[PK__jobs__117F9D94]),`
	`SEEK:([jobs].[job_id]=Convert([@1]))`
	`ORDERED FORWARD)`
	Note that a seek is much better than a scan as in the previous query.

The following table contains more examples of queries that suppress an index on columns of different type and how you can rewrite them for optimal performance.

Query With Suppressed Index	Optimized Query Using Index

`DECLARE @job_id VARCHAR(5)`
`SELECT  @job_id = '2'`
`SELECT  *`
`FROM    jobs`
`WHERE   CONVERT( VARCHAR(5),`
`job_id ) = @job_id`
	`DECLARE @job_id VARCHAR(5)`
	`SELECT  @job_id = '2'`
	`SELECT  *`
	`FROM    jobs`
	`WHERE   job_id = CONVERT(`
	`SMALLINT, @job_id )`

`SELECT  *`
`FROM    authors`
`WHERE   au_fname + ' ' + au_lname`
`= 'Johnson White'`
	`SELECT  *`
	`FROM    authors`
	`WHERE   au_fname = 'Johnson'`
	`AND au_lname = 'White'`

`SELECT  *`
`FROM    authors`
`WHERE   SUBSTRING( au_lname, 1, 2 )`
`= 'Wh'`
	`SELECT  *`
	`FROM    authors`
	`WHERE   au_lname LIKE 'Wh%'`

`CREATE INDEX employee_hire_date`
`ON employee ( hire_date )`
`GO`
`-- Get all employees hired`
`-- in the 1st quarter of 1990:`
`SELECT  *`
`FROM    employee`
`WHERE   DATEPART( year,`
`hire_date ) = 1990`
`AND DATEPART( quarter,`
	`CREATE INDEX employee_hire_date`
	`ON employee ( hire_date )`
	`GO`
	`-- Get all employees hired`
	`-- in the 1st quarter of 1990:`
	`SELECT  *`
	`FROM    employee`
	`WHERE   hire_date >= '1/1/1990'`
	`AND hire_date <  '4/1/1990'`

	hire_date) = 1	
	-- Suppose that hire_date may -- contain time other than 12AM -- Who was hired on 2/21/1990? SELECT * FROM employee **WHERE CONVERT(CHAR(10),** **hire_date, 101)** **= '2/21/1990'**	-- Suppose that hire_date may -- contain time other than 12AM -- Who was hired on 2/21/1990? SELECT * FROM employee **WHERE hire_date >= '2/21/1990'** **AND hire_date < '2/22/1990'**

UPDATE…FROM and DELETE…FROM

T-SQL offers an extension to ANSI-SQL syntax for *UPDATE* and *DELETE* commands that may be very efficient in many cases. It allows you to specify a *FROM* clause and join several tables in an *UPDATE* or *DELETE* command.

Examples

In order to update the `titleauthor` table the ANSI SQL solution below executes two subqueries, while the *UPDATE…FROM* command, shown later, replaces the subqueries with a join.

```
UPDATE titleauthor
SET    royaltyper = 90
WHERE  au_id = (SELECT  au_id FROM  authors
                WHERE au_lname = 'Ringer' AND au_fname = 'Albert')
  AND  title_id = (SELECT  title_id FROM  titles
                   WHERE title = 'Life Without Fear')
```

Which yields a very complex query plan shown here:

```
|--Clustered Index Update(OBJECT:([pubs].[dbo].[titleauthor].[UPKCL_taind]),
   SET:([titleauthor].[royaltyper]=90))
    |--Top(ROWCOUNT est 0)
          |--Merge Join(Inner Join, MERGE:([Expr1014])=([titleauthor].[title_id]),
             RESIDUAL:([titleauthor].[title_id]=[Expr1014]))
              |--Assert(WHERE:(If ([Expr1013]>1) then 0 else NULL))
              |     |--Stream Aggregate(DEFINE:([Expr1013]=Count(*),
                       [Expr1014]=ANY([titles].[title_id])))
              |          |--Index Seek(OBJECT:([pubs].[dbo].[titles].[titleind]),
                              SEEK:([titles].[title]='Life Without Fear')
                              ORDERED FORWARD)
              |--Sort(ORDER BY:([titleauthor].[title_id] ASC))
                   |--Nested Loops(Inner Join, OUTER REFERENCES:([Expr1012]))
                        |--Assert(WHERE:(If ([Expr1011]>1) then 0 else NULL))
                        |     |--Stream Aggregate(DEFINE:([Expr1011]=Count(*),
                                 [Expr1012]=ANY([authors].[au_id])))
                        |          |--Index Seek(OBJECT:([pubs].[dbo].[authors].[aunmind]),
                                        SEEK:([authors].[au_lname]='Ringer' AND
                                        [authors].[au_fname]='Albert') ORDERED FORWARD)
                        |--Index Seek(OBJECT:([pubs].[dbo].[titleauthor].[auidind]),
                           SEEK:([titleauthor].[au_id]=[Expr1012]) ORDERED FORWARD)
```

On the other hand, we can exploit the Transact-SQL extension allowing *FROM* in the *UPDATE* statement:

```
UPDATE  titleauthor
SET     royaltyper = 90
FROM    authors a, titles t
WHERE   titleauthor.au_id = a.au_id
  AND a.au_lname = 'Ringer'
  AND a.au_fname = 'Albert'
  AND titleauthor.title_id = t.title_id
  AND t.title = 'Life Without Fear'
```

Which yields a much simpler query plan:

```
|--Clustered Index Update(OBJECT:([pubs].[dbo].[titleauthor].[UPKCL_taind]),
   SET:([titleauthor].[royaltyper]=90))
    |--Top(ROWCOUNT est 0)
         |--Table Spool
              |--Nested Loops(Inner Join, OUTER REFERENCES:([titleauthor].[title_id]))
                   |--Nested Loops(Inner Join, OUTER REFERENCES:([a].[au_id]))
                   |    |--Index Seek(OBJECT:([pubs].[dbo].[authors].[aunmind] AS [a]),
                           SEEK:([a].[au_lname]='Ringer' AND [a].[au_fname]='Albert')
                           ORDERED FORWARD)
                   |    |--Index Seek(OBJECT:([pubs].[dbo].[titleauthor].[auidind]),
                           SEEK:([titleauthor].[au_id]=[a].[au_id]) ORDERED FORWARD)
                   |--Index Seek(OBJECT:([pubs].[dbo].[titles].[titleind] AS [t]),
                      SEEK:([t].[title]='Life Without Fear' AND
                      [t].[title_id]=[titleauthor].[title_id]) ORDERED FORWARD)
```

In the next example, we update a row in the *titles* table that has a specific order recorded in the *sales* table. Note that ANSI SQL solution has to execute essentially the same subquery twice, because column *title_id* is needed for the *WHERE* clause and the column *qty* is used in the *SET* clause.

ANSI SQL:

```
UPDATE  titles
SET     ytd_sales = ytd_sales + (
    SELECT  qty
    FROM    sales s
    WHERE   s.stor_id = '9999'
        AND s.ord_num = '999999' )
WHERE   title_id = (
    SELECT  title_id
    FROM    sales s
    WHERE   s.stor_id = '9999'
        AND s.ord_num = '999999' )
```

The query plan for the ANSI SQL query:

```
  |--Clustered Index Update(OBJECT:([pubs].[dbo].[titles].[UPKCL_titleidind]),
SET:([titles].[ytd_sales]=[Expr1008]))
       |--Top(1)
            |--Compute Scalar(DEFINE:([Expr1008]=[titles].[ytd_sales]+Convert([Expr1010])))
                 |--Nested Loops(Left Outer Join)
                      |--Nested Loops(Inner Join, OUTER REFERENCES:([Expr1012]))
                      |    |--Assert(WHERE:(If ([Expr1011]>1) then 0 else NULL))
                      |    |    |--Stream Aggregate(DEFINE:([Expr1011]=Count(*),
                      |    |    |  [Expr1012]=ANY([s].[title_id])))
                      |    |         |--Clustered Index Seek(OBJECT:([pubs].[dbo].
                      |    |         |  [sales].[UPKCL_sales] AS [s]), SEEK:
                      |    |         |  ([s].[stor_id]='9999' AND [s].
                      |    |         |  [ord_num]='999999') ORDERED FORWARD)
                      |    |--Clustered Index Seek(OBJECT:([pubs].[dbo].[titles].
                      |    |  [UPKCL_titleidind]), SEEK:([titles].[title_id]=
                      |    |  [Expr1012]) ORDERED FORWARD)
                      |--Assert(WHERE:(If ([Expr1009]>1) then 0 else NULL))
                           |--Stream Aggregate(DEFINE:([Expr1009]=Count(*),
                           |  [Expr1010]=ANY([s].[qty])))
                                |--Clustered Index Seek(OBJECT:([pubs].[dbo].[sales].
                                |  [UPKCL_sales] AS [s]), SEEK:([s].[stor_id]='9999'
                                |  AND [s].[ord_num]='999999') ORDERED FORWARD)
```

Now compare the expansive ANSI SQL update operation show above and the resultant query plan with the SQL Server Transact-SQL extension:

```
UPDATE  titles
SET     ytd_sales = ytd_sales + s.qty
FROM    sales s
WHERE   titles.title_id = s.title_id
    AND s.stor_id = '9999'
    AND s.ord_num = '999999'
```

This produces a query plan with only seven major operations where the ANSI SQL plan had 12:

```
  |--Clustered Index Update(OBJECT:([pubs].[dbo].[titles].[UPKCL_titleidind]),
  |  SET:([titles].[ytd_sales]=[Expr1005]))
       |--Table Spool
            |--Compute Scalar(DEFINE:([Expr1005]=[titles].[ytd_sales]+Convert([s].[qty])))
                 |--Top(ROWCOUNT est 0)
                      |--Nested Loops(Inner Join, OUTER REFERENCES:([s].[title_id]))
                           |--Clustered Index Seek(OBJECT:([pubs].[dbo].[sales].
                           |  [UPKCL_sales] AS [s]), SEEK:([s].[stor_id]='9999' AND
                           |  [s].[ord_num]='999999') ORDERED FORWARD)
                           |--Clustered Index Seek(OBJECT:([pubs].[dbo].[titles].
                           |  [UPKCL_titleidind]), SEEK:([titles].[title_id]=[s].
                           |  [title_id]) ORDERED FORWARD)
```

The following queries demonstrate you can apply the same technique to *DELETE* commands.

In ANSI SQL:

```
DELETE  sales
WHERE   EXISTS (
    SELECT 1
    FROM   titles t
    WHERE  sales.title_id = t.title_id
      AND  t.title = 'Life Without Fear' )
```

Compared to the somewhat shorter Transact-SQL extension:

```
DELETE  sales
FROM    titles t
WHERE   sales.title_id = t.title_id
  AND t.title = 'Life Without Fear'
```

SET NOCOUNT ON

The phenomenon of speeding up T-SQL code by using *SET NOCOUNT ON* was discussed at length in the last white paper; however, it bears repeating. You have already noticed that successful queries return a system message about the number of rows that they affect. In many cases you don't need this information. Command *SET NOCOUNT ON* allows you to suppress the message for all subsequent transactions in your session, until you issue the *SET NOCOUNT OFF* command. We know that this is a double negative, but T-SQL was not created by English majors.

This option has more than a cosmetic effect on the output generated by your script. It reduces the amount of information passed from the server to the client. Therefore, it helps to lower network traffic and improves the overall response time of your transactions. Time to pass a single message may be negligible, but think about a script that executes some queries in a loop and sends Kilobytes of useless information to a user.

As an example, the enclosed file has a T-SQL batch that inserts 9999 rows into the `big_sales` table.

insert into big_sales tbl.sql

When run with *SET NOCOUNT OFF*, the elapsed time was 5176 milliseconds. When run with *SET NOCOUNT ON*, the elapsed time was 1620 milliseconds! Consider adding *SET NOCOUNT ON* at the beginning of every stored procedure and script that doesn't require row counts in the output.

TOP AND SET ROWCOUNT

The *TOP* clause of the *SELECT* statement limits the number of rows returned by a single query, while the *SET ROWCOUNT* limits the number of rows affected by all subsequent queries. These commands provide great efficiencies in numerous programming tasks.

SET ROWCOUNT sets the maximum number of rows that may be affected by a *SELECT*, *INSERT*, *UPDATE*, or *DELETE* statement. The setting is immediately effective upon execution of the command and only impacts the current session. In order to remove this limit, execute *SET ROWCOUNT 0*.

Some practical tasks are much more efficient to program with *TOP* or *SET ROWCOUNT* than with standard SQL commands. Let us demonstrate it on several examples.

TOP n

One of the most popular queries in almost any database is a request for the top N items from a list. In case of the `pubs` database we could search for the top 5 best-selling titles. Compare the three solutions – with *TOP*, with *SET ROWCOUNT* and using ANSI SQL.

Pure ANSI SQL:

```
SELECT  title, ytd_sales
FROM    titles a
WHERE   ( SELECT  COUNT(*)
          FROM    titles b
          WHERE   b.ytd_sales >
                  a.ytd_sales
        ) < 5
ORDER BY ytd_sales DESC
```

The pure ANSI SQL solution executes a correlated subquery that may be inefficient, especially in this case, where there is no index on `ytd_sales` to support it. Additionally, the pure ANSI SQL command does not filter out NULL values in `ytd_sales`, nor does it discriminate in the case of a tie between multiple titles.

Using *SET ROWCOUNT*:

```
SET ROWCOUNT 5

SELECT   title, ytd_sales
FROM     titles
ORDER BY ytd_sales DESC

SET ROWCOUNT 0
```

Using *TOP n*:

```
SELECT TOP 5 title, ytd_sales
FROM     titles
ORDER BY ytd_sales DESC
```

The second solution uses *SET ROWCOUNT* to stop the *SELECT* query, while the third solutions use *TOP n* to terminate after it has found the first 5 rows. In this case, we also have an *ORDER BY* clause that forces sorting of the whole table before results may be retrieved. Both queries have virtually identical query plans. However, the key advantage of *TOP* over *SET ROWCOUNT* is that *SET* must process the worktable required by an *ORDER BY* clause where *TOP* does not.

On a large table we would create an index on *ytd_sales* to avoid sorting. The query would then use the index to find the first five rows and stop. Compare this to the first solution that would scan the whole table and execute a correlated subquery for every row. The difference in performance is negligible on a small table. But on a large table it may amount to hours of processing time for the first solution versus seconds for the last two solutions.

When determining the needs of your query, consider whether you only need to review a few of the rows retrieved. If that is the case, the *TOP* clause will be a valuable time saver.

Assumptions About Temporary Table Size

Temporary tables created at run-time within a stored procedure can be problematic. In SQL Server 7.0, the optimizer was unable to accurately estimate the size of a temporary table, instead assuming that the temporary table had only 100 rows and uses 10 data pages. Obviously, this may be wrong in many cases. Now, SQL Server 2000 is stronger with temporary tables. However, the same advice holds true – create and populate the temp table, including building indexes and constraints, before executing any conditional processing on the temp table.

One problem you may encounter is that the optimizer may refuse to recognize indexes and foreign key constraints that you build on dynamic temporary tables (those created with # or ##). It *will* recognize constraints and indexes built explicitly using a *CREATE TABLE* statement in the `tempdb` database, as well as non-foreign key constraints on dynamically built temporary tables.

Don't forget that all transactions against temporary tables are logged in the `tempdb` transaction log. Although logging has reduced overhead in `tempdb` and may be as much as four times as fast as a comparable transaction against a permanent table, you should still take this into account when sizing the `tempdb` transaction log.

In order to allow the optimizer to take actual table size into account you can use a technique where you split your code into a separate stored procedures or T-SQL batches, especially by explicitly creating the temp table and its supporting indexes before any conditional code is executed. The optimizer will then know the size of the temporary table and whether any good indexes exist before the procedure is executed and will choose the best plan based on the accurate information.
Example:

Table Created in the Same Script

```
CREATE TABLE #pub (
    pub_name VARCHAR(40) NOT NULL,
    title    VARCHAR(80) NULL,
    employee VARCHAR(51) NULL
)
GO

INSERT  #pub
SELECT  p.pub_name,
        t.title,
        e.fname + ' ' + e.lname
FROM    publishers p,
        titles     t,
        employee   e
WHERE   p.pub_id *= t.pub_id
    AND p.pub_id *= e.pub_id

CREATE CLUSTERED INDEX pubind
ON #pub ( pub_name )
```

Table Passed to a Stored Procedure

```
CREATE TABLE #pub (
    pub_name VARCHAR(40) NOT NULL,
    title    VARCHAR(80) NULL,
    employee VARCHAR(51) NULL
)
GO
CREATE PROC get_pub
AS
SELECT  *
FROM    #pub
WHERE   pub_name =
            'New Moon Books'
GO
INSERT  #pub
SELECT  p.pub_name,
        t.title,
        e.fname + ' ' + e.lname
FROM    publishers p,
        titles     t,
SELECT   *
FROM     #pub
WHERE    pub_name =
             'New Moon Books'
GO
DROP TABLE #pub
GO
```

```
            employee   e
WHERE    p.pub_id *= t.pub_id
    AND p.pub_id *= e.pub_id

CREATE CLUSTERED INDEX pubind
ON #pub ( pub_name )

EXEC get_pub WITH RECOMPILE
GO
DROP TABLE #pub
GO
DROP PROC get_pub
GO
```

```
-- plan of the highlighted SELECT:

  |--Table Scan(OBJECT:([tempdb].[dbo].
     [#pub_01D]),
     WHERE:([#pub].[pub_name]=
     'New Moon Books'))
```

```
-- plan of the highlighted SELECT:

   |--Clustered Index Scan(OBJECT:
      ([tempdb].[dbo].[#pub_01D].
      [pubind]))
```

Loop Optimization

More Invariant Operations Outside of the Loop

If you are familiar with other programming languages, then you are probably aware of loop optimization techniques. You should try to put all operations outside of the loop if they don't change inside. This reduces the amount of unnecessary repetitive work. SQL Server optimizer doesn't automatically recognize such inefficiencies and clean the code for you (compilers of some other languages do). You have to write efficient loops yourself as in the following example.

These scripts print a table of square roots for all numbers from 1 to 100.

Inefficient Loop Operations

```
SET NOCOUNT ON
DECLARE @message VARCHAR(25),
        @counter SMALLINT
SELECT  @counter = 0
WHILE @counter < 100
BEGIN
    SET @counter = @counter + 1
    SET @message = REPLICATE( '-', 25 )
    PRINT  @message
    SET @message =
       str( @counter, 10 ) +
       str( SQRT( CONVERT( FLOAT,
            @counter ) ), 10, 4 )
    PRINT  @message
END
```

```
Elapsed time: 40 ms
```

Optimized Script

```
SET NOCOUNT ON
DECLARE @separator VARCHAR(25),
        @message   VARCHAR(25),
        @counter   SMALLINT
SET  @counter = 0,
     @separator = REPLICATE( '-', 25 )
WHILE @counter < 100
BEGIN
    SET @counter = @counter + 1
    PRINT  @separator
    SET @message =
       Str( @counter, 10 ) +
       Str( SQRT( CONVERT( FLOAT,
            @counter ) ), 10, 4 )
    PRINT  @message
END
```

```
Elapsed time: 30 ms
```

The second script executes function *REPLICATE('-', 25)* only once, compared to 100 times in the first script. Results produced by both scripts are identical:

```
-------------------------
         1    1.0000
-------------------------
         2    1.4142
-------------------------
         3    1.7321
-------------------------
         4    2.0000
               . . .
-------------------------
        99    9.9499
-------------------------
       100   10.0000
```

Replace Loops With Queries

It may often be possible to replace loops with SQL queries. A single query is almost always more efficient than multiple iterations because relational databases based upon set operations.

For instance, we could rewrite the loop shown in the previous example as follows:

```
SELECT  REPLICATE( '-', 25 ) + '
' + STR( ( a.id - 1 ) * 10 + b.id, 10 )
  + STR( SQRT( CONVERT( FLOAT, ( a.id - 1 ) * 10 + b.id ) ), 10, 4 )
FROM    sysobjects a, sysobjects b
WHERE   a.id <= 10
    AND b.id <= 10
```

This command *executes in 0 ms* compared to 30 ms and 40 ms for each of the earlier scripts!

This script uses the fact that there are rows in table `sysobjects` with `id` column values of 1 through 10. If we join this table to itself and apply filters on column `id` values to take 10 rows from one instance of `sysobjects` and 10 rows from the second instance, then we get 100 rows (10 times 10). In order to produce numbers 1 through 100 we use expression (a.id–1)*10+b.id. The code may look tricky, but it returns the same results much faster than a loop.

Querying Against Composite Keys

In an earlier white paper and e-seminar, I proposed that composite keys are problematic for SQL Server. Composite indexes, as you will recall, are composed of several columns of a table. The problem is that composite indexes are used from leftmost column to right.

The following examples show that SQL Server 2000 now handles poorly ordered *WHERE* clauses much better than earlier versions of the product. That is, in earlier versions of the product SQL Server might ignore indexes when all the columns of an index were addressed in the *WHERE* clause solely because the columns were not referenced in the same order as they appeared in the index. This is no longer a problem in SQL Server 2000. However, the problem

Consider this composite index that contains three columns:

```
       ALTER TABLE add CONSTRAINT [UPKCL_sales] PRIMARY KEY CLUSTERED
       ([stor_id], [ord_num], [title_id] )
```

Depending on your *WHERE* clause conditions, SQL Server may use all or fewer columns of the index, or not use the index at all, as shown below:

Table 1. Usage of Composite Key Columns

WHERE Clause Conditions	**Query Plan**
`WHERE stor_id = @a AND ord_num = @b AND title_id = @c`	`|--Clustered Index Seek(OBJECT:([pubs].[dbo].[big_sales].[UPKCL_big_sales]), SEEK:([big_sales].[stor_id]=[@a] AND [big_sales].[ord_num]=[@b] AND [big_sales].[title_id]=[@c]) ORDERED FORWARD)`
`WHERE stor_id = @a AND ord_num = @b`	`|--Clustered Index Seek(OBJECT:([pubs].[dbo].[big_sales].[UPKCL_big_sales]), SEEK:([big_sales].[stor_id]=[@a] AND [big_sales].[ord_num]=[@b] ORDERED FORWARD)`
`WHERE ord_num = @b AND stor_id = @a`	`|--Clustered Index Seek(OBJECT:([pubs].[dbo].[big_sales].[UPKCL_big_sales]), SEEK:([big_sales].[stor_id]=[@a] AND [big_sales].[ord_num]=[@b]) ORDERED FORWARD)` Compare this to the previous query and you can see they are the same plan.
`WHERE stor_id = @a`	`|--Clustered Index Seek(OBJECT:([pubs].[dbo].[big_sales].[UPKCL_big_sales]), SEEK:([big_sales].[stor_id]=[@a] ORDERED FORWARD)`
`WHERE stor_id = @a AND title_id = @c`	`|--Clustered Index Seek(OBJECT:([pubs].[dbo].[big_sales].[UPKCL_big_sales]), SEEK:([big_sales].[stor_id]=[@a]), WHERE:([big_sales].[title_id]=[@c]) ORDERED FORWARD)` This query was not able to use the third column of the clustered index and instead had to use a separate nonclustered index on `title_id`.
`WHERE ord_num = @b AND title_id = @c`	`|--Bookmark Lookup(BOOKMARK:([Bmk1000]), OBJECT:([pubs].[dbo].[big_sales])) |--Index Seek(OBJECT:([pubs].[dbo].[big_sales].[ndx_sales_ttlID]), SEEK:([big_sales].[title_id]=[@c]), WHERE:([big_sales].[ord_num]=[@b]) ORDERED FORWARD)` This query was not able to use the clustered index at all, but was able to find a highly performant alternate plan.

The key point to remember is that you should know the order of columns appearing within a composite index. Once you know the order of the columns, you should always structure your *WHERE* clause to analyze columns starting with the leftmost column in the composite index and work towards the right.

Summary

This white paper has presented a collection of tips and trick to help you get the most out of your queries on a SQL Server 2000 database. Some ideas presented in the white paper include:

- Subquery optimization
- *UNION* versus *UNION ALL*
- Functions and expressions that suppress indexes

- The advantages of *UPDATE…FROM* and *DELETE…FROM* over ANSI standard syntax
- *SET NOCOUNT ON*
- *TOP* and *SET ROWCOUNT*
- Temporary table considerations
- Loop optimization
- Recap of querying against concatenated keys

And the key thing to remember in summary is to test, test, retest!

About Quest Software