UNDERSTANDING SQL SERVER EXECUTION PLANS

Rami Reddy
ramireddydasaradha@gmail.com
**Heap**: A table without an clustered index is called heap. In a heap, the rows are inserted in unordered fashion. It inserts the row in the first page which has enough space.

Eg:

```sql
CREATE TABLE HeapTest
(
    Id int identity(1,1),
    Name varchar(3000)
)

INSERT INTO HeapTest VALUES(REPLICATE('a',3000))
INSERT INTO HeapTest VALUES(REPLICATE('b',3000))
INSERT INTO HeapTest VALUES(REPLICATE('c',3000))
INSERT INTO HeapTest VALUES('d')

SELECT sys.fn_PhysLocFormatter(%%Physloc%%,* FROM HeapTest
```

**Clustered Index**: If a table has a clustered index defined on it, rows are always inserted in the order of key columns specified while creating the index.

```sql
CREATE TABLE IndexTest
(
    Id int identity(1,1) primary key,
    Name varchar(6000)
)

INSERT INTO IndexTest VALUES(REPLICATE('a',6000)),(REPLICATE('b',6000)),(REPLICATE('c',6000)),('d')

SELECT %%physloc%%.sys.fn_PhysLocFormatter(%%physloc%%) AS RID,* FROM IndexTest
```
• SQL server will use the B-tree to store the index. For a clustered index, in leaf level (Level-0), it consists of all the data and in its above levels, it consists of the starting record key values of its below page.

• Suppose, in a table, we have say 80 records, and each page can accommodate only 20 records. Then the table will have 4 pages in level-0 and one page in level-1, which consists of key columns information of starting records of each page. If the data grows in table, the index levels also grow.
Non Clustered Index:

Difference between clustered index and non-clustered index is the leaf level of index. Clustered index consists of data at the leaf level, whereas Non-Cl consists of only Key values at leaf level. But in case of Heap, it will have an additional column "PageRID", which enables to bookmark the heap quickly.
When a query is submitted to SQL Server for execution, it will go through the below phases.

1. Verifies the syntax of the queries
2. Expands the views, synonyms etc.
3. DDL statements that are not optimizable (e.g., DDL) directly will be translated to an internal form and passed to query executor.
4. Forms the parse-tree which will be used by optimizer.

1. Create an cost-based optimized plan based on available indexes and statistics.
2. Stores the plan in plan cache for reuse and pass it to query executor.

1. If statistics are not up to date or if any schema changes happened, query will pass to optimizer, which will again creates a plan.
2. With the help of the storage engine, query engine starts to execute the query.

Execution plans will give basic insight about how the query has been executed internally by SQL Server. It gives you the information like below (and much more).

1. Which indexes were used to fetch the data from tables.
2. How the data is joined together.
3. How aggregation such as sum, count are evaluated.
4. Estimated Costs of each of theses operations etc.
There are 2 types of plans,

1. Estimated Execution plan  
2. Actual Execution plan

• Estimated execution plan is the plan estimated by sql server before the query is executed, Where as Actual execution plan is the plan used by sql server once the query is executed.

• Some times, the actual execution plan is different from the estimated plan, when the execution engine determines that statistics are out of date or when the query executor decides to change parallelism or change in dependencies or set options change.

• Actual execution plan consists of additional runtime parameters, such as no of rows affected, degree of parallelism etc.

• Its always advisable to use Actual plan. But estimated plan will be used in some scenarios, when the procedure is taking too much time.

Various ways of collecting Plans:

1. Ssms query window.
2. Set Statistics profile on ,set showplan_all on, set showplan_text on
3. We can use 2 DMVs also to sys.dm_exec_cached_plans contains the cached plan before the query is executed. Once the query is executed, sys.dm_exec_query_stats contains the actual executed plan.

Given Stored Procedure Execution Plan:

select ES.query_plan from sys.dm_exec_cached_plans EC  
cross apply sys.dm_exec_plan_attributes(EC.plan_handle)EP  
cross apply sys.dm_exec_query_plan(EC.plan_handle) ES  
where EP.attribute = 'objectid' and EP.value = OBJECT_ID('Stored Proc Name')
SQL Server breaks queries down into a set of fundamental building blocks that we call operators or iterators. Each iterator implements a single basic operation such as scanning data from a table, updating data in a table, filtering or aggregating data, or joining two data sets. There are a few hundreds of these.

Scans and seek are the iterators used to read the data from the table. There are various types of scans and seek iterators.

1. **Table Scan**
2. **Index Scan/Clusted Index Scan**
3. **Index Seek/Clustered Index Seek**

**Table Scan**:
This iterator scans each row in the heap and evaluate the predicate (if any). If the row qualifies, it returns the row. This touches almost each and every row. The cost of this operator is almost proportional to number of rows in table.

**Estimated CPU cost**:
- Initial cost: 0.0000785 = 785 * power(10, -7) (To Read IAM Page)
- Additional cost: 0.0000011 = 11 * power(10, -7) per each record.

**Estimated IO Cost**:
- Initial cost: 0.0032035 = 3125 * power(10, -7) + 785 * power(10, -7)
  - Disk can make 320 Random I/O per second. So, one I/O will take 1/320 = 0.0003125sec.
  - Additional cost: for each page is 0.00074074 = 1/1350, 1350 is the number of sequential I/Os per second
ITERATORS - SCAN AND SEEK

Index Scan/Clustered Index Scan:
- This iterator scans each row in the index and evaluates the predicate (if any). If the row qualifies, it will return the row. This touches almost each and every row. The cost of this operator is almost proportional to the number of rows in the table.

Estimated CPU cost:
- Initial cost = \(2 \times 785 \times 10^{-7} = 0.0001581\),
- Additional cost = \(11 \times 10^{-7}\) per each record.

Estimated IO Cost:
- Initial cost = \(3125 \times 10^{-7} + 785 \times 10^{-7}\)
  - Disk can make 320 Random I/O per second. So, one I/O will take \(1/320 = 0.0003125\) sec.
- Additional cost = for each page is \(0.00074074 = 1/1350\), 1350 is the number of sequential I/Os per second.

Index Seek /Clustered Index Seek:
- Seek can be used directly to navigate to records. Seek will directly traverse through index levels to reach the exact page that the qualifying records has. It will almost read only the pages which have qualifying records. Because of this, SEEK cost is proportional to the number of qualifying records.
  - Seeks are of 2 types.
    1. Singleton lookup: Looks for only one record. This will occur in case of when we are searching for a single record or there is an unique index.
    2. Range Scan: Initially it will traverse through index levels to reach the initial record that qualifies, and then it performs a range scan until it reaches the end of scan range.
We can’t differentiate the Singleton lookup or range scan from the execution plan. We can use the DMV, “sys.dm_db_index_operational_stats” to track these. This DMV has 2 columns, “range_scan_count”, “singleton_lookup_count”, which will shows the number of times the specific index has been performed range scan or singleton lookups.

Cost details are same like Index Scan/Clustered Index Scan

**Seek Procedure**: While searching for record-4

As the Page 1 starts at 1, page2 starts at 3. It will directly go to page 2

As the Page 5 starts at 1, next page starts at 5. It will go to page 5.
These iterators are used to fetch other additional columns information. The Heap or CI consists of all the columns of the table. In practise, Non-CI will consists of only few columns, which makes more rows fit in a page, thus reduces disk space and improves efficiency of the queries. However, if we need some other columns in query, by using lookup iterators, it will get the data.

Eg:
```sql
if exists (select 1 from INFORMATION_SCHEMA.TABLES where TABLE_NAME = 'TT')
    drop table TT
create table TT (myID int identity(1,1) PRIMARY KEY, ID int, Name varchar(1000))
declare @I int = 1
while (@I <= 1000)
    begin
        insert into TT values (@I,REPLICATE('a',1000))
        set @I = @I + 1
    end
create index idx_ID on TT(ID)
select * from TT where ID = 1000
select * from TT where ID = 1000 and Name = 'RR'
```

If the table is Heap, RID lookup will be used to fetch the additional columns data. If the table is CI, Key lookup will be used to fetch the additional columns data. Non-CI index consists of RID in case of Heap and Key columns in case of CI.

Cost: For 1 lookup, it will cost 0.0003125 I/O cost and 0.0001581 CPU cost. However, in case of more than 1 lookup, TotalSubtreecost need to be considered instead of I/O and CPU cost.
Sql server implements the join physically in 3 ways

1. Nested Loops join
2. Merge join
3. Hash join

**Nested Loops join:**

In general, Nested loop join compares each row in a table with another row in table. Its algorithm is like below

```
for each row R1 in outer table
begin
    for each row R2 in inner table
    begin
        if R1 joins with R2
        begin
            return (R1,R2)
        end
    end
end
```

Cost of this query is proportional to product of rows in outer and inner tables. If Outer table consists of M rows, inner table consists of N rows, it will take M * N iterations, O(n^2) time complexity.

Option (loop join ) or inner loop join will force the query to use Nested Loops. However, by adding an index to inner table, we can force seek on inner table, which reduces to only 1 iteration for each outer row. Hence, it will take M iterations only.

However, Nested loops join will work well only when there are few number of rows. As row count increases, the cost of this iterator will increase exponentially.
Implementing Left outer join:
for each row R₁ in the outer table
begin
   for each row R₂ in the inner table
      if R₁ joins with R₂
         return (R₁, R₂)
      if R₁ did not join
         return (R₁, NULL)
   end

Implementing Right Outer join:
By interchanging outer and inner tables, it will use the same algorithm like LOJ

Implementing Full Outer join:
It implements full outer join with the combination of LOJ + left-anti-semi-join (By reversing tables).

Customers FULL OUTER JOIN sales will be implemented like
   Customers Left outer join Sales + sales left anti semi join Customers.

Left anti semi join, will returns the rows in the outer table which are not having matched row with inner table.

Cost:
Estimated I/O cost is 0.
Estimated CPU cost is 0.0000042 for each comparison.
**Merge Join** is the one of the physical join used by SQL server. The pre-requisite is both the datasets should be in sort-order. Another pre-requisite is it should have at least one equi-join predicate (one equals condition).

**Algorithm**

1. Get first row $R_1$ from input 1
2. Get first row $R_2$ from input 2
3. While not at the end of either input
   - Begin
   - If $R_1$ joins with $R_2$
     - Begin
     - Return ($R_1$, $R_2$)
     - Get next row $R_2$ from input 2
     - End
   - Else if $R_1 < R_2$
     - Get next row $R_1$ from input 1
   - Else
     - Get next row $R_2$ from input 2
   - End

In Merge join, each table is read at most once, so its cost is proportional to the sum of rows. We can say $O(n)$ time complexity.

Option (Merge join) or inner merge join will force the query to use merge join.
If Outer table rows are not guaranteed to be unique, query engine implements Many-Many version of the merge join. In this version, when the outer row joins with inner row, it takes the copy of that row and writes to tempdb. Later when it finds same row in outer table, it copies all the rows from tempdb to back.

Implementation of LOJ:

get first row \( R_1 \) from input 1, get first row \( R_2 \) from input 2
initialize counter to 0
while not at the end of either input
  begin
    if \( R_1 \) joins with \( R_2 \)
      begin
        increment counter
        return \((R_1, R_2)\)
        get next row \( R_2 \) from input 2
      end
    else if \( R_1 < R_2 \)
      If counter = 0
        return \((R_1, null)\)
        get next row \( R_1 \) from input 1
      Initialize counter to 0
    else
      get next row \( R_2 \) from input 2
  end

Merge join’s Estimated I/O cost is 0. But for Many-Many version, it has some I/O cost

Estimated CPU cost is \( 56000 \times \text{power}(10, -7) \). For every outer row, it costs, \( 43 \times \text{power}(10, -7) \) and for every inner row, it costs, \( 21 \times \text{power}(10, -7) \).
Hash join will be mostly used when dealing with huge amount of data and not in sort order. One pre-requisite here is it should have at least one equi-join predicate (one equals condition).

Hash join works in 2 phases.
1. Build Phase: In this phase, it reads all rows from the input table (usually smaller table) and apply the hash function on the equi-join keys and builds a hash table.
2. Probe phase: In this phase, it reads all rows from the other input table (usually larger table) and apply the hash function on the equi-join keys and will check the resultant hashtable is not empty or not. If it not empty, then it will again checks the conditions (because of collisions), then returns the row.

Algorithm
for each row R₁ in the build table
  begin
    calculate hash value on R₁ join key(s)
    insert R₁ into the appropriate hash bucket
  end
for each row R₂ in the probe table
  begin
    calculate hash value on R₂ join key(s)
    for each row R₁ in the corresponding hash bucket
      if R₁ joins with R₂
        return (R₁, R₂)
    end
  end
Unlike Merge join, hash join doesn’t require any order. In same way, Its output rows also need not to be in sort order.

Unlike the other join types, this is the blocking iterator. It will not return any row until it completes the probe phase.

This iterator also requires memory to build the hash table. If the memory is not enough to build the hash table, it will spills the some part of the hash table to disk. For next rows, it will write to disk or in-memory based on partition of hash. Then in probe phase also, based on hash-partition, it will check on-memory or disk partitions.

Option (Hash join) or inner Hash join will force the query to use merge join.

Estimated CPU cost is $177500 \times 10^{-7}$. For Each Build table record, it costs, $189 \times 10^{-7}$ and for every probe table row, it costs, $46 \times 10^{-7}$

Estimated I/o cost is 0 generally. However, when it spills rows to disk, it costs some I/o.
ITERATORS – SEMI JOINS

Left Semi Join Showplan Operator
The Left Semi Join operator returns each row from the first (top) input when there is a matching row in the second (bottom) input. If no join predicate exists in the Argument column, each row is a matching row.

You can see this operator in queries, where exists has been used.

Left Anti Semi Join Showplan Operator
The Left Anti Semi Join operator returns each row from the first (top) input when there is no matching row in the second (bottom) input. If no join predicate exists in the Argument column, each row is a matching row.

You can see this operator in queries, where not exists has been used.

Right Anti Semi Join Showplan Operator
The Right Anti Semi Join operator outputs each row from the second (bottom) input when a matching row in the first (top) input does not exist. A matching row is defined as a row that satisfies the predicate in the Argument column (if no predicate exists, each row is a matching row).

You can see this operator in queries, where exists has been used and outer table is smaller compared to inner table.

Right Semi Join Showplan Operator
The Right Semi Join operator returns each row from the second (bottom) input when there is a matching row in the first (top) input. If no join predicate exists in the Argument column, each row is a matching row.

You can see this operator in queries, where not exists has been used and outer table is smaller compared to inner table.
DECLARE @t TABLE(ID INT)
INSERT INTO @t VALUES (1),(2),(3)
DECLARE @t1 TABLE(ID INT)
INSERT INTO @t1 VALUES (1),(4),(5)
SELECT * FROM @t WHERE EXISTS (SELECT Id FROM @t1 WHERE ID = t.ID)
SELECT * FROM @t WHERE NOT EXISTS (SELECT Id FROM @t1 WHERE ID = t.ID)

CREATE TABLE t (ID INT)
WITH N AS
(SELECT 0 AS Num UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL SELECT 4
UNION ALL SELECT 0 AS Num UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL
SELECT 4)

INSERT INTO t SELECT rn FROM Num

CREATE TABLE t1(Id INT)
INSERT INTO t1 VALUES (1),(4),(5)
SELECT * FROM t WHERE EXISTS (SELECT Id FROM t1 WHERE ID = t.ID)
SELECT * FROM t WHERE NOT EXISTS (SELECT Id FROM t1 WHERE ID = t.ID)
ITERATORS – STREAM & HASH AGGREGATE

SQL server implements aggregates by using 2 iterators.

1. Stream Aggregate
2. Hash Aggregate

**Stream Aggregate:**
Stream Aggregate requires rows to be in sort order of the Grouping columns specified in the query. If they are not in sort order, it will explicitly sort them. Otherwise, it will use the appropriate index, if any row is present. Its output rows will also be in sort order.

Stream Aggregate reads row by row in order. When it finds a same group of columns like previous record, it will update the aggregate results. If it finds a different group, it will return the previous group aggregate results and start aggregating a new group.

**Algorithm:**
1. Clear the current aggregate results
2. Clear the current group by columns
3. For each input row:
   1. Begin
   2. If the input row does not match the current group by columns:
      1. Begin
      2. Output the aggregate results
      3. Clear the current aggregate results
      4. Set the current group by columns to the input row
      End
   3. Update the aggregate results with the input row
End
**Query Hint:**

select ID1,sum(ID) from A group by ID1  
option (Order Group)

**Cost:**

Estimated CPU : For each Record, it costs $11 \times \text{power}(10,-7)$

**Hash Aggregate:**

In general stream aggregate will be used when there are fewer number of rows. However, in case of large data Hash Aggregate is generally used by query engine.

Hash Aggregate does not require rows to be in sort order. It will not preserve the order of rows while outputting rows.

Hash Aggregate Requires memory to build the hash table and again it’s the blocking iterator. It will not release any rows, until it completes the process for all rows.

**Query Hint:**

select ID1,sum(ID) from A group by ID1  
option (Hash Group)

**Estimated CPU Cost:**  
Initial cost is $177500 \times \text{power}(10,-7)$  
Then for every Build element it takes $244 \times \text{power}(10,-7)$, and for every probe element it costs $64 \times \text{power}(10,-7)$
Algorithm:
for each input row
  begin
    calculate hash value on group by column(s)
    check for a matching row in the hash table
    if we do not find a match
      insert a new row into the hash table
    else
      update the matching row with the input row
  end
output all rows in the hash table
ITERATORS – SEGMENT

Segment iterator will distinguish the records into groups based on one or more column values. We can see this operator in action in ranking functions such as row_number(), rank() etc. This iterator requires the rows to be in sort order. It adds a new column to the input rows and send it as output to its parent. That column will indicates whether the element is new group or not. Usually it takes very less unless we process very huge amount of rows. Its Estimated CPU cost is $2 \times 10^{-8}$ for each record

Eg:

```sql
create table RowNumberTest
(
    Col1 int,
    Col2 int
)

insert into RowNumberTest values (1,10),(1,20),(2,15),(2,30),(1,25),(2,25)

select *,ROW_NUMBER() over (partition by col1 order by Col2) as rn from RowNumberTest

select RANK() over (partition by col1 order by Col2) as rn from RowNumberTest
```
Spools are used by query engine to save the intermediate results of a query to a temporary table. There are different types of spools like Eager spool, Lazy Spool, rowcount spool.

Eager Spool: This iterator will read all the rows from table at once and write it to the tempdb. This is blocking iterator. It will not return any data to its parent until it reads all rows from input.

We can see this iterator in action in mostly the scenarios like Remote Scan, Halloween protection, scenarios where read cursor is affecting write cursor (inserts/updates based on columns other than clustered key).

```sql
create table Orders
(OrderId int identity(1,1) primary key,
OrderCost int,
CustomerId int)
GO
insert into Orders values
(ABS(BINARY_CHECKSUM(newid())%100),ABS(BINARY_CHECKSUM(newid())%10000))
GO 10000
-- Reordering orders of a customer
insert into Orders
select OrderCost,CustomerId from Orders where CustomerId = 1
```
Lazy Spool:

We can see this iterator in action in mostly the scenarios where a subquery is taking more cost and there are more chances that outer values will repeat again. We can see lazy spools in action in recursive CTEs also.

Unlike Eager spools, lazy spools are not the blocking operators. They will write the records into tempdb on demand only.
Example: Lazy spools in recursive CTE

```sql
with cte as

    select EmpId, MgrID, 0 as Level from Employees where MgrID is null
    union all
    select E.EmpId, E.MgrID, c.Level + 1 from cte c
    inner join Employees E on c.EmpID = E.MgrID

select * from cte
```

Query 1: Query cost (relative to the batch): 100%

```
with cte as ( select EmpId, MgrID, 0 as Level from Employees where MgrID is null union all select E.EmpId, E.MgrID, c.Level + 1 from cte c inner join Employees E on c.EmpID = E.MgrID
```
Example: Lazy spools in subquery

Create table Orders

```
    OrderId int identity(1,1) primary key,
    OrderCost int,
    CustomerId int
```

GO

Insert into Orders values
ABS(BINARY_CHECKSUM(newid())%100), ABS(BINARY_CHECKSUM(newid())%10000))
GO 10000

- Reordering orders of a customer
- Insert into Orders
  Select OrderCost, CustomerId from Orders where CustomerId = 1

Query 1: Query cost (relative to the batch): 100%
select * from Orders o where OrderCost > ( select AVG(OrderCost) from Orders where CustomerId = o.CustomerId )
RECURSIVE CTE EXECUTION

Step 1:
Execute Anchor Part
Send rows as o/p

Step 2:
Writes O/p rows to temporary table in TempDB

Step 3:
Picks last inserted element from temporary table.

Step 4:
Executes recursive part for the selected element and gets newly qualified rows
Send rows as o/p

Step 5:
Removes selected element from temporary table.

Step 6:
If any rows in temp_table?
Yes
No
Query Execution completed.