MANNING

SAMPLE CHAPTER

SQL SERVER MVP DEEP DIVES

EDITED BY

Paul Nielsen • Kalen Delaney • Greg Low • Adam Machanic • Paul S. Randal • Kimberly L. Tripp

MVP CONTRIBUTORS

John Baird • Bob Beauchemin • Itzik Ben-Gan • Glenn Berry • Aaron Bertrand • Phil Brammer • Robert C. Cain • Michael Coles • John Paul Cook • Hilary Cotter • Louis Davidson • Christopher Fairbairn • Rob Farley • Denis Gobo • Bill Graziano • Dan Guzman • Paul Ibison • Tibor Karaszi • Kathi Kellenberger • Don Kiely • Kevin Kline • Hugo Kornelis • Alex Kuznetsov • Matija Lah • Cristian Lefter • Andy Leonard • Greg Linwood • Bruce Loehle-Conger • Brad McGehee • Paul Nielsen • Pawel Potasinski • Matthew Roche • Dejan Sarka • Edwin Sarmiento • Gail Shaw • Linchi Shea • Richard Siddaway • Jasper Smith • Erland Sommarskog • Scott Stauffer • Tom van Stiphout • Gert-Jan Strik • Ron Talmage • William R. Vaughn • Joe Webb • John Welch • Erin Welker • Allen White

WAR child

Author royalties go to support War Child International

SQL Server MVP
Deep Dives

Edited by
Paul Nielsen, Kalen Delaney , Greg Low,
Adam Machanic, Paul S. Randal, Kimberly L. Tripp

Chapter 5

brief contents

5 Gaps and islands

Itzik Ben-Gan

This chapter describes problems known as gaps and islands and their solutions. I start out with a description of gaps and islands problems, describe the common variations on the problems, and provide sample data and desired results. Then I move on to ways of handling gaps and islands problems, covering multiple solutions to each problem and discussing both their logic and performance. The chapter concludes with a summary of the solutions.

Description of gaps and islands problems

Gaps and islands problems involve missing values in a sequence. Solving the gaps problem requires finding the ranges of missing values, whereas solving the islands problem involves finding the ranges of existing values.

The sequences of values in gaps and islands problems can be numeric, such as a sequence of order IDs, some of which were deleted. An example of the gaps problem in this case would be finding the ranges of deleted order IDs. An example of the islands problem would be finding the ranges of existing IDs.

The sequences involved can also be temporal, such as order dates, some of which are missing due to inactive periods (weekends, holidays). Finding periods of inactivity is an example of the gaps problem, and finding periods of activity is an example of the islands problem. Another example of a temporal sequence is a process that needs to report every fixed interval of time that it is online (for example, every 4 hours). Finding unavailability and availability periods is another example of gaps and islands problems.

Besides varying in terms of the data type of the values (numeric and temporal), sequences can also vary in terms of the uniqueness of values. For example, the sequence can have unique values, that is, unique keys, or non-unique values, that is, order dates. When discussing solutions, for simplicity's sake I'll present them against a numeric sequence with unique values. I'll explain the changes you need to make to apply the solution to the variants.

Sample data and desired results

To demonstrate the logical aspects of the solutions to the gaps and islands problems, I'll use a table called NumSeq. Run the code in listing 1 to create the table NumSeq and populate it with sample data.

Listing 1 Code creating and populating table NumSeq

```
SET NOCOUNT ON;
USE tempdb;

-- dbo.NumSeq (numeric sequence with unique values, interval: 1)
IF OBJECT_ID('dbo.NumSeq', 'U') IS NOT NULL
  DROP TABLE dbo.NumSeq;

CREATE TABLE dbo.NumSeq
(
  seqval INT NOT NULL CONSTRAINT PK_NumSeq PRIMARY KEY
);

INSERT INTO dbo.NumSeq(seqval) VALUES(2);
INSERT INTO dbo.NumSeq(seqval) VALUES(3);

INSERT INTO dbo.NumSeq(seqval) VALUES(11);
INSERT INTO dbo.NumSeq(seqval) VALUES(12);
INSERT INTO dbo.NumSeq(seqval) VALUES(13);

INSERT INTO dbo.NumSeq(seqval) VALUES(31);

INSERT INTO dbo.NumSeq(seqval) VALUES(33);
INSERT INTO dbo.NumSeq(seqval) VALUES(34);
INSERT INTO dbo.NumSeq(seqval) VALUES(35);

INSERT INTO dbo.NumSeq(seqval) VALUES(42);
```

The column `seqval` is a unique column that holds the sequence values. The sample data represents a sequence with ten unique values with four gaps and five islands. The solutions to the gaps problem should return the ranges of missing values, as table 1 shows.

start_range	end_range
4	10
14	30
32	32
36	41

Table 1 Desired result for gaps problem

The solutions to the islands problem should return the ranges of existing values, as table 2 shows.

When discussing performance, I'll provide information based on tests I did against a table called BigNumSeq with close to 10,000,000 rows, representing a numeric sequence with unique values, with 10,000 gaps. Run the code in listing 2 to create the

start_range	end_range
2	3
11	13
31	31
33	35
42	42

Table 2 Desired result for islands problem

BigNumSeq table and populate it with sample data. Note that it may take a few minutes for this code to finish.

Listing 2 Code creating and populating the BigNumSeq table

```
-- dbo.BigNumSeq (big numeric sequence with unique values, interval: 1)
IF OBJECT_ID('dbo.BigNumSeq', 'U') IS NOT NULL
  DROP TABLE dbo.BigNumSeq;

CREATE TABLE dbo.BigNumSeq
(
  seqval INT NOT NULL CONSTRAINT PK_BigNumSeq PRIMARY KEY
);

-- Populate table with values in the range 1 through to 10,000,000
-- with a gap every 1000 (total 10,000 gaps)
WITH
L0   AS(SELECT 1 AS c UNION ALL SELECT 1),
L1   AS(SELECT 1 AS c FROM L0 AS A, L0 AS B),
L2   AS(SELECT 1 AS c FROM L1 AS A, L1 AS B),
L3   AS(SELECT 1 AS c FROM L2 AS A, L2 AS B),
L4   AS(SELECT 1 AS c FROM L3 AS A, L3 AS B),
L5   AS(SELECT 1 AS c FROM L4 AS A, L4 AS B),
Nums AS(SELECT ROW_NUMBER() OVER(ORDER BY (SELECT 0)) AS n FROM L5)
INSERT INTO dbo.BigNumSeq WITH(TABLOCK) (seqval)
  SELECT n
  FROM Nums
  WHERE n <= 10000000
    AND n % 1000 <> 0;
```

Now that you understand the problems and the desired results and have created and populated the sample tables, you are ready to start working on solutions. I encourage you to come up with your own solutions and test and tune them against the big table before you look at my solutions.

Solutions to gaps problem

I'll present four solutions to the gaps problem: two using subqueries, one using ranking calculations, and (how can I not?) one using the notorious cursors. I'll conclude this section with a performance summary.

Gaps—solution 1 using subqueries

Listing 3 shows the first solution to the gaps problem.

Listing 3 Gaps—solution 1 using subqueries

```
SELECT
  seqval + 1 AS start_range,
  (SELECT MIN(B.seqval)
   FROM dbo.NumSeq AS B
   WHERE B.seqval > A.seqval) - 1 AS end_range
FROM dbo.NumSeq AS A
WHERE NOT EXISTS
  (SELECT *
   FROM dbo.NumSeq AS B
   WHERE B.seqval = A.seqval + 1)
  AND seqval < (SELECT MAX(seqval) FROM dbo.NumSeq);
```

This solution is based on subqueries. In order to understand it you should first focus on the filtering activity in the `WHERE` clause and then proceed to the activity in the `SELECT` list. The purpose of the `NOT EXISTS` predicate in the `WHERE` clause is to filter only points that are a point before a gap. You can identify a point before a gap when you see that for such a point, the value plus 1 doesn't exist in the sequence. The purpose of the second predicate in the `WHERE` clause is to filter out the maximum value from the sequence because it represents the point before infinity, which does not concern us.

That filter left only points that are a point before a gap. What remains is to relate to each such point the next point that exists in the sequence. A subquery in the `SELECT` list is used to return for each point the minimum value that is greater than the current point. This is one way to implement the concept of *next* in SQL.

Each pair is made of a point before a gap, and the next point that exists in the sequence represents a pair of values bounding a gap. To get the start and end points of the gap, add 1 to the point before the gap, and subtract 1 from the point after the gap.

This solution performs well, and I must say that I'm a bit surprised by the efficiency of the execution plan for this query. Against the BigNumSeq table, this solution ran for only about 8 seconds on my system, and incurred 62,262 logical reads. To filter points before gaps, I expected that the optimizer would apply an index seek per each of the 10,000,000 rows, which could have meant over 30 million random reads. Instead, it performed two ordered scans of the index, costing a bit over 30,000 sequential reads, and applying a merge join between the two inputs for this purpose. For each of the 10,000 rows that remain (for the 10,000 points before gaps), an index seek is used to return the next existing point. Each such seek costs 3 random reads for the three levels in the index b-tree, amounting in total to about 30,000 random reads. The total number of logical reads is 62,262 as mentioned earlier. If the number of gaps is fairly small, as in our case, this solution performs well.

To apply the solution to a temporal sequence, instead of using +1 or -1 to add or subtract the interval 1 from the integer sequence value, use the DATEADD function to add or subtract the applicable temporal interval from the temporal sequence.

To apply the solution to a sequence with duplicates, you have several options. One option is to substitute the reference to the table that is aliased as A with a derived table based on a query that removes duplicates: (SELECT DISTINCT seqval FROM dbo.TempSeq) AS A. Another option is to add a DISTINCT clause to the SELECT list.

Gaps—solution 2 using subqueries

Listing 4 shows the second solution for the gaps problem.

Listing 4 Gaps—solution 2 using subqueries

```
SELECT cur + 1 AS start_range, nxt - 1 AS end_range
   FROM (SELECT
           seqval AS cur,
           (SELECT MIN(B.seqval)
            FROM dbo.NumSeq AS B
            WHERE B.seqval > A.seqval) AS nxt
         FROM dbo.NumSeq AS A) AS D
WHERE nxt - cur > 1;
```

As with solution 1, this solution is also based on subqueries. The logic of this solution is straightforward. The query that defines the derived table D uses a subquery in the SELECT list to produce current-next pairs. That is, for each current value, the subquery returns the minimum value that is greater than the current. The current value is aliased as cur, and the next value is aliased as nxt. The outer query filters the pairs in which the difference is greater than 1, because those pairs bound gaps. By adding 1 to cur and subtracting 1 from nxt, you get the gap starting and ending points. Note that the maximum value in the table (the point before infinity) will get a NULL back from the subquery, the difference between cur and nxt will yield a NULL, the predicate NULL > 1 will yield UNKNOWN, and the row will be filtered out. That's the behavior we want for the point before infinity. It is important to always think about the three-valued logic and ensure that you get desired behavior; and if you don't, you need to add logic to get the behavior you are after.

The performance measures for this solution are not as good as solution 1. The plan for this query shows that the index is fully scanned to retrieve the current values, amounting to about 16,000 sequential reads. Per each of the 10,000,000 current values, an index seek operation is used to return the next value to produce the current-next pairs. Those seeks are the main contributor to the cost of this plan, as they amount to about 30,000,000 random reads. This query ran for about 48 seconds on my system, and incurred 31,875,478 logical reads.

To apply the solution to a temporal sequence, use the DATEADD function instead of using +1 and -1 and the DATEDIFF function to calculate the difference between cur and nxt.

To apply the solution to a sequence with duplicates, use guidelines similar to those in the previous solution.

Gaps—solution 3 using ranking functions

Listing 5 shows the third solution to the gaps problem.

Listing 5 Gaps—solution 3 using ranking functions

```
WITH C AS
(
  SELECT seqval, ROW_NUMBER() OVER(ORDER BY seqval) AS rownum
  FROM dbo.NumSeq
)
SELECT Cur.seqval + 1 AS start_range, Nxt.seqval - 1 AS end_range
FROM C AS Cur
  JOIN C AS Nxt
    ON Nxt.rownum = Cur.rownum + 1
WHERE Nxt.seqval - Cur.seqval > 1;
```

This solution implements the same logic as the previous solution, but it uses a different technique to produce current-next pairs. This solution defines a common table expression (CTE) called *C* that calculates row numbers to position sequence values. The outer query joins two instances of the CTE, one representing current values, and the other representing next values. The join condition matches current-next values based on an offset of 1 between their row numbers. The filter in the outer query then keeps only those pairs with a difference greater than 1.

The plan for this solution shows that the optimizer does two ordered scans of the index to produce row numbers for current and next values. The optimizer then uses a merge join to match current-next values. The two ordered scans of the index are not expensive compared to the seek operations done in the previous solution. However, the merge join appears to be many-to-many and is a bit expensive. This solution ran for 24 seconds on my system, and incurred 32,246 logical reads.

To apply the solution to a temporal sequence, use the `DATEADD` function instead of using +1 and -1, and the `DATEDIFF` function to calculate the difference between `Nxt.seqval` and `Cur.seqval`.

To apply the solution to a sequence with duplicates, use the `DENSE_RANK` function instead of `ROW_NUMBER`, and add `DISTINCT` to the `SELECT` clause of the inner query.

Gaps—solution 4 using cursors

Listing 6 shows the fourth solution to the gaps problem.

Listing 6 Gaps—solution 4 using cursors

```
SET NOCOUNT ON;

DECLARE @seqval AS INT, @prvseqval AS INT;
DECLARE @Gaps TABLE(start_range INT, end_range INT);
DECLARE C CURSOR FAST_FORWARD FOR
  SELECT seqval FROM dbo.NumSeq ORDER BY seqval;

OPEN C;

FETCH NEXT FROM C INTO @prvseqval;
IF @@FETCH_STATUS = 0 FETCH NEXT FROM C INTO @seqval;

WHILE @@FETCH_STATUS = 0
BEGIN
  IF @seqval - @prvseqval > 1
    INSERT INTO @Gaps(start_range, end_range)
      VALUES(@prvseqval + 1, @seqval - 1);

  SET @prvseqval = @seqval;
  FETCH NEXT FROM C INTO @seqval;
END

CLOSE C;

DEALLOCATE C;

SELECT start_range, end_range FROM @Gaps;
```

This solution is based on cursors, which represent the ordered sequence values. The code fetches the ordered sequence values from the cursor one at a time, and identifies a gap whenever the difference between the previous and current values is greater than one. When a gap is found, the gap information is stored in a table variable. When finished with the cursor, the code queries the table variable to return the gaps.

This solution is a good example of the overhead involved with using cursors. The I/O work involved here is a single ordered scan of the index, amounting to 16,123 logical reads. However, there's overhead involved with each record manipulation that is multiplied by the 10,000,000 records involved. This code ran for 250 seconds on my system and is the slowest of the solutions I tested for the gaps problem.

To apply the solution to a temporal sequence, use the `DATEADD` function instead of using +1 and -1 and the `DATEDIFF` function to calculate the difference between `@seqval` and `@prvseqval`.

Nothing must be added to handle a sequence with duplicates.

Performance summary for gaps solutions

Table 3 shows a summary of the performance measures I got for the four solutions presented.

Table 3 Performance summary of solutions to gaps problem

Solution	Runtime in seconds	Logical reads
Solution 1—using subqueries	8	62,262
Solution 2—using subqueries	48	31,875,478
Solution 3—using ranking functions	24	32,246
Solution 4—using cursors	250	16,123

As you can see, the first solution using subqueries is by far the fastest, whereas the fourth solution using cursors is by far the slowest.

Solutions to islands problem

I'll present four solutions to the islands problem: using subqueries and ranking calculations; using a group identifier based on subqueries; using a group identifier based on ranking calculations; and using cursors. I'll also present a variation on the islands problem, and then conclude this section with a performance summary.

Islands—solution 1 using subqueries and ranking calculations

Listing 7 shows the first solution to the islands problem.

Listing 7 Islands—solution 1 using subqueries and ranking calculations

```
WITH StartingPoints AS
(
  SELECT seqval, ROW_NUMBER() OVER(ORDER BY seqval) AS rownum
  FROM dbo.NumSeq AS A
  WHERE NOT EXISTS
    (SELECT *
     FROM dbo.NumSeq AS B
     WHERE B.seqval = A.seqval - 1)
),
EndingPoints AS
(
  SELECT seqval, ROW_NUMBER() OVER(ORDER BY seqval) AS rownum
  FROM dbo.NumSeq AS A
  WHERE NOT EXISTS
    (SELECT *
     FROM dbo.NumSeq AS B
     WHERE B.seqval = A.seqval + 1)
)
SELECT S.seqval AS start_range, E.seqval AS end_range
FROM StartingPoints AS S
  JOIN EndingPoints AS E
    ON E.rownum = S.rownum;
```

This solution defines two CTEs—one called `StartingPoints`, representing starting points of islands, and one called `EndingPoints`, representing ending points of islands. A point is identified as a starting point if the value minus 1 doesn't exist in the sequence. A point is identified as an ending point if the value plus 1 doesn't exist in the sequence. Each CTE also assigns row numbers to position the starting/ending points. The outer query joins the CTEs by matching starting and ending points based on equality between their row numbers.

This solution is straightforward, and also has reasonable performance when the sequence has a fairly small number of islands. The plan for this solution shows that the index is scanned four times in order—two ordered scans and a merge join are used to identify starting points and calculate their row numbers, and similar activity to identify ending points. A merge join is then used to match starting and ending points.

This query ran for 17 seconds against the BigNumSeq table, and incurred 64,492 logical reads.

To apply the solution to a temporal sequence, use `DATEADD` to add or subtract the appropriate interval instead of using +1 and -1.

As for a sequence with duplicates, the existing solution works as is; no changes are needed.

Islands—solution 2 using group identifier based on subqueries

Listing 8 shows the second solution to the islands problem.

Listing 8 Islands—solution 2 using group identifier based on subqueries

```
SELECT MIN(seqval) AS start_range, MAX(seqval) AS end_range
FROM (SELECT seqval,
        (SELECT MIN(B.seqval)
         FROM dbo.NumSeq AS B
         WHERE B.seqval >= A.seqval
           AND NOT EXISTS
             (SELECT *
              FROM dbo.NumSeq AS C
              WHERE C.seqval = B.seqval + 1)) AS grp
      FROM dbo.NumSeq AS A) AS D
GROUP BY grp;
```

This solution uses subqueries to produce, for each point, a group identifier (`grp`). The group identifier is a value that uniquely identifies the island. Such a value must be the same for all members of the island, and different than the value produced for other islands. When a group identifier is assigned to each member of the island, you can group the data by this identifier, and return for each group the minimum and maximum values.

The logic used to calculate the group identifier for each current point is this: return the next point (inclusive) that is also a point before a gap. Try to apply this logic and figure out the group identifier that will be produced for each point in the NumSeq table. For the values 2, 3 the `grp` value that will be produced will be 3, because for both values the next point (inclusive) before a gap is 3. For the values 11, 12, 13, the next point before a gap is 13, and so on. Recall the techniques used previously to identify the next point—the minimum value that is greater than the current. In our case, the next point should be inclusive; therefore you should use the >= operator instead of the > operator. To identify a point before a gap, use a `NOT EXISTS` predicate that ensures that the value plus 1 doesn't exist. Now combine the two techniques and you get the next point before a gap. The query defining the derived table `D` does all the work of producing the group identifiers. The outer query against `D` is then left to group the data by `grp`, and return for each island the first and last sequence values.

If you think that the logic of this solution is complex, I'm afraid its performance will not comfort you. The plan for this query is horrible—it scans the 10,000,000 sequence values, and for each of those values it does expensive work that involves a merge join between two inputs that identifies the next point before a gap. I aborted the execution of this query after letting it run for about 10 minutes.

Islands—solution 3 using group identifier based on ranking calculations

Listing 9 shows the third solution.

Listing 9 Islands—solution 3 using group identifier based on ranking calculations

```
SELECT MIN(seqval) AS start_range, MAX(seqval) AS end_range
FROM (SELECT seqval, seqval - ROW_NUMBER() OVER(ORDER BY seqval) AS grp
      FROM dbo.NumSeq) AS D
GROUP BY grp;
```

This solution is similar to solution 2 in the sense that it also calculates a group identifier that uniquely identifies the island; however this solution is dramatically simpler and more efficient. The group identifier is calculated as the difference between the sequence value and a row number that represents the position of the sequence value. Within an island, both the sequence values and the row numbers keep incrementing by the fixed interval of 1. Therefore, the difference between the two is constant. When moving on to the next island, the sequence value increases by more than 1, while the row number increases by 1. Therefore, the difference becomes higher than the previous island. Each island will have a higher difference than the previous island. Note that the last two characteristics that I described of this difference (constant within an island, and different for different islands) are the exact requirements we had for the group identifier. Hence, you can use this difference as the group identifier. As in solution 2, after the group identifier is calculated, you group the data by this identifier, and for each group, return the minimum and maximum values.

For performance, this solution is efficient because it required a single ordered scan of the index to calculate the row numbers, and an aggregate operator for the grouping activity. It ran for 10 seconds against the BigNumSeq table on my system and incurred 16,123 logical reads.

To apply this solution to a temporal sequence, subtract from the sequence value as many temporal intervals as the row number representing the position of the sequence value. The logic is similar to the integer sequence, but instead of getting a unique integer per island, you will get a unique time stamp for each island. For example, if the interval of the temporal sequence is 4 hours, substitute the expression `seqval - ROW_NUMBER() OVER(ORDER BY seqval) AS grp` in listing 9 with the expression `DATEADD(hour, -4 * ROW_NUMBER() OVER(ORDER BY seqval), seqval) AS grp`.

For a numeric sequence with duplicates, the trick is to have the same rank for all duplicate occurrences. This can be achieved by using the `DENSE_RANK` function instead of `ROW_NUMBER`. Simply substitute the expression `seqval - ROW_NUMBER() OVER(ORDER BY seqval) AS grp` in listing 9 with the expression `seqval - DENSE_RANK() OVER(ORDER BY seqval) AS grp`.

Islands—solution 4 using cursors

For kicks, and for the sake of completeness, listing 10 shows the fourth solution, which is based on cursors.

Listing 10 Islands—solution 4 using cursors

```
SET NOCOUNT ON;

DECLARE @seqval AS INT, @prvseqval AS INT, @first AS INT;
DECLARE @Islands TABLE(start_range INT, end_range INT);

DECLARE C CURSOR FAST_FORWARD FOR
  SELECT seqval FROM dbo.NumSeq ORDER BY seqval;

OPEN C;

FETCH NEXT FROM C INTO @seqval;
SET @first = @seqval;
SET @prvseqval = @seqval;

WHILE @@FETCH_STATUS = 0
BEGIN
  IF @seqval - @prvseqval > 1
  BEGIN
    INSERT INTO @Islands(start_range, end_range)
      VALUES(@first, @prvseqval);
    SET @first = @seqval;
  END

  SET @prvseqval = @seqval;
  FETCH NEXT FROM C INTO @seqval;
END

IF @first IS NOT NULL
  INSERT INTO @Islands(start_range, end_range)
    VALUES(@first, @prvseqval);

CLOSE C;

DEALLOCATE C;

SELECT start_range, end_range FROM @Islands;
```

The logic of this solution is to scan the sequence values in order, and at each point, check if the difference between the previous value and the current is greater than 1. If it is, you know that the last island ended with the previous value closed, and a new one just started with the current value. As expected, the performance of this solution is not good—it ran for 217 seconds on my system.

Variation on the islands problem

Before I present a performance summary for the different solutions, I want to show a variation on the islands problem, and a solution based on the fast technique with the ranking calculation. I'll use a table called *T1* with new sample data to discuss this problem. Run the code in listing 11 to create the table T1 and populate it with sample data.

Listing 11 Code creating and populating table T1

```
SET NOCOUNT ON;
USE tempdb;

IF OBJECT_ID('dbo.T1') IS NOT NULL
  DROP TABLE dbo.T1;

CREATE TABLE dbo.T1
(
  id  INT         NOT NULL PRIMARY KEY,
  val VARCHAR(10) NOT NULL
);
GO

INSERT INTO dbo.T1(id, val) VALUES(2, 'a');
INSERT INTO dbo.T1(id, val) VALUES(3, 'a');
INSERT INTO dbo.T1(id, val) VALUES(5, 'a');
INSERT INTO dbo.T1(id, val) VALUES(7, 'b');
INSERT INTO dbo.T1(id, val) VALUES(11, 'b');
INSERT INTO dbo.T1(id, val) VALUES(13, 'a');
INSERT INTO dbo.T1(id, val) VALUES(17, 'a');
INSERT INTO dbo.T1(id, val) VALUES(19, 'a');
INSERT INTO dbo.T1(id, val) VALUES(23, 'c');
INSERT INTO dbo.T1(id, val) VALUES(29, 'c');
INSERT INTO dbo.T1(id, val) VALUES(31, 'a');
INSERT INTO dbo.T1(id, val) VALUES(37, 'a');
INSERT INTO dbo.T1(id, val) VALUES(41, 'a');
INSERT INTO dbo.T1(id, val) VALUES(43, 'a');
INSERT INTO dbo.T1(id, val) VALUES(47, 'c');
INSERT INTO dbo.T1(id, val) VALUES(53, 'c');
INSERT INTO dbo.T1(id, val) VALUES(59, 'c');
```

This variation involves two attributes: one represents a sequence (`id` in our case), and the other represents a kind of status (`val` in our case). The task at hand is to identify the starting and ending sequence point (`id`) of each consecutive status (`val`) segment. Table 4 shows the desired output.

This is a variation on the islands problem. Listing 12 shows the solution to this problem.

mn	mx	val
2	5	A
7	11	B
13	19	A
23	29	C
31	43	A
47	59	C

Table 4 Desired result for variation on the islands problem

Listing 12 Solution to variation on the islands problem

```
WITH C AS
(
  SELECT id, val,
    ROW_NUMBER() OVER(ORDER BY id)
      - ROW_NUMBER() OVER(ORDER BY val, id) AS grp
  FROM dbo.T1
)
SELECT MIN(id) AS mn, MAX(id) AS mx, val
FROM C
GROUP BY val, grp
ORDER BY mn;
```

The query defining the CTE C calculates the difference between a row number representing `id` ordering and a row number representing `val`, `id` ordering, and calls that difference `grp`. Within each consecutive status segment, the difference will be constant, and smaller than the difference that will be produced for the next consecutive status segment. In short, the combination of status (`val`) and that difference uniquely identifies a consecutive status segment. What's left for the outer query is to group the data by `val` and `grp`, and return for each group the status (`val`), and the minimum and maximum `id`s.

Performance summary for islands solutions

Table 5 shows a summary of the performance measures I got for the four solutions presented.

Table 5 Performance summary of solutions to islands problem

Solution	Runtime in seconds	Logical reads
Solution 1—using subqueries and ranking calculations	17	64,492
Solution 2—using group identifier based on subqueries	Aborted after 10 minutes	
Solution 3—using group identifier based on ranking calculations	10	16,123
Solution 4—using cursors	217	16,123

As you can see, solution 3 that uses the row number function to calculate a group identifier is the fastest. Solution 4 using a cursor is slow, but not the slowest. Solution 2 that uses subqueries to calculate a group identifier is the slowest.

Summary

In this chapter I explained gaps and islands problems and provided different solutions to those problems. I compared the performance of the different solutions and as you could see, the performance of the solutions varied widely. The common theme was that the cursors performed badly, and the solutions that were based on ranking calculations performed either reasonably or very well. Some of the solutions based on subqueries performed well, whereas some not so well.

One of my goals in this chapter was to cover the specifics of the gaps and islands problems and identify good performing solutions. However, I also had an additional goal. It is common to find that for any given querying problem there are several possible solutions that will vary in complexity and performance. I wanted to emphasize the importance of producing multiple solutions, and analyzing and comparing their logic and performance.

About the author

Itzik Ben-Gan is a Mentor and Co-Founder of Solid Quality Mentors. A SQL Server Microsoft MVP (Most Valuable Professional) since 1999, Itzik has delivered numerous training events around the world focused on T-SQL Querying, Query Tuning, and Programming. Itzik is the author of several books, including *Microsoft SQL Server 2008: T-SQL Fundamentals*, *Inside Microsoft SQL Server 2008: T-SQL Querying*, and *Inside Microsoft SQL Server 2008: T-SQL Programming*. He has written many articles for SQL Server Magazine as well as articles and white papers for MSDN. Itzik's speaking activities include TechEd, DevWeek, SQLPASS, SQL Server Magazine Connections, various user groups around the world, and Solid Quality Mentors' events, to name a few. Itzik is the author of Solid Quality Mentors' Advanced T-SQL Querying, Programming and Tuning and T-SQL Fundamentals courses along with being a primary resource within the company for their T-SQL related activities.

SQL SERVER/DATABASE

SQL SERVER MVP DEEP DIVES

EDITORS: Paul Nielsen • Kalen Delaney • Greg Low • Adam Machanic • Paul S. Randal • Kimberly L. Tripp
TECHNICAL EDITOR: Rod Colledge

This is no ordinary SQL Server book. In *SQL Server MVP Deep Dives*, the world's leading experts and practitioners offer a masterful collection of techniques and best practices for SQL Server development and administration. 53 MVPs each pick an area of passionate interest to them and then share their insights and practical know-how with you.

SQL Server MVP Deep Dives is organized into five parts: Design and Architecture, Development, Administration, Performance Tuning and Optimization, and Business Intelligence. In each, you'll find concise, brilliantly clear chapters that take on key topics like mobile data strategies, Dynamic Management Views, or query performance.

What's Inside

- Topics important for SQL Server pros
- Accessible to readers of all levels
- New features of SQL Server 2008

Whether you're just getting started with SQL Server or you're an old master looking for new tricks, this book belongs on your bookshelf.

The authors of this book have generously donated 100% of their royalties to support War Child International.

About War Child International

WAR child

War Child works in conflict areas around the world, advancing the cause of peace by helping hundreds of thousands of children every year. Visit www.warchild.org for more information.

For online access to the authors go to manning.com/SQLServerMVPDeepDives.
For a free ebook for owners of this book, see insert.

ISBN 13: 978-1-935182-04-7
ISBN 10: 1-935182-04-8
55999
9 781935 182047

MANNING $59.99 / Can $74.99 [INCLUDING eBOOK]