SQL Best Practices

- Long Hoang

- Aug 2023

Course Objective and target participants

- Provide guidance and best practices on how to write SQL effectively and correctly

- Target
  - All developers

Best practice #1 Using Explicit column names in SELECT:

Avoid using SELECT *, specify the column names you need.

→ improves query readability and reduces potential issues.

```
SET STATISTICS IO ON
SET STATISTICS TIME ON
SELECT [Name], TerritoryID
FROM  Sales.SalesTerritory AS st
WHERE  st.[Name] = 'Australia' ;
SET STATISTICS TIME ON
SET STATISTICS IO OFF
```

Table 'SalesTerritory'. Scan count 0, logical **reads 2, physical reads 0**

CPU time = 0 ms, elapsed time = 0 ms

```
SET STATISTICS IO ON
SET STATISTICS TIME ON
SELECT *
FROM  Sales.SalesTerritory AS st
WHERE  st.[Name] = 'Australia' ;
SET STATISTICS TIME OFF
SET STATISTICS IO OFF
```

Table 'SalesTerritory'. Scan count 0, logical **reads 4, physical reads 0**

CPU time = 0 ms, elapsed time = 39 ms

Best practice #2 Filtering with LEFT JOIN

```
select *
from public.departments d
left join public.employees e
on d.department_id =e.department_id
and e.hire_date = '1998-03-07’
-- a non-matching ON condition will generate a row where all columns from right table contain NULL.
```

```
select *
from public.departments d
left join public.employees e
on d.department_id =e.department_id
where e.hire_date = '1998-03-07’
--a non-matching WHERE clause will eliminate the row completely regardless of join type.
```

Best practice #3 Avoid abusing SELECT DISTINCT

Problem: Using SELECT DISTINCT can be resource-intensive.

→ Consider using appropriate joins and conditions instead

```
SELECT distinct *
FROM
(SELECT
  FROM Employees
  WHERE JobTitleID  in (…)
union
SELECT
FROM Employees
WHERE JobTitleID  = xyz
)
```

Best practice #4 Use index properly

- Identify columns used frequently in WHERE, JOIN, and ORDER BY clauses, and create indexes to improve query performance

Script used to identify missing indexes

```
SELECT
statement AS [database.scheme.table],
column_id , column_name, column_usage,
migs.user_seeks, migs.user_scans,
migs.last_user_seek, migs.avg_total_user_cost,
migs.avg_user_impact
FROM sys.dm_db_missing_index_details AS mid
CROSS APPLY sys.dm_db_missing_index_columns (mid.index_handle)
INNER JOIN sys.dm_db_missing_index_groups AS mig
ON mig.index_handle = mid.index_handle
INNER JOIN sys.dm_db_missing_index_group_stats  AS migs
ON mig.index_group_handle=migs.group_handle
ORDER BY migs.avg_user_impact DESC
```

Best practice #5 Use Prepared Statements or parameterized queries

- Use parameterized queries to prevent SQL injection attacks

```
USE AdventureWorks;

-- Example using Prepared Statement (Parameterized Query)
DECLARE @ProductName NVARCHAR(50);
SET @ProductName = 'Mountain-200 Black, 42';

-- Using a parameterized query
SELECT ProductID, Name, ProductNumber, Color
FROM Production.Product
WHERE Name = @ProductName;
```

```
USE AdventureWorks;

-- Example without Prepared Statement (Not Recommended)
DECLARE @ProductName NVARCHAR(50);
SET @ProductName = 'Mountain-200 Black, 42';

-- Using concatenated input (NOT RECOMMENDED)
DECLARE @SQL NVARCHAR(MAX);
SET @SQL = 'SELECT ProductID, Name, ProductNumber, Color
FROM Production.Product WHERE Name = ''' + @ProductName
+ '''';
EXEC sp_executesql @SQL;
```

Best practice #6 Avoid using implicit data type conversion

- Explicitly cast data types to avoid implicit conversions, which can affect query performance
- Add overhead to the query performance
- Always use the same data type for both expression (column and value/constant)

Best practice #7 Using NOT EXISTS instead NOT IN

- They are not equivalent in all cases.
- When NULLs are involved, they will return different results
  - when the subquery returns even one null, NOT IN will not match any rows

Best practice #8 Avoid using OR

- ✓ Performance: In some cases, using OR can lead to suboptimal query execution plans, especially when combined with other conditions. It might result in table scans or index scans rather than more efficient index seeks.
- ✓ Complexity: Using multiple OR conditions can make the query logic more complex and harder to understand.

Less efficient

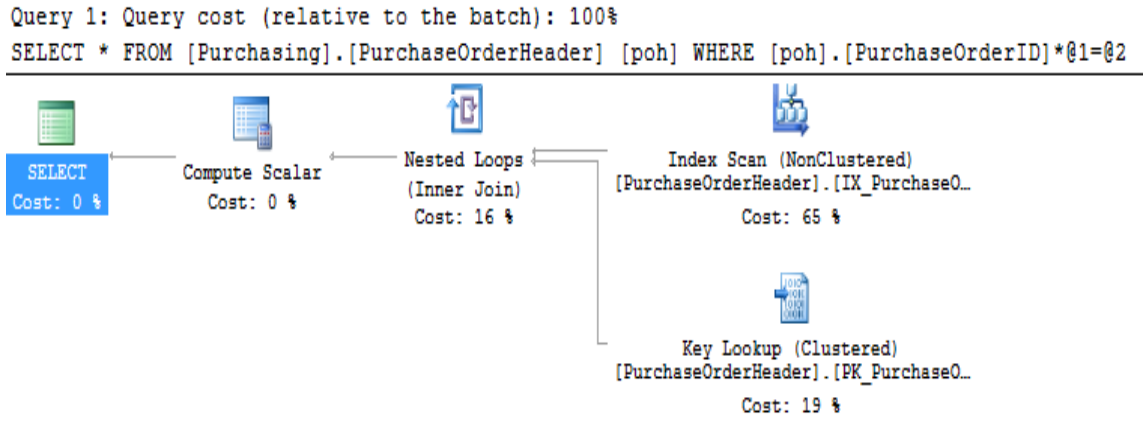
```
SELECT *
FROM TableData
WHERE (@MinDate is null OR @MinDate <= Date)
   AND  (@MaxDate is null OR @MaxDate >= Date)
   AND (@CompanyID is null OR CompanyID = @CompanyID
```

More efficient

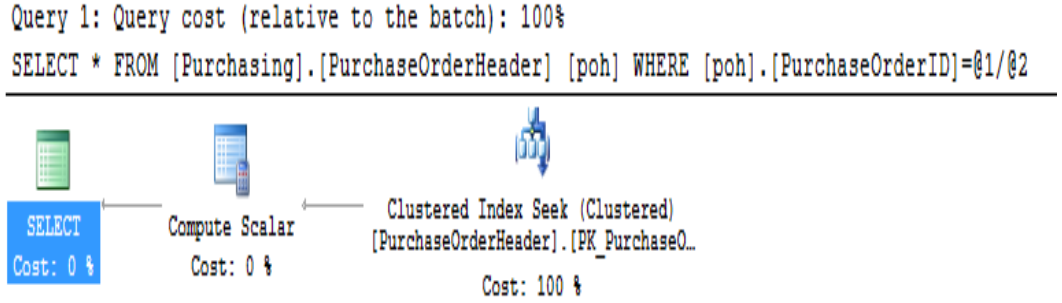
```
SELECT *
FROM TableData
WHERE Date BETWEEN Isnull(@MinDate, '1/1/1900') AND Isnull(@MaxDate, '12/31/2999')
AND CompanyID BETWEEN Isnull(@CompanyID,0) AND Isnull(@CompanyID,999999999)
```

Best practice #9 Avoid Arithmetic Operators on the WHERE clause

```
SELECT *
FROM PurchaseOrderHeader as poh
WHERE poh.PurchaseOrderID * 2 = 3400
```



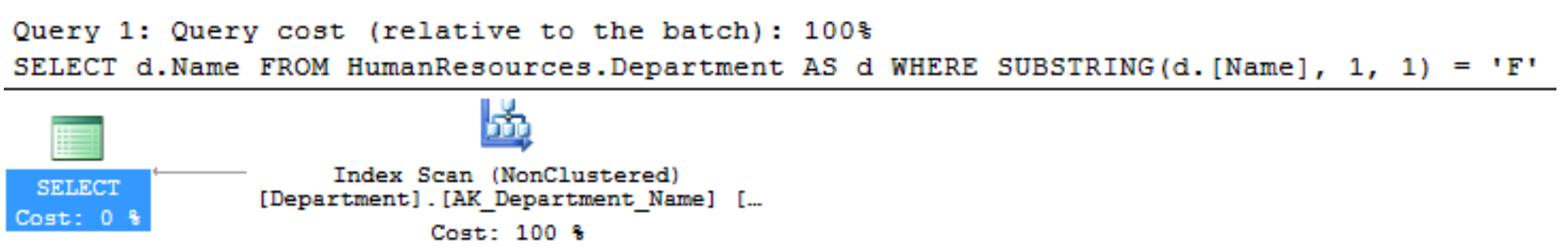

```
SELECT *
FROM PurchaseOrderHeader as poh
WHERE poh.PurchaseOrderID = 3400 / 2
```

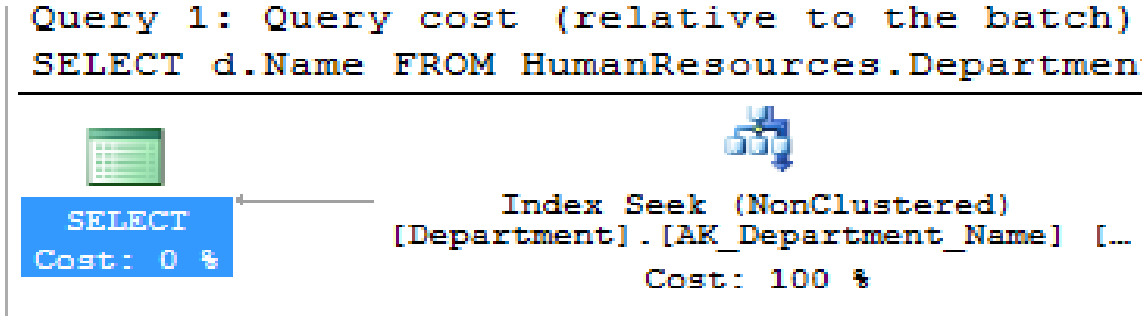


Best practice #10 Avoid Function on the WHERE clause

```
SELECT d.Name
FROM HumanResources.Department AS d
WHERE SUBSTRING(d.[Name], 1, 1) = 'F'
```

```
SELECT d.Name
FROM HumanResources.Department AS d
WHERE d.[Name] LIKE 'F%' ;
```





Best practice #11 Use JOINs instead of Subqueries

- Joins are generally more efficient than subqueries. Use the JOIN syntax for readability and performance.
- Demo

Best practice #12 Use EXISTS over COUNT(*) to check if data exists

Check if record exists

```
DECLARE @n INT ;
SELECT @n = COUNT(*)
FROM  Sales.SalesOrderDetail AS sod
WHERE  sod.OrderQty = 1 ;
IF @n > 0
        PRINT 'Record Exists' ;
```

Table 'SalesOrderDetail'. Scan count 1, logical reads 1240, physical reads 0, read-ahead reads 0
CPU time = 15 ms, elapsed time = 11 ms

```
IF EXISTS ( SELECT 1
FROM  Sales.SalesOrderDetail AS sod
WHERE  sod.OrderQty = 1 )
        PRINT 'Record Exists';
```

Table 'SalesOrderDetail'. Scan count 1, logical reads 3, physical reads 0, read-ahead reads 0
CPU time = 0 ms, elapsed time = 0 ms.

Best practice #13 Use Union ALL instead of Union

If duplicates are not an issue, use UNION ALL instead of UNION, as it's more efficient.

```
SELECT *
FROM  Sales.SalesOrderHeader AS soh
WHERE  soh.SalesOrderNumber LIKE '%47808'
UNION
SELECT *
FROM  Sales.SalesOrderHeader AS soh
WHERE  soh.SalesOrderNumber LIKE '%65748'
```

Best practice #13 Use indexes for aggregate and sort operations

```
DBCC FREEPROCCACHE
DBCC DROPCLEANBUFFERS
set statistics IO on
set statistics  TIME on
go
SELECT SOH.CustomerID,
    sum(SOH.SubTotal) AS TotalSales
FROM Sales.SalesOrderHeader AS SOH
GROUP BY SOH.CustomerID
ORDER BY SOH.CustomerID
go
set statistics IO off
set statistics  TIME off
```

```
CREATE INDEX idex_test ON Sales.SalesOrderHeader(CustomerID, SubTotal)
```

Best practice #14 Group by technique

```
DBCC FREEPROCCACHE
DBCC DROPCLEANBUFFERS
set statistics IO on
set statistics  TIME on
go
SELECT
  C.CustomerID, C.AccountNumber,
  SUM(D.OrderQty) as TotalSales
FROM
  Sales.Customer C
INNER JOIN Sales.SalesOrderHeader S
  ON C.CustomerID = S.CustomerID
INNER JOIN Sales.SalesOrderDetail D
  ON S.SalesOrderID=D.SalesOrderID
GROUP BY
   C.CustomerID, C.AccountNumber
set statistics IO off
set statistics  TIME off
```



```
DBCC FREEPROCCACHE
DBCC DROPCLEANBUFFERS
set statistics IO on
set statistics  TIME on
go
SELECT
  C.CustomerID, C.AccountNumber,SS.TotalSales
FROM
  Sales.Customer C
INNER JOIN
  (SELECT S.CustomerID ,SUM(D.OrderQty) as TotalSales
   FROM  Sales.SalesOrderHeader S
   INNER JOIN Sales.SalesOrderDetail D
   ON S.SalesOrderID=D.SalesOrderID
   GROUP BY   S.CustomerID ) SS
ON C.CustomerID = SS.CustomerID
  set statistics IO off
set statistics  TIME off
```

should only be grouping on CustomerID, and not on all those other columns.
Push the grouping down a level, into a derived table

Best practice #15 Don’t use scalar function in SELECT

- **Query 1: use user-define inline scalar function**

```
DBCC FREEPROCCACHE
DBCC DROPCLEANBUFFERS
set statistics io on
set statistics time on
SELECT soh.SalesOrderID,soh.OrderDate,
      dbo.[get_totalamt_by_Salesorderid](soh.SalesOrderID)
as Qty
FROM Sales.SalesOrderHeader soh
set statistics time off
set statistics io off
```

```
(31465 rows affected)
Table 'SalesOrderHeader'. Scan count 1, logical reads 689,
physical reads 3

 SQL Server Execution Times:
   CPU time = 1250 ms,  elapsed time = 1457 ms.
```

Query 2: use Table-valued-function with cross apply

```
DBCC DROPCLEANBUFFERS
go
set statistics io on
set statistics time on
GO
SELECT soh.SalesOrderID, soh.OrderDate, s.TotalAmt
FROM Sales.SalesOrderHeader soh
CROSS APPLY
GetTotalAmountbySalesorderid(soh.SalesOrderID) s
go
set statistics time off
set statistics io off
```

```
(31465 rows affected)
Table 'SalesOrderHeader'. Scan count 1, logical reads 689, physical
reads 3
Table 'SalesOrderDetail'. Scan count 1, logical reads 1247, physical
reads 3,
 SQL Server Execution Times:
   CPU time = 63 ms,  elapsed time = 152 ms.
```

Best practice #16 Avoid using cursor

- Cursors can be slow and resource-intensive. Whenever possible, use set-based technique instead.

```
-- Plain Cursor ----------------------------------
DECLARE @OrderAmount DECIMAL(24,4)
DECLARE @TotalOrders DECIMAL(24,4)
SET @TotalOrders = 0

DECLARE c1 CURSOR
FOR SELECT OrderAmount = OrderQty * UnitPrice
FROM       Sales.SalesOrderDetail
WHERE      SalesOrderID = 47018

OPEN c1
FETCH NEXT FROM c1 INTO @OrderAmount
WHILE (@@fetch_status <> -1)
BEGIN
            IF (@@fetch_status <> -2)
            BEGIN
                        SET @TotalOrders = @TotalOrders +
@OrderAmount
            END
            FETCH NEXT FROM c1 INTO @OrderAmount
END

CLOSE c1
DEALLOCATE c1
```



```
-- Straight SELECT
SELECT SELECT_Total = SUM(OrderQty * UnitPrice)
FROM       Sales.SalesOrderDetail
WHERE      SalesOrderID = 47018
GO
```

Best practice #17 Using View vs Stored procedure

View:

- View abstract complex queries, making them easier to use by encapsulating the underlying logic.
- View allow you to control user access by limiting which columns or rows they can see.
- View can be reused across multiple queries, promoting consistency and reducing redundancy.

Stored procedure

- Use stored procedures for encapsulating business logic, improving performance through precompiled code, and facilitating parameterized input.
- Stored procedures are suitable when you need to control execution plans and reuse code across multiple queries.

- **Views are used as a guard to provide only specific columns/rows to users.**
- **Stored procedures are used for business data processing needs.**

Best practice #18 Check and delete data duplicated with CTE

	UserSkillMetricId	EmployeeID	Skillmetricid	Expertiseid	Experienceid	Lastyearused	Status
1	2	792	713	5	2	2023	1
2	3	792	713	3	2	2023	1
3	4	792	718	5	2	2023	1
4	5	792	736	4	2	2023	1
5	6	792	788	2	2	2023	1
6	7	792	841	1	2	2023	0
7	8	792	841	1	2	2023	0
8	9	792	926	3	2	2023	1
9	10	792	927	3	2	2023	1
10	11	792	1816	3	6	2023	1
11	12	792	432	5	2	2023	1
12	13	792	434	5	2	2023	1

- **Use case**: a user can add multiple skills with level of expertise and experience for every skill he/she has. Write a query to check duplicate and delete it
- **Two possible approaches**:
  - Use GROUP BY with HAVING COUNT(*) >1
  - Use CTE with Row_number and partition by

  →CTE is more flexible as we can replace SELECT from output by DELETE statement

```
--CTE approach
WITH CTE(employeeid,
    skillmetricID,
    duplicatecount)
AS (SELECT employeeid,
          skillmetricID,
          ROW_NUMBER() OVER(PARTITION BY employeeid,
                skillmetricID
          ORDER BY expertiseid desc) AS DuplicateCount
    FROM UserSkillMetric
)
select * from cte where DuplicateCount >1 ;
```

Best practice #19 Update or Delete from multiple tables

- Use appropriate join conditions when updating or deleting data from multiple tables
- Update or Delete with alias.
- Demo

```
Update d
set d.UnitPrice=9999
from [Sales].[SalesOrderDetail] d
inner join [Sales].[SalesOrderHeader] o
on d.SalesOrderID=o.SalesOrderID
and o.SalesOrderNumber='SO43659'
```

```
Delete d
from [Sales].[SalesOrderDetail] d
inner join [Sales].[SalesOrderHeader] o
on d.SalesOrderID=o.SalesOrderID
and o.SalesOrderNumber='SO43659'
```

Q&A


Thank you
Nash
Tech.