## Queries Analysis and optimization

We choose to take a deeper look at the queries 3.c, 3.m, and 3.n. The main criterion for this choice where:

* Long enough running time to be worth it. We believe it is not worth your time to optimize workloads that are good enough for the usage we make of them. In real development environments one would of course use the percentages of global running time and not the running time for a single run as a criterion (ie: if a fast, small query is 99% of your load and a big slow one represents 1% of your load; you should optimize the small one)
* After a quick glance at the execution plan that the DB reported we found issues with the queries.
* They presented different challenges that could be solved in different ways. Hence being more valuable as an academic experience.

### 3.C

### Statement

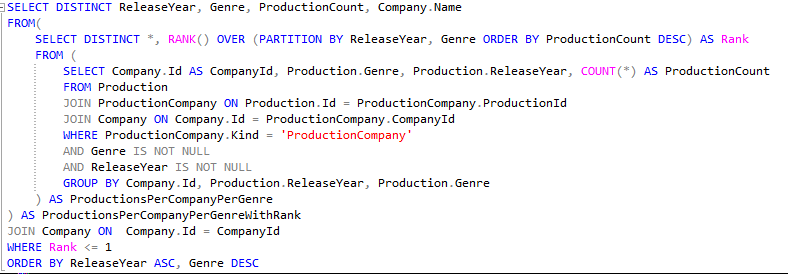
Given a year, list the company with the highest number of productions in each genre.

*NOTE: we decided to generate the results for each year*

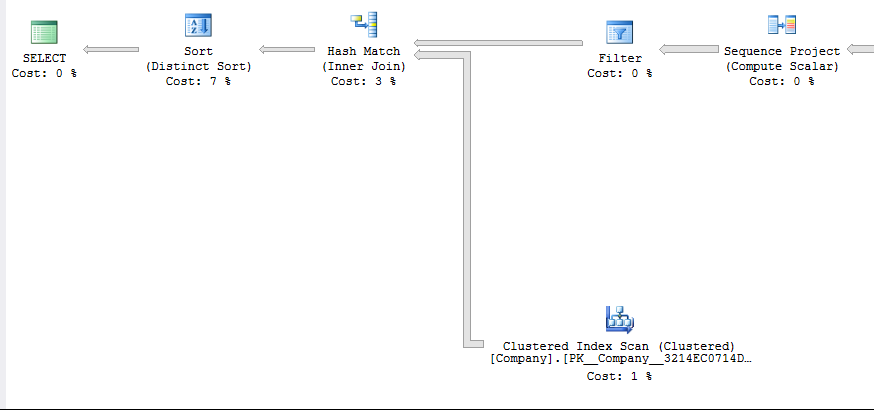
### Error in Query

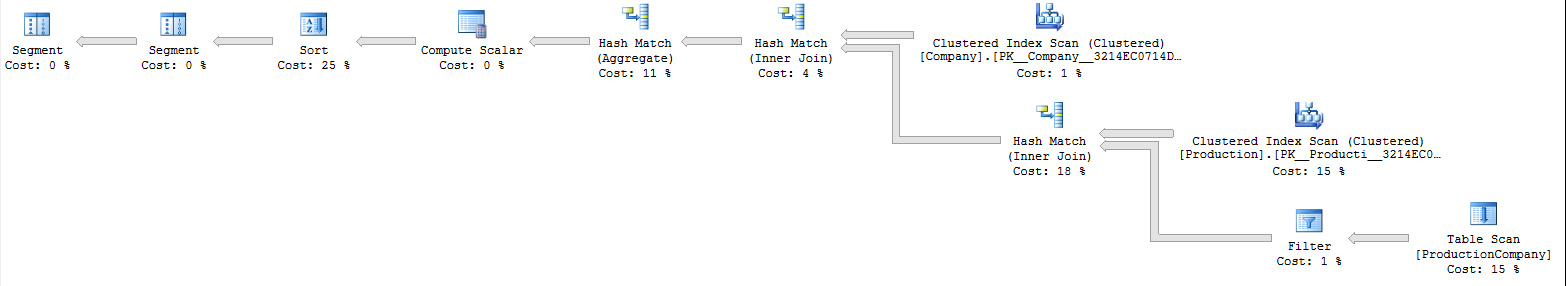
While trying to optimize our query, we found an error. This was a valuable experience of how taking a close look at an execution plan can help you understand what your query really does and find errors.

Here is the original (wrong) query:



And its execution plan:

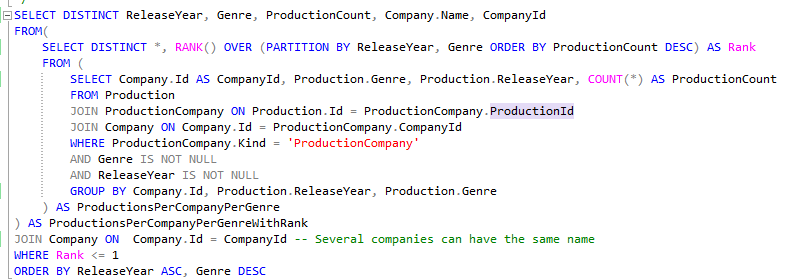




Our query contains distinct clauses, which in our minds were free, as the combination of release year, genre and company name should be unique. We found a difference in how the DB engine planned our query with or without the DISTINCT clause. Even the results where different.

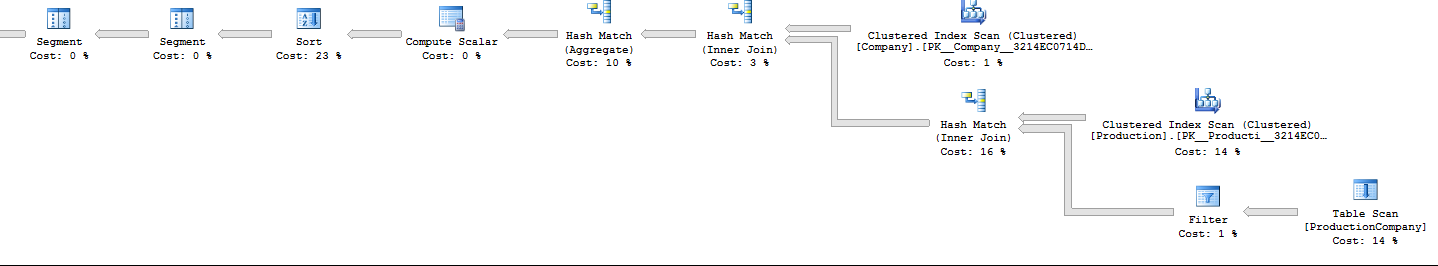
The error was in the wrong assumption that the company names are unique.

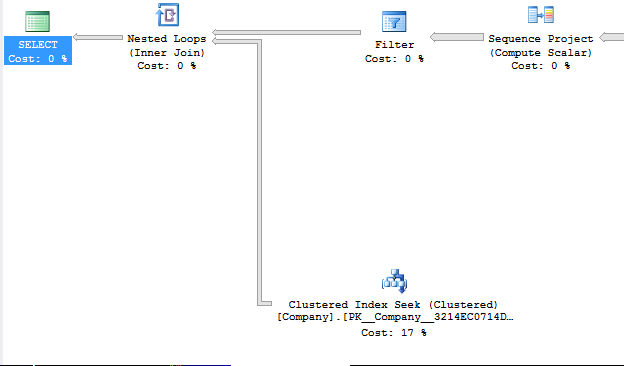
When we changed the query to:



A seamingly beging change (we added CompanyId) but the entire execution plans was changed ! Now there are now distinct sorts, which makes sense, in both SELECT DISTINCT statements, primary keys are included, which by definition are unique.

### Current Execution Plan





### Current Running Time

 SQL Server Execution Times:

   CPU time = 0 ms,  elapsed time = 0 ms.

SQL Server parse and compile time:

   CPU time = 0 ms, elapsed time = 0 ms.

    (6528 row(s) affected)

Table 'Company'. Scan count 1, logical reads 22545, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'ProductionCompany'. Scan count 1, logical reads 26783, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Production'. Scan count 1, **logical reads 28651**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

    (1 row(s) affected)

 SQL Server Execution Times:

   CPU time = **5312 ms**,  elapsed time = 5384 ms.

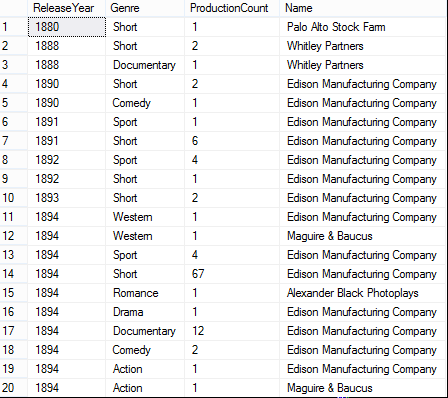
SQL Server parse and compile time:

   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:

   CPU time = 0 ms,  elapsed time = 0 ms.

### Output

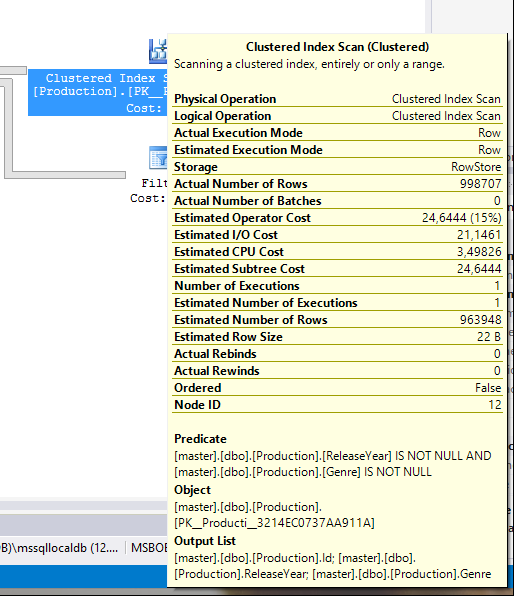


### Analysis and Optimization

MS SQL server recommends us to create a non-clustered index on *Production* (*ReleaseYear, Genre*), including *Id*

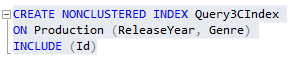
Looking at the query we can guess that this would make our GROUP BY clause (in the innermost query) run faster. The inclusion of *Id* is important, it enables the database engine to only use the index (i.e. not the actual table data), since all the data needed for the query is in the index.

In the execution plan we can see a clustered scan on *Production.Id*:

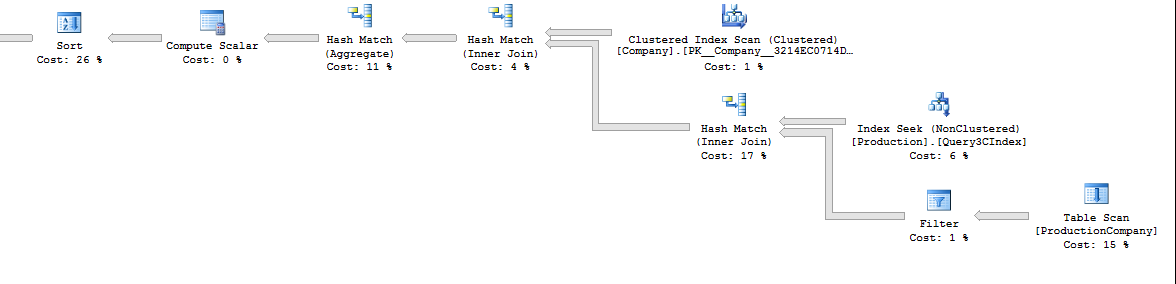


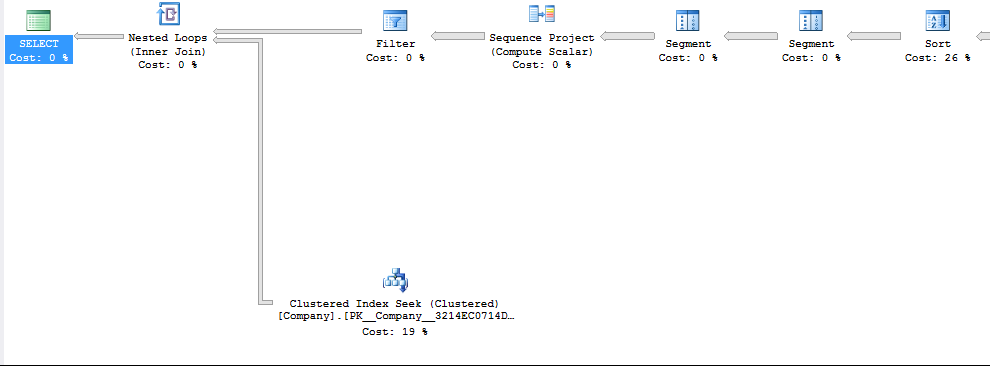
With an index our NOT NULL clause would be almost free.

### Added index



### New Execution Plan





### New Execution Time

 SQL Server Execution Times:

   CPU time = 0 ms,  elapsed time = 0 ms.

SQL Server parse and compile time:

   CPU time = 0 ms, elapsed time = 0 ms.

    (6528 row(s) affected)

Table 'Company'. Scan count 1, logical reads 22545, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'ProductionCompany'. Scan count 1, **logical reads 26783**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Production'. Scan count 1, logical reads 7382, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

    (1 row(s) affected)

 SQL Server Execution Times:

   CPU time = **5125 ms**,  elapsed time = 5163 ms.

SQL Server parse and compile time:

   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:

   CPU time = 0 ms,  elapsed time = 0 ms.

### Difference

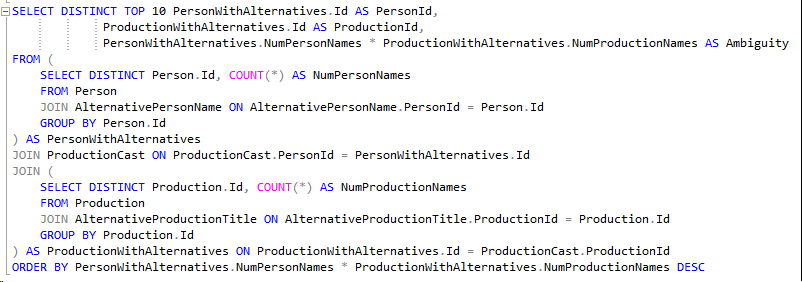
Our execution time went from 5350ms to 5130ms on average, a 4% speed increase, although small it can make a huge difference. The same order of magnitude of change can be seen in the number of logical reads on ProductionCast (where we added the index)

## 3.m

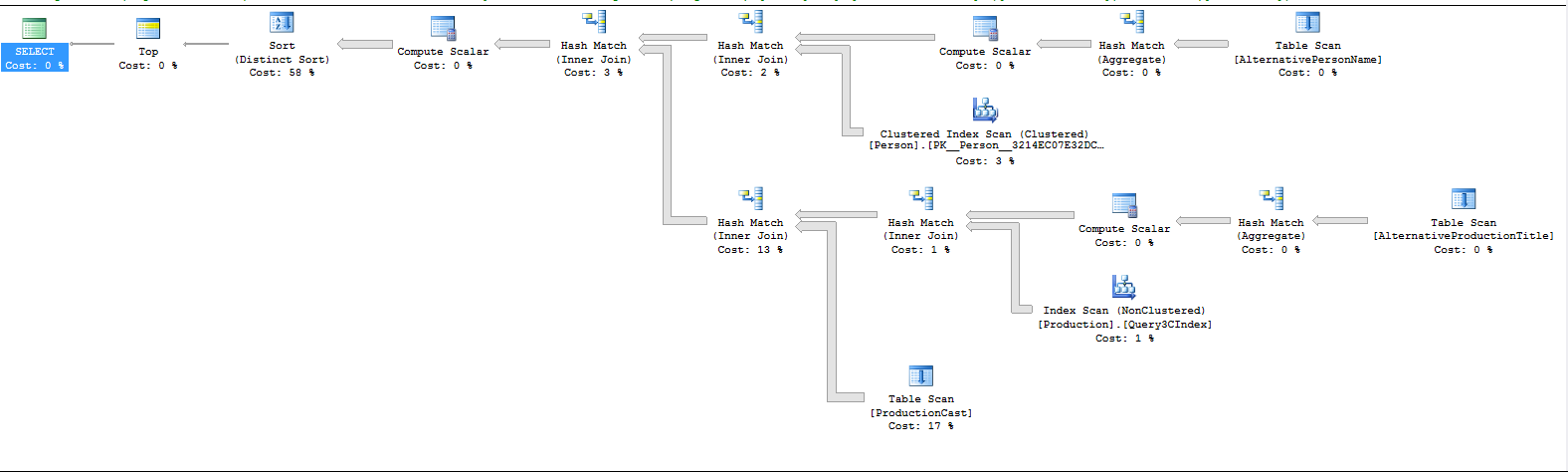
### Statement

List 10 most ambiguous credits (pairs of people and productions) ordered by the degree of ambiguity. A credit is ambiguous if either a person has multiple alternative names or a production has multiple alternative titles. The degree of ambiguity is a product of the number of possible names (real name + all alternatives) and the number of possible titles (real + alternatives).

### Query



### Current Execution Plan



### Current Running Time

 SQL Server Execution Times:

   CPU time = 0 ms,  elapsed time = 0 ms.

SQL Server parse and compile time:

   CPU time = 0 ms, elapsed time = 0 ms.

SQL Server parse and compile time:

   CPU time = 21 ms, elapsed time = 21 ms.

    (10 row(s) affected)

Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'ProductionCast'. Scan count 1, **logical reads 300485**, physical reads 0, read-ahead reads 300485, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Production'. Scan count 1, logical reads 7649, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'AlternativeProductionTitle'. Scan count 1, logical reads 3649, physical reads 0, read-ahead reads 3649, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Person'. Scan count 1, logical reads 66751, physical reads 0, read-ahead reads 66503, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'AlternativePersonName'. Scan count 1, logical reads 6703, physical reads 0, read-ahead reads 6703, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

    (1 row(s) affected)

 SQL Server Execution Times:

   CPU time = **19922 ms**,  elapsed time = 20656 ms.

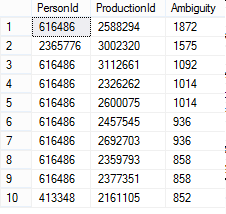
SQL Server parse and compile time:

   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:

   CPU time = 0 ms,  elapsed time = 0 ms.

### Output



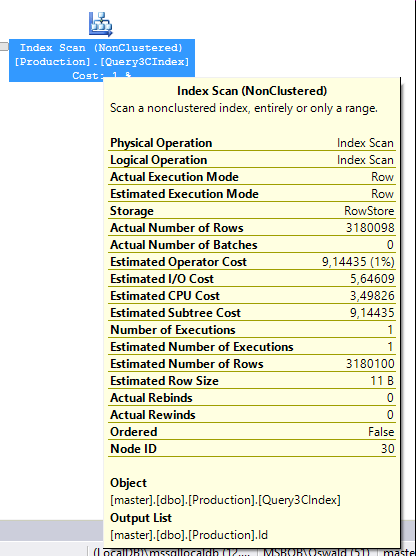
*Note: we list the ids since by definition of ambiguity we would have to list a lot of confusing names.*

### Analysis & Optimization

First we see that 58% of the time is spent on a distinct sort, taking the TOP 10. Just to be sure we didn’t mess things up with DISTINCT again, we compared the query without DISTINCT (Person.Id and Production.Id should be unique so DISTINCT shouldn’t matter).

The name of the step changed to Sort (TOP N Sort) but the cost didn’t change. Indeen the DISTINCT clause is here just an indication that they are distinct.

Secondly our execution plan uses Query3CIndex created earlier. Which is quite weird since there is an index on Production.Id (it’s a primary key) and the Index scan only outputs Production.Id.



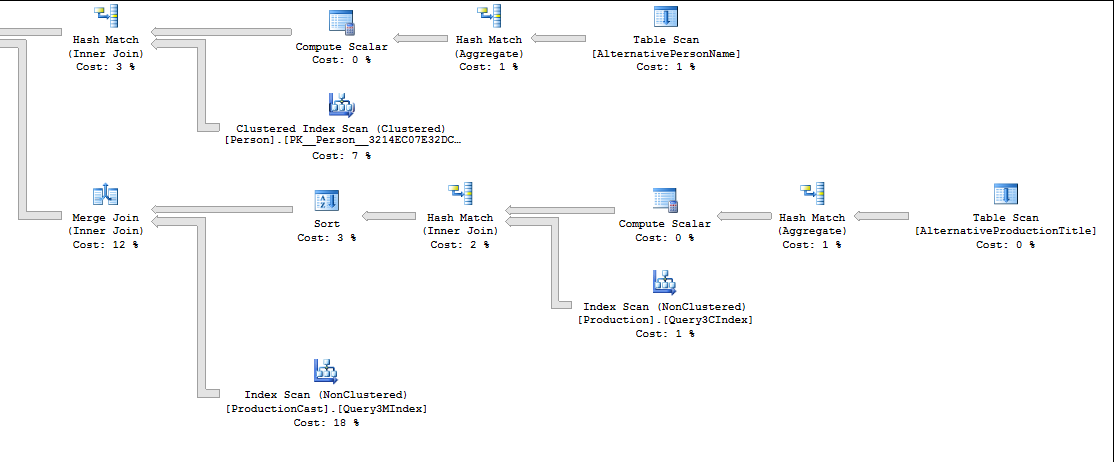
Since it is only 1% of the query cost, we won’t try to fix it/understand why the DB engine choose to use that index instead of the primary key one.

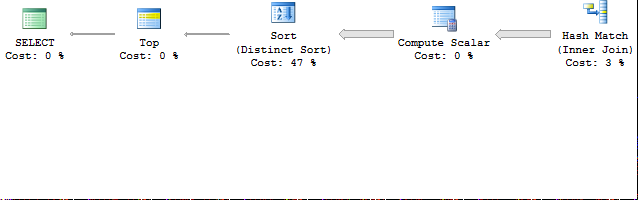
Lastly in the Execution plan we can see that the second most expensive step is the scan of Production cast & the join, using ProductionCast as join table from Production to Person. This is a very common use of ProductionCast throughout our queries, hence adding an index would be valuable.

### Added index



### New Execution Plan





### New Running Time

 SQL Server Execution Times:

   CPU time = 0 ms,  elapsed time = 0 ms.

SQL Server parse and compile time:

   CPU time = 0 ms, elapsed time = 0 ms.

    (10 row(s) affected)

Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'ProductionCast'. Scan count 1, **logical reads 119989**, physical reads 2662, read-ahead reads 109840, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Production'. Scan count 1, logical reads 7649, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'AlternativeProductionTitle'. Scan count 1, logical reads 3649, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Person'. Scan count 1, logical reads 66751, physical reads 9, read-ahead reads 62458, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'AlternativePersonName'. Scan count 1, logical reads 6703, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

    (1 row(s) affected)

 SQL Server Execution Times:

   CPU time = **15094 ms**,  elapsed time = 15588 ms.

SQL Server parse and compile time:

   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:

   CPU time = 0 ms,  elapsed time = 0 ms.

### Difference

The table scan on ProductionCast has been switched with a NonClustered index scan. Additionally, the Hash join has been switched to a Merge Join (only works if tables are suitably sorted).

Lastly, our running time went from 19900ms to 15150ms (average of 3 runs), a 24% gain!

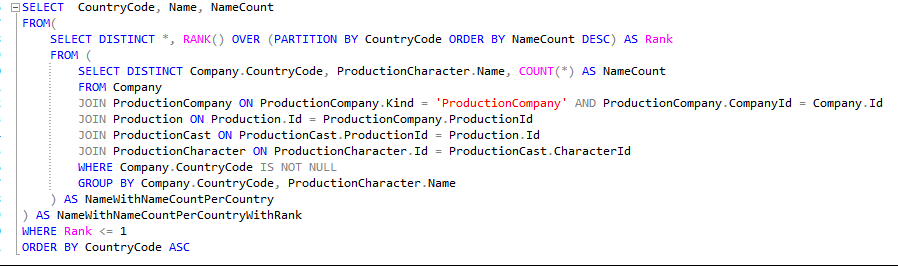
Which is also reflected in the IO report (ProductionCast: logical reads 300485 -> 119989 logical reads)

## 3.N

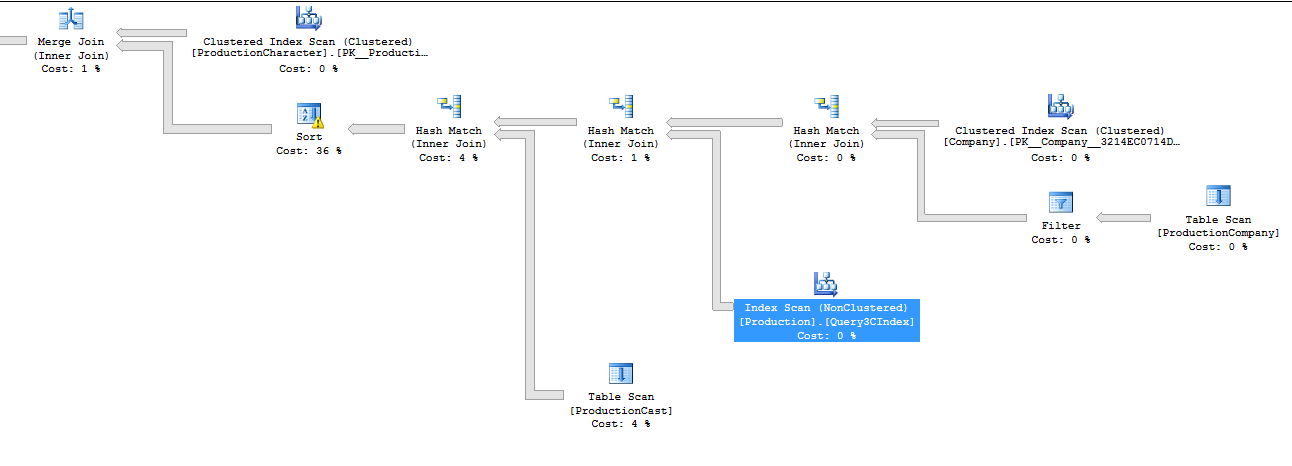
### Statement

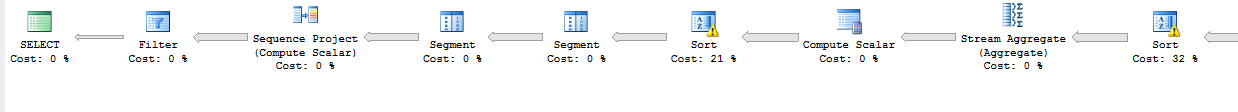
For each country, list the most frequent character name that appears in the productions of a production company (not a distributor) from that country.

### Query



### Current Execution Plan





### Current Running Time

 SQL Server Execution Times:

   CPU time = 0 ms,  elapsed time = 0 ms.

SQL Server parse and compile time:

   CPU time = 0 ms, elapsed time = 0 ms.

    (632 row(s) affected)

Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'ProductionCast'. Scan count 1, **logical reads 300485**, physical reads 0, read-ahead reads 300279, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Production'. Scan count 1, **logical reads 7649**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'ProductionCompany'. Scan count 1, **logical reads 26783**, physical reads 0, read-ahead reads 26783, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Company'. Scan count 1, **logical reads 2544**, physical reads 3, read-ahead reads 2540, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'ProductionCharacter'. Scan count 1, **logical reads 27687**, physical reads 3, read-ahead reads 27681, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

    (1 row(s) affected)

 SQL Server Execution Times:

   CPU time = **191734 ms**,  elapsed time = 196209 ms.

SQL Server parse and compile time:

   CPU time = 0 ms, elapsed time = 0 ms.

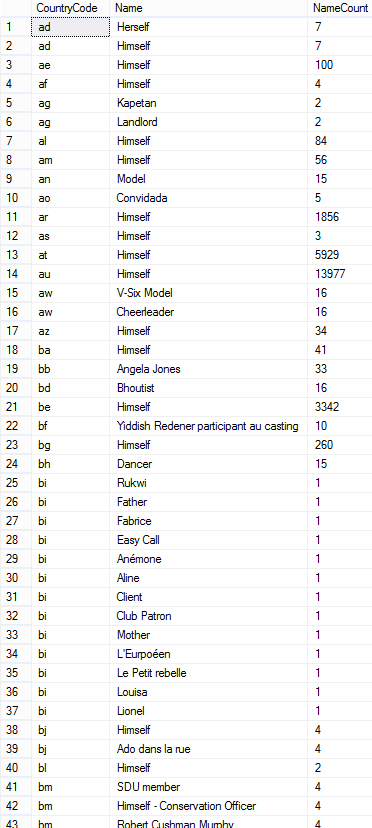
 SQL Server Execution Times:

   CPU time = 0 ms,  elapsed time = 0 ms.

 SQL Server Execution Times:

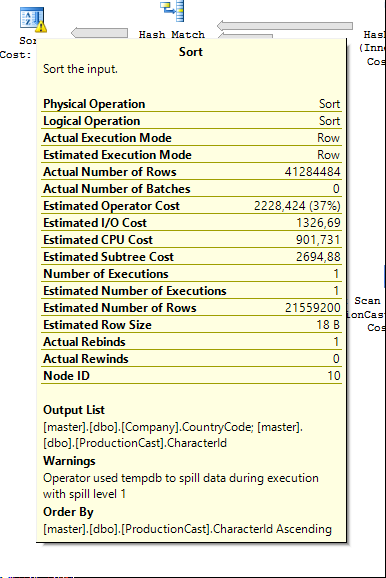
   CPU time = 0 ms,  elapsed time = 0 ms.

### Output



### Analysis & Optimization

In the execution plan, we discovered a whole bunch of yellow warning triangles. After a closer inspection, these are spill warnings from the DB Engine.

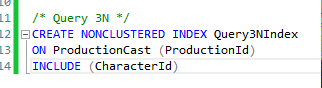


There is a big discrepancy between the estimated number of rows and the actual number of rows. Since MS SQL Server allocates memory before execution, there is no choice but to spill to tempdb.

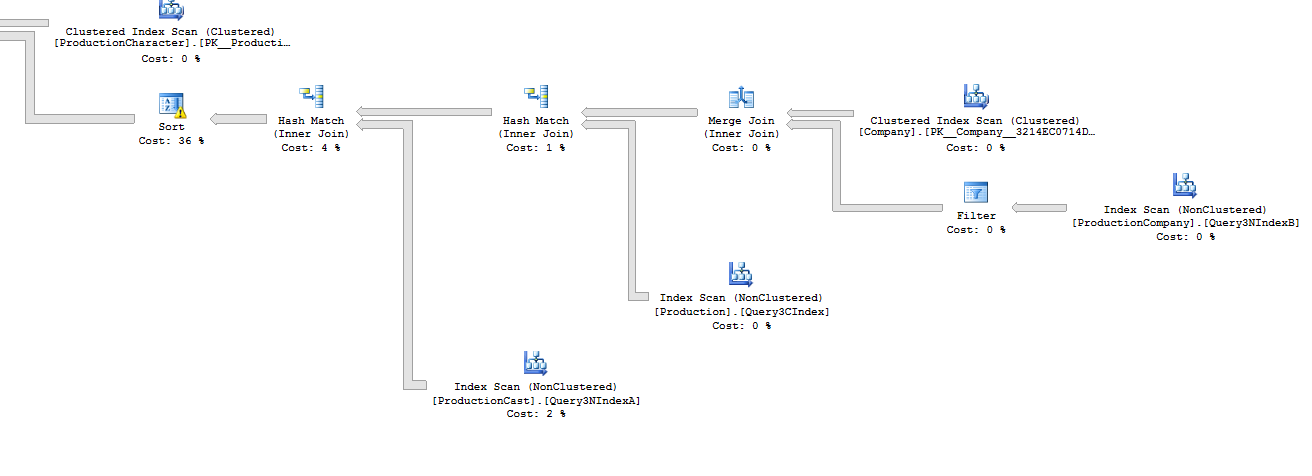
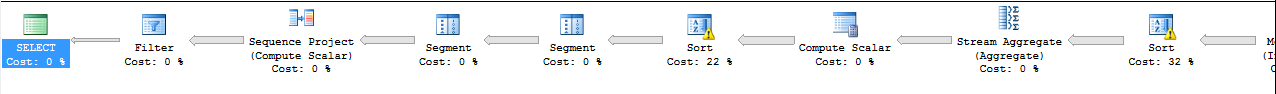
Unfortunately, it is very hard to get those statistics correctly.

We did accelerate the query marginally by adding an index on ProductionCast.

### Added index



### New Execution Plan

### New Running Time

 SQL Server Execution Times:

   CPU time = 0 ms,  elapsed time = 0 ms.

SQL Server parse and compile time:

   CPU time = 30 ms, elapsed time = 30 ms.

    (632 row(s) affected)

Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'ProductionCast'. Scan count 1, **logical reads 120005**, physical reads 0, read-ahead reads 119692, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Production'. Scan count 1, **logical reads 7649**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'ProductionCompany'. Scan count 1, **logical reads 26783**, physical reads 0, read-ahead reads 26783, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'Company'. Scan count 1, **logical reads 2544**, physical reads 0, read-ahead reads 2533, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table 'ProductionCharacter'. Scan count 1, **logical reads 27687**, physical reads 0, read-ahead reads 27582, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

    (1 row(s) affected)

 SQL Server Execution Times:

   CPU time = **171360** **ms**,  elapsed time = 174317 ms.

SQL Server parse and compile time:

   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:

   CPU time = 0 ms,  elapsed time = 0 ms.

### Difference

We can see a drastic reduction in logical reads on ProductionCast, accompanied by a small decrease in total running time (191500 ms -> 172 000ms). This can be explained by the spilling to tempdb, that dominates the running time.

## Cost

Index size before optimizations



### Index size after optimizations



We see that the indexes are a trade-off, on one side they can improve query performance, on the other they take disk/RAM space.

## Conslusion

* There is no magic way to optimize a query, you need to take a deep look at what the DB engine actually does.
* It’s all about tradeoffs.
* Taking a good look at the execution plan can help you find bugs in queries