DBSophic

PHYSICAL JOIN OPERATORS in SQL SERVER

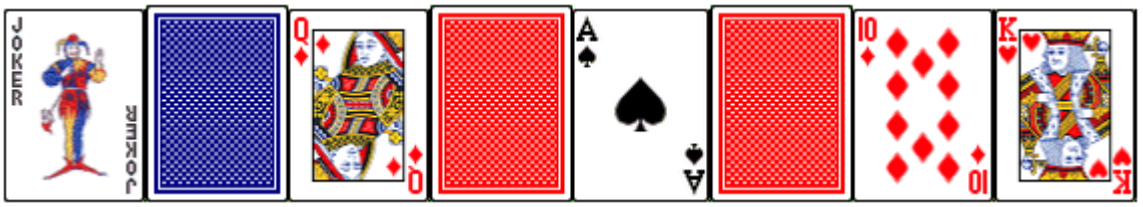
AUTHOR:

Ami Levin

Ami Levin is a Microsoft SQL Server MVP, with over 20 years of experience in the IT industry. For the past 16 years he has been consulting, teaching and speaking on SQL Server worldwide. He moderates the Israeli MSDN SQL Server support forum, and is a regular speaker at Microsoft conferences. Levin's areas of expertise are data modeling, database design, T-SQL and performance tuning.

TABLE OF CONTENTS:

CHAPTER 1:
Introduction



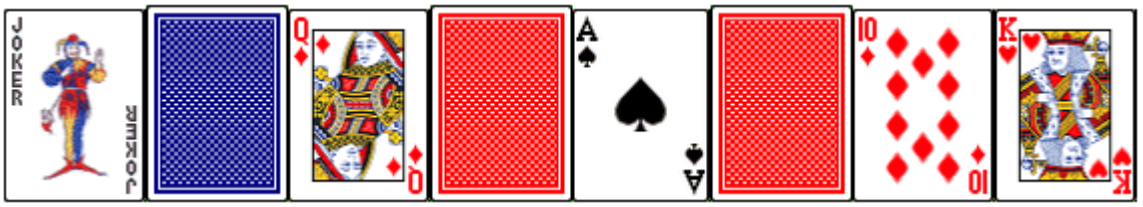
SQL Server implements three different physical operators to perform joins. In this eBook we will examine how each operators works, its advantages and challenges. We will try to understand the logic behind the optimizer's decisions on which operator to use for various joins using (semi) real life examples and how to avoid common pitfalls.

In this eBook, I'll focus on the most common type of join, the 'E-I-J' (or 'Equi-Inner-Join') which is just a cool sounding name for an inner join that uses only equality predicates for the join conditions. With sadness, I will skip some extremely interesting issues concerning joins such as logical processing order of outer joins, NULL value issues, join parallelism and others. This eBook focuses on helping you make your E-I-J joins faster. We'll accomplish this by helping you understand how the SQL Server optimizer decides which physical operators it will use to carry out your query joins. We'll show you situations where the SQL Server optimizer is occasionally tricked into choosing a slower method by the characteristics of the data and SQL you wrote. If you understand these pitfalls, you can code to overcome them, thus speeding up your queries - and impressing your colleagues and bosses.

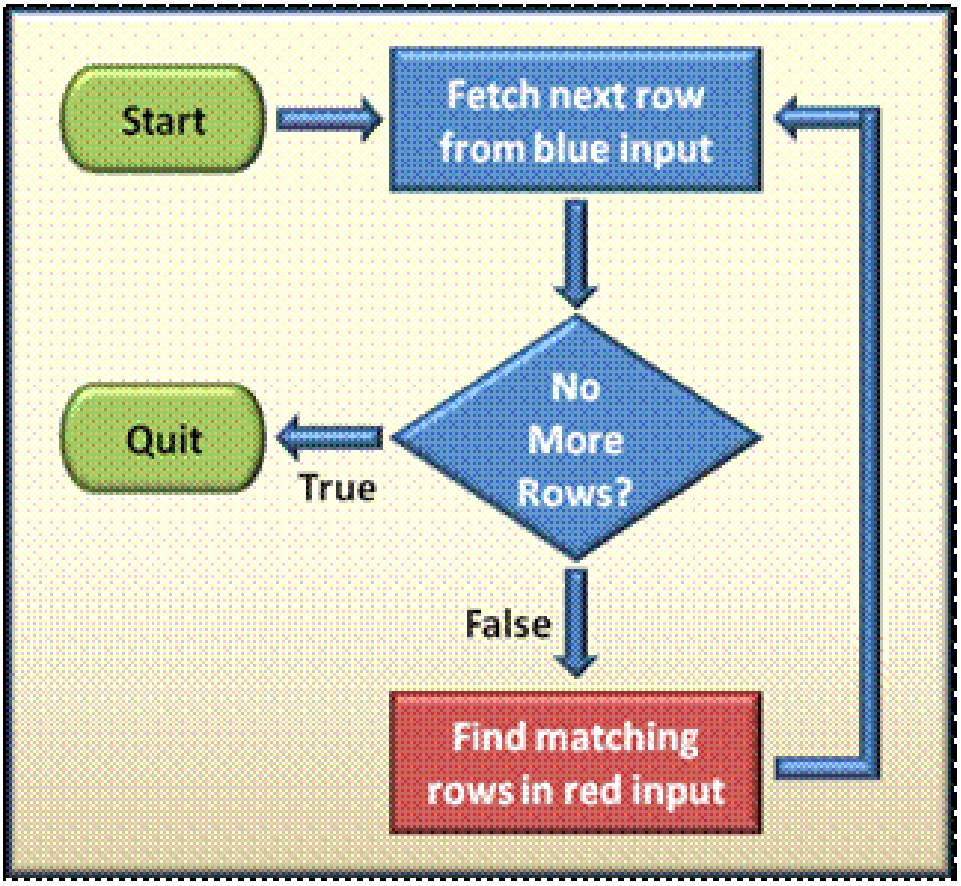
As amazing as the SQL Server optimizer can sometimes be, it's not making its decisions intuitively. It's using the information it has to work with. Recall that a join is basically 'looking for matching rows' from two inputs, based on the join condition. The creators of the optimizer have made three techniques available to it for carrying out these joins:

- Nested Loops
- Merge
- Hash Match

For this eBook I will use an analogy of two sets of standard playing cards. One set with a blue back and another with a red back that need to be 'joined' together according to various join conditions. The two sets of cards simply represent the rows from the two joined tables, the red input and the blue input - "Now Neo - which input do you choose"?

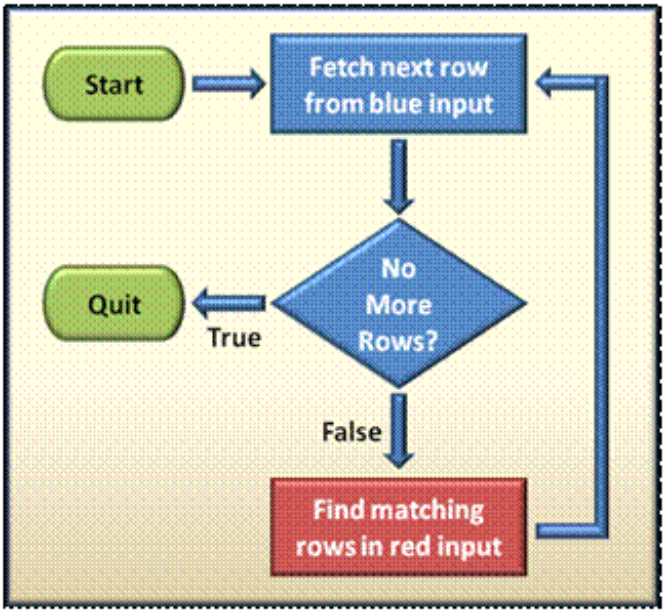


CHAPTER 2:
Nested Loops



Probably the simplest and most obvious operator is the nested loops operator. The algorithm for this operator is as follows:

The 'outer loop' consists of going through all rows from the blue input and for each row; some mysterious 'inner operator' is performed to find the matching rows from the red input. If we would use this operator to join our sets of cards, we would browse through all the blue-back cards and for each one we would have to go through all the red-back cards to find its matches. Not the most efficient way of matching cards but it probably would be most people's first choice... **if** there are only a handful of cards to match.



Flowchart 1 – Nested Loops

Let's take a closer look and see when nested loops are a good choice. The first parameter to consider, as with any iterative loop, is the number of required iterations. For nested loops to be efficient, it requires at least one relatively small row set to be used as the outer loop input that determines the number of iterations.

Remember that this does not necessarily reflect the number of rows in the table but the number of rows that satisfy the query filters.

The second parameter to consider is the (intentionally) vague "Find matching rows in red input" part in Flow Chart 1 above. Assuming that we do have a small input for the outer loop, hence a small number of required iterations, we now must consider how much work is required to actually find the matching rows in every iteration. "Find matching rows" might consist of a highly efficient index seek but it might require a full table scan if the join columns are not properly indexed; making nested loops a far less optimal plan. Since it is very common for joins to be performed on one-to-many relationships, and since the parent node in these relationships is often the smaller input (the 'one' in the 'one-to-many'), and since this parent node is always indexed (must be a primary key or a unique constraint), the well known best practice rule that requires that FK columns should be indexed, now makes perfect sense, doesn't it?

Let’s see a few examples

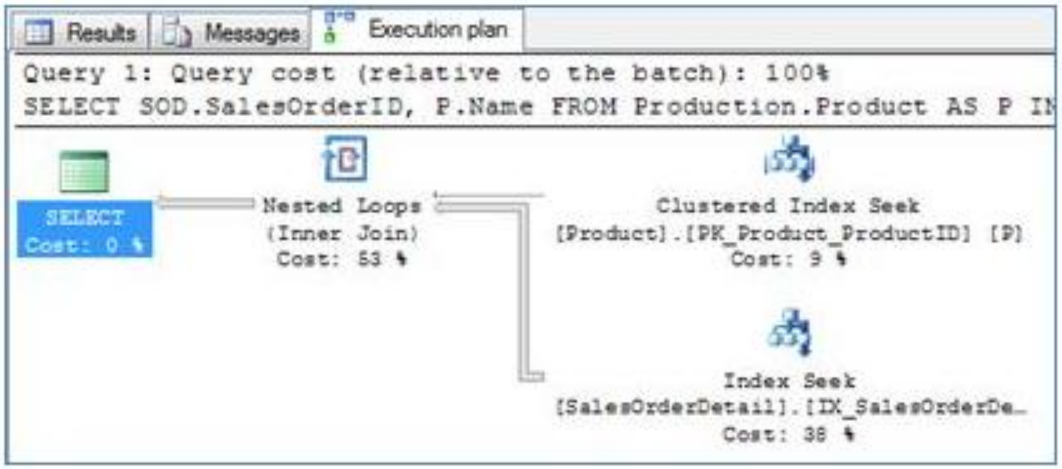
* **Note:** All the queries used in this chapter can be found in the demo code file, plus a few more. I highly recommend that you play around with the code, change parameters and select list columns, try to use outer joins and even create some indexes and see how they affect the query plans. Just remember to drop your indexes before proceeding to the next query so that the demo plans will remain valid.

SMALLER OUTER LOOP WITH A WELL INDEXED INNER OUTPUT

The most obvious case for nested loops is when one input is very small (is 1 row small enough?) and the other input is ideally indexed for the join. In Execution Plan 1 you can see that the optimizer retrieved the single row for product 870 from the *Product* table using a PK clustered index seek (needed to retrieve the *Product* Name) and all the matching *Order ID*s were retrieved using the non clustered index on the *Product ID* column of the *Order Detail* table.

```
SELECT      SOD.SalesOrderID,
            P.Name
FROM        Production.Product AS P
            INNER JOIN
            Sales.SalesOrderDetail AS SOD
            ON P.ProductID = SOD.ProductID
WHERE       P.ProductID    = 870
```

Query 1



Execution Plan 1

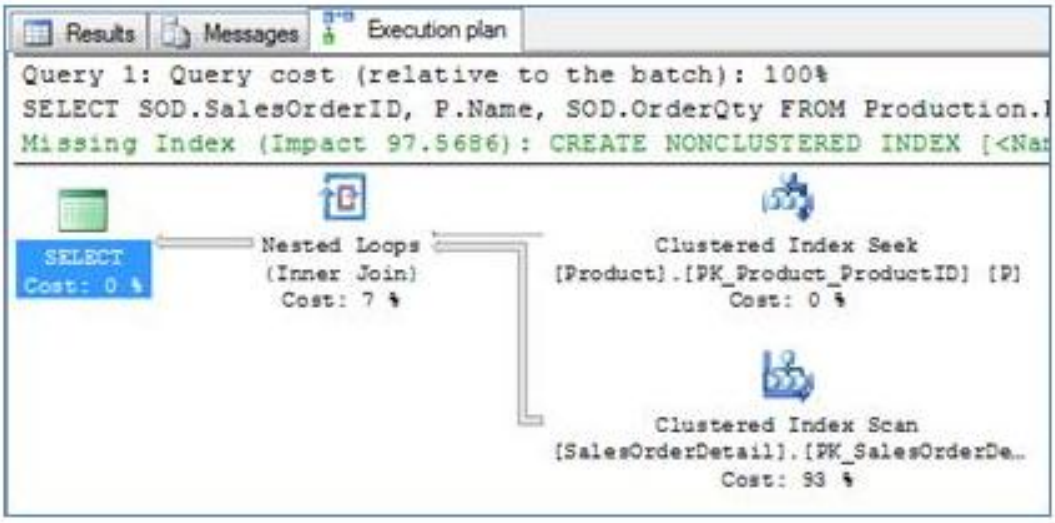
*** Tip:** The optimizer was clever enough to duplicate the filter for Product ID to both tables although the predicate specifies only the Product table. Since the filter and the join condition are based on the same column and since the join is an equi-join, the optimizer knows that when constructing its physical operators, it can add a second predicate 'AND SOD.ProductID = 870' to optimize the data access for the Order Detail table.

NESTED LOOPS JOINS WITH TABLE SCANS

If we just add the Order Quantity column to the select list as in Query 2, the index on Product ID (which does not contain the Order Quantity column) is now not enough to 'cover' the query, meaning that it does not include all the data required to satisfy the query. The optimizer can either perform a 'lookup' (use the pointer from the index to fetch the full row from the table itself) for each Order Detail row to get it (nearly 5,000 lookups...) or alternatively it can simply scan the Order Detail table, retrieve both Product ID and Order Quantity. In this case, since there is just one Product row to join hence the table only needs to be scanned once, and since the table is not very large (which would make the scan expensive), the single scan option is probably better than the alternative of performing thousands of lookups. Indeed, this is exactly what the optimizer chooses to do in this situation as you can see in Execution Plan 2.

```
SELECT      SOD.SalesOrderID,
            P.Name,
            SOD.OrderQty
FROM        Production.Product AS P
            INNER JOIN
            Sales.SalesOrderDetail AS SOD
            ON P.ProductID = SOD.ProductID
WHERE       P.ProductID = 870
```

Query 2



Execution Plan 2

* **Exercise:** Try to change the predicate to 'P.ProductID IN (870,871)'. This would mean that now, to use the same plan as above will require two full scans instead of just one. Do you think that this will still be the most efficient plan? Try it and see what plan the optimizer chooses. You can find the code for this exercise (Code sample #2b) in the demo code.

NESTED LOOP JOINS WITH LOOKUPS

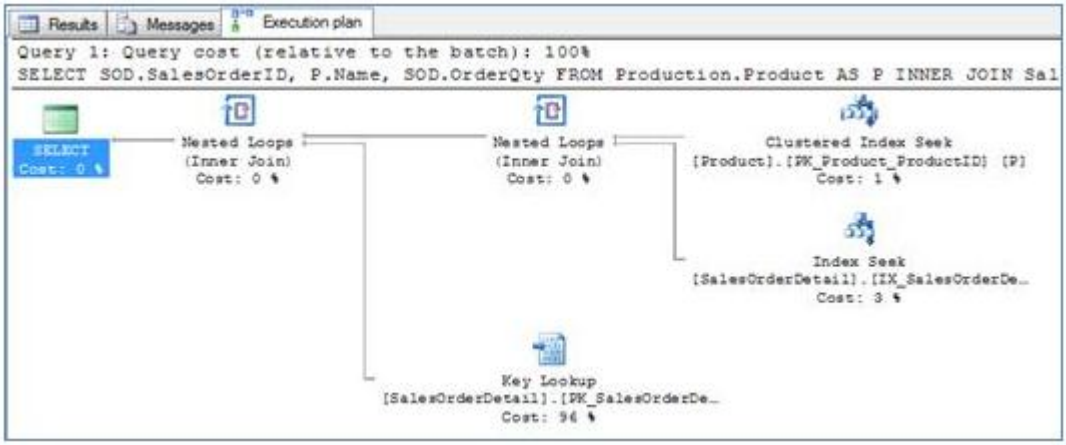
Now, let's play around a bit with the parameters. In Query 3, I have changed the predicate to filter for three products instead of one, but I used products that are far less commonly ordered. Think of three colors of 'White-Out' ('Tipp-Ex' for you Europeans)

for the computer screen - not the hottest seller... Now, a slightly different use of nested loops emerges. Note that product 870 exists in nearly 5,000 *Order Detail* rows, but only 13 *Order Detail* rows contain one or more of Query 3's three products (897,942,943). The optimizer always consults the column statistical histograms so it is very aware of this fact. To join only product 870, the optimizer chose to perform a single scan instead of an index seek + 5,000 additional lookups. For the three products in Query 3, it could either perform three full scans or revert to using an index seek + 13 additional lookups. What do you think is the right choice?

The answer is graphically portrayed in Execution Plan 3. Note that the execution plan incorporates two nested loop physical operators, and that both constitute logical inner joins. The one on the right is our actual table join, and the one on the left denotes the full-record lookup from the *Order Detail* table (required in order to retrieve the *Order Quantity*) which, in a sense, is also a join between the *Order Detail* table and the result of the first join.

```
SELECT      SOD.SalesOrderID,
            P.Name,
            SOD.OrderQty
FROM        Production.Product AS P
            INNER JOIN
            Sales.SalesOrderDetail AS SOD
            ON P.ProductID = SOD.ProductID
WHERE       P.ProductID IN (897,942,943)
```

Query 3



Execution Plan 3

Beware of the 'Estimation Error Devil'

Probably one of the most common pitfalls of the query optimizer is under-estimating the number of required iterations for a nested loops operator due to (partial list):

- Statistical deviations
- Outdated statistical data

- 'Hard to estimate' predicates such as compound expressions, sub queries, functions or type conversions on filter columns
- Multiple predicates estimation errors

When analyzing a poorly performing query, one of the first things I do is to look at the 'Estimated number of rows' vs. the 'Actual number of rows' figures of the outer input of the nested loop operator. The outer input is the top one in the graphical query execution plan. See for example Execution Plan 4. I've seen many cases where the optimizer estimated that only a few dozen iterations will be required, making nested loops operator a very good choice for the join, but in fact tens (or even hundreds) of thousands of rows satisfied the filters, causing the query to perform very badly.

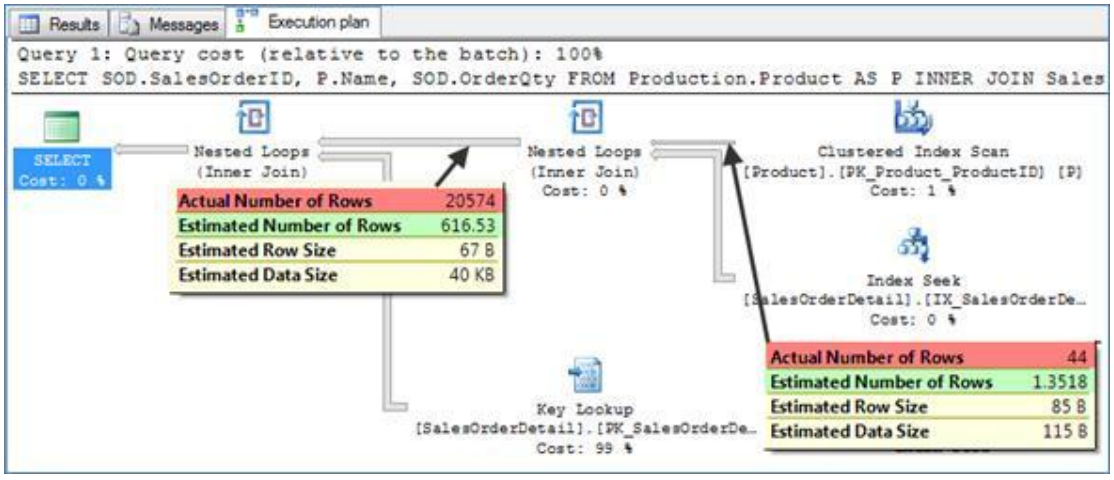
For Query 4 below, the optimizer estimated that ~1.3 products will satisfy the query filters when in fact, 44 products did. This seemingly small error led the optimizer to believe that only 616 rows will be matched from the *Order Detail* table and that estimation made it choose the same plan as the previous example, using an index seek and additional lookups. The optimizer estimated that it will require only 616 key lookups when in fact, 20,574 were

required. This might seem at first like a flaw in the optimizer, but only to the degree that mind reading is hard to program. You'll see why in the answer to Challenge #1 at the end of this section.

* **Challenge #1:** Can you guess what misled the optimizer to make the estimation error? See the answer at the end of this section.

```
SELECT      SOD.SalesOrderID,
            P.Name,
            SOD.OrderQty
FROM        Production.Product AS P
            INNER JOIN
            Sales.SalesOrderDetail AS SOD
            ON P.ProductID = SOD.ProductID
WHERE       P.DaysToManufacture > 2
            AND
            ListPrice > 1350
            AND
            StandardCost > 750
```

Query 4



Execution Plan 4

* **Exercise:** The demo code contains Query 4 as shown here, plus hints that force the optimizer to use nested loops, hash match or merge joins respectively (Queries 4b, 4c and 4d). Use the demo code to execute these query variations. Trace the executions with profiler to benchmark these alternatives and see if the optimizer was correct to prefer nested loops for this query. Remember that logical reads might be misleading so pay close attention to both CPU and duration when evaluating the queries' true efficiency.

WICKED A-SEQUENTIAL PAGE CACHE FLUSHES

One more issue we need to consider with nested loops is the issue of sequential vs. a-sequential page access patterns. You might recall from my article that nested loops are characterized with a high number of logical reads, as the same data page might be accessed multiple times. For small row sets, this is usually not an issue as the pages are cached once and subsequent reads are performed in memory. But, if the system experiences memory pressure and in the common case where the outer loop consists of a large number of rows where the distribution of the data (in respect to the order of the join columns

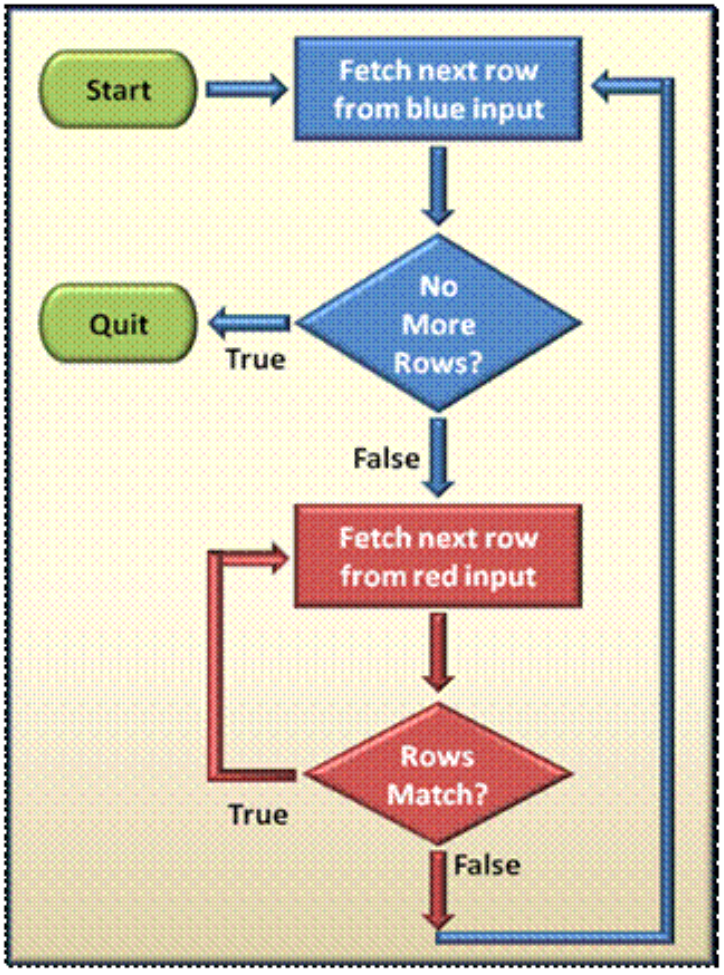
retrieved for the outer loop) within the pages is more or less random, a real performance nightmare might occur when a page is **repeatedly** flushed from the buffer cache only to be physically retrieved a few seconds later for retrieval of another row, for the same join! If you were the query optimizer - go with the flow for a moment - how would you avoid such a performance 'nightmare'?

*** Challenge #1 answer:** *The filter consists of three predicates. If you inspected each predicate by itself (armed with the statistical distribution histograms of the values in each column), you would see that the predicates are highly selective, meaning that they filter out a large percentage of the rows as the values used are very close to the maximum values for each of the columns. Using AND logical operators for these three highly selective predicates leads the optimizer to estimate that only a few rows will satisfy all three predicates. This is correct from a statistical perspective under the assumption that there is no correlation between the predicates. But... unbeknownst to the optimizer, the values in these three predicates are actually closely related. The query filters for the products that take the longest to manufacture, making them the products with the highest cost, and naturally the ones with the highest list price. So it's more or less the same group of products that satisfy each*

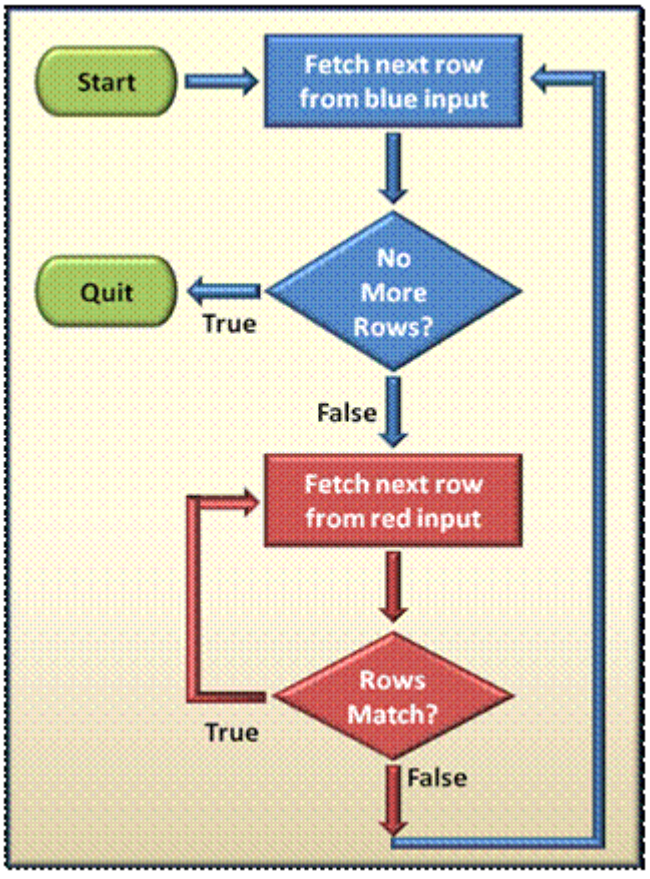
predicate. These and other conditions which make selectivity extremely hard to estimate are much more common in production systems than most people would think.

I want to use this opportunity to congratulate the Microsoft SQL Server optimizer team for producing an unbelievably intelligent optimization engine which is (IMHO) by far the best of its kind on the market, and it just keeps getting better with every new version. It is getting harder and harder for me to come up with such examples where I manage to make it err.

CHAPTER 3:
Merge Operator



A merge operator can be used only when both sets of rows are pre-sorted according to the join expression(s). For example, a *Product* table index and *Order Detail* table index both sorted by *Product ID* (recall Execution Plan 3 from the previous chapter). The algorithm is extremely simple, elegant, and efficient:



Flowchart 2 – Merge

Since the rows are sorted according to the join expression(s), we can immediately begin the matching process. Simply get the first row from the blue input and the first row from the red input. If they match, output them and continue to the next row from the red input. If not, fetch the next row from the blue input and repeat the processes until all rows from the blue input have been processed. If we were to join our cards this way, we would first lay both sets on the table, sorted according to the join condition, let's

say suit and rank. We will probably spread them one row below the other and simply start picking the cards, moving from left to right on both rows and matching as we progress. Of course, we could decide to sort them just for this purpose even if they weren't sorted to begin with.

MERGE IS A HIGHLY EFFICIENT OPERATOR

The merge join is probably the most efficient of all three operators. It combines the advantage of hash match where the actual data needs to be accessed only once with the advantages of nested loops - low CPU consumption and enabling of fast output of matched rows for further query processing. Moreover, it tops them both by eliminating the potential for a-sequential page flushes. Since the days of SQL Server 6.5, I have witnessed how the query optimizer tends to favor merge joins more and more with every new release. In SQL 2000, we would see merge joins almost exclusively for joins that had the appropriately pre-sorted indexes, and for queries that included an ORDER BY clause that required the sort. In SQL 2008 the optimizer cleverly realizes that the advantages of this join operator justify pre-sorting of one or both inputs just for the sake of using merge in many more cases than previous versions did.

Let's check out a few examples that use the Merge join operator.

MERGE JOIN WITH CLUSTERED INDEXES

The most obvious example, as can be seen in ***Query 5*** , is a join that uses the keys of the clustered indexes of both tables. Since a clustered index is actually the table itself, sorted in the order of the clustered index key(s), the clustered index covers all queries and can be used to retrieve any (non BLOB) column in the table you specify in your SELECT clause. Even the SELECT * in the query below will not require any additional table lookups. But remember that there is still a penalty to pay for retrieving all columns unnecessarily as both tables will need to be fully loaded into memory and sent over the network.

```
SELECT      *
FROM        Sales.SalesOrderHeader AS SOH
            INNER JOIN
            Sales.SalesOrderDetail AS SOD
            ON SOH.SalesOrderID = SOD.SalesOrderID
```

Query 5

Execution Plan 5

* **Challenge #2:** There are no expression computations in this query. Try to guess what the 'compute scalar' operators in Execution Plan 5 above are for…

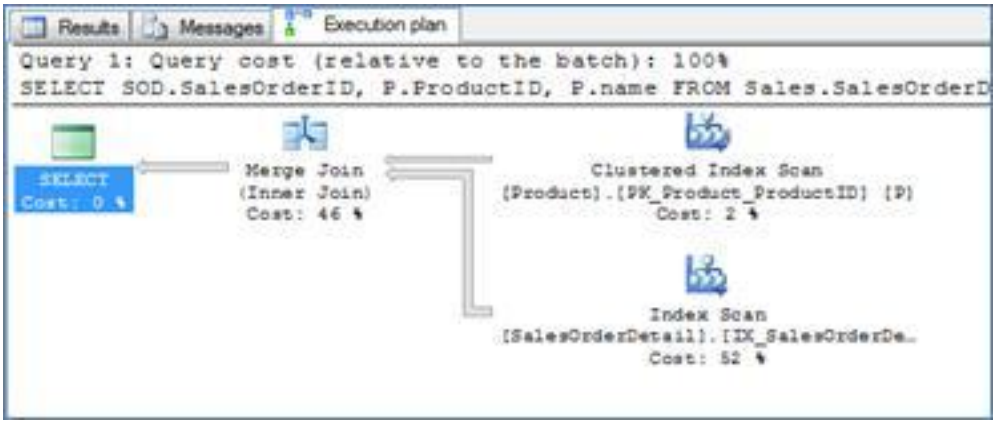
Hint: The answer is at the columns node of these tables in SSMS object explorer.

MERGE JOIN WITH NON CLUSTERED INDEXES

Merge will be very efficient when one or both tables have a non clustered index that sorts the join column. This is twice as true if the index covers the query, as in Query 6.

```
SELECT      SOD.SalesOrderID,
            P.ProductID,
            P.name
FROM        Sales.SalesOrderDetail AS SOD
            INNER JOIN
            Production.Product AS P
            ON SOD.ProductID = P.ProductID
```

Query 6



Execution Plan 6

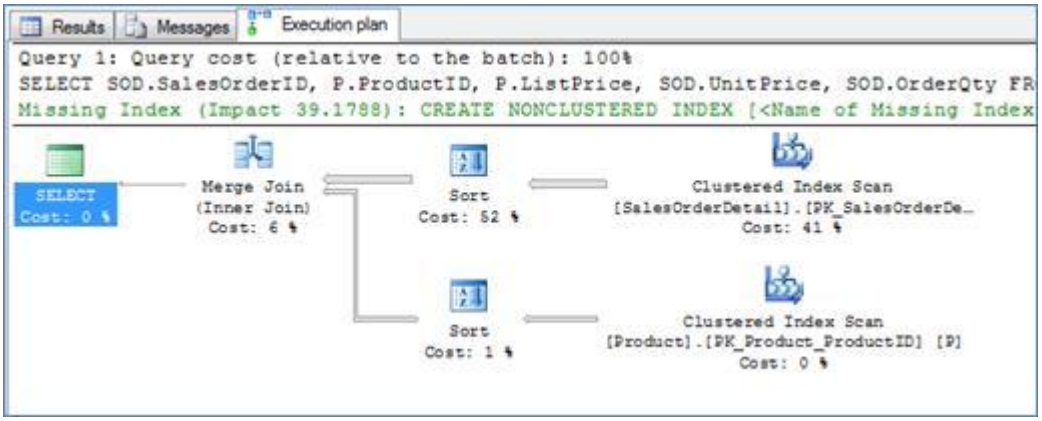
* **Exercise:** In the demo code I've added two more examples (Queries 6b and 6c) where a small change to the query changes the physical operator chosen by the SQL Server optimizer. Try to play around with the parameters and see how the optimizer changes its decisions.

ORDER BY CLAUSE MAY COAX A MERGE JOIN

As I mentioned earlier, in many cases you will see that the optimizer decides to sort one or both inputs just to use the merge operator. This will usually happen when the inputs are not very large and the alternatives are worse. This decision becomes even easier if the optimizer sees that performing a sort provides an additional benefit, allowing the optimizer to '*kill two birds with one stone*' (see disclaimer below). For example, the pre sorting may help facilitate the highly efficient stream aggregate for GROUP BY or DISTINCT clauses, UNION operators, analytical rankings or for delivering the result set in the order of the ORDER BY clause, as is the case in Query 7 .

```
SELECT      SOD.SalesOrderID,
            P.ProductID,
            P.ListPrice,
            SOD.UnitPrice,
            SOD.OrderQty
FROM        Sales.SalesOrderDetail AS SOD
            INNER JOIN
            Production.Product AS P
            ON SOD.UnitPrice = P.Listprice
WHERE       SOD.OrderQty > 5
ORDER BY    P.ListPrice DESC
```

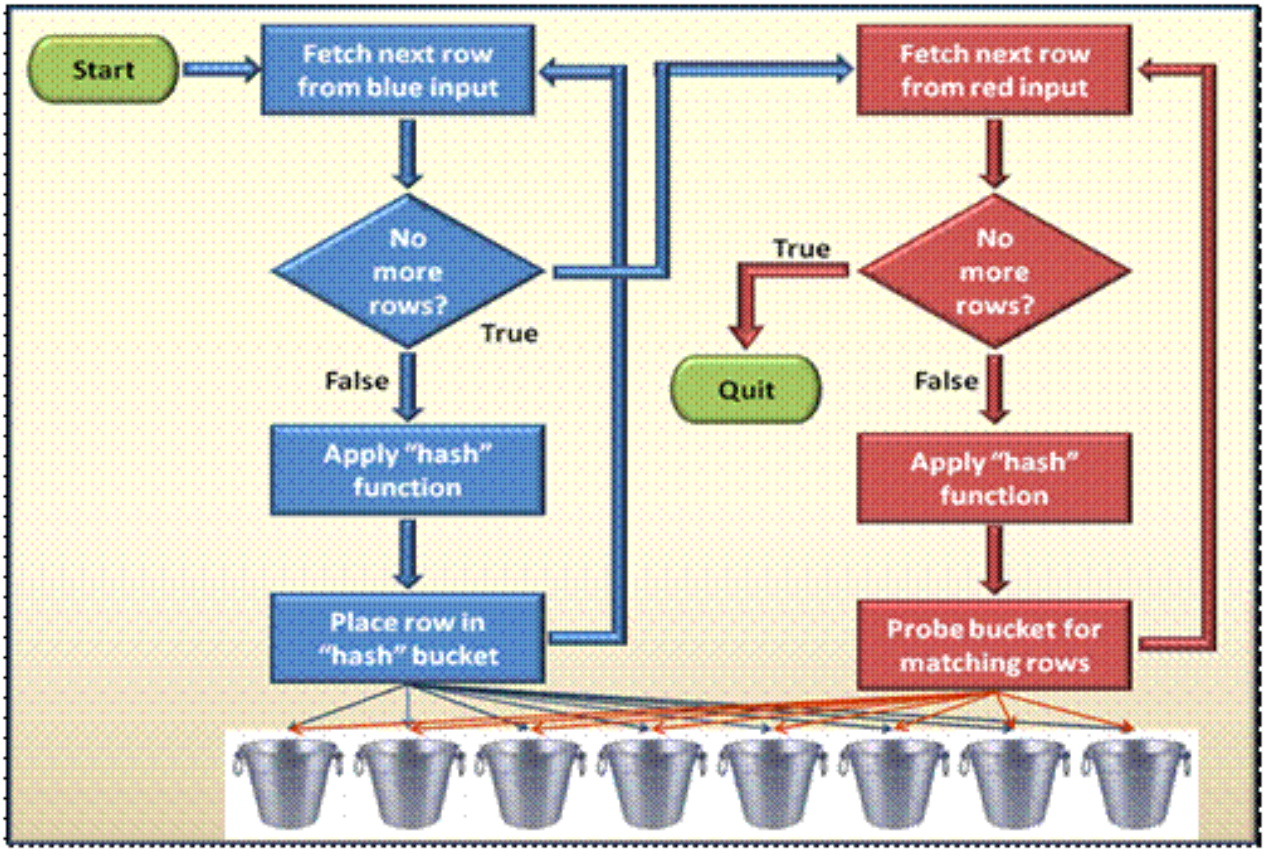
Query 7



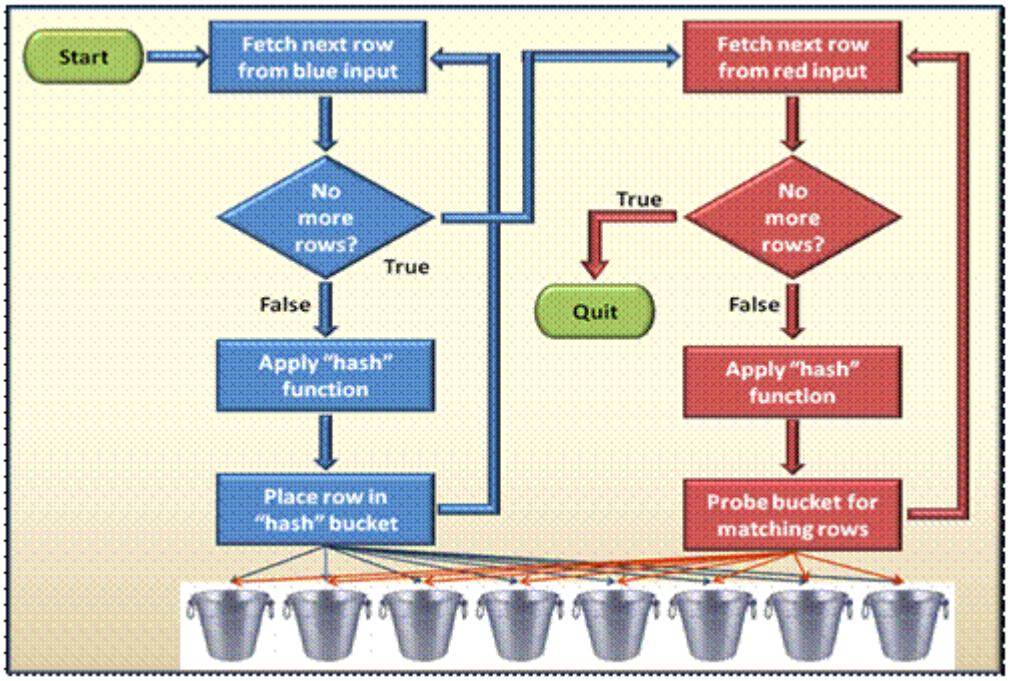
Execution Plan 7

*** DISCLAIMER:** No birds (nor any other animals) were harmed for the making of this script.

CHAPTER 4:
Hash Operator



The algorithm for a hash match join operator is a little more complicated:



Flowchart 3 – Hash Match

Each row in the blue input is fetched and a hash function (explained soon) is applied on the join expression. The row (in full, part or just a pointer) is placed in a 'bucket' which represents the result of the hash function. After all relevant rows have been 'hashed' and placed in their appropriate buckets; the rows from the red input are fetched one by one. For each row, the same hash function is applied to the join expression and matches are looked for (probed) within the appropriate bucket only. If we would use this operator to join our playing cards, we would need

to decide on an appropriate hash function. For example, let's assume our hash function is the card's suit. In this case, we would first separate the blue-back cards into 4 piles (buckets) by suit - spades, clubs, hearts and diamonds. Then, we would pick the red-back cards, one by one, look at their suit and try to find their match within the appropriate pile only.

The tricky part of achieving high efficiency with the hash match operator is choosing the right hash function for a particular data set. This is a highly challenging task which is handled by expert mathematicians and is one of the most secret aspects of the query optimizer. Imagine what would happen, if in the card example above, we would use the same hash function (card suit) for a set of 1 million cards that consists of spades only... On the other hand, imagine what would happen if we used the same function when the same million cards consisted of a million different (hypothetical) suits? Remember that a join may be performed on any comparable data type with highly varying distribution patterns and with highly varying filter patterns... It's an extremely complicated and delicate balance.

The hash match operator has some additional overhead we need to consider as well. Besides the

obvious CPU overhead for applying the potentially complicated hash function on every row in both inputs, memory pressures may have devastating performance results for this operator as well. The hash buckets must be persisted until the whole operation completes and all rows are matched. This requires significant memory resources, in addition to the actual data pages in the buffer cache. In case that memory is needed for other concurrent operations or in case there is simply not enough memory to hold all buckets for large hash joins, the buckets are flushed from cache and physically written to TempDB. Of course, they will need to be physically retrieved into memory when their content needs to be updated or probed which might prove to be quite painful for those people who like their results delivered in less time than if sent by first class mail. The query optimizer is aware of this fact and may consider the amount of free memory when deciding between hash match and alternative operators.

So when is a hash match join a good choice? Well, I would say far less than it's actually being used, and not due to the optimizer fault... The main advantage of hash match over nested loops is that the data is (seemingly) accessed only once. But, remember that the probing of the buckets is in fact repeated access of the data (or part of it) which does not constitute

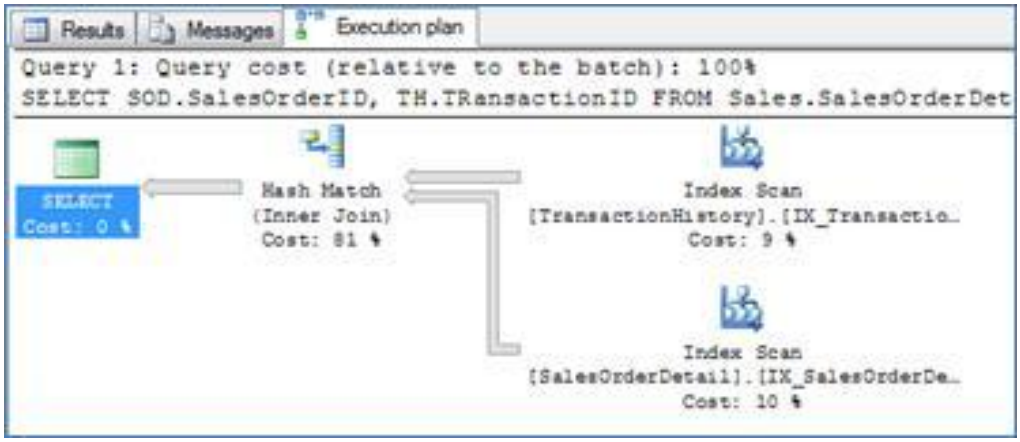
logical reads and therefore is a 'hidden' from our eyes and most monitoring tools. Hash match might be the best choice when both inputs are very large and using nested loops may cause 'a-sequential page flushes'. In most practical cases, the optimizer will revert to hash match when the inputs are not properly indexed (intentionally or not). Optimizing your indexes will, in many cases, cause the optimizer to change its choice of execution plans to use nested loops instead of hash match, significantly reducing CPU and memory consumption, potentially affecting the performance of the whole workload. Let's check out a few examples where the optimizer chooses to use hash match.

HASH MATCH USED WITH VERY LARGE INPUTS

The simplest case is when both inputs are simply too large for nested loops. In Query 8 , both inputs are ~120,000 rows (source not shown, but you can trust me). Although both are 'ideally' indexed for the join column and although the indexes cover the query so no lookups are required, nested loops will simply require too many iterations. Remember that even an efficient index seek might consist of a few page accesses for traversing the non leaf level of the index.

```
SELECT      SOD.SalesOrderID,
            TH.TRansactionID
FROM        Sales.SalesOrderDetail AS SOD
            INNER JOIN
            Production.TransactionHistory AS TH
            ON SOD.SalesOrderID = TH.ReferenceOrderID
```

Query 8



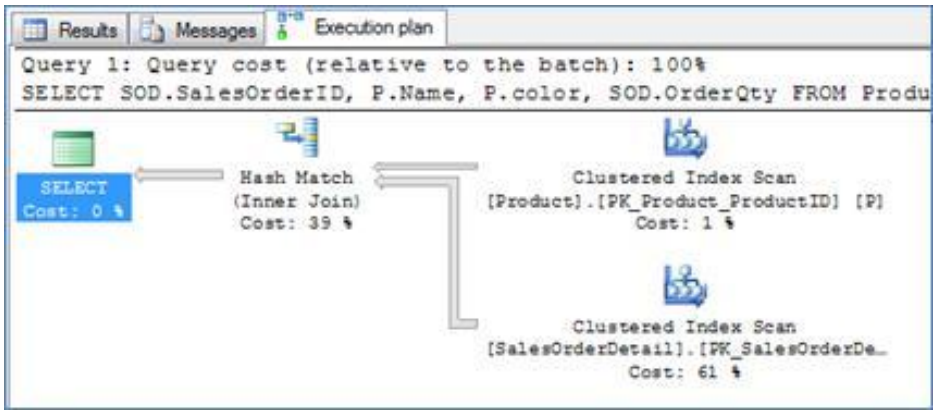
Execution Plan 8

HASH MATCH USED WHEN MULTIPLE LOOKUPS ARE 2nd BEST ALTERNATIVE

In Query 9 below, both inputs are not small enough (P ~500 rows, SOD ~120,000 rows) to make nested loops efficient. It's interesting to see that even though *Product ID* is indexed, nested loops will probably not be a good choice here. The main reason is that the index on *Product ID* does not cover the query and *OrderQty* will need to be looked up for each order which sums up to ~120,000 lookups.

```
SELECT      SOD.SalesOrderID,
            P.Name,
            P.color,
            SOD.OrderQty
FROM        Production.Product AS P
            INNER JOIN
            Sales.SalesOrderDetail AS SOD
            ON P.ProductID = SOD.ProductID
```

Query 9



Execution Plan 9

* **Exercise:** In the demo code, you will find the same query without the OrderQty column in the select list (Query 9b). Try to guess what join operator will the optimizer choose when the need for lookups is eliminated? Try it…

PHYSICAL VS. LOGICAL SCAN ORDER

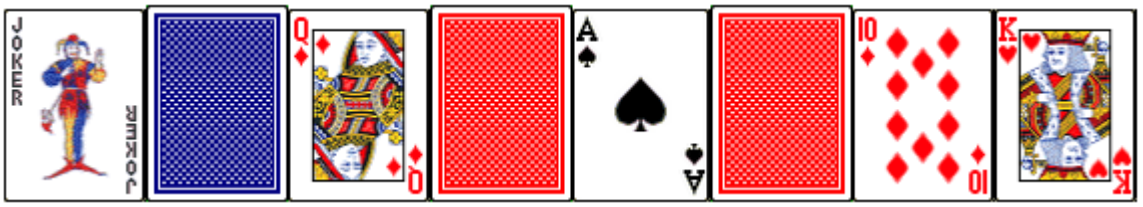
I would also like to draw your attention to another interesting property of the execution plan. If you look at the properties of either index scan above, you will see the value 'false' for the scan property 'ordered'.

This means that the storage engine is not required to follow the logical chain links between the index leaf pages that point to the 'next' and 'previous' pages. Since the order of the retrieval of the rows is of no significance to the hash match operator, the storage engine will optimize the scan performance by scanning the index pages in their physical order (by following the IAM pages). This might prove to be a significant performance gain, especially for indexes with a high level of logical fragmentation. Go back and

look at this property for index scans of the execution plans of the merge examples above.

Another thing we should note about hash joins is the fact that, in contrast to both nested loops and merge operators where the join operator may immediately start outputting joined rows for further processing, no rows can be outputted when using hash match until the whole 'blue input' is fully hashed.

CHAPTER 4:
Summary



In a nutshell, we can sum up the main properties of the three physical operators available to the SQL Server optimizer when carrying out Equi-Inner-Joins.

	Nested Loops	Merge	Hash Match
A Good choice when	Small outer input Inner input well indexed	Pre-sorted inputs Sorting required anyway	Large inputs Inputs not well indexed
CPU consumption	Low	Low *Unless requires sorting	High
Memory usage	Low	Low *Unless requires sorting	High
Logical reads	High	Low	Low *Hidden cost of probes
Output matches	Fast	Fast	Slow

Table 1 - Summary

I hope this short discussion of the seemingly simple SQL construct raised your curiosity. There are many more aspects and considerations to joins which were not even mentioned in this article. This is a fascinating and highly complex subject.

If you are interested to dive deeper and learn more on joins and their implementation in SQL Server, I highly recommend Craig Freedman's SQL Server Blog on MSDN. Craig has published a series of excellent, in depth articles regarding many more aspects and types of joins.