Now, let's create the sample tables and check all these solutions for efficiency:

01.SET NOCOUNT ON

02.GO

03.DROP TABLE [20090716\_cross].table1

04.DROP TABLE [20090716\_cross].table2

05.DROP SCHEMA [20090716\_cross]

06.GO

07.CREATE SCHEMA [20090716\_cross]

08.CREATE TABLE table1

09.        (

10.        id INT NOT NULL PRIMARY KEY,

11.        row\_count INT NOT NULL

12.        )

13.CREATE TABLE table2

14.        (

15.        id INT NOT NULL PRIMARY KEY,

16.        value VARCHAR(20) NOT NULL

17.        )

18.GO

19.BEGIN TRANSACTION

20.DECLARE @cnt INT

21.SET @cnt = 1

22.WHILE @cnt <= 100000

23.BEGIN

24.        INSERT

25.        INTO    [20090716\_cross].table2 (id, value)

26.        VALUES  (@cnt, 'Value ' + CAST(@cnt AS VARCHAR))

27.        SET @cnt = @cnt + 1

28.END

29.INSERT

30.INTO    [20090716\_cross].table1 (id, row\_count)

31.SELECT  TOP 5

32.        id, id % 2 + 1

33.FROM    [20090716\_cross].table2

34.ORDER BY

35.        id

36.COMMIT

37.GO

table2 contains **100,000** rows with sequential ids.

table1 contains the following:

|  |  |
| --- | --- |
| **id** | **row\_count** |
| 1 | 2 |
| 2 | 1 |
| 3 | 2 |
| 4 | 1 |
| 5 | 2 |

Now let's run the first query (with COUNT):

01.SELECT  \*

02.FROM    [20090716\_cross].table1 t1

03.JOIN    (

04.        SELECT  t2o.\*,

05.                (

06.                SELECT  COUNT(\*)

07.                FROM    [20090716\_cross].table2 t2i

08.                WHERE   t2i.id <= t2o.id

09.                ) AS rn

10.        FROM    [20090716\_cross].table2 t2o

11.        ) t2

12.ON      t2.rn <= t1.row\_count

13.ORDER BY

14.        t1.id, t2.id

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **row\_count** | **id** | **value** | **rn** |
| 1 | 2 | 1 | Value 1 | 1 |
| 1 | 2 | 2 | Value 2 | 2 |
| 2 | 1 | 1 | Value 1 | 1 |
| 3 | 2 | 1 | Value 1 | 1 |
| 3 | 2 | 2 | Value 2 | 2 |
| 4 | 1 | 1 | Value 1 | 1 |
| 5 | 2 | 1 | Value 1 | 1 |
| 5 | 2 | 2 | Value 2 | 2 |
| 8 rows fetched in 0.0000s (498.4063s) | | | | | |

Table 'table1'. Scan count 2, logical reads 200002, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 100000, logical reads 8389920, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'table2'. Scan count 4, logical reads 1077, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

CPU time = 947655 ms, elapsed time = 498385 ms.

This query, as was expected, is very unoptimal. It runs for more than **500** seconds.

Here's the query plan:

SELECT

Sort

Compute Scalar

Parallelism (Gather Streams)

Inner Join (Nested Loops)

Inner Join (Nested Loops)

Clustered Index Scan ([20090716\_cross].[table2])

Compute Scalar

Stream Aggregate

Eager Spool

Clustered Index Scan ([20090716\_cross].[table2])

Clustered Index Scan ([20090716\_cross].[table1])

For each row selected from table2, it counts all previous rows again an again, never recording the intermediate result. The complexity of such an algorithm is O(n^2), that's why it takes so long.

Let's run he second query, which uses ROW\_NUMBER():

01.SELECT  \*

02.FROM    [20090716\_cross].table1 t1

03.JOIN    (

04.        SELECT  t2o.\*, ROW\_NUMBER() OVER (ORDER BY id) AS rn

05.        FROM    [20090716\_cross].table2 t2o

06.        ) t2

07.ON      t2.rn <= t1.row\_count

08.ORDER BY

09.        t1.id, t2.id

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **row\_count** | **id** | **value** | **rn** |
| 1 | 2 | 1 | Value 1 | 1 |
| 1 | 2 | 2 | Value 2 | 2 |
| 2 | 1 | 1 | Value 1 | 1 |
| 3 | 2 | 1 | Value 1 | 1 |
| 3 | 2 | 2 | Value 2 | 2 |
| 4 | 1 | 1 | Value 1 | 1 |
| 5 | 2 | 1 | Value 1 | 1 |
| 5 | 2 | 2 | Value 2 | 2 |
| 8 rows fetched in 0.0006s (0.5781s) | | | | | |

Table 'Worktable'. Scan count 1, logical reads 214093, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'table2'. Scan count 1, logical reads 522, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'table1'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

CPU time = 578 ms, elapsed time = 579 ms.

This is much faster, only **0.5 ms**.

Let's look into the query plan:

SELECT

Inner Join (Nested Loops)

Clustered Index Scan ([20090716\_cross].[table1])

Lazy Spool

Sequence Project (Compute Scalar)

Compute Scalar

Segment

Clustered Index Scan ([20090716\_cross].[table2])

This is much better, since this query plan keeps the intermediate results while calculating the ROW\_NUMBER.

However, it still calculates ROW\_NUMBERs for all 100,000 of rows in table2, then puts them into a temporary index over rn created by Lazy Spool, and uses this index in a nested loop to range the rns for each row from table1.

Calculating and indexing all ROW\_NUMBERs is quite expensive, that's why we see **214,093** logical reads in the query statistics.

Finally, let's try a CROSS APPLY:

01.SELECT  \*

02.FROM    [20090716\_cross].table1 t1

03.CROSS APPLY

04.        (

05.        SELECT  TOP (t1.row\_count) \*

06.        FROM    [20090716\_cross].table2

07.        ORDER BY

08.                id

09.        ) t2

10.ORDER BY

11.        t1.id, t2.id

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **row\_count** | **id** | **value** |
| 1 | 2 | 1 | Value 1 |
| 1 | 2 | 2 | Value 2 |
| 2 | 1 | 1 | Value 1 |
| 3 | 2 | 1 | Value 1 |
| 3 | 2 | 2 | Value 2 |
| 4 | 1 | 1 | Value 1 |
| 5 | 2 | 1 | Value 1 |
| 5 | 2 | 2 | Value 2 |
| 8 rows fetched in 0.0004s (0.0008s) | | | | |

Table 'table2'. Scan count 5, logical reads 10, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'table1'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 1 ms.

This query is instant, as it should be.

The plan is quite simple:

SELECT

Inner Join (Nested Loops)

Clustered Index Scan ([20090716\_cross].[table1])

Top

Clustered Index Scan ([20090716\_cross].[table2])

For each row from table1, it just takes first row\_count rows from table2. So simple and so fast.

**Summary:**

While most queries which employ CROSS APPLY can be rewritten using an INNER JOIN, CROSS APPLY can yield better execution plan and better performance, since it can limit the set being joined yet before the join occurs.