Rami Reddy
ramireddydasaradha@gmail.com

UNDERSTANDING SQL SERVER EXECUTION PLANS

Chennai SQL Server User Group

STORAGE ENGINE

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
```

Results | Messages

	(No column name)	Id	Name
1	(1:470:0)	1	aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...
2	(1:470:1)	2	bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb...
3	(1:470:2)	4	d
4	(1:1176:0)	3	ccccccccccccccccccccccccccccccccccccccccccccccc...

Clustered Index : If a table has an 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
```

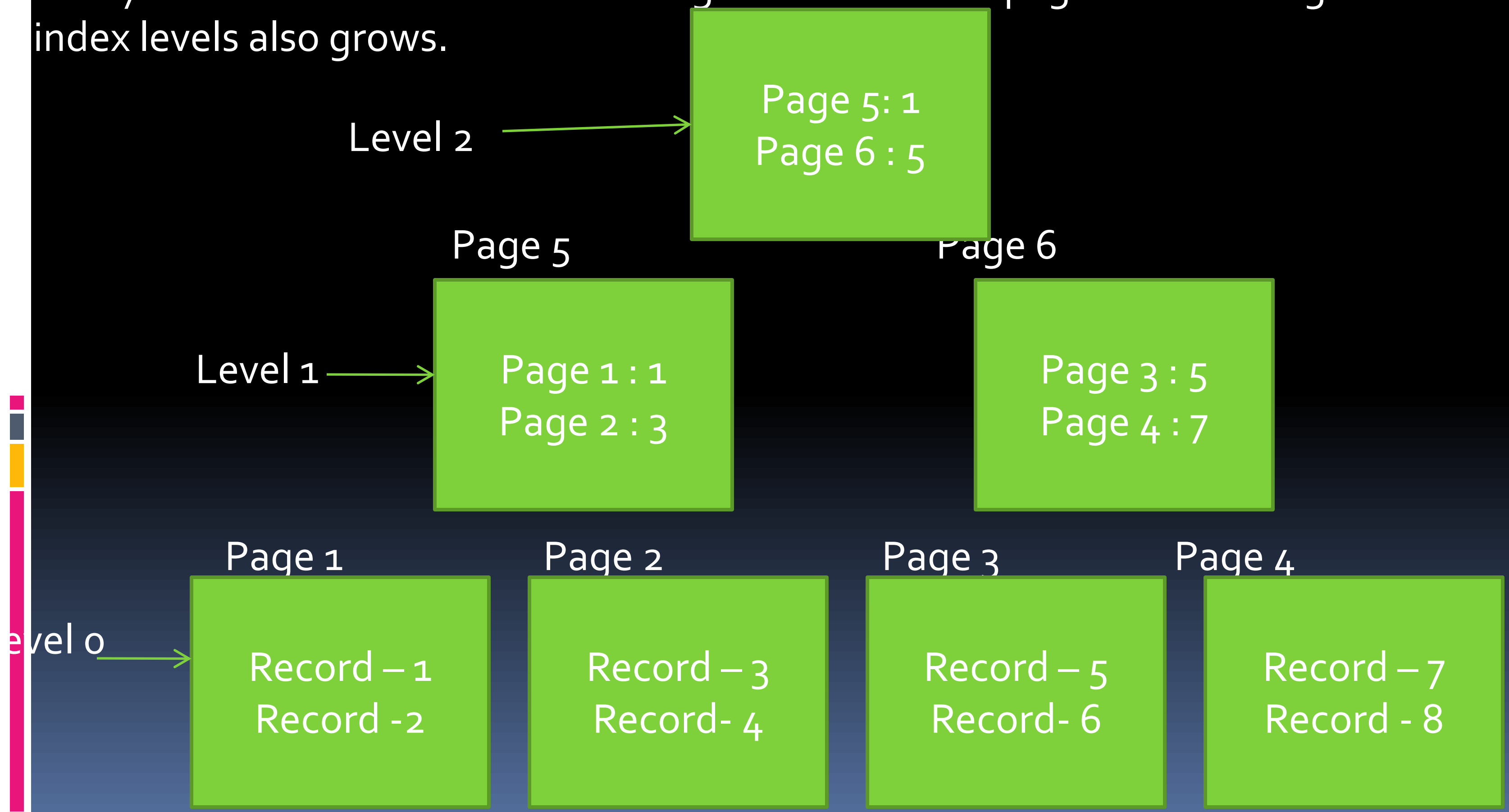
Results | Messages

	(No column name)	RID	Id	Name
1	0x9A04000001000000	(1:1178:0)	1	aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...
2	0x9D04000001000000	(1:1181:0)	2	bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb...
3	0x9E04000001000000	(1:1182:0)	3	ccccccccccccccccccccccccccccccccccccccccccccccc...
4	0x9E04000001000100	(1:1182:1)	4	d

STORAGE ENGINE

- 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 grows.



STORAGE ENGINE

DBCC IND can be used to see the pages of an index or table.

```
dbcc ind('Test','IndexTest',1,1)
```

Results | Messages

	PageFID	PagePID	IAMFID	IAMPID	ObjectID	IndexID	PartitionNumber	PartitionID	iam_chain_type	PageType	IndexLevel	NextPageFID	NextPagePID	PrevPageFID	PrevPagePID
1	1	1179	NULL	NULL	1532610040	1	1	72057595479064576	In-row data	10	NULL	0	0	0	0
2	1	1178	1	1179	1532610040	1	1	72057595479064576	In-row data	1	0	1	1181	0	0
3	1	1180	1	1179	1532610040	1	1	72057595479064576	In-row data	2	1	0	0	0	0
4	1	1181	1	1179	1532610040	1	1	72057595479064576	In-row data	1	0	1	1182	1	1178
5	1	1182	1	1179	1532610040	1	1	72057595479064576	In-row data	1	0	0	0	1	1181

```
dbcc traceon(3604)
dbcc page('Test',1,1180,3)
```

Results | Messages

	FileId	PageId	Row	Level	ChildFileId	ChildPageId	Id (key)	KeyHashValue
1	1	1180	0	1	1	1178	NULL	NULL
2	1	1180	1	1	1	1181	2	NULL
3	1	1180	2	1	1	1182	3	NULL

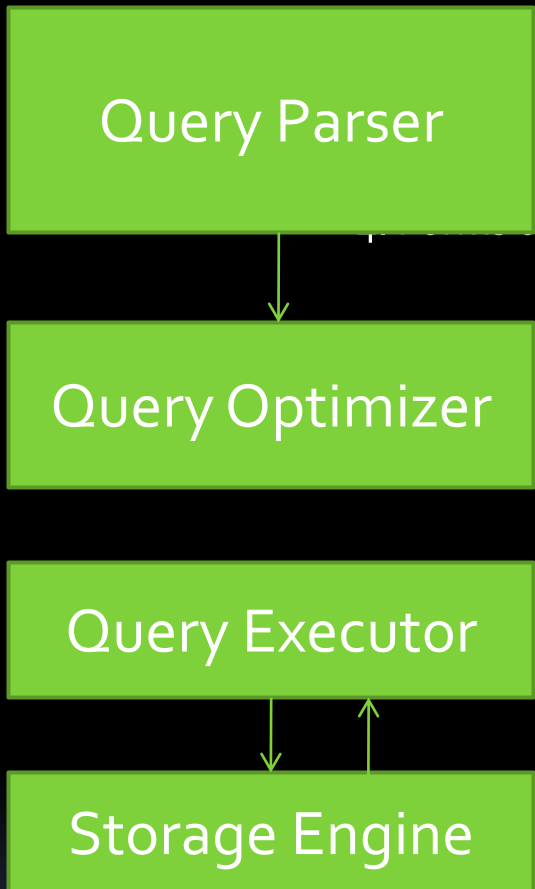
Non Clustered Index :

Difference between clustered index and non-clustered index is the leaf level of index. Clustered index consists of data at the leaf level, where as Non-CI consists of only Key … umn "PageRID", which …

	FileId	PageId	Row	Level	ChildFileId	ChildPageId	ID (key)	HEAP RID (key)	KeyHashValue
1	1	1195	0	1	1	1192	NULL	NULL	NULL
2	1	1195	1	1	1	1194	450	0x473D000001000000	NULL
3	1	1195	2	1	1	1196	899	0x873D000001000100	NULL

QUERY EXECUTION - BASICS

When a query is submitted to sql server for execution, it will go through the below phases.



Query Parser:

1. Verifies the syntax of the queries
2. Expands the views, synonyms etc .
3. DDL statements that are not optimizable(Eg:DDL) directly will be translated to an internal form and passed to query executor.

he parse-tree which will be used by optimizer.

Query 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.

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

EXECUTION PLAN - BASICS

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')
```

ITERATORS - SCAN AND SEEK

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 rom a table, updating data in a table, filtering or aggregating data, or joining two data sets. There re a few hundreds of these.

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

. Table Scan
. Index Scan/Clusted Index Scan
. Index Seek/Clustered Index Seek

able Scan :

This iterator scans each row in the heap and evaluate the predicate(if any). If the row qualifies, it vill returns the row. This touches almost each and every row. The cost of this operator is almost roportional 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

ndex Scan/Clustered Index Scan :

This iterator scans each row in the index and evaluate the predicate(if any). If the row qualifies, it vill returns the row. This touches almost each and every row. The cost of this operator is almost roportional to number of rows in table.

Estimated CPU cost :

Initial cost - 2 * 785 * power(10,-7) = 0.0001581,

Additional cost - 11 * power(10,-7) per each record.

Estimated IO Cost :

Initial cost - 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 econd

ndex 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 has qualifying records. Because of this, SEEK cost is proportional to 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 single record or there is an unique index.

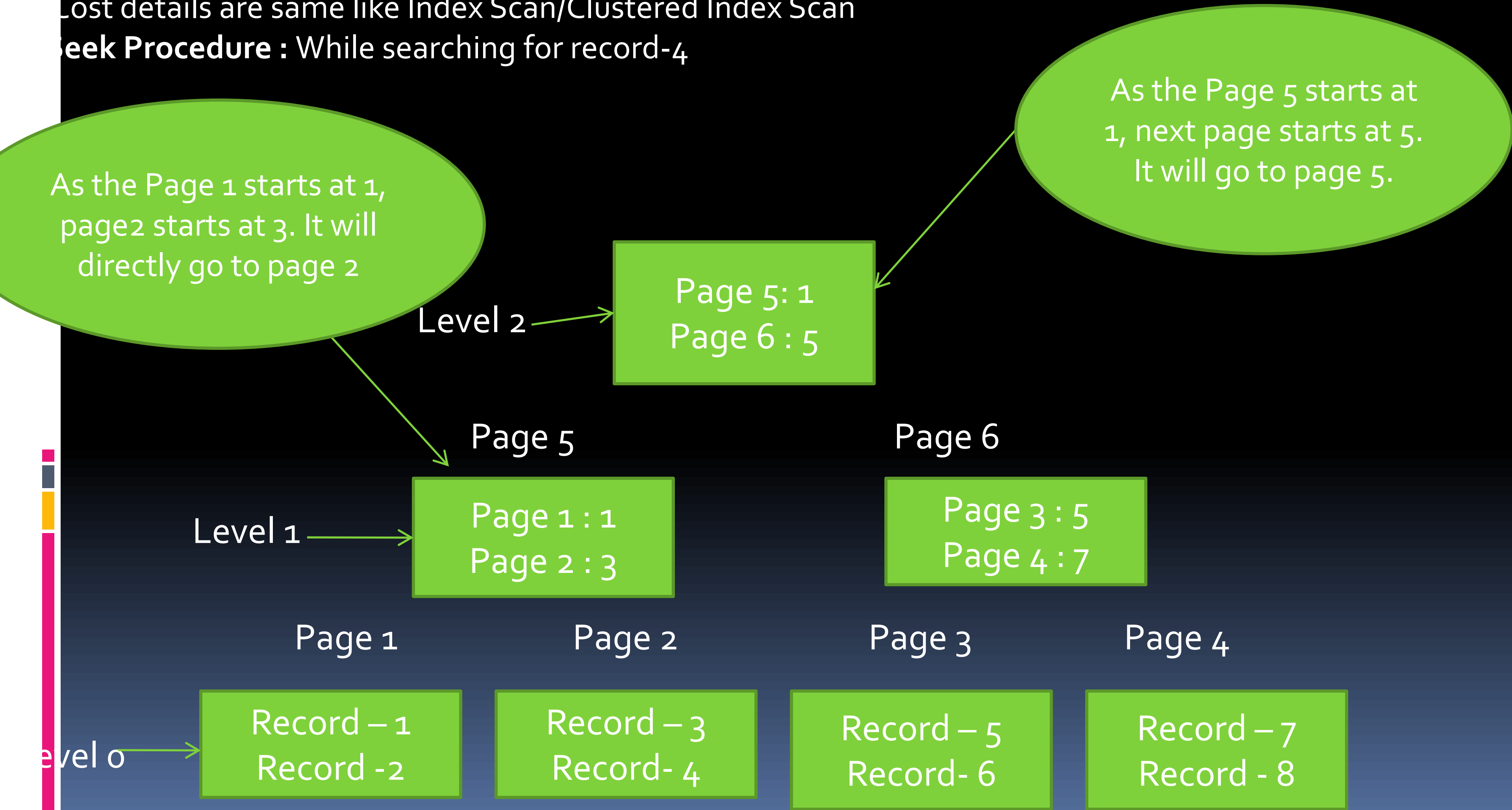
2. Range Scan : Initially it will traverse through index levels to reach the initial record that qualifies, and then it performs a range scan until it reaches end of scan range.

ITERATORS - SCAN AND SEEK

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 ndex has been performed range scan or singleton lookups.

Cost details are same like Index Scan/Clustered Index Scan

eek Procedure : While searching for record-4



ITERATORS – KEY LOOKUP,RID LOOKUP

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:

```
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.

ITERATORS – NESTED LOOPS JOIN

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
               return (R1,R2)
     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

ITERATORS – NESTED LOOPS JOIN

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.

ITERATORS – MERGE JOIN

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 atleast one equi-join predicate(One Equals condition).

Algorithm

```
get first row R1 from input 1
get first row R2 from input 2
while not at the end of either input
  begin
    if R1 joins with R2
      begin
        return (R1, R2)
        get next row R2 from input 2
      end
    else if R1 < R2
      get next row R1 from input 1
    else
      get next row R2 from input 2
  end
```

In Merge join, each table is read at most once, so its cost is proportional to 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.

ITERATORS – 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 R1 from input 1 ,get first row R2 from input 2
Initialize counter to 0
while not at the end of either input
 begin
    if R1 joins with R2
       begin
             increment counter
           return (R1, R2)
           get next row R2 from input 2
       end
    else if R1 < R2
             If counter = 0
                   return (R1, null)
           get next row R1 from input 1
             Initialize counter to 0
    else
       get next row R2 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 * power(10,-7). For Every outer row, it costs, 43 * power(10,-7) and for every inner row, it costs 21 * power(10,-7)

ITERATORS – HASH JOIN

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 he hash function on the equi-join keys and builds a hash table

. Probe phase : In this phase, it reads all rows from the other input table(Usually larger table) and pply the hash function on the equi-join keys and will check the resultant hashtable is not empty or ot. If it not empty, then it will again checks the conditions(Because of collisions), then returns the ow.

Algorithm

```
or 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
```

ITERATORS – HASH JOIN

- 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 he probe phase.

This iterator also requires memory to build the hash table. If the memory is not enough to build the ash table, it will spills the some part of the hash table to disk. For next rows, it will write to disk or n-memory based on partition of hash. Then in probe phase also, based on hash-partition, it will heck 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 * power(10,-7). For Each Build table record, it costs, 189 * power(10,-7) and for every probe table row, it costs, 46 * power(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

ITERATORS – SEMI JOINS

```
declare @t table(ID int)
insert into @t values (1),(2),(3)
eclare @t1 table(ID int)
nsert into @t1 values (1),(4),(5)
elect * from @t t where exists (select Id from @t1 where ID = t.ID)
elect * from @t t where not exists (select Id from @t1 where ID = t.ID)

reate 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
elect 4

Num as

		select ROW_NUMBER() over (order by N1.Num) as rn from N N1,N N2,N N3,N N4
)
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 implement 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 query. If they are not in sort order, it will explicitly sort them. Otherwise it will use the appropriate index, if any row present. Its output rows also will be in sort order.

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

Algorithm :

```
Clear the current aggregate results
   clear the current group by columns
   for each input row
     begin
       if the input row does not match the current group by columns
         begin
           output the aggregate results
           clear the current aggregate results
           set the current group by columns to the input row
         end
       update the aggregate results with the input row
     end
```

ITERATORS – STREAM & HASH AGGREGATE

Query Hint :

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

Cost :

Estimated CPU : For each Record, it costs 11 * 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 * power(10,-7)
Then for every Build element it takes 244 * power(10,-7), and for every probe element it costs 64 * power(10,-7)

ITERATORS – STREAM & HASH AGGREGATE

Algorithm :

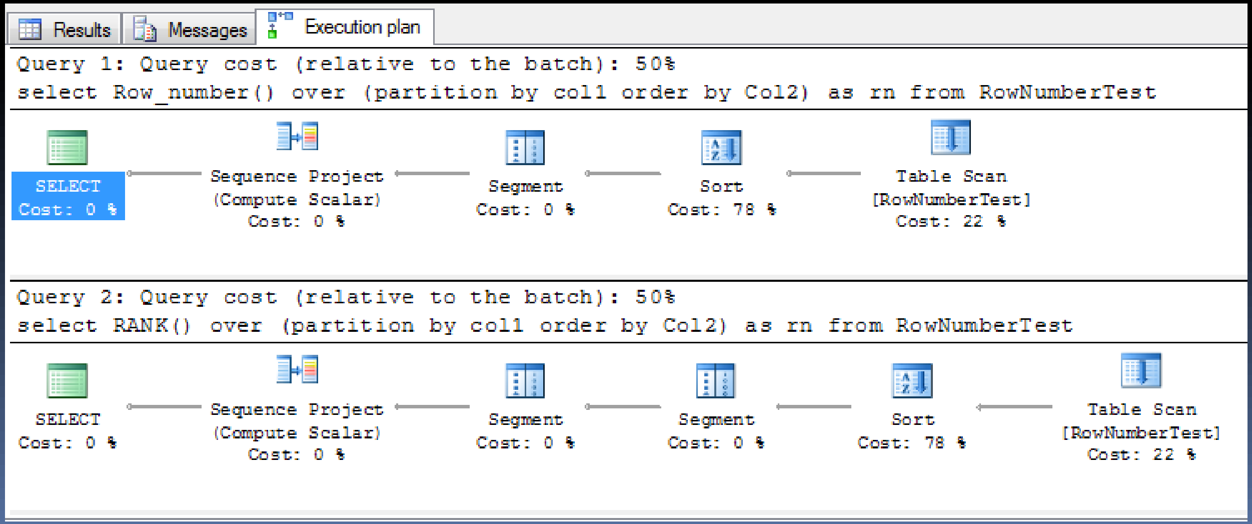
```
for each input row
    begin
      calculate hash value on group by column(s)
      check for a matching row in the hash table
      if we do not find a match
        insert a new row into the hash table
      else
        update the matching row with the input row
    end
  output all rows in the 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* power(10,-8) for each record

Eg :

```
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
```



ITERATORS – EAGER SPOOL,LAZY SPOOL

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).

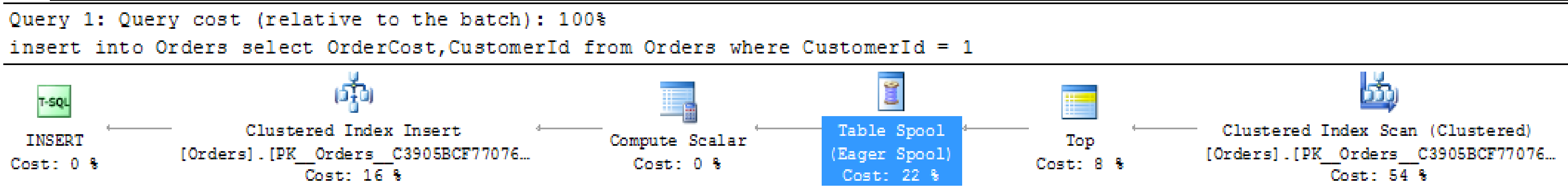
```
ig :
reate table Orders

            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
```

ITERATORS – EAGER SPOOL,LAZY SPOOL



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.

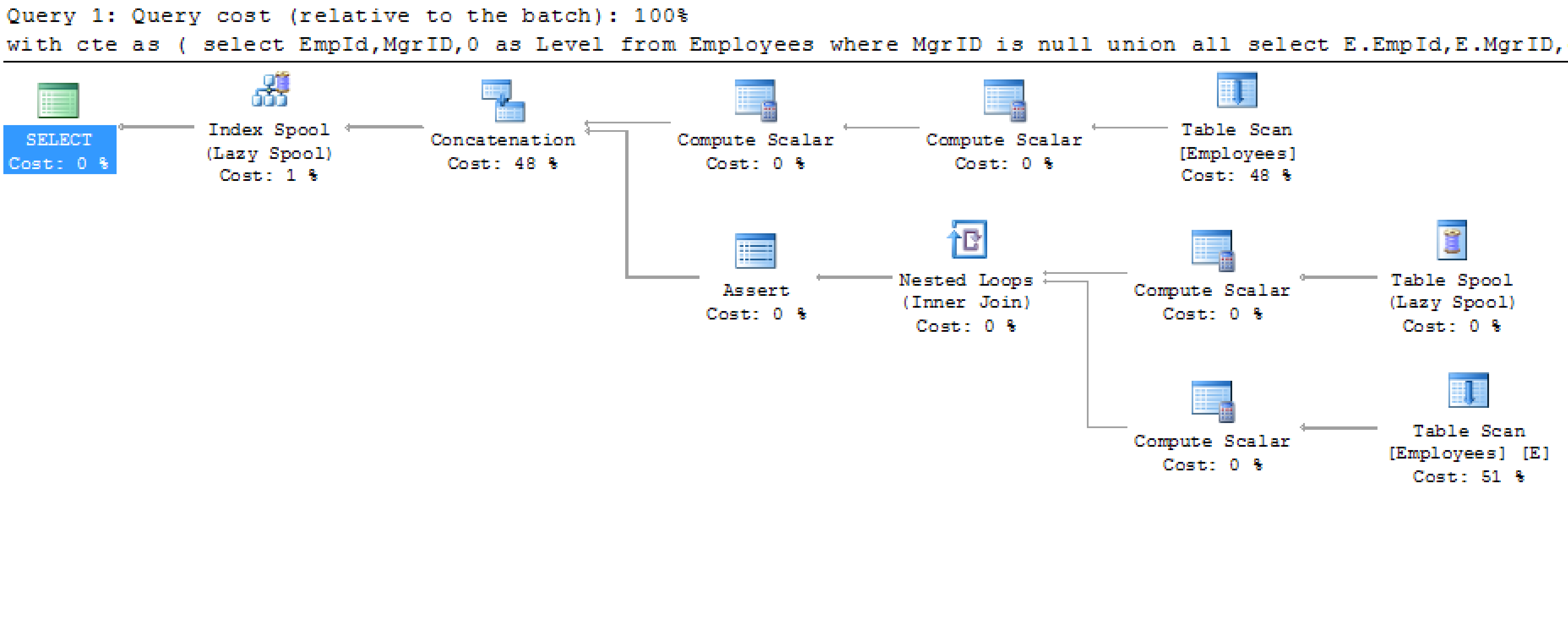
ITERATORS – EAGER SPOOL & LAZY SPOOL

Eg : **Lazy spools in recursive CTE**

```
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

elect * from cte
```



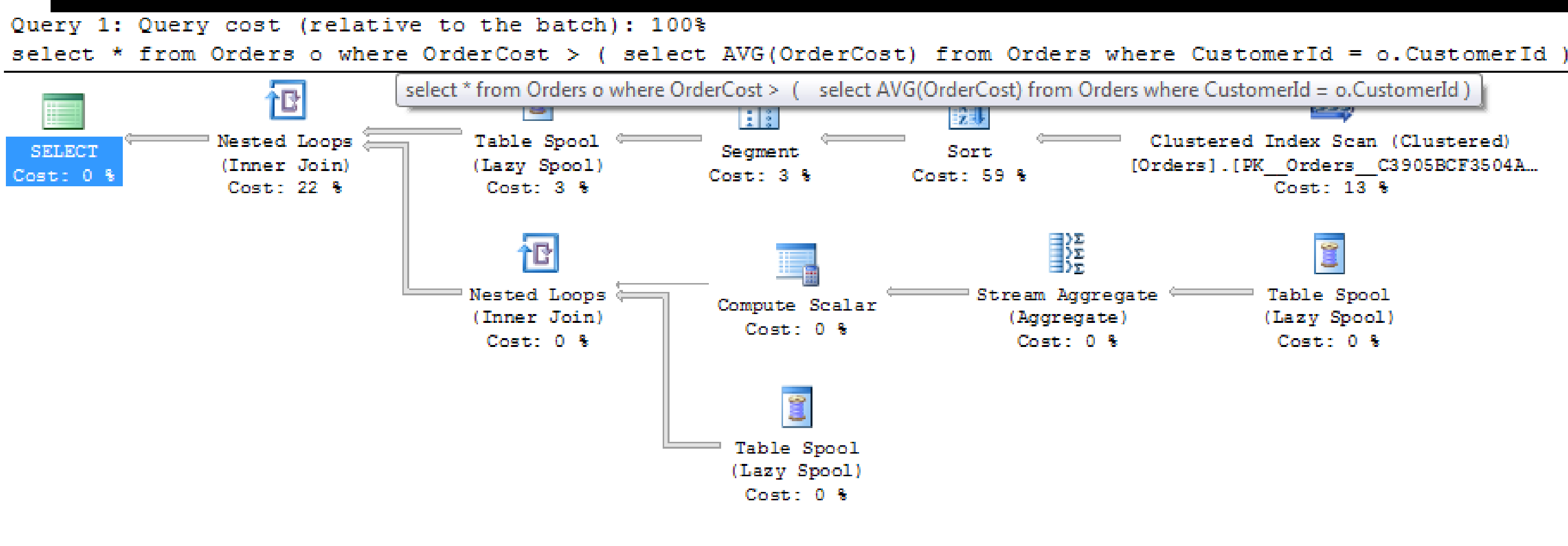
ITERATORS – EAGER SPOOL & LAZY SPOOL

Eg : **Lazy spools in subquery**

create table Orders

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

GO
nsert into Orders values
ABS(BINARY_CHECKSUM(newid())%100),ABS(BINARY_CHECKSUM(newid())%10000))
Go 10000
- Reordering orders of a customer
nsert into Orders
elect OrderCost,CustomerId from Orders where CustomerId = 1



RECURSIVE CTE EXECUTION