20250307 - Review of Paper - Exploring Scalar Solutions to Complex Data Math

Exploring Scalar Solutions to Complex Data Math
Simple Talk

Key Insights on the Article's Approach

The article describes a clever non-iterative approach using bitmaps to count specific days between date ranges. Here's my analysis:

1. **Bitmap Representation**: The approach represents days of the week as bits (Sunday=1, Monday=2, etc.), allowing for efficient day counting with bitwise operations
2. **Formula Breakdown**:
   - Calculate complete weeks × number of target days
   - Add any target days in the remainder period using bitmap operations
3. **Performance**: The non-iterative approach has constant execution time regardless of date range size, unlike iterative methods whose performance degrades with wider ranges
4. **Trade-offs**: The bitmap approach sacrifices readability for performance, but is worth it for frequently executed queries or UDFs

The key advantage is the O(1) time complexity - whether you're calculating days between dates 1 week apart or 100 years apart, the execution time remains virtually identical. This makes it ideal for high-performance scenarios, especially when embedded in scalar functions as the article discusses.

Simple Iteration

The Big-O time complexity of this iterative approach is **O(n)**, where **n** is the number of days between the start and end dates.

The algorithm has:

- A WHILE loop that executes once for each day in the range
- Fixed-time operations within each iteration

For your specific example with dates '1/1/2024' to '6/17/2024':

- This represents approximately 168 days
- The loop will execute 168 times
- As the date range increases, the execution time grows linearly

This is why the article advocates for the non-iterative bitmap approach, which has O(1) time complexity (constant time) regardless of the date range. The bitmap solution will execute in the same amount of time whether you're checking 1 week or 100 years, while this iterative approach's execution time would grow proportionally with the date range size.

Tactical Iteration

Approach Explained

1. The first WHILE loop advances by 7 days at a time (full weeks), adding 1 to the result for each week (since we know each week contains exactly one Monday)
2. The second WHILE loop handles the remaining days (less than 7) by checking each day individually to see if it's a Monday

Time Complexity Analysis

The time complexity is **O(n/7 + r)** where:

- **n** is the total number of days between dates
- **r** is the remainder days (less than 7)

For practical purposes, this simplifies to **O(n/7)** or **O(n)** in Big-O notation since constants are dropped.

However, this is a significant optimization over the basic iteration:

- The first approach requires **n** iterations (one per day)
- This approach requires approximately **n/7 + r** iterations (one per week + remainder days)

For your date range example ('1/1/2024' to '6/17/2024'):

- First approach: ~168 iterations
- This approach: ~24 iterations (number of weeks) + at most 6 remainder days = ~30 iterations

This represents a ~5.6x performance improvement, although it's still O(n) linear time complexity. By comparison, the bitmap-based approach in the article achieves O(1) constant time complexity, making it significantly faster for very large date ranges.

Bitmap Approach

The author presents an elegant non-iterative approach to count specific days of the week between two dates using bitmap operations:

1. **Core Insight**: The effort needed should not be linear to the timeframe analyzed - a one-century span should require the same computational effort as a one-month span.
2. **Two-Step Calculation**:
   - Count days in complete weeks: `(days in range ÷ 7) × (number of target days per week)`
   - Count target days in the remainder period (0-6 days)

3. **Bitmap Implementation**:
    - Represent days of week as bits in a 7-bit number (Sunday=1, Monday=2, etc.)
    - Create a "target days" bitmap (which days to count)
    - Create a "remainder days" bitmap (which days exist in the remainder)
    - Use bitwise AND to find overlaps between target days and remainder days
    - Count set bits in the result to determine how many target days appear in the remainder
4. **Key Advantage**: This achieves O(1) constant time complexity regardless of date range size.

The approach transforms what would be an O(n) iteration problem into a constant-time calculation using basic arithmetic and bitwise operations, making it ideal for scalar functions and large-scale queries.

Testing Mr. Pollack's Ideas - Code

Iterative Approach

Here are some examples I made for fun.

TEST NAME	TOTAL SUBTREE COST	RANK
CROSS APPLY	0.0076783	6
CROSS JOIN	56.0692	8
CROSS JOIN - WITH TOP	0.0072591	5
GENERATE_SERIES (SQL Server 2022+)	0.0001089	2
RECURSIVE CTE	0.000013	1
TABLE VALUE CONSTRUCTOR & NUMBERS TABLE	0.0007172	3

TEMP TABLE	0.0132982	7
TVC - MULTIPLE CJs (EXAMPLE)	0.0007908	4

5KB iterative_vs_all_at_once_approaches_tsql.sql

235KB iterative_vs_all_at_once_approaches_tsql.sqlplan

Database.Schema.Table: WideWorldImporters.Sales.Order

I created two simple scripts to give use that first and last order date's and id for each customer in the table. I use the function shown in the articel: dbo.ReturnDaysBetweenDates_Iteration();

Tests:

```
use WideWorldImporters;
go
```

Test 1:

24KB Test1.sqlplan

```
select
      CustomerID
    , min(OrderDate)
    , max(OrderDate)
    , dbo.ReturnDaysBetweenDates_Iteration(min(OrderDate), max(OrderDate)
from Sales.Orders
group by CustomerID
order by CustomerID;
```

Test2:

61KB Test2.sqlplan

```
with order_ranked
as
(
    select
          CustomerID
        , OrderID
        , OrderDate
        , row_number() over(partition by CustomerID order by OrderDate as
        , row_number() over(partition by CustomerID order by OrderDate de
    from Sales.Orders
)
select
      f.customer_id
    , f.order_id
    , f.order_date
    , l.order_id
    , l.order_date
    , dbo.ReturnDaysBetweenDates_Iteration(f.order_date, l.order_date )
from
    order_ranked             as f
    inner join order_ranked  as l
        on f.customer_id = l.customer_id
            and f.first_order_rnk = 1
            and l.last_order_rnk  = 1
order by
    f.customer_id;
```

All at Once: BitMap Approach

1. The starting day of the week is less than or equal to the ending day of the week.
2. The starting day of the week is greater than the ending day of the week.
    a. There are two more problems to solve here:
        - A generic solution for any day of the week
            - A generic solution for any combination of days of the week

Test3:

Test3.sqlplan

```
/*
ALL-AT-ONCE : BITMAP
*/
select
      CustomerID
    , min(OrderDate)
    , max(OrderDate)
    , dbo.DaysWithinDateRangeCount(min(OrderDate), max(OrderDate), 1,0,0,
from Sales.Orders
group by CustomerID
order by CustomerID;
```

Test4:

Test4.sqlplan

```
with order_ranked
as
(
    select
          CustomerID
        , OrderID
        , OrderDate
        , row_number() over(partition by CustomerID order by OrderDate as
        , row_number() over(partition by CustomerID order by OrderDate de
    from Sales.Orders
)
select
      f.customer_id
    , f.order_id
    , f.order_date
    , l.order_id
    , l.order_date
    , dbo.DaysWithinDateRangeCount(f.order_date, l.order_date, 1,0,0,0,0,
from
    order_ranked             as f
    inner join order_ranked  as l
        on f.customer_id = l.customer_id
            and f.first_order_rnk = 1
            and l.last_order_rnk  = 1
order by
    f.customer_id;
```

My Own T-SQL All-at-Once Approach

My function (I placed it in WideWorldImporters database).

This function provides a simplified implementation of the day counting logic while maintaining the O(1) time complexity. Let me analyze it:

Function Explanation

`DaysWithinDateRangeCount_CarrollExample` counts occurrences of a specific day of the week (like Monday or Friday) between two dates.

The function works in 3 steps:

1. **Calculate complete weeks**: It divides the total days by 7 to find how many complete weeks are in the range, which gives us how many times the specific day occurs in complete weeks.
2. **Determine if the day appears in the remainder period**: It uses a conditional check to determine if the target day (@day_to_find) appears in the remainder days by comparing it with the start and end day of week.
3. **Return the sum**: The function returns complete weeks count + extra day count (0 or 1).

Time Complexity Analysis: O(1)

This function has O(1) constant time complexity because:

- All operations (date difference, division, comparisons) are scalar operations
- The number of operations is fixed regardless of input range
- Execution time does not depend on the size of the date range

The Key Logic Explained

The most interesting part is the condition that checks if the target day appears in the remainder:

This handles two cases:

1. When the remainder period doesn't wrap around the week (e.g., Monday to Thursday)
2. When it does wrap around (e.g., Friday to Monday)

```
if @day_to_find
    between @start_dow and @end_dow
    or (
        @start_dow > @end_dow
        and (
            @day_to_find >= @start_dow
            or @day_to_find <= @end_dow
        )
    )
```

dbo.DaysWithinDateRangeCount_CarrollExample

```
create function dbo.DaysWithinDateRangeCount_CarrollExample
(     @start_dt     date
    , @end_dt       date
    , @day_to_find  tinyint
)
returns integer
as
begin
declare
      @total_days       int = 0
    , @complete_weeks   int = 0
    , @start_dow        tinyint = 0
    , @end_dow      tinyint = 0
    , @extra_day_count  int = 0;

set @total_days     = datediff(day, @start_dt, @end_dt) + 1;
set @complete_weeks = @total_days / 7;
set @start_dow      = datepart(weekday, @start_dt);
set @end_dow        = datepart(weekday, @end_dt);

-- Count target days in complete weeks
declare @complete_week_count int = @complete_weeks;

-- Check if target day appears in remainder days
-- Use bitmap approach similar to article but simplified
if @day_to_find
    between @start_dow
        and @end_dow
        or (
                @start_dow > @end_dow
                and
                    (
                        @day_to_find >= @start_dow
                        or @day_to_find <= @end_dow
                    )
            )
    set @extra_day_count = 1;

    return @complete_week_count + @extra_day_count
end;
```

Test5:

Test5.sqlplan

25KB

```
/*
ALL-AT-ONCE : JEREMY CARROLL
*/
select
      CustomerID
    , min(OrderDate)
    , max(OrderDate)
    , dbo.DaysWithinDateRangeCount_CarrollExample(min(OrderDate), max(Ord
from Sales.Orders
group by CustomerID
order by CustomerID;
```

Test6:

61KB Test6.sqlplan

```
with order_ranked
as
(
    select
          CustomerID
        , OrderID
        , OrderDate
        , row_number() over(partition by CustomerID order by OrderDate as
        , row_number() over(partition by CustomerID order by OrderDate de
    from Sales.Orders
)
select
      f.customer_id
    , f.order_id
    , f.order_date
    , l.order_id
    , l.order_date
    , dbo.DaysWithinDateRangeCount_CarrollExample(f.order_date, l.order_d
from
    order_ranked             as f
    inner join order_ranked  as l
        on f.customer_id = l.customer_id
            and f.first_order_rnk = 1
            and l.last_order_rnk  = 1
order by
    f.customer_id;
```