

SQL Server 2012 Internal

[!NOTE]

Highlights information that users should take into account, even when skimming.

[!TIP] Optional information to help a user be more successful.

[!IMPORTANT] Crucial information necessary for users to succeed.

[!WARNING]

Critical content demanding immediate user attention due to potential risks.

[!CAUTION] Negative potential consequences of an action.

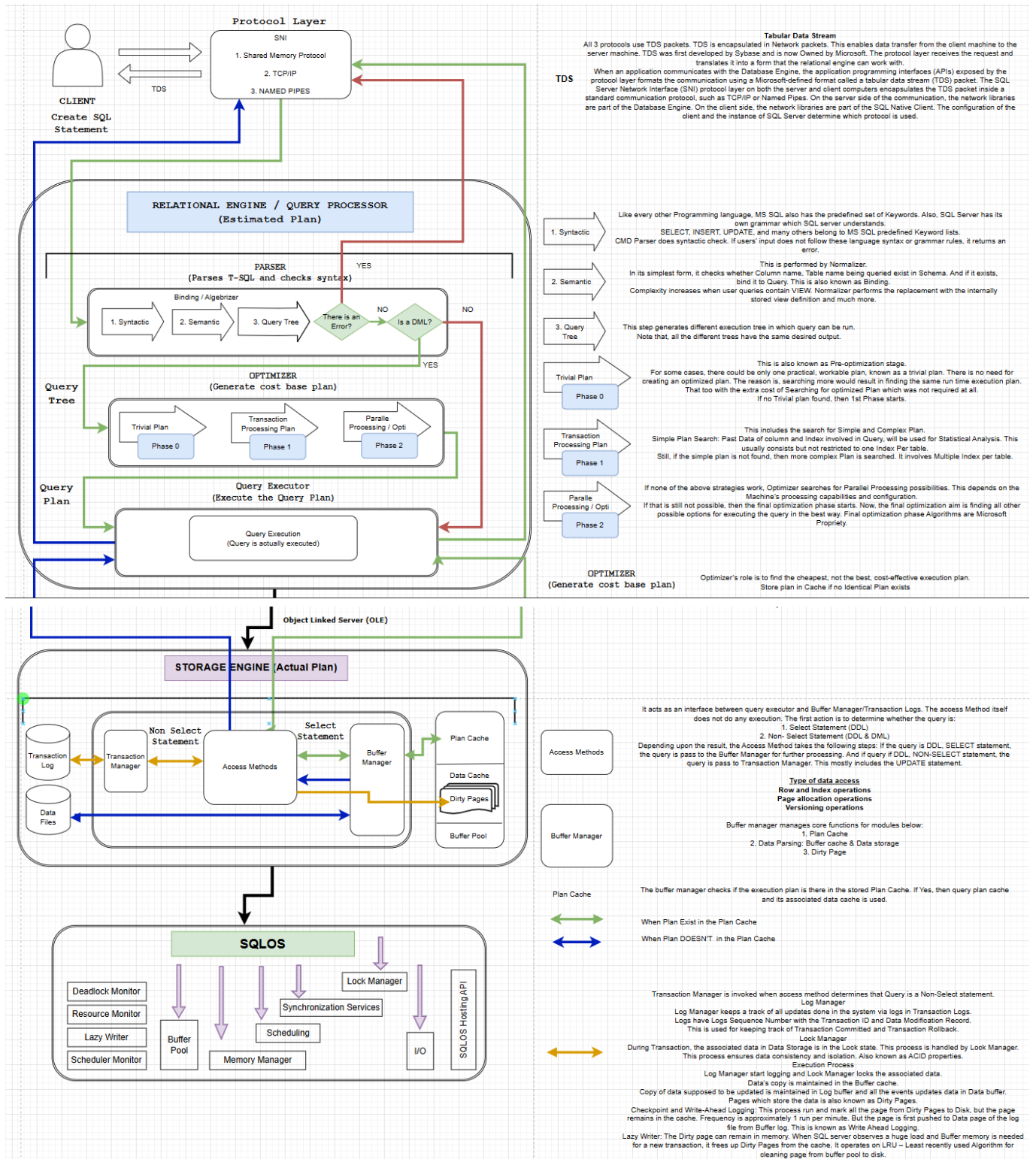
Index

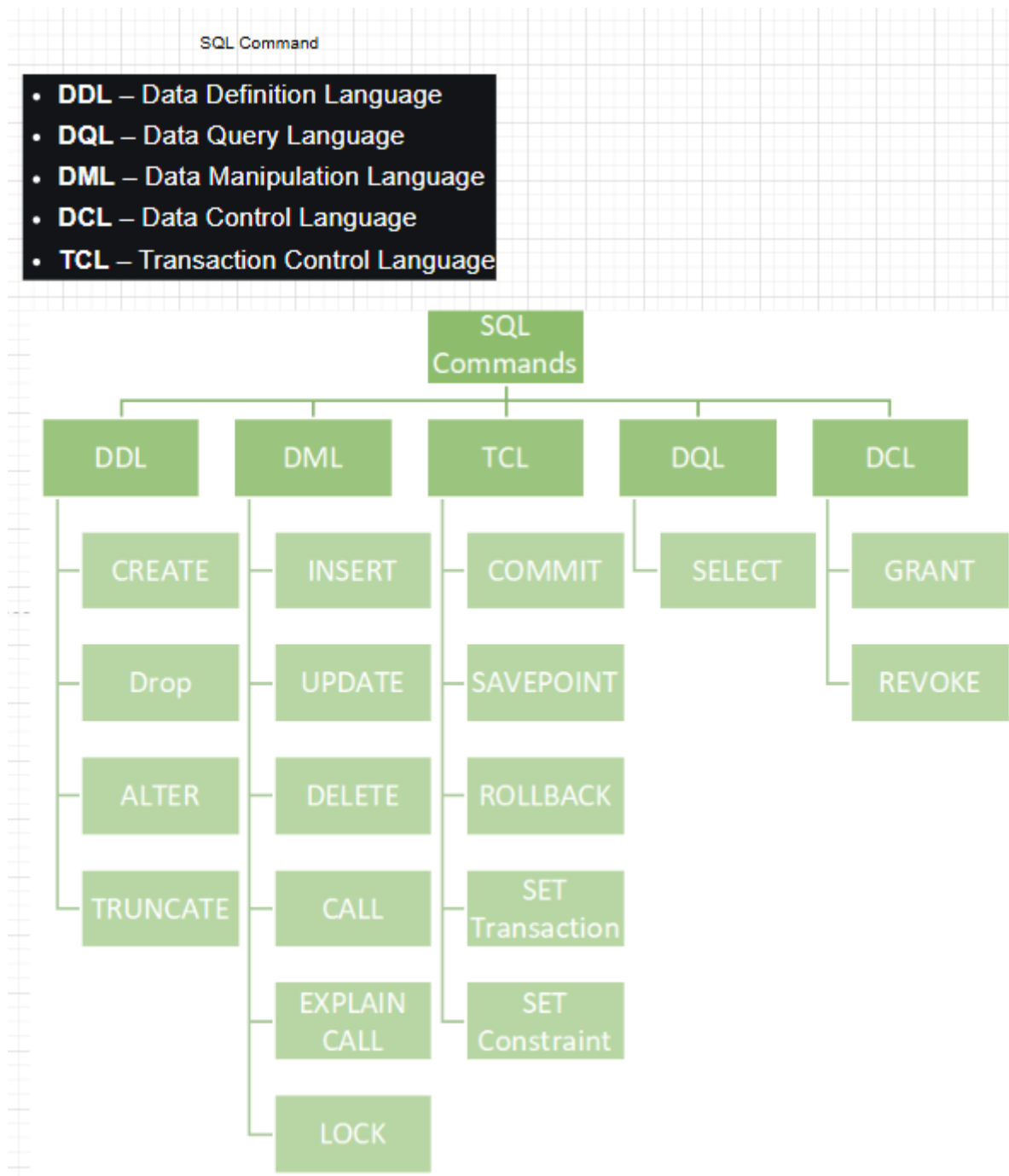
- [SQL Server 2012 architecture and configuration](#)
 - [1. Architecture](#)
 - [1.1. MSSQL Server Physical CLIENT-SERVER Architecture Diagram](#)
 - [1.2. Components of the SQL Server Engine](#)
 - [1.3. The Protocol Layer](#)
 - [1.4. The Query Processor](#)
 - [1.4.1. PARSER](#)
 - [1.4.2. OPTIMIZER](#)
 - [1.4.3. EXECUTOR](#)
 - [1.5. The Storage Engine](#)
 - [1.5.1. Access Method](#)
 - [1.5.2. Buffer Manager](#)
 - [1.5.3. Transaction Manager](#)
 - [1.6. Plan Caching and Recompilation](#)
 - [1.7. Transactions and Concurrency](#)

SQL Server 2012 architecture and configuration

1. Architecture

1.1. MSSQL Server Physical CLIENT-SERVER Architecture Diagram





1.2. Components of the SQL Server Engine

The four major components of SQL Server engine are: **the protocol layer**, **the query processor**, **the storage engine**, and **the SQLOS**. Every batch submitted to SQL Server for execution, from any client application, must interact with these four components.

The protocol layer receives the request and translates it into a form that the relational engine can work with. It also takes the final results of any queries. **The query processor** accepts T-SQL batches and determines what to do with them. For T-SQL queries and programming constructs, it parses, compiles, and optimizes the request and oversees the process of executing the batch. As the batch is executed, if data is needed, a request for that data is passed to the storage engine. **The storage engine** manages all data access, both through transaction-based commands and bulk operations such as backup, bulk insert, and certain Database Console Commands (DBCCs). **The SQLOS** layer handles activities normally considered to be operating system responsibilities, such as thread management (scheduling), synchronization primitives, deadlock detection, and memory management, including the buffer pool.

1.3. The Protocol Layer

Receives the request and translates it into a form that the relational engine can work with. It also takes the final results of any queries.

1.4. The Query Processor

It includes the SQL Server components that determine exactly what your query needs to do and the best way to do it. This has two primary components: Query Optimization and query execution. This layer also includes components for parsing and binding. By far the most complex component of the query processor—and maybe even of the entire SQL Server product—is the Query Optimizer, which determines the best execution plan for the queries in the batch.

1.4.1. PARSER

To be Complete

1.4.2. OPTIMIZER

- Intro

The Query Optimizer takes the query tree and prepares it for optimization. Statements that can't be optimized, such as flow-of-control and **Data Definition Language (DDL)** commands, are compiled into an internal form. Optimizable (not DDL) statements are marked as such and then passed to the Query Optimizer.

The Query Optimizer is concerned mainly with the **Data Manipulation Language (DML)** statements SELECT, INSERT, UPDATE, DELETE, and MERGE, which can be processed in more than one way; the Query Optimizer determines which of the many possible ways is best. It compiles an entire command batch and optimizes queries that are optimizable. The query optimization and compilation result in an execution plan.

The first step in producing such a plan is to normalize each query, which potentially breaks down a single query into multiple, fine-grained queries. After the Query Optimizer normalizes a query, it optimizes it, which means that it determines a plan for executing that query. **Query optimization is cost-based**, the Query Optimizer chooses the plan that it determines would cost the least based on internal metrics that include estimated memory requirements, CPU utilization, and number of required I/Os.

The Query Optimizer considers the type of statement requested, checks the amount of data in the various tables affected, looks at the indexes available for each table, and then looks at a sampling of the data values kept for each index or column referenced in the query. The sampling of the data values is called distribution statistics. Based on the available information, the Query Optimizer considers the various access methods and processing strategies that it could use to resolve a query and chooses the most cost-effective plan.

After normalization and optimization are completed, the normalized tree produced by those processes is compiled into the execution plan, which is actually a data structure.

- Overview

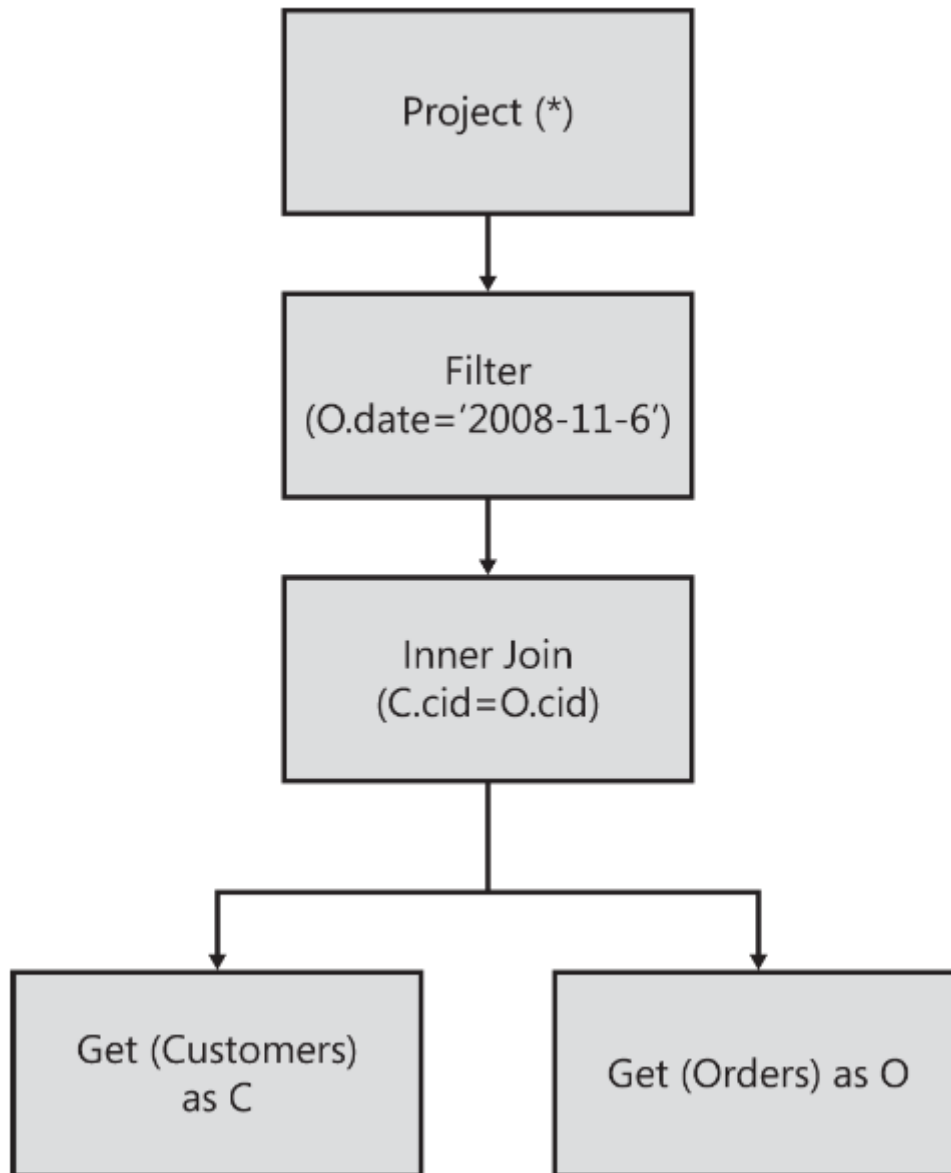
When a query is compiled, the SQL Statement is first parsed into an equivalent tree representation. For queries with valid SQL syntax, the next stage performs a series of validation steps on the query, generally called binding, where the columns and tables in the tree are compared to database metadata to ensure that those columns and tables exist and are visible to the current user. This stage also performs semantic checks on the query to ensure that it's valid, such as making sure that the columns bound to a GROUP BY operation are valid. After the query tree is bound, the QO takes the query and starts evaluating different possible query plans. The QO performs this search, selects the query plan to be executed, and then returns it to the system to execute. The execution component runs the query plan and returns the query results.

- Understanding the tree format

When you submit a SQL query to the QP, the SQL is parsed into a tree representation. Each node in the tree represents a query operation to be performed. e.g. the query

```
SELECT * FROM Customers C INNER JOIN Orders O ON C.cid = O.cid WHERE O.date  
= '2008-11-06'
```

might be represented internally as:



- Understanding optimization

Another major job of the QO is to find an efficient query plan. At first you might think that every SQL query would have an obvious best plan. Unfortunately, finding an optimal query plan is actually a much more difficult algorithmic problem for SQL Server. As the number of tables increases, the set of alternatives to consider quickly grows to be larger than what any computer can count. The storage of all possible query plans also becomes a problem. The Query Optimizer solves this problem using heuristics and statistics to guide those heuristics.

[!WARNING] What heuristics means on this contexts?

- Search space and heuristics

The QO uses a framework to search and compare many different possible plan alternative efficiently.

- Rules

The QO is a search framework. The QO considers transformation of a specific query tree from the current state to a different. In the framework used in SQL Server, the transformations are done via RULES, which are very similar to the mathematical theorems. Rules are matches to tree

patterns and are applied if they are suitable to generate new alternatives. The QO has different kinds of RULES:

1. SUBSTITUTION RULE. Rules that heuristically rewrite a query tree into a new shape.
2. EXPLORATION RULES. These rules generate new tree shapes but can't be directly executed.
3. IMPLEMENTATION RULES. Rules that convert logical trees into physical trees to be executed.

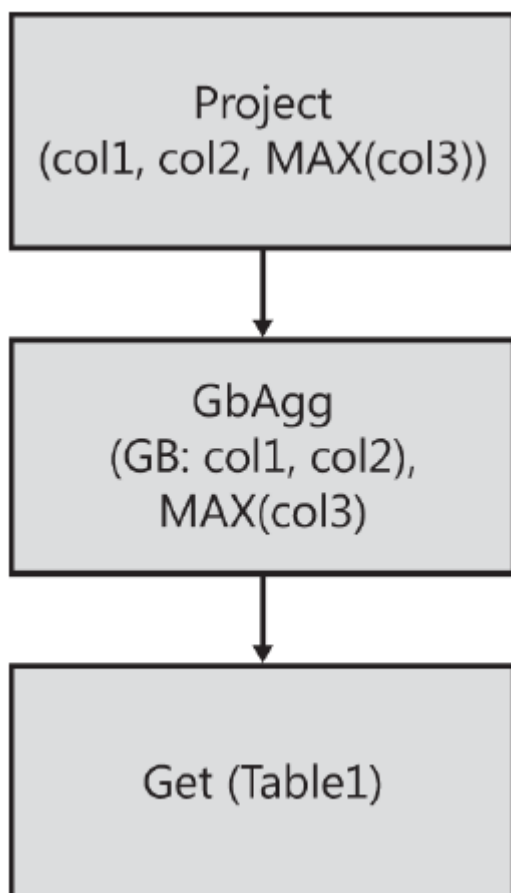
The best of these generated physical alternatives from implementation rules is eventually output by the QO as the final query execution plan.

- Properties

The search framework collects information about the query tree in a format that can make it easier for rules to work. e.g. one property used in SQL Server is the set of columns that make up a UQ on the data. Consider the following query:

```
SELECT col1, col2, MAX(col3) FROM Table1 GROUP BY col1, col2;
```

This query is represented internally as a tree, as shown below:



If the columns (col1, col2) make up a unique key on table group by, doing grouping isn't necessary at all because each group has exactly one row. So, writing a rule that removes the group by from the query tree completely is possible. Figure below shows this rule in action

```
CREATE TABLE groupby (col1 int, col2 int, col3 int);
ALTER TABLE groupby ADD CONSTRAINT unique1 UNIQUE(col1, col2);
SELECT col1, col2, MAX(col3) FROM groupby GROUP BY col1, col2;
```

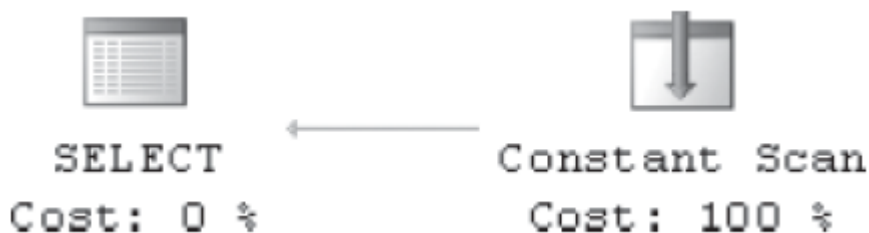


By looking at the final query plan you can see that the QO performs no grouping operation, even though the query uses a GROUP BY. The properties collected during optimization enable this rule to perform a trees transformation to make the resulting query plan complete more quickly.

One useful app of this scalar property is in CONTRADICTION DETECTION. The QO can determine whether the query is written in such a way as to never return any rows at all. When the QO detects a contradiction, it requires the query to remove the portion of the query containing the contradiction. Figure below shows an example of a contradiction detected during optimization.

```
CREATE TABLE dbo.DomainTable ([col1] INT);
GO

SELECT *
FROM   dbo.DomainTable D1
JOIN   dbo.DomainTable D2
ON     D1.col1 = D2.col1
WHERE  D1.col1 > 5
AND    D2.col1 < 0;
```



The final QP doesn't even reference the table at all; it's replaced with a special Constant Scan operator that doesn't access the storage engine and, in this case, returns zero rows. This means that the query runs faster, consumes less memory, and doesn't need to acquire locks against the resources referenced in the section containing the contradiction when being executed.

Like with rules, both logical and physical properties are available.

- Logical properties cover things like the output column set, key columns, and whether or not a column can output any nulls.

- Physical properties are specific to a single plan, and each plan operator has a set of physical properties associated with it.
- The Memo (Storage of alternatives)

Earlier, this chapter mentioned that the storage of all the alternatives considered during optimization could be large for some queries. The QO contains a mechanism to avoid storing duplicate information. The structure is called the Memo, and one of its purposes is to find previously explored subtrees and avoid reoptimizing those areas of the plan it exists for the life of one optimization.

The Memo works by storing equivalent trees in groups. This model is used to avoid storing trees more than once during query optimization and enables the QO to avoid searching the same possible plan alternatives more than once.

The Memo stores all considered plans.

If the QO is about to run out of memory while searching the set of plans, it contains logic to pick a **good enough** query plan rather than run out of memory. After the QO finishes searching for a plan it goes through the Memo to select the best alternative from each group that satisfies the query's requirements. These operators are assembled into the final query plan, but it's very close to the showplan output generated for the query plan.

- Operators

SQL Server has around 40 operators and even more physical operators.

Traditionally operators in SQL Server follow the model shown below:

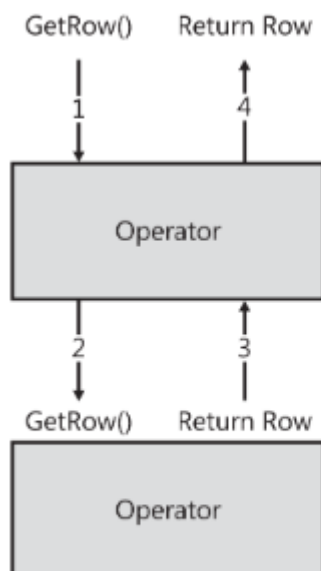


FIGURE 11-6 SQL Server operator data flow model.

This **row-based** model works by requesting rows from one or more children and then producing rows to return to the caller. The caller can be another operator or can be sent to the user if it's the uppermost operator in the query tree. Each operator **returns one row at a time**, meaning that the caller must call for each row.

I will cover a few of the more rare and exotic operators here and will reference them later.

- **Compute Scalar:** Project Is a simple operator that attempts to declare a set of columns, compute some value, or perhaps restrict columns from other operators in the query tree. These operators correspond to the SELECT list in the SQL Language.
- **Compute Sequence:** Sequence Project Is somewhat similar to a compute Scalar in that it computes a new value to be added into the output stream. The key difference is that this works on an ordered stream and contains state that is preserved from row to row.
- **semi-join:** It describes an operator that performs a join but returns only values from one of its inputs. The QP uses this internal mechanism to handle most subqueries. Contrary to popular belief, a subquery isn't always executed and cached in a temporary table, its treated much like a regular join. e.g. Suppose that you need to ask a sales tracking system for a store to show you all customers who have placed an order in the last 30 days so that you can send them a thank you email.

```
CREATE TABLE Customers (
    custid INT IDENTITY
    , name NVARCHAR(100)
);

CREATE TABLE Orders (
    orderid INT IDENTITY
    , custid INT
    , orderdate DATE
    , amount MONEY
);

INSERT INTO Customers(name) VALUES ('Conor Cunningham');
INSERT INTO Customers(name) VALUES ('Paul Randal');

INSERT INTO Orders(custid, orderdate, amount) VALUES (1, '2025-07-12', 49.23);
INSERT INTO Orders(custid, orderdate, amount) VALUES (1, '2025-07-04', 65.00);
INSERT INTO Orders(custid, orderdate, amount) VALUES (2, '2025-07-12', 123.44);

truncate table Orders
drop table Orders

-- Let's find out customers who have ordered something in the last month
-- Semantically wrong way to ask the question - returns duplicate names
SELECT
    name
FROM Customers C
JOIN Orders O
ON C.custid = O.custid
```

```

WHERE DATEDIFF(M, O.orderdate, '2025-06-30') < 1

-- and then people try to "fix" by adding a distinct
SELECT
    DISTINCT name
FROM Customers C
JOIN Orders O
ON C.custid = O.custid
WHERE DATEDIFF("m", O.orderdate, '2025-06-30') < 1;

-- this happens to work, but it is fragile, hard to modify, and it
is usually not done properly.
-- the subquery way to write the query returns one row for each
matching Customer
SELECT
    name
FROM Customers C
WHERE EXISTS ( SELECT 1
                FROM Orders O
                WHERE C.custid = O.custid
                AND DATEDIFF("m", O.orderdate, '2008-08-30') < 1 );
-- note that the subquery plan has a cheaper estimated cost result
-- and should be faster to run on larger systems

--SELECT DATEDIFF("m", '2025-07-12', '2008-08-30'), DATEDIFF("m",
'2025-07-04', '2008-08-30'), DATEDIFF("m", '2025-07-12', '2008-08-
30');

SELECT * FROM Customers
SELECT * FROM Orders

-- exercise
SET STATISTICS IO, TIME ON
-- do the same with CTE
WITH cteRecentOrdes AS (
    SELECT
        custid
    FROM Orders
    WHERE DATEDIFF("m", orderdate, '2008-08-30') < 1
    /* GROUP BY custid */
)
SELECT name
FROM Customers C
WHERE EXISTS ( SELECT 1
                FROM cteRecentOrdes R
                WHERE C.custid = R.custid );

-- CUAL ES LA MAS PERFORMANTE???
SELECT
    name
FROM Customers C
WHERE EXISTS ( SELECT 1
                FROM Orders O

```

```
WHERE C.custid = O.custid
AND DATEDIFF("m", O.orderdate, '2008-08-30') < 1 );
```

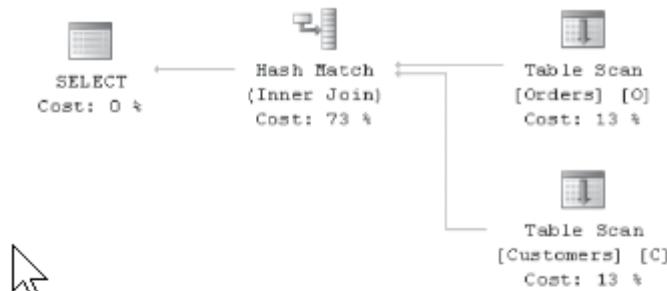


FIGURE 11-7 Query plan using an *INNER JOIN* instead of a subquery (the section commented with "Semantically wrong way to ask the question - returns duplicate names").



FIGURE 11-8 Query plan using *DISTINCT* and *INNER JOIN* instead of a subquery (the section commented with "try to 'fix' by adding a distinct").

Microsoft SQL Server 2012 Internals

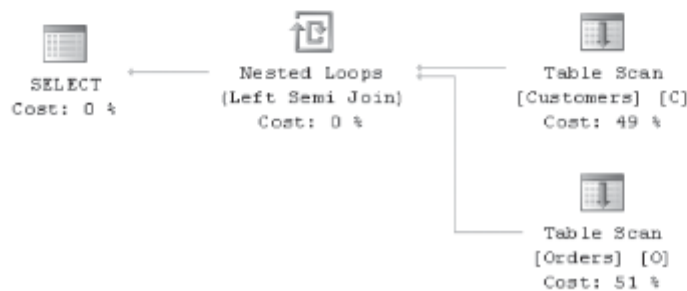


FIGURE 11-9 Query plan using subquery.

o Apply

Since SQL server 2005, *CROSS APPLY* and *OUTER APPLY* represent a special kind of subquery. The most common application for this feature is to do an index lookup join.

```
CREATE TABLE idx1 (col1 INT PRIMARY KEY, col2 INT);
CREATE TABLE idx2 (col1 INT PRIMARY KEY, col2 INT);
GO

SELECT * FROM idx1
CROSS APPLY ( SELECT * FROM idx2 WHERE idx1.col1 = idx2.col1 ) AS a;
```

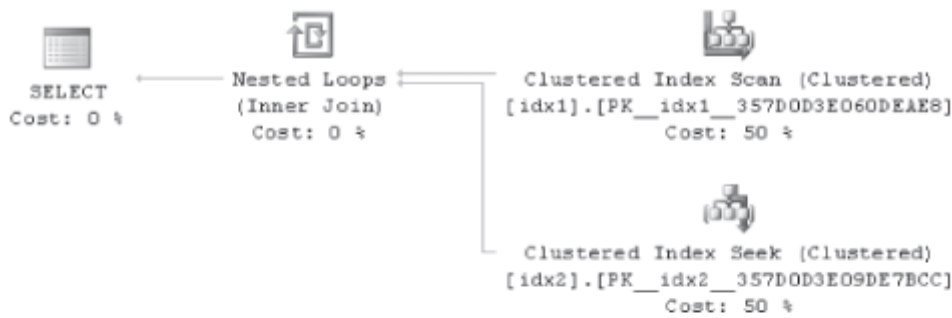
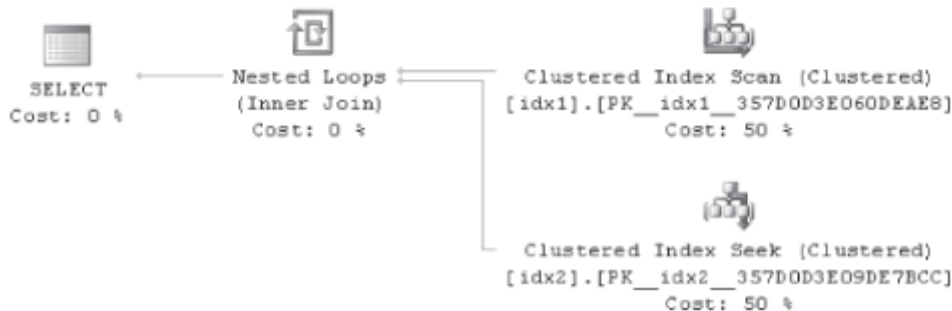


FIGURE 11-10 *APPLY* query plan.

This query is logically equivalent to an *INNER JOIN*, and Figure 11-11's query plan is identical in SQL Server 2012.

```
SELECT * FROM idx1 INNER JOIN idx2
ON idx1.col1=idx2.col1;
```



In both cases, a value from the outer table is referenced as an argument to the seek on the inner table. The apply operator is almost like a function call in a procedural language. For Each row from the outer (left) side, some logic on the inner (right) side is evaluated and zero or more rows are returned for that invocation of the right subtree.

- Spools

SQL Server has a number of different, specialized spools, each one highly tuned for some scenario. Conceptually, they all do the same thing—they read all the rows from the input, store them in memory or spill it to disk, and then allow operators to read the rows from this cache.

Spools exist to make a copy of the rows.

The most exotic spool operation is called a **common subexpression**. This spool can be written once and then read by multiple, different children in the query. It's currently the only operator that can have multiple parents in the final query plan. **Common subexpression spools** have only one client at a time. So, the first instance populates the spool, and each later reference reads from this spool in sequence. **Common subexpression spools** are used most frequently in wide update plans, they are also used in windowed aggregate functions.

```
CREATE TABLE window1 (col1 INT, col2 INT);
GO
```

```

DECLARE @i INT = 0;

WHILE @i < 100
BEGIN
    INSERT INTO window1 (col1, col2)
    VALUES (@i/10, RAND() * 1000);
    SET @i += 1;
END;

SELECT
    col1, SUM(col2)
    OVER(PARTITION BY col1)
FROM window1

```

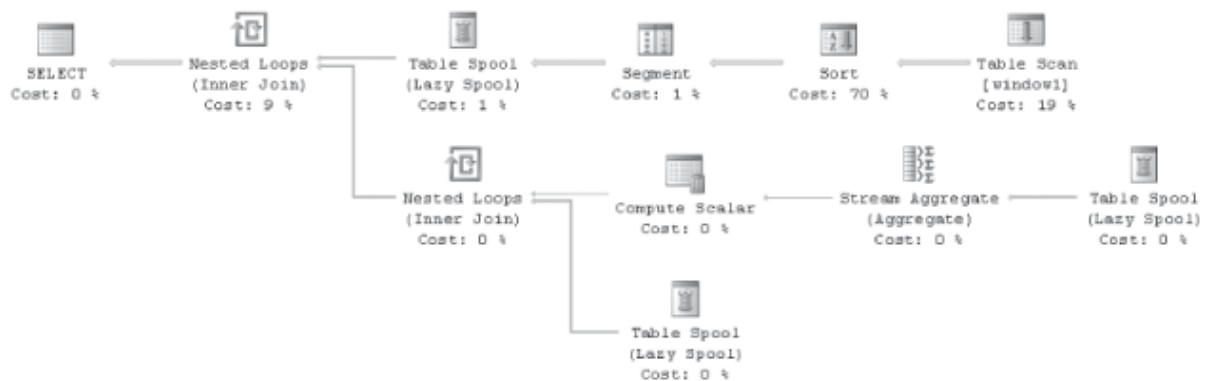


FIGURE 11-12 Query plan containing common subexpression spool.

- Exchange

The Exchange operator is used to represent parallelism in query plans. This can be seen in the show-plan as a Gather Streams, Repartition Streams, or Distribute Streams operation, based on whether it's collecting rows from threads or distributing rows to threads, respectively.

- Optimizer architecture

The QO contains many optimization phases that each perform different functions. The major phases in the optimization of a query, as show below are as follows:

- Simplification
- Trivial plan
- Auto-stats create/update
- Exploration/Implementation(phases)
- Convert to executable plan

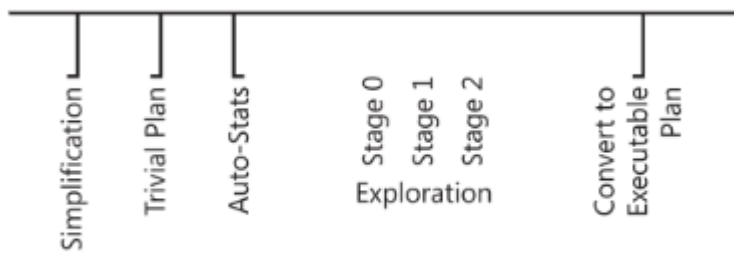


FIGURE 11-14 Query Optimizer pipeline.

- Before optimization The SQL Server QP performs several steps before actual optimization process begins. View expansion is one major preoptimization activity. Coalescing adjacent UNION operations is another preoptimization transformation that is performed to simplify the tree.

- Simplification

Early in optimization, the tree is normalized in the Simplification phase to convert the tree from a form linked closely to the user syntax into one that helps later processing. The Simplification phase also performs a number of other tree rewrites, including the following:

- Grouping joins together and picking an initial join order, based on cardinality data for each table.
- Finding contradictions in queries that can allow portions of a query not to be executed
- Performing the necessary work to rewrite SELECT lists to match computed columns

- Trivial plan/auto-parameterization

The main optimization path in SQL Server is a very powerful cost-based model of a query's execution time. To be able to satisfy small query applications well, SQL Server use a fast path to identify queries where cost based optimization isn't needed. This means that only one plan is available to execute or an obvious best plan can be identified. In these cases, The QO directly generates the best plan and returns it to the system to be executed.

The SQL Server QP actually takes this concept one step further. When simple queries are compiled and optimized, the QP attempts to rewrite them into an equivalent parameterized query instead. If the plan is determined to be **trivial**, the parameterized query is turned into an executable plan. Then, future queries that have the same shape except for constants in well-known locations in the query text just run the existing compiled query and avoid going through the QO at all.

```

SELECT [text]
FROM    sys.dm_exec_query_stats AS qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS st
WHERE st.text LIKE '%Table1%';

-----
(@1 tinyint)SELECT [col1] FROM [Table1] WHERE [col2]=@1
  
```

The other choice is **full**, meaning that cost-based optimization was performed.

- Limitations

Using more complex features can disqualify a query from being considered trivial because those features always have a cost-based plan choice or are too difficult to identify as trivial. Examples of query features that cause a query not to be considered trivial include Distributed Query, Bulk Insert, XPath queries, queries with joins or subqueries, queries with hints, some cursor queries, and queries over tables containing filtered indexes.

SQL Server 2005 added another feature, **forced parameterization**, to auto-parameterize queries more aggressively. This feature parameterizes all constants, ignoring cost-based considerations. The benefit of this feature is that it can reduce compilations, compilation time, and the number of plans in the procedure cache. All these things can improve system performance. On the other hand, this feature can reduce performance when different parameter values would cause different plans to be selected. These values are used in the Query Optimizer's cardinality and property framework to decide how many rows to return from each possible plan choice, and forced parameterization blocks these optimizations.

- The Memo: exploring multiple plans efficiently

The core structure of the QO is the Memo. This structure helps store the result of all the rules run in the Query Optimizer, and it also helps guide the search of possible plans to find a good plan quickly and to avoid searching a subtree more than once. the Memo consist of a series of groups. Rules are the mechanism that allow the memo to explore new alternatives during the optimization process.

An optimization search pass is split into two parts. In the first part of the search, exploration rules match logical trees and generate new, equivalent alternative logical trees that are inserted into the Memo. Implementation rules run next, generating physical trees from the logical trees. After a physical tree is generated, it's evaluated by the costing component to determine the cost for this query tree. The resulting cost is stored in the Memo for that alternative. When all physical alternatives and their costs are generated for all groups in the Memo, the Query Optimizer finds the one query tree in the Memo that has the lowest cost and then copies that into a standalone tree. The selected physical tree is very close to the showplan form of the tree.

The optimization process is optimized further by using multiple search passes. The QO can quit optimization at the end of a phase if a sufficiently good plan has been found. This calculation is done by comparing the estimated cost of the best plan found so far against the actual time spent optimizing so far. If the current best plan is still very expensive, another phase is run to try to find a better plan. This model allows the QO to generate plans efficiently for a wide range of workloads. By the end of the search, the QO has selected a single plan to be returned to the system. This plan is copied from the Memo into a separate tree format that can be stored in the Procedure Cache. During this process, a few small, physical rewrites are performed. Finally, the plan is copied into a new piece of contiguous memory and is stored in the procedure cache

- Statistics, cardinality estimation, and costing
- Statistics

The QO uses a model with estimated costs of each operator to determine which plan to choose. The costs are based on statistical information used to estimate the number of rows processed in each

operator. By default, statistics are generated automatically during the optimization process to help generate these cardinality estimates. The QO also determines which columns need statistics on each table. When a set of columns is identified as needing statistics, the QO tries to find a preexisting statistics object for that column. If it doesn't find one, the system samples the table data to create a new statistics object. If one already exists, it's examined to determine whether the sample was recent enough to be useful for the compilation of the current query. If it's considered out of date, a new sample is used to rebuild the statistics object. This process continues for each column where statistics are needed. Both auto-create and auto-update statistics are enabled by default.

Although these settings are left enabled, some reasons for disabling the creation or update behavior of statistics, include the following:

- The table is very large, and the time to update the statistics automatically is too high.
- The tables has many unique values, and the sample rate used to generate statistics isn't high enough to capture all the statistical information needed to generate a good query plan.
- The DB app has a short query timeout defined and doesn't want automatic statistics to cause a query to require noticeably more time than average to compile because it could cause that timeout to abort the query.

SQL Server 2005 introduced a feature called asynchronous statistics update, or `ALTER DATABASE...SET AUTO_UPDATE_STATISTICS_ASYNC {ON | OFF}`. This allows the statistics update operation to be performed on a background thread in a different transaction context.

- Statistics design

Statistics are stored in the system metadata and are composed primarily of **a histogram** (a representation of the data distribution for a column) Statistics can be created over most, but not all. As a general rule, data types that support comparisons (such as `>`, `=` and so on) support the creation of statistics. Also SQL Server supports statistics on computed column. The following code creates statistics on a persisted computed column created on a function of an otherwise noncomparable UDT.

```
CREATE TABLE Geog (col1 INT IDENTITY, col2 GEOGRAPHY);
INSERT INTO Geog (col2) VALUES (NULL);
INSERT INTO Geog (col2) VALUES (GEOGRAPHY::Parse('LINESTRING(0 0, 0 10, 10
10, 10 0, 0 0)'));

ALTER TABLE Geog
ADD col3 AS col2.STStartPoint().ToString() PERSISTED;

CREATE STATISTICS s2 ON Geog(col3);
DBCC SHOW_STATISTICS('Geog', 's2');
```

For small tables, all pages are sampled. For large tables a smaller percentage of pages are sampled. So that the histogram remains a reasonable size, it's limited to 200 total steps.

- Density/Frequency information

In addition to a histogram, the QO keeps track of the number of unique values for a set in columns. This information, called the **DENSITY INFORMATION**, is stored in the statistics objects. Density is

calculated by the formula **1/frequency**, with frequency being the average number of duplicates for each value in a table. For multicolumn statistics, the statistics object stores density information for each combination of columns in the statistic object.

```
CREATE TABLE MULTIDENSITY (col1 INT, col2 INT);
go

DECLARE @i INT;
SET @i=0;
WHILE @i < 10000
BEGIN
    INSERT INTO MULTIDENSITY(col1, col2)
    VALUES (@i, @i+1);

    INSERT INTO MULTIDENSITY(col1, col2)
    VALUES (@i, @i+2);

    INSERT INTO MULTIDENSITY(col1, col2)
    VALUES (@i, @i+3);

    set @i+=1;
END;
GO

-- create multi-column density information
CREATE STATISTICS s1 ON MULTIDENSITY(col1, col2);
GO
```

In col1 are 10,000 unique values, each duplicated three times. In col2 are actually 10,002 unique values. For the multicolumn density, each set of (col1, col2) in the table is unique.

```
DBCC SHOW_STATISTICS ('MULTIDENSITY', 's1')
```

The density information for col1 is 0.0001 1/0.0001 = 10,000 wich is the number of unique values of col1. The density information for (col1/2) is about 0.00003 (the number are stored as floating points and are imprecise).

```
SET STATISTICS PROFILE ON
SELECT COUNT(*) AS CNT FROM MULTIDENSITY GROUP BY col1
```

	Rows	Executes	StmtText	EstimateRows
1	10000	1	SELECT COUNT(*) AS CNT FROM MULTIDENSITY GROUP ...	10000
2	0	0	-Compute Scalar[DEFINE:([Expr1004]=CONVERT_IMPLICIT(l...	10000
3	10000	1	-Hash Match(Aggregate, HASH:([s1].[dbo].[MULTIDENSI...	10000
4	30000	1	-Table Scan[OBJECT:([s1].[dbo].[MULTIDENSITY]]]	30000

```
SET STATISTICS PROFILE ON
SELECT COUNT(*) AS CNT FROM MULTIDENSITY GROUP BY col1, col2
```

Rows	Executes	StmtText	EstimateRows
30000	1	SELECT COUNT(*)AS CNT FROM MULTIDENSITY GROUP ...	30000
0	0	[-Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(L...	30000
30000	1	[-Hash Match(Aggregate, HASH:([s1].[dbo].[MULTIDENSI...	30000
30000	1	[-Table Scan(OBJECT:([s1].[dbo].[MULTIDENSITY]))]	30000

- Filtered Statistics SQL 2008 introduced the Filtered Index and Filtered Statistics feature. The statistics object is created over a subset of the rows in a table based on a filter predicate. Filtered statistics can avoid a common problem in cardinality estimation in which estimates become skewed because of data correlation between columns. e.g. if you create a table called CARS, you might have a column called MAKE and a column called MODEL. The following table shows that multiple models of cars are made by Ford.

CAR_ID MAKE MODEL 1 Ford F-150 2 Ford Taunus 3 BMW M3

Also, assume that you want to run a query like the following:

```
SELECT * FROM CARS WHERE MAKE = 'Ford' AND MODEL = 'F-150';
```

When the query processor tries to estimate the selectivity for each condition in an AND clause, it usually assumes that each condition is independent. This allows the selectivity of each predicate to be multiplied together to form the total selectivity for the complete WHERE clause. For this example, it would be $2/3 * 1/3 = 2/9$. The actual selectivity is really $1/3$ for this query because every F-150 is a Ford. This kind of estimation error can be large in some data sets.

In addition to the Independence assumption, the QO contains other assumptions that are used both to simplify the estimation process and to be consistent in how estimates are made across all operators. Another assumption in the QO is **Uniformity**. This means that if a range of values is being considered but the values aren't known, they are assumed to be uniformly distributed over the range in which they exist. e.g. if a query has an IN list with different parameters for each value, the values of the parameters aren't assumed to be grouped. The final assumption in the QO is **Containment**. This says that if a range of values is being joined with another range of values, the default assumption is that query is being asked because those ranges overlap and qualify rows. Without this assumption, many common queries would be underestimated and poor plans would be chosen.

- String statistics

SQL Server 2005 introduced a feature to improve cardinality estimation for strings called **String Statistics or trie trees**. SQL Server histograms can have up to 200 steps, or unique values, to store information about the overall distribution of a table. Although this works well for many numeric types, the string data types often have many more unique values. Trie trees were created to store efficiently a sample of the strings in a column.

- Cardinality estimation details

During optimization, each operator in the query is evaluated to estimate the number of rows processed by that operator. This helps the QO make proper tradeoffs based on the costs of different query plans. This process is done bottom up, with the base table cardinalities and statistics being used as input to tree nodes above it. e.g. to explain how the cardinality derivation process works see:

```
CREATE TABLE Table3(col1 INT, col2 INT, col3 INT);
GO
SET NOCOUNT ON;
BEGIN TRANSACTION;

    DECLARE @i INT=0;

    WHILE @i< 10000
    BEGIN
        INSERT INTO Table3(col1, col2, col3) VALUES (@i, @i,@i % 50);

        SET @i+=1;
    END;
COMMIT TRANSACTION;
GO

SELECT col1, col2 FROM Table3 WHERE col3 < 10;
```

For this query, the filter operator requests statistics on each column participating in the predicate (col3 in this query). The request is passed down to Table3, where an appropriate statistics object is created or updated. That statistics object is then passed to the filter to determine the operator’s selectivity. Selectivity is the fraction of rows that are expected to be qualified by the predicate and then returned to the user. When the selectivity for an operator is computed, it’s multiplied by the current number of rows for the query. The selectivity of this filter operation is based on the histogram loaded for column col3, as shown in Figure below

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	Sting Index	Filter Expression	Unfiltered Rows
1	_WA_Sys_00000003_1088795B	Nov 27 2008 10:30AM	10000	10000	50	0	4	NO	NULL	10000
	All density	Average Length	Columns							
1	0.02	4	col3							
	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS					
1	0	0	200	0	1					
2	1	0	200	0	1					
3	2	0	200	0	1					
4	3	0	200	0	1					
5	4	0	200	0	1					
6	5	0	200	0	1					
7	6	0	200	0	1					
8	7	0	200	0	1					
9	8	0	200	0	1					
10	9	0	200	0	1					
11	10	0	200	0	1					

The distribution on col3 is uniformly distributed from 0 to 49, and 10/50 values are less than 10, or 20 percent of the rows. Therefore, the selectivity of this filter in the query is 0.2, and the calculation of the number of rows resulting from the filter is as follows:

```
(# rows in operator below) * (selectivity of this operator)
10000 * 0.2 = 2000 rows
```

The estimate for the operator is taken by looking at the histogram, counting the number of sampled rows matching the criteria (in this case, 10 histogram steps with 200 equal rows are required for values that match the filter condition). Then, the number of qualifying rows (2,000) is normalized against the number of rows sampled when the histogram was created (10,000) to create the selectivity for the operator (0.2). This is then multiplied by the current number of rows in the table (10,000) to get the estimated query output cardinality. The cardinality estimation process is continued for any other filter conditions, and the results are usually just multiplied to estimate the total selectivity for each condition.

When a multicolumn statistics object is created, it computes density information for the sets of columns being evaluated in the order of the statistics object. So a statistics object created on (col1, col2, col3) has density information stored for ((col1), (col1, col2), and (col1, col2, col3)).

```
CREATE TABLE Table4(col1 int, col2 int, col3 int)
GO
DECLARE @i int=0
WHILE @i< 10000
BEGIN
    INSERT INTO Table4(col1, col2, col3)
    VALUES (@i % 5, @i % 10,@i % 50);

    SET @i+=1
END
CREATE STATISTICS s1 on Table4(col1, col2, col3)
DBCC SHOW_STATISTICS (Table4, s1)
```

	All density	Average Length	Columns
1	0.2	4	col1
2	0.1	8	col1, col2
3	0.02	12	col1, col2, col3

If a similar table is created with random data in the first two columns, the density looks quite different. This would imply that every combination of col1, col2, and col3 is actually unique in that case.

- Limitations The cardinality estimation of SQL server is usually very good. Unfortunately, you can understand that the calculations explained earlier in this sections don't work perfectly in every query.
 - Multiple predicates in an operator
 - Deep query tree. The process of tree-based cardinality estimation is good, but it also means that any errors lower in the query tree are magnified as the calculation proceeds higher up the query tree to more and more operators.
 - Less common operators. The QO uses many operators. Most of the common operators have extremely deep support developed over multiple versions of the product. Some of the lesser-

used operators, however, don't necessarily have the same depth of support for every single scenario. So if you're using an infrequent operator or one that has only recently been introduced, it might not provide cardinality estimates that are as good as most core operators.

- Costing

The process of estimating cardinality is done using the logical query trees. Costing is the process of determining how much time each potential plan choice will require to run, and it's done separately for each physical plan considered. Because the QO considers multiple different physical plans that return the same results, this makes sense. Costing is the component that picks between hash joins and loops joins or between one join order and another. The idea behind costing is actually quite simple. Using the cardinality estimates and some additional information about the average and maximum width of each column in a table, it can determine how many rows fit on each database page. This value is then translated into a number of disk reads that a query requires to complete. The total costs for each operator are then added to determine the total query cost, and the QO can select the fastest (lowest-cost) query plan from the set of considered plans during optimization. To make the QO more consistent, the development team used several assumptions when creating the costing model.

- A query is assumed to start with a cold cache. This means that the QP assumes that each initial I/O for a query requires reading from disk.
- Random I/Os are assumed to be evenly dispersed over the set of pages in a table or index. If a nonindexed base table (a heap) has 100 disk pages and the query is doing 100 random bookmark-based lookups from a nonclustered index into that heap, the QO assumes that 100 random I/Os occur in the query against that heap because it assumes that each target row is on a separate page.

The QO has other assumptions built into its costing model. One assumption relates to how the client reads the query results. Costing assumes every query reads every row in the query result. However, some clients read only a few rows and then close the query. e.g. if you are using an application that shows you pages of rows on screen at a time, that application can read 40 rows even though the original query might have returned 10,000 rows. If the QO knows the number of rows the user will consume, it can optimize for the number in the plan selection process to pick a faster plan. SQL Server exposes a hint called FAST N for just this case.

```
CREATE TABLE A(col1 INT);
CREATE CLUSTERED INDEX i1 ON A(col1);
GO
SET NOCOUNT ON;
BEGIN TRANSACTION;
DECLARE @i INT=0;
WHILE @i < 10000
BEGIN
    INSERT INTO A(col1) VALUES (@i);
    SET @i+=1;
END;
COMMIT TRANSACTION;
GO
```

```

SELECT A1.*
FROM   A AS A1
JOIN   A AS A2
ON     A1.col1 = A2.col1;

SELECT A1.*
FROM   A AS A1
JOIN   A AS A2
ON     A1.col1 = A2.col1 OPTION (FAST 1);

/* FAST N
Specifies that the query is optimized for fast retrieval of the first
integer_value number of rows. This result is a non-negative integer. After
the first integer_value number of rows are returned,
the query continues execution and produces its full result set.
*/

USE AdventureWorks2022
GO
SELECT ProductID, OrderQty, SUM(LineTotal) AS Total
FROM   Sales.SalesOrderDetail
WHERE  UnitPrice < $5.00
GROUP BY ProductID, OrderQty
ORDER BY ProductID, OrderQty
OPTION (HASH GROUP, FAST 10);
GO

SET STATISTICS IO, TIME ON
SELECT * FROM Sales.SalesOrderDetail
--Table 'SalesOrderDetail'. Scan count 1, logical reads 1238, physical
reads 0
--CPU time = 125 ms, elapsed time = 787 ms.

SELECT * FROM Sales.SalesOrderDetail OPTION (FAST 1)
--Table 'SalesOrderDetail'. Scan count 1, logical reads 1238, physical
reads 2, page server reads 0, read-ahead reads 1225
--CPU time = 47 ms, elapsed time = 787 ms.

```

- Index selection

Index selection is one of the most important aspects of QO. The basic idea behind index matching is to take predicates from a WHERE clause, join condition, or other limiting operation in a query and to convert that operation so that it can be performed against an index. Two basic operations can be performed against an index.

- Seek for a single value or a range of values on the index key
- Scan the index forward or backward

The job of the QO is to figure out which predicates can be applied to the index to return rows as quickly as possible. Some predicates can be applied to an index, whereas others can't.

Predicates that can be converted into an index operation are often called **sargable, or “search-ARGument-able.”** This means that the form of the predicate can be converted into an index operation. Predicates that can never match or don’t match the selected index are called **non-sargable** predicates. Predicates that are non-sargable would be applied after any index seek or range scan operations so that the query can return rows that match all predicates. Making things somewhat confusing is that SQL Server usually evaluates non-sargable predicates within the seek/scan operator in the query tree. This is a performance optimization; if this weren’t done, the series of steps performed would be as follows:

1. Seek Operator: Seek to a key in an index’s B-tree.
2. Latch the page.
3. Read the row.
4. Release the latch on the page.
5. Return the row to the filter operator.
6. Filter: Evaluate the non-sargable predicate against the row. If it qualifies, pass the row to the parent operator. Otherwise, repeat step 2 to get the next candidate row

The actual operation in SQL Server looks like this:

1. Seek Operator: Seek to a key in an index’s B-tree.
2. Latch the page.
3. Read the row.
4. Apply the non-sargable predicate filter. If the row doesn’t pass the filter, repeat step 3. Other-wise, continue to step 5.
5. Release the latch on the page.
6. Return the row

This is called pushing non-sargable predicates.

Not all predicates can be evaluated in the seek/scan operator. Because the latch operation prevents other users from even looking at a page in the system, this optimization is reserved for predicates that are very cheap to perform. This is called non-pushable, non-sargable predicates. Examples include the following.

- Predicates on large objects (including varbinary(max), varchar(max), nvarchar(max))
 - Common language runtime (CLR) functions
 - Some T-SQL functions
- Filtered Indexes

At first glance, the Filtered Indexes feature is a subset of the functionality already contained in indexed views. Nevertheless, this feature exists for good reasons.

- Indexed views are more expensive to use and maintain than filtered indexes.
- The matching capability of the Indexed View feature isn’t supported in all editions of SQL Server.

Filtered Indexes are created using a new WHERE clause on a CREATE INDEX statement.


```

CREATE TABLE dbo.TestFilter1 (col1 INT, col2 INT);
GO

DECLARE @i INT = 0;

SET NOCOUNT ON;

BEGIN TRANSACTION;
    WHILE @i < 40000
    BEGIN
        INSERT INTO testfilter1(col1, col2) VALUES (rand()*1000, rand()*1000);
        SET @i+=1;
    END;
COMMIT TRANSACTION;
GO

CREATE INDEX i1 ON dbo.TestFilter1 (col2)
WHERE col2 > 800;

SELECT col2 FROM testfilter1
WHERE col2 > 800;

SELECT col2 FROM testfilter1
WHERE col2 > 799;

```

Filtered indexes can handle several scenarios (When can we use it?).

- If you are querying a table with a small number of distinct values and are using a multicolumn predicate in which some of the elements are fixed, you can create a filtered index to speed up this specific query. This might be useful for a regular report run only for your boss; it speeds up a small set of queries without slowing down updates as much for everyone else.
- The index can be used when an expensive query on a large table has a known query condition
- Indexed Views

Traditional, NONindexed views have been used for goals such as simplifying SQL queries, abstracting data models from user models, and enforcing user security. From an optimization perspective, SQL Server doesn't do much with these views because they are expanded, or in-lined, before optimization begins. SQL Server exposes a CREATE INDEX command on views that creates a materialized form of the query result. The resulting structure is physically identical to a table with a clustered index. Nonclustered indexes also are supported on this structure. The QO can use this structure to return results more efficiently to the user. The WITH(NOEXPAND) hints tells the query processor not to expand the view definition.

```

-- Create two tables for use in our indexed view
CREATE TABLE dbo.Table1 (
    id          INT PRIMARY KEY
    , submitdate DATETIME
    , comment   NVARCHAR(200)

```

```
);

CREATE TABLE dbo.Table2 (
    id          INT PRIMARY KEY IDENTITY
    , commentid INT
    , product   NVARCHAR(200)
);
GO

-- submit some data into each table
INSERT INTO dbo.Table1(id, submitdate, comment) VALUES (1, '2008-08-21',
'Conor Loves Indexed Views');
INSERT INTO dbo.Table2(commentid, product) VALUES (1, 'SQL Server');
GO

-- create a view over the two tables
CREATE VIEW dbo.v1 WITH SCHEMABINDING
AS
    SELECT
        t1.id
        , t1.submitdate
        , t1.comment
        , t2.product
    FROM dbo.Table1 t1
    JOIN dbo.Table2 t2
    ON   t1.id = t2.commentid;
GO

-- Indexed the view
CREATE UNIQUE CLUSTERED INDEX i1 ON v1(id);

-- Query the view directly --> matches
SELECT * FROM dbo.v1;

-- Query the statement used in the view definition --> matches as well
SELECT
    t1.id
    , t1.submitdate
    , t1.comment
    , t2.product
    FROM dbo.table1 t1
    JOIN dbo.table2 t2
    ON   t1.id = t2.commentid;

-- Query a logically equivalent statement used in the view definition that
(is written differently) --> matches as well
SELECT
    t1.id
    , t1.submitdate
    , t1.comment
    , t2.product
    FROM dbo.table2 t2
    JOIN dbo.table1 t1
    ON   t2.commentid = t1.id;
```

Query 1: Query cost (relative to the batch): 100%
 SELECT * FROM dbo.v1;



FIGURE 11-38 A direct reference match of an indexed view.

Microsoft SQL Server 2012 Internals

Query 1: Query cost (relative to the batch): 100%
 SELECT t1.id, t1.submitdate, t1.comment, t2.product F

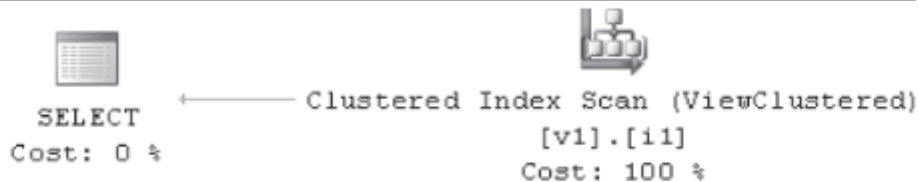
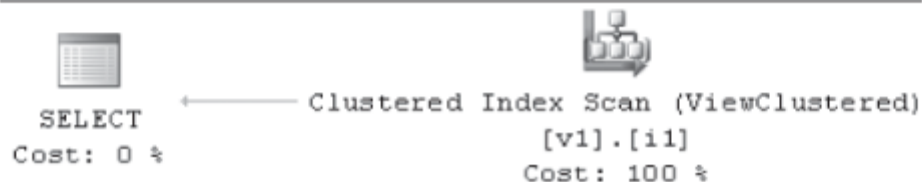


FIGURE 11-39 An indexed view match when the query is a match to the view definition.

Query 1: Query cost (relative to the batch): 100%
 SELECT t1.id, t1.submitdate, t1.comment, t2.product F



Although they are often the best plan choice, this isn't always the case.

```
CREATE TABLE dbo.Table5 (
    col1 INT PRIMARY KEY IDENTITY
    , col2 INT
);

INSERT INTO dbo.Table5(col2) VALUES (10);
INSERT INTO dbo.Table5(col2) VALUES (20);
INSERT INTO dbo.Table5(col2) VALUES (30);
GO

-- Create a view that returns values of col2 > 20
CREATE VIEW dbo.v2 WITH SCHEMABINDING
AS
SELECT
```

```

        t5.col1
    , t5.col2
FROM    dbo.Table5 t5
WHERE   t5.col2 > 20;
GO

-- materialize the view
CREATE UNIQUE CLUSTERED INDEX i1 ON v2(col1);
GO

-- Query the view and filter the results to have col2 values equal to 10.
-- The optimizer can detect this is a contradiction and avoid matching the
indexed view
-- (the trivial plan feature can "block" this optimization)
SELECT *
FROM    dbo.v2
WHERE   col2 = CONVERT(INT, 10)

```


[!IMPORTANT] **What is the contradiction on this query??**

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

```

SELECT * FROM dbo.v2 WHERE col2 = CONVERT(INT, 10);

```



Clustered Index Scan (Clustered)
[table3].[PK__table3__357D0D3E595B4...]
Cost: 100 %

SQL Server also supports matching indexed views in cases beyond exact matches of the query text to the view definition. It also supports using an indexed view for inexact matches in which the definition of the view is broader than the query submitted by the user. SQL Server then applies residual filters, projections.

The code below demonstrates view matching.

```

-- Base Table
CREATE TABLE dbo.BaseTbl1 (
    col1 INT
    , col2 INT
    , col3 BINARY(4000)
);

CREATE UNIQUE CLUSTERED INDEX i1 ON dbo.BaseTbl1(col1);
GO

-- Populate base table
SET NOCOUNT ON;

DECLARE @i INT = 0;

WHILE @i < 50000
BEGIN

```

```

INSERT INTO basetbl1(col1, col2) VALUES (@i, 50000-@i);

SET @i += 1;
END;
GO

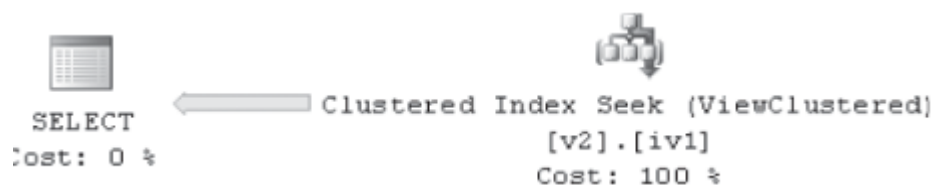
-- Ceate a view over the 2 integer columns
CREATE VIEW dbo.v3 WITH SCHEMABINDING
AS
    SELECT
        col1
        , col2
    FROM dbo.basetbl1;
GO

-- Index that on col2 (base table is only indexed on col1)
CREATE UNIQUE CLUSTERED INDEX iv1 on dbo.v3(col2);

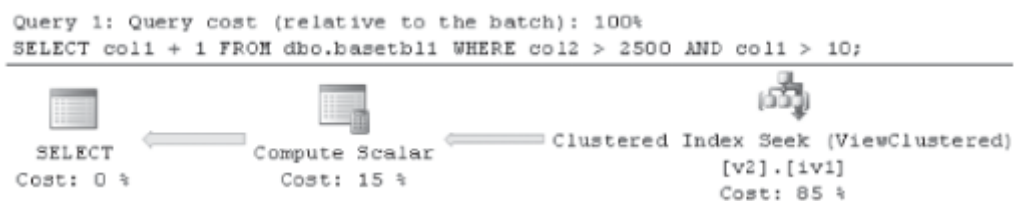
-- The indexed view still matches for both a restricted -- column set and a
restricted row set
SELECT
    col1
FROM dbo.basetbl1
WHERE col2 > 2500

SELECT
    col1 + 1
FROM dbo.basetbl1
WHERE col2 > 2500

```



The projection isn't explicitly listed as a separate Compute Scalar operator in this query because SQL Server 2012 has special logic to remove projections that don't compute an expression. The filter operator in the index matching code is translated into an index seek against the view. If you modify the query to compute an expression, Figure 11-44 demonstrates the residual Compute Scalar added to the plan.



- Partitioned tables

Table and index partitioning can help you manage large databases better and minimize downtime. Physically, partitioned tables and indexes are really N tables or N indexes that store a fraction of the rows. When comparing this to their nonpartitioned equivalents, the difference in the plan is often that the partitioned case requires iterating over a list of tables or a list of indexes to return all the rows. SQL Server represents partitioning in most cases by storing the partitions within the operator that accesses the partitioned table or index. This provides a number of benefits, such as enabling parallel scans to work properly.

```
-- Create Partition Function
CREATE PARTITION FUNCTION pf2008(date) AS RANGE RIGHT FOR VALUES ('2012-10-01', '2012-11-01', '2012-12-01');

-- Check partition function
SELECT * FROM sys.partition_functions WHERE name = 'pf2008';

-- Create Scheme Partition
CREATE PARTITION SCHEME ps2008 AS PARTITION pf2008 ALL TO ([PRIMARY]);

-- Check Partition Scheme
SELECT * FROM sys.partition_schemes WHERE name = 'ps2008';

-- Create Table on Partition Scheme
CREATE TABLE dbo.PTNSales(
    saledate DATE
    , salesperson INT
    , amount MONEY
) ON ps2008(saledate);

INSERT INTO dbo.PTNSales (saledate, salesperson, amount) VALUES ('2012-10-20', 1, 250.00);
INSERT INTO dbo.PTNSales (saledate, salesperson, amount) VALUES ('2012-11-05', 2, 129.00);
INSERT INTO dbo.PTNSales (saledate, salesperson, amount) VALUES ('2012-12-23', 2, 98.00);
INSERT INTO dbo.PTNSales (saledate, salesperson, amount) VALUES ('2012-10-03', 1, 450.00);

SELECT * FROM dbo.PTNSales WHERE (saledate) NOT BETWEEN '2012-11-01' AND '2012-11-30';
```

You can see that the base case doesn't require an extra join with a Constant Scan. This makes the query plans look like the nonpartitioned cases more often, which should make understanding the query plans easier. One benefit of this model is that getting parallel scans over partitioned tables is now possible. The following example creates a large partitioned table and then performs a COUNT(*) operation that generates a parallel scan.

```
CREATE PARTITION FUNCTION pfparallel(INT) AS RANGE RIGHT FOR VALUES (100, 200, 300);
```

```

GO

CREATE PARTITION SCHEME psparallel AS PARTITION pfparallel ALL TO
([PRIMARY]);
GO

CREATE TABLE testscan(randomnum INT, value INT, data BINARY(3000))
ON psparallel(randomnum);
GO

SET NOCOUNT ON;

BEGIN TRANSACTION;
  DECLARE @i INT=0;

  WHILE @i < 100000
  BEGIN
    INSERT INTO testscan(randomnum, value)
    VALUES (rand()*400, @i);

    SET @i+=1;
  END;
COMMIT TRANSACTION;
GO

-- now let's demonstrate a parallel scan over a partitioned table in SQL
Server 2012
SELECT COUNT(*) FROM testscan;

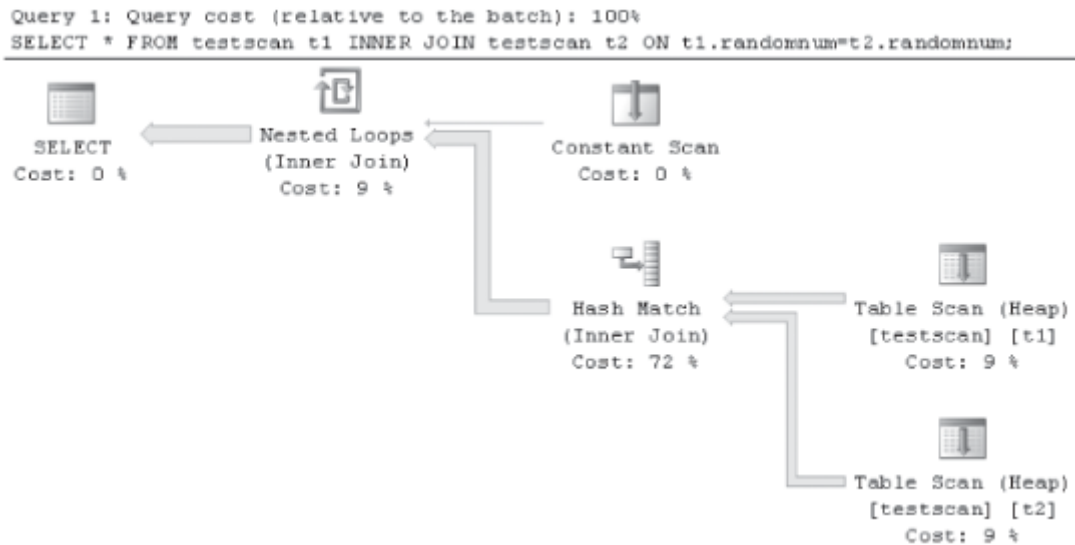
```

In SQL server, JOIN collocation is still represented by using the APPLY/NESTED LOOPS JOIN, but other cases use the traditional representation. The following example builds on the last example to demonstrate that joining with the same partitioning scheme can be done using the collocated **join technique**. What DOES THIS MEAN?

```

-- SQL Server join collocation uses the constant scan + apply model
SELECT *
FROM testscan t1
JOIN testscan t2
ON t1.randomnum = t2.randomnum;

```



- Windowing functions
- Data warehousing

SQL Server contains a number of special optimizations that speed the execution of data warehouse queries. Fact tables are usually so large that the use of nonclustered indexes is limited because of the large storage requirements to store these structures. Dimension tables are often indexed. First, SQL Server orders joins differently in data warehouses to try to perform as many limiting operations against the dimension tables as possible before performing a scan of the fact table. SQL Server also contains special bitmap operators that help reduce data movement across threads in parallel queries when using the star join pattern. SQL Server 2012 introduces significant changes in the space of relational data warehouse processing. First is a new kind of index called a columnstore. Second is a new query execution model called Batch Mode. Together, these two improvements can significantly improve runtime for data warehouse queries that use the star join pattern.

- Columnstore indexes

Columnstores in SQL Server 2012 are nonclustered indexes that use less space than a traditional B-tree index in SQL Server. They are intended to be created on the fact table of a data warehouse (and potentially also on large dimension tables). They achieve space savings by ignoring the standard practice of physically collocating all the column data for an index together for each row. Instead, they collocate values from each column together. The Sales fact table in Figure 11-52 conceptually shows how data is stored in different types of indexes. In traditional indexes, data is stored sequentially per row, aligned with the horizontal rectangle in Figure 11-52. In the columnstore index, data is stored sequentially per column.

Date	DepartmentID	ProductID	SalesAmt
12/21/2011	1	23	1000.00
12/21/2011	1	125	120.00
12/21/2011	2	3	3000.00

Date	DepartmentID	ProductID	SalesAmt
12/21/2011	1	23	1000.00
12/21/2011	1	125	120.00
12/21/2011	2	3	3000.00

FIGURE 11-52 Column orientation versus row orientation highlighted in a Sales fact table,

From the perspective of the QO, having a significantly smaller index reduces the I/O cost to read the fact table and process the query. In traditional (non-columnstore) indexes, data warehouse configurations are designed so that the dimension tables fit into buffer pool memory, but the fact table doesn't.

- Columnstore limitations and workarounds

One limitation is that tables must be marked as read-only as long as the columnstore exists. In other words, you can't perform INSERT, UPDATE, DELETE, or MERGE operations on the table while the columnstore index is active. With such a pattern, the index can potentially be dropped and re-created each evening around the daily extract, transform, and load (ETL) operation. Even when this isn't true, most large fact tables use table partitioning to manage the data volume. You can load a partition of data at a time because the SWITCH PARTITION operation is compatible with the columnstore index. The use of partitioning allows data to be loaded in a closer-to-real-time model, with no downtime of the index on the fact table. The other columnstore restrictions in SQL Server 2012 relate to data types and which operations block the use of batch processing. Columnstore indexes support data types that are commonly used in data warehouses. Some of the more complex data types, including varchar(max), nvarchar(max), CLR types, and other types not often found in fact tables are restricted from using columnstore indexes.

- Batch mode processing

The changes made for this new execution model significantly reduce the CPU requirements to execute each query. The batch execution model improves CPU performance in multiple ways. First, it reduces the number of CPU instructions needed to process each row, often by a factor of 10 or more. The batch execution model specifically implements techniques that greatly reduce the number of blocking memory references required to execute a query, allowing the system to finish queries much more quickly than would be otherwise possible with the traditional rowbased model.

- Grouping rows for repeated operations

In batch mode, data is processed in groups of rows instead of one row at a time. The number of rows per group depends on the query's row width and is designed to try to keep each batch

around the right size to fit into the internal caches of a CPU core.

- Column orientation within batches

Like columnstore index, data within batches is allocated by column instead of by row. This allocation model allows some operations to be performed more quickly. Continuing the filter example, a filter in a rowbased processing model would have to call down to its child operator to get each row. This code could require the CPU cache to load new instructions and new data from main memory. In the batch model, the instructions to execute the filter instructions will likely be in memory already because this operation is being performed on a set of rows within a batch.

- Data encoding

The third major difference in the batch model is that data is stored within memory using a probabilistic representation to further reduce the number of times the CPU core needs to access memory that isn't already in the CPU's internal caches. For example, a 64-bit nullable int field technically takes 64 bits for the data and another 1 bit for the null bit. Rather than store this in two CPU registers, the user data is encoded so that the most common values for 64 bit fields are stored within the main 64 bit word, and uncommon fields are stored outside the main 64 bit storage with a special encoding so that SQL Server can determine when the data is in batch vs out of batch.

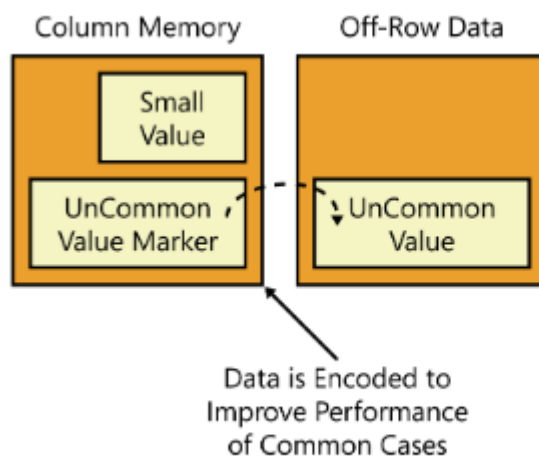


FIGURE 11-56 Uncommon data stored outside of the main batch.

- Logical database design best practices

- You need to determine whether each column needs to support NULLs. If the column can be declared as NOT NULL, this helps batch processing fit values more easily into a CPU register.
- You must design data warehouses to use the supported set of data types.
- Data uniqueness must be enforced elsewhere, either through a constraint or in a UNIQUE (B-tree) index in the physical database design

- Plan shape

Taken in total, the typical desired shape for data warehouse star join plans in SQL Server 2012 will be as follows.

- Join all dimension tables before a single scan of the fact table.
- Use hash joins for all these joins.
- Create bitmaps for each dimension and use them to scan the fact table.
- Use GROUP BY both at the top (row-based) and pushed down toward the source operations (local aggregates in batch mode).
- Use parallelism for the entire batch section of the query.

Figure below shows the common query plan shape for batch execution plans in this scenario.

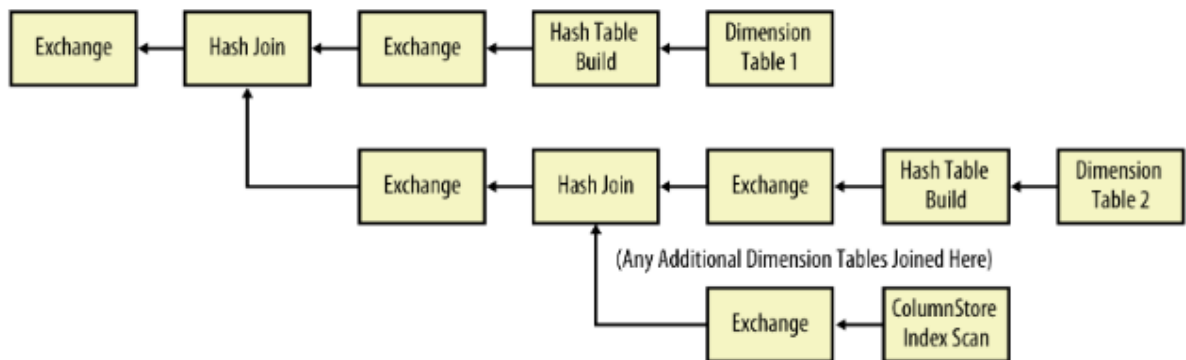


FIGURE 11-59 Typical columnstore plan shape.

When batch processing is used in a query, it usually outperforms the row-based approach, often by a significant amount (10x improvement is possible).

• Updates

Updates are an interesting area within query processing. In addition to many of the challenges faced while optimizing traditional SELECT queries, update optimization also considers physical optimizations such as how many indexes need to be touched for each row, whether to process the updates one index at a time or all at once, and how to avoid unnecessary deadlocks while processing changes as quickly as possible. The term update processing actually includes all top-level commands that change data, such as INSERT, UPDATE, DELETE, and MERGE. As you see in this section, SQL Server treats these commands almost identically. Every update query in SQL Server is composed of the same basic operations.

- It determines what rows are changed (inserted, updated, deleted, merged).
- It calculates the new values for any changed columns.
- It applies the change to the table and any nonclustered index structures.

```
CREATE TABLE update1 (col1 INT PRIMARY KEY IDENTITY, col2 INT, col3 INT);
INSERT INTO update1 (col2, col3) VALUES (2, 3);
```

Query 1: Query cost (relative to the batch): 100%
 INSERT INTO update1 (col2, col3) VALUES (2, 3);

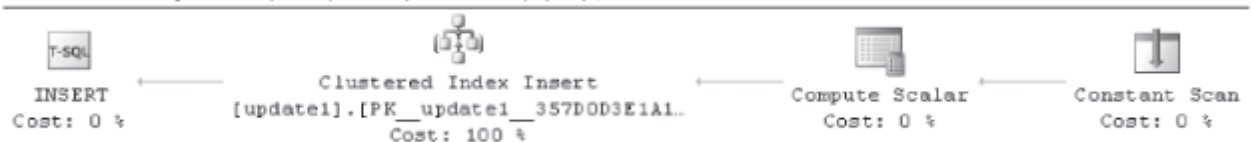


FIGURE 11-60 Basic INSERT query plan.

The INSERT query uses a special operator called a **Constant Scan** that, in relational algebra, generates rows without reading them from a table. If you are inserting a row into a table, it doesn't really have an existing table, so this operator creates a row for the insert operator to process. The **Compute Scalar** operation evaluates the values to be inserted. In the example, these are constants, but they could be arbitrary scalar expressions or scalar subqueries. Finally, the insert operator physically updates the primary key clustered index

```
UPDATE update1 SET col2 = 5;
```

Query 1: Query cost (relative to the batch): 100%
UPDATE update1 SET col2 = 5;



FIGURE 11-61 UPDATE query plan.

The UPDATE query reads values from the clustered index, performs a Top operation, and then updates the same clustered index. The Top operation is actually a placeholder for processing ROWCOUNT and does nothing unless you've executed a SET ROWCOUNT N operation in your session. Also note that in the example, the UPDATE command doesn't modify the key of the clustered index, so the row in the index doesn't need to be moved.

```
DELETE FROM update1 WHERE col3 = 10;
```

Query 1: Query cost (relative to the batch): 100%
DELETE FROM update1 WHERE col3 = 10;

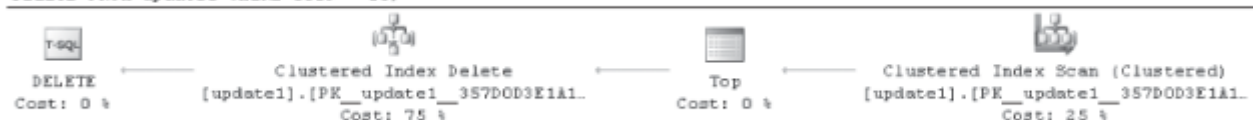


FIGURE 11-62 DELETE query plan.

```
CREATE TABLE update2 (col1 INT, col2 INT, col3 INT);
INSERT INTO update2 (col2, col3) VALUES (2, 3);
```

Query 1: Query cost (relative to the batch): 100%
INSERT INTO update2 (col2, col3) VALUES (2, 3);

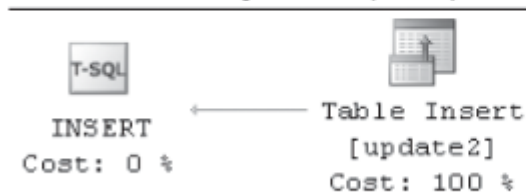


FIGURE 11-63 Simple INSERT query plan.

When the table is a heap (it has no clustered index), a special optimization occurs that can collapse the operations into a smaller form. This is called a simple update (the word update is used generically here to refer to insert, update, delete, and merge plans), and it's obviously faster. This single operator does all the work to insert into a heap, but it doesn't support every feature in Update.

```
CREATE TABLE update3 (col1 INT, col2 INT, col3 INT);

CREATE INDEX i1 ON update3(col1);
CREATE INDEX i2 ON update3(col2);
CREATE INDEX i3 ON update3(col3);

INSERT INTO update3(col1, col2, col3) VALUES (1, 2, 3);
```

Query 1: Query cost (relative to the batch): 100%
 INSERT INTO update3(col1, col2, col3) VALUES (1, 2, 3);

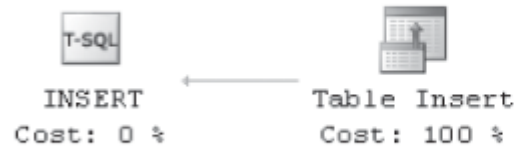


FIGURE 11-64 All-in-one *INSERT* query plan.

This query needs to update all the indexes because a new row has been created. However, Figure shows that the plan has only one operator. If you look at the properties for this operator in Management Studio, as shown in Figure below, you can see that it actually updates all indexes in one operator. This is another one of the physical optimizations done to improve the performance of common update scenarios. This kind of insert is called an all-in-one or a per-row insert.

Object	[s1].[dbo].[update3], [s1].[dbo].[update3].[i1]
[1]	[s1].[dbo].[update3]
[2]	[s1].[dbo].[update3].[i1]
[3]	[s1].[dbo].[update3].[i2]
[4]	[s1].[dbo].[update3].[i3]

FIGURE 11-65 Multiple indexes updated by a single operator.

By using the same table, you can try an UPDATE command to update some, but not all, of the indexes. Figure below shows the resulting query plan

```
UPDATE update3 SET col2=5, col3=5;
```



FIGURE 11-66 A query plan that modifies only some of the indexes on a table.

Now things are becoming a bit more complex. The query scans the heap in the Table Scan operator, performs the ROWCOUNT Top (in versions before SQL Server 2012), performs two Compute Scalars, and then performs a Table Update. If you examine the properties for the Table Update, notice that it lists only indexes i2 and i3 because the Query Optimizer can statically determine that this command

won't change i1. One of the Compute Scalars calculates the new values for the columns. The other is yet another physical optimization that helps compute whether each row needs to modify each and every index.

- Halloween Protection

Halloween Protection describes a feature of relational databases that's used to provide correctness in update plans. One simple way to perform an update is to have an operator that iterates through a B-tree index and updates each value that satisfies the filter. This works fine as long as the operator assigns a value to a constant or to a value that doesn't apply to the filter. However, if the query attempts more complex operations such as increasing each value by 10%, in some cases the iterator can see rows that have already been processed earlier in the scan because the previous update moved the row ahead of the cursor iterating through the B-tree

The typical protection against this problem is to scan all the rows into a buffer, and then process the rows from the buffer. In SQL Server, this is usually implemented by using a spool or a Sort operator.

- Split/Sort/Collpase

SQL Server contains a physical optimization called Split/Sort/Collapse, which is used to make wide update plans more efficient. The feature examines all the change rows to be changed in a batch and determines the net effect that these changes would have on an index. Unnecessary changes are avoided.

```
CREATE TABLE update5 (col1 INT PRIMARY KEY);

INSERT INTO update5(col1) VALUES (1), (2), (3);

UPDATE update5
SET col1 = col1 + 1;
```



FIGURE 11-67 Split/Sort/Collapse UPDATE query plan.

This query is modifying a clustered index that has three rows with values 1, 2, and 3. After this query, you would expect the rows to have the values 2, 3, and 4. Rather than modify three rows, you can determine that you can just delete 1 and insert 4 to make the changes to this query.

Now walk through what happens in each step. Before the split, the row data is shown in Table below.

TABLE 11-1 Pre-split update data representation

Action	Old Value	New Value
UPDATE	1	2
UPDATE	2	3
UPDATE	3	4

Split converts each *UPDATE* into one *DELETE* and one *INSERT*. Immediately after the split, the rows appear as shown in Table 11-2.

TABLE 11-2 Post-split data representation

Action	Value
DELETE	1
INSERT	2
DELETE	2
INSERT	3
DELETE	3
INSERT	4

The Sort sorts on (value, action), in which *DELETE* sorts before *INSERT*. After the sort, the rows appear as shown in Table 11-3.

TABLE 11-3 Post-sort data representation

Action	Value
DELETE	1
DELETE	2
INSERT	2
DELETE	3
INSERT	3
INSERT	4

The Collapse operator looks for (DELETE, INSERT) pairs for the same value and removes them. In this example, it replaces the DELETE and INSERT rows with UPDATE for the rows with the values 2 and 3.

- Merge

Like the other queries, the source data is scanned, filtered, and modified. However, in the case of MERGE, the set of rows to be changed is then joined with the target source to determine what should be done with each row. An existing table is going to be updated with new data, some of which might already exist in the table. Therefore, MERGE is used to determine only the set of rows that are missing.

```
CREATE TABLE AnimalsInMyYard (  
    sightingdate DATE  
    , Animal NVARCHAR(200)
```

```

);
GO

INSERT INTO AnimalsInMyYard(sightingdate, Animal) VALUES ('2012-08-12',
'Deer');
INSERT INTO AnimalsInMyYard(sightingdate, Animal) VALUES ('2012-08-12',
'Hummingbird');
INSERT INTO AnimalsInMyYard(sightingdate, Animal) VALUES ('2012-08-13',
'Gecko');
GO

CREATE TABLE NewSightings (
    sightingdate DATE
    , Animal NVARCHAR(200)
);
GO

INSERT INTO NewSightings(sightingdate, Animal) VALUES ('2012-08-13',
'Gecko');
INSERT INTO NewSightings(sightingdate, Animal) VALUES ('2012-08-13',
'Robin');
INSERT INTO NewSightings(sightingdate, Animal) VALUES ('2012-08-13',
'Dog');
GO

-- Insert values we have not yet seen - do nothing otherwise
MERGE AnimalsInMyYard A
USING NewSightings N
ON (    A.sightingdate = N.sightingdate
      AND A.Animal      = N.Animal
      ) WHEN NOT MATCHED
THEN
    INSERT (sightingdate, Animal) VALUES (sightingdate, Animal);

SELECT * FROM AnimalsInMyYard;

```

Take care when deciding to use MERGE. Although it's a powerful operator, it's also easily misused. MERGE is best-suited for OLTP workloads with small queries that use a two-query pattern, like this:

- Check for whether a row exists.
 - If it doesn't exist, INSERT
- Wide update plans

SQL Server also has special optimization logic to speed the execution of large batch changes to a table. If a query is changing a large percentage of a table, SQL Server can create a plan that avoids modifying each B-tree with many individual updates. Instead, it can generate a per-index plan that determines all the rows that need to be changed, sorts them into the order of the index, and then applies the changes in a single pass through the index. These plans are called **per index or wide update plans**


```

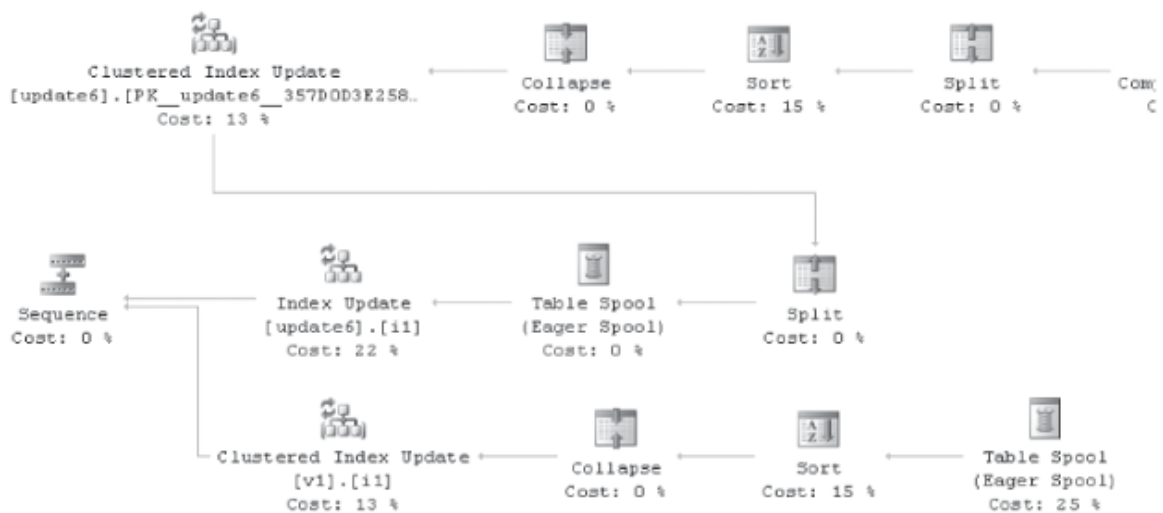
CREATE TABLE dbo.update6(col1 INT PRIMARY KEY, col2 INT, col3 INT);
CREATE INDEX i1 ON update6(col2);
GO

CREATE VIEW v1 WITH SCHEMABINDING AS
    SELECT col1, col2 FROM dbo.update6;
GO

CREATE UNIQUE CLUSTERED INDEX i1 ON v1(col1);

UPDATE update6 SET col1 = col1 + 1

```



- Non-updating updates

Nothing important. Search on AI

- Sparse column updates

SQL Server provides a feature called sparse columns that supports creating more columns in a table than were previously supported, as well as creating rows that were technically greater than the size of a database page.

- Partitioned updates

Updating partitioned tables is somewhat more complicated than nonpartitioned equivalents. Instead of a single physical table (heap) or B-tree, the query processor has to handle one heap or B-tree per partition. The following examples demonstrate how partitioning fits into these plans. The first example creates a partitioned table and then inserts a single row into it. Figure BELOW shows the plan.

```

CREATE PARTITION FUNCTION pfinert(INT) AS RANGE RIGHT FOR VALUES (100,
200, 300);

CREATE PARTITION SCHEME psinsert AS PARTITION pfinert ALL TO
([PRIMARY]);

```

```

go

CREATE TABLE testinsert(ptncol INT, col2 INT)
ON psinsert(ptncol);
go

INSERT INTO testinsert(ptncol, col2) VALUES (1, 2);

```

Looking at the query plan notice that this matches the behaviour you would expect from a nonpartitioned table. Changing partitions can be a somewhat expensive operation, especially when many of the rows in the table are being changed in a single statement. The Split/Sort/Collapse logic can also be used to reduce the number of partition switches that happen, improving runtime performance. In the following example, a large number of rows are inserted into the partitioned table, and the QO chooses to sort on the virtual partition ID column before inserting to reduce the number of partition switches at run time. Figure below shows the Sort optimization in the query plan.

```

CREATE TABLE #nonptn(ptncol INT, col2 INT)

DECLARE @i int = 0
WHILE @i < 10000
BEGIN
    INSERT INTO #nonptn(ptncol) VALUES (RAND()*1000)
    SET @i+=1
END
GO

INSERT INTO testinsert
SELECT * FROM #nonptn

```

- Locking

One special locking mode is called a U (for Update) lock. This special lock type is compatible with other S (shared) locks but incompatible with other U locks. The following code demonstrates how to examine the locking behavior of an update query plan. Figure below shows the query plan used in this example, and the other shows the locking output from sp_lock.

```

CREATE TABLE lock(col1 INT, col2 INT);
CREATE INDEX i2 ON lock(col2);

INSERT INTO lock (col1, col2) VALUES (1, 2);
INSERT INTO lock (col1, col2) VALUES (10, 3);

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN TRANSACTION;
    UPDATE lock
    SET col1 = 5

```

```
WHERE col1 > 5;

EXEC sp_lock;
ROLLBACK;
```



FIGURE 11-82 The update plan used in the locking example.

	spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
1	52	21	0	0	DB		S	GRANT
2	52	1	1131151075	0	TAB		IS	GRANT
3	52	21	693577509	0	PAG	1:75676	IX	GRANT
4	52	21	693577509	0	RID	1:75676:1	X	GRANT
5	52	21	693577509	0	TAB		IX	GRANT

You can run a slightly different query that shows that the locks vary based on the query plan selected. Figure below shows a seek based update plan. In the second example, the U lock is taken by the nonclustered index, whereas the base table contains the X lock. So, this U lock protection works only when going through the same access paths because it's taken on the first access path in the query plan. Figure 11-85 shows the locking behavior of this query.

```
BEGIN TRANSACTION;
UPDATE lock
SET col1 = 5
WHERE col2 > 2;
EXEC sp_lock;
```



FIGURE 11-84 Locking behavior of an update plan with a seek.

	spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
1	52	21	0	0	DB		S	GRANT
2	52	21	693577509	2	KEY	{a0004dc87aeb}	U	GRANT
3	52	1	1131151075	0	TAB		IS	GRANT
4	52	21	693577509	2	PAG	1:75678	IU	GRANT
5	52	21	693577509	0	PAG	1:75676	IX	GRANT
6	52	21	693577509	0	RID	1:75676:1	X	GRANT
7	52	21	693577509	0	TAB		IX	GRANT

FIGURE 11-85 The `sp_lock` output for a seek-based update plan.

- Partition-Level lock escalation Nothing important. Search on AI

- Distributed query

SQL Server includes a feature called Distributed Query, which accesses data on different SQL Server instances, other database sources, and non-database tabular data such as Microsoft Office Excel files or comma-separated text files.

Distributed Query supports several distinct use cases.

- You can use Distributed Query to move data from one source to another.
- You can use Distributed Query to integrate nontraditional sources into a SQL Server query.
- You can use Distributed Query for reporting. Because multiple sources can be queried in a single query, you can use Distributed Query to gather data into a single source and generate reports.
- You can use Distributed Query for scale-out scenarios.

Distributed Query is implemented within the Query Optimizer's plan-searching framework. Distributed queries initially are represented by using the same operators as regular queries. Each base table represented in the QO tree contains metadata collected from the remote source. The information collected is very similar to the information that the QP collects for local tables, including column data, index data, and statistics.

```

EXEC sp_addlinkedserver 'remote', N'SQL Server';
go

SELECT * FROM remote.Northwind.dbo.customers
WHERE ContactName = 'Marie Bertrand';
    
```



FIGURE 11-86 A fully remotable Distributed Query.

The Distributed Query feature, introduced in SQL Server 7.0, has some limitations that you should consider when designing scenarios that use it:

- Not every feature in SQL Server is supported via the remote query mechanism, such as some XML and UDT-based functionality.
- The costing model used within SQL Server is good for general use but sometimes generates a plan that's substantially slower than optimal.
- Plan hinting
 - Debugging plan issues

Determining when to use a hint requires an understanding of the workings of the QO and how it might not be making a correct decision, and then an understanding of how the hint might change the plan generation process to address the problem. This section explains how to identify cardinality estimation errors and then use hints to correct poor plan choices because these are usually something that can be fixed without buying new hardware or otherwise altering the host machine. The primary tool to identify cardinality estimation errors is the statistics profile output in SQL Server.

Other tools exist to track down performance issues with query plans. Using SET STATISTICS TIME ON is a good way to determine runtime information for a query.

```
SET STATISTICS PROFILE ON;
SELECT * FROM sys.objects;
```

Looking at the statistics profile output to find an error can help identify where the QO has used bad information to make a decision. If the QO makes a poor decision even with up-to-date statistics, this might mean that the QO has an out-of-model condition.

- {HASH | ORDER} GROUP

SQL Server has two possible implementations for GROUP BY (and DISTINCT).

- It can be implemented by sorting the rows and then grouping adjacent rows with the same grouping values.
- It can hash each group into a different memory location.

This hint is a good way to affect system performance, especially in larger queries and in situations when many queries are being run at once on a system.

- {MERGE | HASH | CONCAT} UNION

UNION ALL is a faster operation, in general because it takes rows from each input and simply returns them all. UNION must compare rows from each side and ensure that no duplicates are returned. Essentially, UNION performs a UNION ALL and then a GROUP BY operation over all output columns.

```
CREATE TABLE t1 (col1 INT);
CREATE TABLE t2 (col1 INT);
go

INSERT INTO t1(col1) VALUES (1), (2);
INSERT INTO t2(col1) VALUES (1);

SELECT * FROM t1
UNION
SELECT * FROM t2
OPTION (MERGE UNION)
```

As you can see, each hint forces a different query plan pattern:

- With common input sizes, MERGE UNION is useful
 - CONCAT UNION is best at low-cardinality plans (one sort)
 - HASH UNION works best when a small input can be used to make a hash table against which the other inputs can be compared
- FORCE ORDER, {LOOP | MERGE | HASH} JOIN

When estimating the number of rows that qualify a join, the best algorithm depends on factors such as the cardinality of the inputs, the histograms over those inputs, The available memory to store data in memory such as hash tables, and what indexes are available. If the cardinality or histograms aren't representative of the input, a poor join order or algorithm can result.

Places where I've seen these hints be appropriate in the past are as follows.

- Small, OLTP-like queries where locking is a concern.
 - Larger DW with many joins, complex data correlations, and enough of a fixed query pattern that you can reason about the join order in ways that make sense for all queries.
 - Systems that extend beyond traditional relational application design. Examples SQL store with Full-Text or XQuery components.
- INDEX = | Is very effective in forcing the QO to use a specific index when compiling a plan.
 - FORCESEEK

It tells the QO that it needs to generate a seek predicate when using an index. In a few cases, the QO can determine that an index scan is better than a seek when compiling the query. Earlier version of this hint semantically meant **SEEK ON THE FIRST COLUMN OF THE INDEX**. The primary scenario where this hint is interesting is to avoid locks in OLTP applications. This hint precludes an index scan, so it can be effective if you have a high-scale OLTP application where

locking is a concern in scaling and concurrency. The hint avoids the possibility of the plan taking more locks than desired.

- FAST

The QO assumes that the user will read every row produced by a query. Although this is often true, some user scenarios, such as manually paging through results, don't follow this pattern; in these cases, the client reads some small number of rows and then closes the query result. So, when a client wants only a few rows but doesn't specify a query that returns only a few rows, the latency of the first row can be slower because of the startup costs for stop-and-go operators such as hash joins, spools, and sorts. The FAST <number_rows> hint supplies the costing infrastructure with a hint from the user about how many rows the user will want to read from a query.

- MAXDOP

MAXDOP stands for maximum degree of parallelism, which describes the preferred degree of fan-out to be used when this query is run. A parallel query can consume memory and threads, blocking other queries that want to begin execution. In some cases, reducing the degree of parallelism for one or more queries is beneficial to the overall health of the system to lower the resources required to run a long-running query.

- OPTIMIZE FOR

When estimating cardinality for parameterized queries, the QO usually uses a less accurate estimate of the average number of distinct values in the column or sniffs the parameter value from the context. This sniffed value is used for cardinality estimation and plan selection. So parameter sniffing can help pick a plan that's good for a specific case. Because most data sets have nonuniform column distributions, the value sniffed can affect the runtime of the query plan. If a value representing the common distribution is picked, this might work very well in the average case and less optimally in the outlier case. If the outlier is used to sniff the value, the plan picked might perform noticeably worse than it would have if the average case value had been sniffed. The OPTIMIZE FOR hint allows the query author to specify the actual values to use during compilation. This can be used to tell the QO.

```
CREATE TABLE param1 (col1 INT, col2 INT);
GO

SET NOCOUNT ON;
BEGIN TRANSACTION;
  DECLARE @a INT = 0;

  WHILE @a < 5000
  BEGIN
    INSERT INTO param1(col1, col2) VALUES (@a, @a);
    SET @a+=1;
  END;

  WHILE @a < 10000
  BEGIN
```

```

        INSERT INTO param1(col1, col2) VALUES (5000, @a);
        SET @a+=1;
    END;
COMMIT TRANSACTION;
GO

CREATE INDEX i1 ON param1(col1);
go
CREATE INDEX i2 ON param1(col2);
go

DECLARE @b INT;
DECLARE @c INT;
SELECT * FROM param1 WHERE col1 = @b AND col2 = @c;

```

```

DECLARE @b INT;
DECLARE @c INT;
SELECT * FROM param1 WHERE col1 = @b AND col2 = @c
OPTION (optimize for (@b=5000))

```

- PARAMETERIZATION {SIMPLE ! FORCED}

FORCED parameterization always replaces most literals in the query with parameters. Because the plan quality can suffer, you should use FORCED with care, and you should have an understanding of the global behavior of your application. Usually, FORCED mode should be used only in an OLTP system with many almost equivalent queries that (almost) always yield the same query plan.

- NOEXPAND

By default, the QP expands view definitions when parsing and binding the query tree. The NOEXPAND hint **causes the QP to force the use of the indexed view** in the final query plan. In many cases, this can speed up the execution of the query plan because the indexed view often precomputes an expensive portion of a query.

- USE PLAN

Directs the QO to try to generate a plan that looks like the plan in the supplied XML string. The common user of this hint is a DBA or database developer who wants to fix a plan regression in the QO. If a baseline of good/expected query plans is saved when the application is developed or first deployed, these can be used later to force a query plan to change back to what was expected if the QO later determines to change to a different plan that's not performing well.

The following example demonstrates how to retrieve a plan hint from SQL Server and then apply it as a hint to a subsequent compilation to guarantee the query plan.

```

CREATE TABLE customers(id INT, name NVARCHAR(100));
CREATE TABLE orders(orderid INT, customerid INT, amount MONEY);
go

```



```
SET SHOWPLAN_XML ON;
go
SELECT * FROM customers c INNER JOIN orders o ON c.id = o.customerid;
```

After you copy the XML, you need to escape single quotes before you can use it in the USE PLAN hint. Usually, I copy the XML into an editor and then search for single quotes and replace them with double quotes. Then you can copy the XML into the query using the OPTION (USE PLAN '<xml . . ./>') hint.

```
SET SHOWPLAN_XML OFF;
SELECT * FROM customers c INNER JOIN orders o ON c.id = o.customerid
OPTION (USE PLAN '<ShowPlanXML
xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan"
Version="1.0" . . .>');
```

1.4.3. EXECUTOR

The QE runs the execution plan that the QO produced, acting as a dispatcher for all commands in the execution plan. This module goes through each command of the execution plan until the batch is complete. Most commands require interaction with the storage engine to modify or retrieve data and to manage transactions and locking. You can find more information on query execution and execution plans in Chapter 10, "Query execution."

- Introducing query processing and execution

This section iterates, discusses how to read and understand query plans, explores some of the most common query execution operators, and shows how SQL Server combines these operators to execute even the most complex queries.

- Iterators

SQL Server breaks queries down into a set of fundamental building blocks called operators or iterators. Iterators can have no children, have one, two, or more children, and can be combined into trees called query plans. An iterator reads input rows either from a data source such as a table or from its children and produces output rows, which it returns to its parent. **All version of SQL Server traditionally use a row at a time model, it is operators process only one row at a time. This approach is now called row-based processing. A new processing approach introduced with SQL Server 2012, called batch mode processing, uses operators that can process batches of rows at a time and will be explained at the end of this chapter in the section Columnstore indexes and batch processing. An iterator doesn't need specialized knowledge of its children or parent.**

- Properties of Iterators

Three important properties of iterators can affect query performance and are worth special attention: memory consumption, nonblocking vs. blocking, and dynamic cursor support.

- Memory consumption

All iterators require some small fixed amount of memory to store state, perform calculations, and so forth. SQL Server doesn't track this fixed memory or try to reserve this memory before executing a query. When SQL Server caches an executable plan, it caches this fixed memory so that it doesn't need to allocate it again and to speed up subsequent executions of the cached plan.

The amount of memory required by a memory-consuming operator is generally proportional to the number of rows processed. To ensure that the server doesn't run out of memory and that queries containing memory-consuming iterators don't fail, SQL Server estimates how much memory these queries need and reserves a memory grant before executing such a query.

Memory-consuming iterators can affect performance in a few ways.

- Queries with memory-consuming iterators might have to wait to acquire the necessary memory grant. This waiting can directly affect performance by delaying execution.
- If a memory-consuming iterator requests too little memory, it might need to spill data to disk during execution. Spilling can significantly affect query and system performance adversely because of the extra I/O overhead. Moreover, if an iterator spills too much data, it can run out of disk space on tempdb and fail.

The primary memory-consuming iterators are sort, hash join, and hash aggregation.

- Nonblocking vs. blocking iterators

Iterators can be classified into two categories.

- Iterators that consume input rows and produce output rows at the same time (in the GetRow method) These iterators are often referred to as nonblocking.
- Iterators that consume all input rows (generally in the Open method) before producing any output rows These iterators are often referred to as "blocking" or "stop-and-go."

The compute scalar iterator is a simple example of a nonblocking iterator. It reads an input row, computes a new output value using the input values from the current row, immediately outputs the new value, and continues to the next input row.

The sort iterator is a good example of a blocking iterator. The sort can't determine the first output row until it has read and sorted all input rows.

- Dynamic cursor support

The iterators used in a dynamic cursor query plan have special properties. Among other things, a dynamic cursor plan must be able to return a portion of the result set on each fetch request, must be able to scan forward or backward, and must be able to acquire scroll locks as it returns rows. To support this functionality, an iterator must be able to save and restore its state, must be able to scan forward or backward, must process one input

row for each output row it produces, and must be nonblocking. Not all iterators have all these properties, however.

- Reading query plans

SQL Server supports three showplan options: Graphical, text, and XML

- Graphical Plans

Nothing

- Text Plans

The text showplan option represents each iterator on a separate line. SQL Server uses indentation and vertical bars (| characters) to show the child–parent relationship between the iterators in the query tree. Two types of text plans are available: SET SHOWPLAN_TEXT ON, which displays just the query plan, and SET SHOWPLAN_ALL ON, which displays the query plan as well as most of the same estimates and statistics included in the graphical plan ToolTips window and Properties sheet.

- XML

The ability to nest XML elements makes XML a much more natural choice than text for representing the tree structure of a query plan. XML plans comply with a published XSD schema (at <http://schemas.microsoft.com/sqlserver/2004/07/showplan/showplanxml.xsd>)

- Estimated vs Actual Query Plans

Nothing

- Query Plan display Options

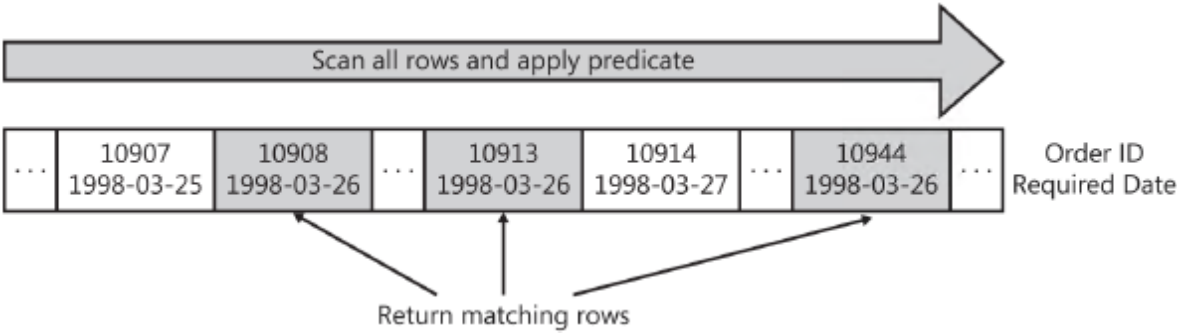
The element for each memory-consuming operator (in this example, just the sort) includes a element, which indicates the portion of the total memory grant used by that operator. Two fractions are available: The input fraction refers to the portion of the memory grant used while the operator is reading input rows, and the output fraction refers to the portion of the memory grant used while the operator is producing output rows. Generally, during the input phase of an operator's execution, it must share memory with its children; during the output phase of an operator's execution, it must share memory with its parent. Because the sort is the only memory-consuming operator in the plan in this example, it uses the entire memory grant. Thus, the fractions are both one.

- Analyzing Plans

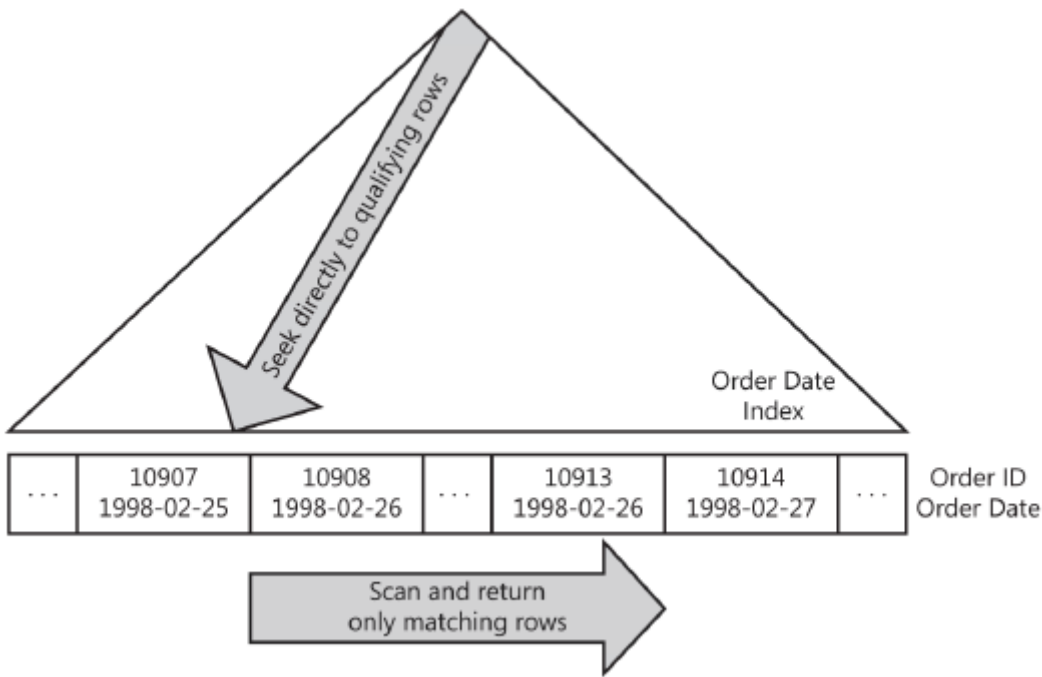
This section focuses on understanding the most common query operators and give some insight into when and how SQL Server uses them to construct a variety of interesting query plans.

1. Scans and seeks

Scans and seeks are the iterators that SQL Server uses to read data from tables and indexes. A scan processes an entire table or the entire leaf level of an index, whereas a seek efficiently returns rows from one or more ranges of an index based on a predicate.



a scan is an efficient strategy if the table is small or if many of the rows qualify for the predicate. However, if the table is large, and most of the rows do not qualify, a scan touches many more pages and rows and performs many more I/Os than is necessary.



A seek is generally a more efficient strategy when using a highly selective seek predicate.

TABLE 10-2 Scan and seek operators as they appear in a query plan

	Scan	Seek
Heap	Table scan	
Clustered index	Clustered index scan	Clustered index seek
Nonclustered index	Index scan	Index seek

1.1 Seekable predicates and covered columns

Before SQL Server can perform an index seek, it must determine whether the keys of the index are suitable for evaluating a predicate in the query.

1.1.1 Single-column indexes

SQL Server can use single-column indexes to answer most simple comparisons including equality and inequality comparisons. More-complex expressions, such as functions over a column and LIKE predicates with a leading wildcard character, generally prevent SQL Server from using an

index seek. Suppose that column Col1 has a single-column index. We can use this index to seek on these predicates.

- [Col1] = 3.14
- [Col1] > 100
- [Col1] BETWEEN 0 AND 99
- [Col1] LIKE 'abc%'
- [Col1] IN (2, 3, 5, 7)

However, we can't use the index to seek on these predicates.

- ABS([Col1]) = 1
- [Col1] + 1 = 9
- [Col1] LIKE '%abc'

1.1.2 Composite indexes

With a composite index, the order of the keys matters. It determines the sort order of the index and affects the set of seek predicates that SQL Server can evaluate using the index.

If we have an index on two columns, we can use only the index to satisfy a predicate on the second column if we have an equality predicate on the first column. Even if we can't use the index to satisfy the predicate on the second column, we might be able to use it on the first column. In this case, we introduce a "residual" predicate for the predicate on the second column. This predicate is evaluated just like any other scan predicate.

In these cases, column Col2 needs a residual predicate

- [Col1] > 100 AND [Col2] > 100
- [Col1] LIKE 'abc%' AND [Col2] = 2

Finally, we can't use the index to seek on the next set of predicates because we can't seek even on column Col1.

- [Col2] = 0
- [Col1] + 1 = 9 AND [Col2] BETWEEN 1 AND 9
- [Col1] LIKE '%abc' AND [Col2] IN (1, 3, 5)

1.1.3 Identifying index keys

Covered columns The heap or clustered index for a table contains all columns in the table.

2. Bookmark lookup (Key Lookup)

Consider a query with a predicate on a nonclustered index key that selects columns not covered by the index. Look at the following query:

```
SELECT OrderId, CustomerId FROM Orders WHERE OrderDate = '1998-02-26'
```

The nonclustered index OrderDate covers only the OrderId column

SQL Server has a solution for this problem. For each row that it fetches from the nonclustered index, it can look up the value of the remaining columns (for instance, the CustomerId column in the example) in the clustered index. This operation is called a bookmark lookup. A bookmark is a pointer to the row in the heap or clustered index.

In a graphical plan, SQL Server uses the Key Lookup icon to make the distinction between a typical clustered index seek and a bookmark lookup very clear.

The Key lookup are I/O random

Bookmark lookup isn't a cheap operation. Assuming (as is commonly the case) that no correlation exists between the nonclustered and clustered index keys, each bookmark lookup performs a random I/O into the clustered index or heap. Random I/Os are very expensive.

3. JOINS

SQL Server supports three physical join operators: **Nested Loop Join, Merge Join, and Hash Join**

No "best" join operator exists, and no join operator is inherently good or bad. We can't draw any conclusions about a query plan merely from the presence of a particular join operator.

3.1 Nested Loops Join

It compares each row from one table to each row from the other table, looking for rows that satisfy the join predicate.

The following pseudocode shows the nested loops join algorithm:

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

Consider this query:

```
SELECT O.[OrderId]
FROM [Customers] C JOIN [Orders] O ON C.[CustomerId] = O.[CustomerId]
WHERE C.[City] = N'London'
```

Rows Executes

```
46      1 |--Nested Loops(Inner Join, OUTER REFERENCES:([C].
      [CustomerId]))
6        1 |--Index Seek(OBJECT:([Customers].[City] AS [C]),
      SEEK:([C].[City]=N'London') ORDERED FORWARD)
46      6 |--Index Seek(OBJECT:([Orders].[CustomerId] AS [O]),
      SEEK:([O].[CustomerId]=[C].[CustomerId]) ORDERED FORWARD)
```

The outer table in this plan is Customers; the inner table is Orders. Hence, according to the nested loops join algorithm, SQL Server begins by seeking on the Customers table. The join takes one customer at a time and performs an index seek on the Orders table for each customer. The prior example illustrated two important techniques that SQL Server uses to boost the performance of a nested loops join: correlated parameters and, more importantly, an index seek based on those correlated parameters on the inner side of the join. The rules for determining whether a join predicate is suitable for use with an index seek are identical to the rules for determining whether any other predicate is suitable for an index seek. Consider the following query,

```
``sql
SELECT E1.[EmployeeId], COUNT(*)
FROM [Employees] E1 JOIN [Employees] E2
ON E1.[HireDate] < E2.[HireDate]
GROUP BY E1.[EmployeeId]
````
```

Because the HireDate column has no index, this query generates a simple nested loops join with a predicate but without any correlated parameters and without an index seek:

```
Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(int,[Expr1007],0)))
 Stream Aggregate(GROUP BY:([E1].[EmployeeID]) DEFINE:
([Expr1007]=Count(*)))
 Nested Loops(Inner Join, WHERE:([E1].[HireDate]<[E2].[HireDate]))
 Clustered Index Scan(OBJECT:([Employees].[PK_Employees] AS [E1]))
 Clustered Index Scan(OBJECT:([Employees].[PK_Employees] AS [E2]))
```

Now consider the following identical query that has been rewritten to use a CROSS APPLY:

```
SELECT E1.[EmployeeId], ECnt.Cnt
FROM [Employees] E1 CROSS APPLY
(
 SELECT COUNT(*) Cnt
 FROM [Employees] E2
 WHERE E1.[HireDate] < E2.[HireDate]
) ECnt
```

Although these two queries are identical and will always return the same results, the plan for the query with CROSS APPLY uses a nested loops join with a correlated parameter:

```
--Nested Loops(Inner Join, OUTER REFERENCES:([E1].[HireDate]))
--Clustered Index Scan(OBJECT:([Employees].[PK_Employees] AS [E1]))
--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(int,
[Expr1007],0)))
```

```
--Stream Aggregate(DEFINE:([Expr1007]=Count(*)))
--Clustered Index Scan (OBJECT:([Employees].[
```

- Join predicates and logical join types It supports all join predicates including equijoin (equality) predicates and inequality predicates. The nested loops join supports the following logical join operators.

- Inner join
- Left outer join
- Cross join
- Cross apply and outer apply
- Left semi-join and left anti-semi-join

The nested loops join doesn't support the following logical join operators.

- Right and full outer join
- Right semi-join and right anti-semi-join
- Full outer joins The nested loops join can't directly support full outer join. However, the optimizer can transform [Table1] FULL OUTER JOIN [Table2] into [Table1] LEFT OUTER JOIN [Table2] UNION ALL [Table2] LEFT ANTI-SEMI-JOIN [Table1].
- Costing

**WARNING** QUE ES UN ANTI-SEMIJOIN, SEMI-JOIN, EQUI-JOIN, ETC

### 3.2 Merge Join

The merge join requires at least one equijoin predicate. Moreover, the inputs to the merge join must be sorted on the join keys.

The merge join works by simultaneously reading and comparing the two sorted inputs one row at a time. At each step, it compares the next row from each input. If the rows are equal, it outputs a joined row and continues. If the rows aren't equal, it discards the lesser of the two inputs and continues. As soon as it reaches the end of either input, the merge join stops scanning.

Pseudocode:

```
```sql
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
    output (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
```

```

With a merge join each table is read at most once, and the total cost is proportional to the sum of the number of rows in the inputs.

- One-to-many vs. many-to-many merge join

Nothing

- Sort merge join vs. index merge join

SQL Server can get sorted inputs for a merge join in two ways: It can explicitly sort the inputs using a sort operator or it can read the rows from an index. In general, a plan using an index to achieve sort order is cheaper than a plan using an explicit sort.

- Join predicates and logical join types

Merge joins support multiple equijoin predicates as long as the inputs are sorted on all the join keys.

Merge join also support residual predicates

Merge joins support all outer and semi-join variations

Examples

```

```sql
SELECT O.[OrderId], C.[CustomerId], C.[ContactName]
FROM [Orders] O
JOIN [Customers] C
  ON O.[CustomerId] = C.[CustomerId]
```

```

Because no predicates exist other than the join predicates, both tables must be scanned in their entirety. Moreover, the CustomerId column of both tables has covering indexes. Thus, the optimizer chooses a merge join plan:

```

```sql
|--Merge Join(Inner Join, MERGE:([C].[CustomerId])=([O].[CustomerId]),
RESIDUAL:(...))
|--Clustered Index Scan(OBJECT:([Customers].[PK_Customers] AS [C]), ORDERED
FORWARD)
|--Index Scan(OBJECT:([Orders].[CustomerId] AS [O]), ORDERED FORWARD)
```

```

Now consider a slightly more complex example. The following query returns a list of orders that are shipped to cities different from the city on file for the customer who placed the order:

```

```sql
SELECT O.[OrderId], C.[CustomerId], C.[ContactName]
FROM [Orders] O JOIN [Customers] C
ON O.[CustomerId] = C.[CustomerId] AND O.[ShipCity] <> C.[City]
ORDER BY C.[CustomerId]
```

```

The ORDER BY clause encourages the optimizer to choose a merge join. (This point will be explained in a moment.) Here is the query plan:

```

```sql
|--Merge Join(Inner Join, MERGE:([C].[CustomerId])=([O].[CustomerId]),
  RESIDUAL:(... AND [O].[ShipCity]<>[C].[City]))
|--Clustered Index Scan(OBJECT:([Customers].[PK_Customers] AS [C]), ORDERED
FORWARD)
|--Sort(ORDER BY:([O].[CustomerId] ASC))
|--Clustered Index Scan(OBJECT:([Orders].[PK_Orders] AS [O]))
```

```

### 3.3 Hash Join

Hash joins parallelize and scale better than any other join and are great at minimizing response times for data warehouse queries. Hash joins require at least one equijoin predicate, support residual predicates, and support all outer and semi-joins. Unlike merge joins, hash joins don't require ordered input sets. The hash join algorithm executes in two phases known as build and probe. During the build phase, it reads all rows from the first input, hashes the rows on the equijoin keys, and creates or builds an in-memory hash table. During the probe phase, it reads all rows from the second input, hashes these rows on the same equijoin keys, and looks or probes for matching rows in the hash table. Pseudocode:

```

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
 output (R1, R2)
 end
 end

```

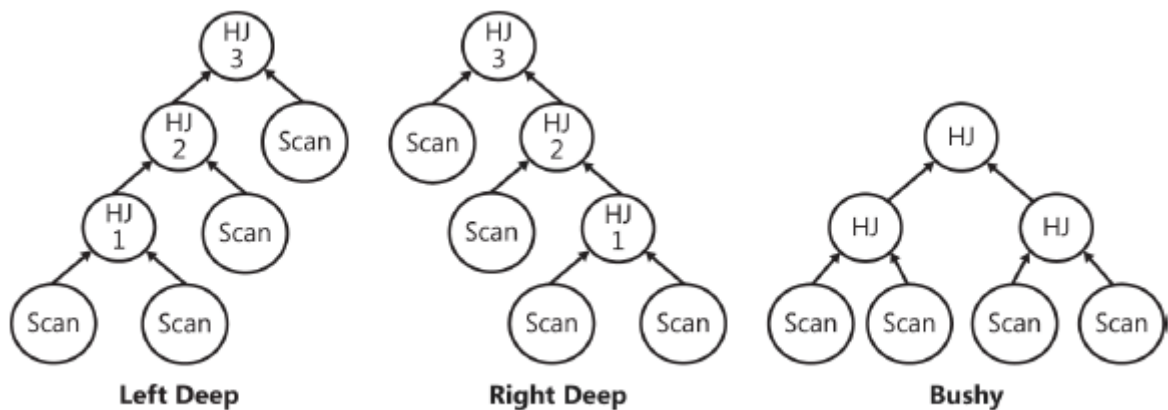
The hash join is blocking on its build input. That is, it must read and process its entire build input before it can return any rows. The hash join requires a memory grant to store the hash table.

- Memory and spilling

Before a hash join begins execution, SQL Server tries to estimate how much memory it will need to build its hash table. It uses the cardinality. To minimize the memory required, the optimizer chooses the smaller of the two tables as the build table. If the hash join might run out of memory during the build phase. It begins spilling to disk. This process of running out of memory and spilling buckets to disk can repeat multiple times until the build phase is complete. The hash join performs a similar process during the probe phase.

- Left deep vs. right deep vs. bushy hash join trees

The shape and order of joins in a query plan can significantly affect the performance of the plan - left deep, right deep, and bushy



**FIGURE 10-10** Three common shapes for query plans involving joins.

Example

```
SELECT O.[OrderId], O.[OrderDate], C.[CustomerId], C.[ContactName]
FROM [Orders] O
JOIN [Customers] C
ON O.[CustomerId] = C.[CustomerId]

|--Hash Match(Inner Join, HASH:([C].[CustomerId])=([O].[CustomerId]),
RESIDUAL:(...))
 |--Clustered Index Scan(OBJECT:([Customers].[PK_Customers] AS [C]))
 |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders] AS [O]))
```

- Summary of join properties

|                          | Nested loops join                                                         | Merge join                                                                                                               | Hash join                                                                        |
|--------------------------|---------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------|
| Best for...              | Relatively small inputs with an index on the inner table on the join key. | Medium to large inputs with indexes to provide order on the equijoin keys and/or where order is required after the join. | Data warehouse queries with medium to large inputs. Scalable parallel execution. |
| Concurrency              | Supports large numbers of concurrent users.                               | Many-to-one join with order provided by indexes (rather than explicit sorts) supports large numbers of concurrent users. | Best for small numbers of concurrent users.                                      |
| Stop and go              | No                                                                        | No                                                                                                                       | Yes (build input only)                                                           |
| Equijoin required        | No                                                                        | Yes (except for full outer join)                                                                                         | Yes                                                                              |
| Outer and semi-joins     | Left joins only (full outer joins via transformation)                     | All join types                                                                                                           | All join types                                                                   |
| Uses memory              | No                                                                        | No (might require sorts that use memory)                                                                                 | Yes                                                                              |
| Uses tempdb              | No                                                                        | Yes (many-to-many join only)                                                                                             | Yes (if join runs out of memory and spills)                                      |
| Requires order           | No                                                                        | Yes                                                                                                                      | No                                                                               |
| Preserves order          | Yes (outer input only)                                                    | Yes                                                                                                                      | No                                                                               |
| Supports dynamic cursors | Yes                                                                       | No                                                                                                                       | No                                                                               |

#### 4. Aggregations

SQL Server supports two physical operators for performing aggregations: **Stream Aggregate** and **Hash Aggregate**

##### 4.1 Scalar Aggregation

Scalar aggregates are queries with aggregate functions in the select list and no GROUP BY clause. Scalar aggregates always return a single row. SQL Server always implements scalar aggregates using the stream aggregate operator.

```

```sql
SELECT COUNT(*) FROM [Orders]

|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1004],0)))
|--Stream Aggregate(DEFINE:([Expr1004]=Count(*)))
|--Index Scan(OBJECT:([Orders].[OrderDate]))

SELECT MIN([OrderDate]), MAX([OrderDate]) FROM [Orders]

|--Stream Aggregate(DEFINE:([Expr1003]=MIN([Orders].[OrderDate]),
[Expr1004]=MAX([Orders].[OrderDate])))
|--Index Scan(OBJECT:([Orders].[OrderDate]))

SELECT AVG([Freight]) FROM [Orders]

|--Compute Scalar(DEFINE:([Expr1003]=CASE WHEN [Expr1004]=(0)
THEN NULL
ELSE [Expr1005]/CONVERT_IMPLICIT(money,[Expr1004],0)
END))

```

```

|--Stream Aggregate(DEFINE:([Expr1004]=COUNT_BIG([Orders].[Freight]),
Expr1005]=SUM([Orders].[Freight])))
|--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))

```

The CASE expression is needed to make sure that SQL Server does not attempt to divide by zero.

```
SELECT SUM([Freight]) FROM [Orders]
```

```

|--Compute Scalar(DEFINE:([Expr1003]=CASE WHEN [Expr1004]=(0)
THEN NULL
ELSE [Expr1005]
END))
|--Stream Aggregate(DEFINE:([Expr1004]=COUNT_BIG([Orders].[Freight]),
[Expr1005]=SUM([Orders].[Freight])))
|--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
...

```

4.1.1 Scalar distinct

```

```sql
SELECT COUNT(DISTINCT [ShipCity]) FROM [Orders]

|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
|--Stream Aggregate(DEFINE:([Expr1006]=COUNT([Orders].[ShipCity])))
|--Sort(DISTINCT ORDER BY:([Orders].[ShipCity] ASC))
|--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
...

```

#### 4.1.2 Multiple distinct

```

```sql
SELECT COUNT(DISTINCT [ShipAddress]), COUNT(DISTINCT [ShipCity])
FROM [Orders]

|--Nested Loops(Inner Join)
|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,
[Expr1009],0)))
|--Stream Aggregate(DEFINE:([Expr1009]=COUNT([Orders].[ShipAddress])))
|--Sort(DISTINCT ORDER BY:([Orders].[ShipAddress] ASC))
|--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))

|--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(int,
[Expr1010],0)))
|--Stream Aggregate(DEFINE:([Expr1010]=COUNT([Orders].[ShipCity])))
|--Sort(DISTINCT ORDER BY:([Orders].[ShipCity] ASC))
|--Clustered Index Scan
(OBJECT:([Orders].[PK_Orders]))
...

```

4.2 Stream Aggregation

Now that we've seen how to compute scalar aggregates, look at how SQL Server computes general-purpose aggregates that involve a GROUP BY clause.

The algorithm Stream aggregate relies on data arriving sorted by the GROUP BY column(s). Like a merge join, if a query includes a GROUP BY clause with more than one column, the stream aggregate can use any sort order that includes all the columns. The sort order might be delivered by an index or by an explicit sort operator. The sort order ensures that sets of rows with the same value for the GROUP BY columns will be adjacent to one another.

Pseudocode

```
```sql
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 current aggregate results (if any)
 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
```
```

For example, to compute a SUM, the stream aggregate considers each input row. If the input row belongs to the current group, the stream aggregate updates the current SUM by adding the appropriate value from the input row to the running total. If the input row belongs to a new group, the stream aggregate outputs the current SUM, resets the SUM to zero, and starts a new group.

Example

```
```sql
SELECT [ShipAddress], [ShipCity], COUNT(*)
FROM [Orders]
GROUP BY [ShipAddress], [ShipCity]

SELECT [ShipAddress], [ShipCity], COUNT(*)
FROM [Orders]
GROUP BY [ShipAddress], [ShipCity]
ORDER BY [ShipAddress], [ShipCity]

SELECT [CustomerId], COUNT(*)
FROM [Orders]
GROUP BY [CustomerId]
```

```

```

Select distinct If we have an index to provide order, SQL Server can also
use the stream aggregate to implement SELECT DISTINCT.

```sql
SELECT DISTINCT [CustomerId] FROM [Orders]

SELECT [CustomerId] FROM [Orders] GROUP BY [CustomerId]

/* Both queries use the same plan: */
|--Stream Aggregate(GROUP BY:([Orders].[CustomerID]))
 |--Index Scan(OBJECT:([Orders].[CustomerID]), ORDERED FORWARD)

SELECT [EmployeeId], COUNT(DISTINCT [CustomerId])
FROM [Orders]
GROUP BY [EmployeeId]

CREATE INDEX [EmployeeCustomer] ON [Orders] (EmployeeId, CustomerId)

|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
 |--Stream Aggregate(GROUP BY:([Orders].[EmployeeID])
 DEFINE:([Expr1006]=COUNT([Orders].[CustomerID])))
 |--Stream Aggregate(GROUP BY:([Orders].[EmployeeID], [Orders].
[CustomerId]))
 |--Index Scan(OBJECT:([Orders].[EmployeeCustomer]), ORDERED FORWARD)

DROP INDEX [Orders].[EmployeeCustomer]

SELECT [EmployeeId], COUNT(*), COUNT(DISTINCT [CustomerId])
FROM [Orders]
GROUP BY [EmployeeId]
```

```

4.3 Hash Aggregation

Is similar to hash join. It doesn't require sort order, requires memory and is blocking. Hash aggregate excels at efficiently aggregating very large data sets and in parallel plans, scales better than stream aggregate.

```

```sql
for each input row
begin
 calculate hash value on group by column(s)
 check for a matching row in the hash table
 if matching row not found
 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
 ...

```

Hash aggregate computes all the groups simultaneously. Like a hash join, a hash aggregate uses a hash table to store these groups. With each new input row, it checks the hash table to see whether the new row belongs to an existing group. If it does, it simply updates the existing group. If it doesn't, it creates a new group. Because the input data is unsorted, any row can belong to any group. Thus, a hash aggregate can't output any results until it finishes processing every input row

**\*\*Memory and spilling\*\*** A hash aggregate, however, stores only one row for each group, so the total memory requirement is actually proportional to the number and size of the output groups or rows. If a hash aggregate runs out of memory, it must begin spilling rows to a workfile in tempdb.

Example

```

```sql
SELECT [ShipCountry], COUNT(*)
FROM [Orders]
GROUP BY [ShipCountry]
```

```

**\*\*The ShipCountry column has only 21 unique values. Because a hash aggregate requires less memory as the number of groups decreases, and a sort requires memory proportional to the number of input rows, this time the optimizer chooses a plan with a hash aggregate:\*\***

```

```sql
|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
  |--Hash Match(Aggregate, HASH:([Orders].[ShipCountry]), RESIDUAL:(...
    DEFINE:([Expr1006]=COUNT(*)))
    |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))

SELECT [ShipCountry], COUNT(*)
FROM [Orders]
GROUP BY [ShipCountry]
ORDER BY [ShipCountry]

|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
  |--Stream Aggregate(GROUP BY:([Orders].[ShipCountry]) DEFINE:
([Expr1006]=Count(*)))
    |--Sort(ORDER BY:([Orders].[ShipCountry] ASC))
    |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

```

**\*\*If the table gets big enough and the number of groups remains small enough, eventually the optimizer will decide that using the hash aggregate and sorting after the aggregation are cheaper.\*\***



```

```sql
SELECT [ShipCountry], COUNT(*)
FROM [BigOrders]
GROUP BY [ShipCountry]
ORDER BY [ShipCountry]
```

```

**\*\*As the following plan shows, the optimizer concludes that using the hash aggregate and sorting 21 rows are better than sorting 4,150 rows and using a stream aggregate:\*\***

```

```sql
|--Sort(ORDER BY:([BigOrders].[ShipCountry] ASC))
|--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(int,[Expr1007],0)))
|--Hash Match(Aggregate, HASH:([BigOrders].[ShipCountry]),
RESIDUAL:(...) DEFINE:([Expr1007]=COUNT(*)))
|--Clustered Index Scan(OBJECT:([BigOrders].[OrderID]))
```

```

**Distinct** Just like a stream aggregate, a hash aggregate can be used to implement distinct operations.

```

SELECT DISTINCT [ShipCountry] FROM [Orders]

|--Hash Match(Aggregate, HASH:([Orders].[ShipCountry]), RESIDUAL:(...))
|--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))

```

Finally, hash aggregate can be used to implement distinct aggregates, including multiple distincts. The basic idea is the same as for stream aggregate. SQL Server computes each aggregate separately and then joins the results.

```

SELECT [ShipCountry], COUNT(DISTINCT [EmployeeId]), COUNT(DISTINCT
[CustomerId])
FROM [HugeOrders]
GROUP BY [ShipCountry]

|--Compute Scalar(DEFINE:([HugeOrders].[ShipCountry]=[HugeOrders].
[ShipCountry]))
|--Hash Match(Inner Join, HASH:([HugeOrders].[ShipCountry])=
([HugeOrders].[ShipCountry]),RESIDUAL:(...))
|--Compute Scalar(DEFINE:([HugeOrders].[ShipCountry]=[HugeOrders].
[ShipCountry]))
|--Compute Scalar(DEFINE:([Expr1005]=CONVERT_IMPLICIT(int,
[Expr1012],0)))
|--Hash Match(Aggregate, HASH:([HugeOrders].[ShipCountry]),RESIDUAL:
(...))
DEFINE:([Expr1012]=COUNT([HugeOrders].[CustomerID]))

```

- This plan uses both a hash aggregate and a stream aggregate in the computation of COUNT(DISTINCT [EmployeeId]). The hash aggregate eliminates duplicate values from the EmployeeId column; the sort and stream aggregate computes the counts.
- SQL Server uses a pair of hash aggregates to compute COUNT(DISTINCT [Customer-Id]). The bottommost hash aggregate eliminates duplicate values from the CustomerId column, whereas the topmost computes the counts.
- Because the hash aggregate doesn't return rows in any particular order, SQL Server can't use a merge join without introducing another sort. Instead, the optimizer chooses a hash join for this plan.

## 5. Unions

Two types of unions queries are available: UNION ALL and UNION

```

SELECT [FirstName] + N' ' + [LastName], [City], [Country] FROM [Employees]
UNION ALL
SELECT [ContactName], [City], [Country] FROM [Customers]

|--Concatenation
|--Compute Scalar(DEFINE:([Expr1003]=([Employees].[FirstName] + N' ')+
 [Employees].[LastName]))
|--Clustered Index Scan(OBJECT:([Employees].[PK_Employees]))
 |--Clustered Index Scan(OBJECT:([Customers].[PK_Customers]))

SELECT [City], [Country] FROM [Employees]
UNION
SELECT [City], [Country] FROM [Customers]

|--Sort(DISTINCT ORDER BY:([Union1006] ASC, [Union1007] ASC))
|--Concatenation
 |--Clustered Index Scan(OBJECT:([Employees].[PK_Employees]))
 |--Clustered Index Scan(OBJECT:([Customers].[PK_Customers]))

```

**Another essentially identical alternative plan is to replace the sort distinct with a hash aggregate. A sort distinct requires memory proportional to the number of input rows before the duplicates are eliminated; a hash aggregate requires memory proportional to the number of output rows after the duplicates are eliminated. Thus, a hash aggregate requires less memory than the sort distinct when many duplicates occur, and the optimizer is more likely to choose a hash aggregate when it expects many duplicates.**

```

SELECT [ShipCountry] FROM [Orders]
UNION
SELECT [ShipCountry] FROM [BigOrders]

|--Hash Match(Aggregate, HASH:([Union1007]), RESIDUAL:(...))
|--Concatenation
 |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))

```

```

|--Clustered Index Scan(OBJECT:([BigOrders].[OrderID]))

SELECT [EmployeeId], [FirstName] + N' ' + [LastName] AS [ContactName],
[City], [Country]
INTO [NewEmployees]
FROM [Employees]

ALTER TABLE [NewEmployees] ADD CONSTRAINT [PK_NewEmployees] PRIMARY KEY
([EmployeeId])

CREATE INDEX [ContactName] ON [NewEmployees]([ContactName])
CREATE INDEX [ContactName] ON [Customers]([ContactName])

SELECT [ContactName] FROM [NewEmployees]
UNION ALL
SELECT [ContactName] FROM [Customers]
ORDER BY [ContactName]

|--Merge Join(Concatenation)
|--Index Scan(OBJECT:([NewEmployees].[ContactName]), ORDERED FORWARD)
|--Index Scan(OBJECT:([Customers].[ContactName]), ORDERED FORWARD)

SELECT [ContactName] FROM [NewEmployees]
UNION
SELECT [ContactName] FROM [Customers]

|--Merge Join(Union)
|--Stream Aggregate(GROUP BY:([NewEmployees].[ContactName]))
|--Index Scan(OBJECT:([NewEmployees].[ContactName]), ORDERED FORWARD)
|--Stream Aggregate(GROUP BY:([Customers].[ContactName]))
|--Index Scan(OBJECT:([Customers].[ContactName]), ORDERED FORWARD)

```

**This time the plan has a merge join (union) or merge union operator. The merge union eliminates duplicate rows that appear in both of its inputs; it does not eliminate duplicate rows from either individual input. That is, if a name appears in both the NewEmployees and the Customers tables, the merge union will eliminate that duplicate name. However, if a name appears twice in the NewEmployees table or twice in the Customers table, the merge union by itself does not eliminate it. Thus, the optimizer has also added stream aggregates above each index scan to eliminate any duplicates from the individual tables. As discussed earlier, aggregation operators can be used to implement distinct operations.**

```

DROP TABLE [NewEmployees]
DROP INDEX [Customers].[ContactName]

```

The hash union operator is similar to a hash aggregate with two inputs. A hash union builds a hash table on its first input and eliminates duplicates from it just like a hash aggregate. It then reads its second input and, for each row, probes its hash table to see whether the row is a duplicate of a row from the first input. If the row isn't a duplicate, the hash union returns it. Because the hash union

doesn't insert rows from the second input into the hash table, it doesn't eliminate duplicates that appear only in its second input. To use a hash union, the optimizer either must explicitly eliminate duplicates from the second input or must know that the second input has no duplicates. Example

```
CREATE TABLE [BigTable] ([PK] int PRIMARY KEY, [Dups] int, [Pad] char(1000))
CREATE TABLE [SmallTable] ([PK] int PRIMARY KEY, [NoDups] int UNIQUE, [Pad]
char(1000))
SET NOCOUNT ON
DECLARE @i int
SET @i = 0
BEGIN TRAN
WHILE @i < 100000
BEGIN
INSERT [BigTable] VALUES (@i, 0, NULL)
SET @i = @i + 1
IF @i % 1000 = 0
BEGIN
COMMIT TRAN
BEGIN TRAN
END
END
COMMIT TRAN
SELECT [Dups], [Pad] FROM [BigTable]
UNION
SELECT [NoDups], [Pad] FROM [SmallTable]

|--Hash Match(Union)
|--Clustered Index Scan(OBJECT:([BigTable].[PK_BigTable]))
|--Clustered Index Scan(OBJECT:([SmallTable].[PK_SmallTable]))
```

## 6. Advanced index operations

- Dynamic index seeks

```
SELECT [OrderId]
FROM [Orders]
WHERE [ShipPostalCode] IN (N'05022', N'99362')

|--Index Seek(OBJECT:([Orders].[ShipPostalCode]), SEEK:([Orders].
[ShipPostalCode]=N'05022'
OR [Orders].[ShipPostalCode]=N'99362') ORDERED FORWARD)

DECLARE @SPC1 nvarchar(20), @SPC2 nvarchar(20)
SELECT @SPC1 = N'05022', @SPC2 = N'99362'
SELECT [OrderId]
FROM [Orders]
WHERE [ShipPostalCode] IN (@SPC1, @SPC2)

|--Nested Loops(Inner Join, OUTER REFERENCES:([Expr1009],[Expr1010],
[Expr1011]))
```

```

|--Merge Interval
| |--Sort(TOP 2, ORDER BY:([Expr1012] DESC, [Expr1013] ASC,
| [Expr1009] ASC, [Expr1014] DESC))
| |--Compute Scalar (DEFINE:([Expr1012]=((4)&[Expr1011]) = (4) AND
| NULL = [Expr1009],[Expr1013]=(4)&[Expr1011],[Expr1014]=(16)&
| [Expr1011]))
| |--Concatenation
| |--Compute Scalar(DEFINE:([@SPC2]=[@SPC2], [@SPC2]=[@SPC2],
| [Expr1003]=(62)))
| |--Constant Scan
| |--Compute Scalar(DEFINE:([@SPC1]=[@SPC1],
| [@SPC1]=[@SPC1], [Expr1006]=(62)))
| |--Constant Scan

```

The more complex plan works by eliminating duplicates from the IN list at execution time. The two constant scans and the concatenation operator generate a “constant table” with the two IN list values. Then the plan sorts the parameter values, and the merge interval operator eliminates the duplicates. Finally, the nested loops join executes the index seek once for each unique value.

```

DECLARE @OD1 datetime, @OD2 datetime
SELECT @OD1 = '1998-01-01', @OD2 = '1998-01-04'
SELECT [OrderId]
FROM [Orders]
WHERE [OrderDate] BETWEEN @OD1 AND DATEADD(day, 6, @OD1)
OR [OrderDate] BETWEEN @OD2 AND DATEADD(day, 6, @OD2)

```

Again, the plan includes a pair of constant scans and a concatenation operator. However, this time, rather than return discrete values from an IN list, the constant scans return ranges. Unlike the prior example, eliminating duplicates is no longer sufficient. Now the plan needs to handle ranges that aren’t duplicates but do overlap. The sort ensures that ranges that might overlap are next to one another and the merge interval operator collapses overlapping ranges. In the example, the two date ranges (1998-01-01 to 1998-01-07 and 1998-01-04 to 1998-01-10) do in fact overlap, and the merge interval collapses them into a single range (1998-01-01 to 1998-01-10). These plans are referred to as dynamic index seeks because the range(s) that SQL Server actually fetches aren’t statically known at compile time and are determined dynamically during execution.

- Index unions

```

SELECT [OrderId]
FROM [Orders]
WHERE [OrderDate] BETWEEN '1998-01-01' AND '1998-01-07'
OR [ShippedDate] BETWEEN '1998-01-01' AND '1998-01-07'

```

SQL Server has two strategies that it can use to execute a query such as this one. One option is to use a clustered index scan (or table scan) and apply the entire predicate to all rows in the table. This strategy is reasonable if the predicates aren't very selective and if the query will end up returning many rows. However, if the predicates are reasonably selective and if the table is large, the clustered index scan strategy isn't very efficient. The other option is to use both indexes. This plan looks like the following:

```
--Sort(DISTINCT ORDER BY:([Orders].[OrderId] ASC))
|--Concatenation
 |--Index Seek(OBJECT:([Orders].[OrderDate]),
 SEEK:([Orders].[OrderDate] >= '1998-01-01' AND
 [Orders].[OrderDate] <= '1998-01-07')
 ORDERED FORWARD)

 |--Index Seek(OBJECT:([Orders].[ShippedDate]),
 SEEK:([Orders].[ShippedDate] >= '1998-01-01' AND
 [Orders].[ShippedDate] <= '1998-01-07')
 ORDERED FORWARD)
```

The optimizer effectively rewrote the query as a union:

```
SELECT [OrderId]
FROM [Orders]
WHERE [OrderDate] BETWEEN '1998-01-01' AND '1998-01-07'
UNION
SELECT [OrderId]
FROM [Orders]
WHERE [ShippedDate] BETWEEN '1998-01-01' AND '1998-01-07'
```

By using both indexes, this plan gets the benefit of the index seek. However, the plan might generate duplicates if, as is likely in this example, any of the orders were both placed and shipped during the first week of 1998. To ensure that the query plan doesn't return any rows twice, the optimizer adds a sort distinct. This type of plan is referred to as an index union.

```
SELECT [OrderId]
FROM [Orders]
WHERE [OrderDate] = '1998-01-01'
OR [ShippedDate] = '1998-01-01'
```

**Now have a merge concatenation and a stream aggregate. Because we have equality predicates on the leading column of each index, the index seeks return rows sorted on the second column (the OrderId column) of each index. Because index seeks return sorted rows, this query is suitable for a merge concatenation. Moreover, because the merge concatenation returns rows sorted on the merge key (the OrderId column), the optimizer can use a stream aggregate instead of a sort to eliminate duplicates. This plan is generally a better choice because the sort distinct uses memory and could spill data to disk if it runs**

**out of memory, whereas the merge concatenation and stream aggregate don't use memory.**

```
SELECT [OrderId], [OrderDate], [ShippedDate]
FROM [BigOrders]
WHERE [OrderDate] = '1998-01-01'
OR [ShippedDate] = '1998-01-01'
```

- Index intersections

```
SELECT [OrderId]
FROM [Orders]
WHERE [OrderDate] = '1998-02-26'
AND [ShippedDate] = '1998-03-04'
```

/\* The optimizer has effectively rewritten this query as a join \*/

```
SELECT O1.[OrderId]
FROM [Orders] O1 JOIN [Orders] O2
ON O1.[OrderId] = O2.[OrderId]
WHERE O1.[OrderDate] = '1998-02-26'
AND O2.[ShippedDate] = '1998-03-04'
```

/\* This query plan is referred to as an index intersection. Just as an index union can use different operators depending on the plan, so can index intersection. \*/

```
SELECT [OrderId]
FROM [BigOrders]
WHERE [OrderDate] BETWEEN '1998-02-01' AND '1998-02-04'
AND [ShippedDate] BETWEEN '1998-02-09' AND '1998-02-12'
```

/\* The inequality predicates mean the index seeks no longer return rows sorted by the OrderId column; therefore, SQL Server can't use a merge join without first sorting the rows. Rather than sort, the optimizer chooses a hash join:\*/

```
--Hash Match(Inner Join, HASH:([BigOrders].[OrderID], [Uniq1002])=
([BigOrders].[OrderID], [Uniq1002]), RESIDUAL:(...))
|--Index Seek(OBJECT:([BigOrders].[OrderDate]),
SEEK:([BigOrders].[OrderDate] >= '1998-02-01' AND
[BigOrders].[OrderDate] <= '1998-02-04')
ORDERED FORWARD)
```

```
--Index Seek(OBJECT:([BigOrders].[ShippedDate]),
SEEK:([BigOrders].[ShippedDate] >= '1998-02-09' AND
[BigOrders].[ShippedDate] <= '1998-02-12')
ORDERED FORWARD)
```

## 7. Subqueries

Subqueries are essentially joins. Subqueries can be categorized in three ways:

- **Noncorrelated vs. correlated subqueries** A noncorrelated subquery has no dependencies on the outer query, can be evaluated independently of the outer query, and returns the same result for each row of the outer query. A correlated subquery does have a dependency on the outer query. It can only be evaluated in the context of a row from the outer query and might return a different result for each row of the outer query.
- **Scalar vs. multirow subqueries** A scalar subquery returns or is expected to return a single row (that is, a scalar), whereas a multirow subquery might return a set of rows.
- **The clause of the outer query in which the subquery appears**

## 7.1 Noncorrelated scalar subqueries

The following query returns a list of orders in which the freight charge exceeds the average freight charge for all orders:

```
```sql
SELECT O1.[OrderId], O1.[Freight]
FROM [Orders] O1
WHERE O1.[Freight] >
(
  SELECT AVG(O2.[Freight])
  FROM [Orders] O2

  SELECT O.[OrderId]
  FROM [Orders] O
  WHERE O.[CustomerId] =
  (
    SELECT C.[CustomerId]
    FROM [Customers] C
    WHERE C.[ContactName] = N'Maria Anders'
  )
)
```
```

The ANY aggregate is a special internal-only aggregate that, as its name suggests, returns any row. Because this plan raises an error if the scan of the Customers table returns more than one row, the ANY aggregate has no real effect. This is the same reason the following query doesn't compile:

```
```sql
SELECT COUNT(*), C.[CustomerId]
FROM [Customers] C
WHERE C.[ContactName] = N'Maria Anders'
```
```

Often, creating a unique index or rewriting the query eliminates the assert operator and improves the query plan. For example:



```

```sql
SELECT O.[OrderId]
FROM [Orders] O JOIN [Customers] C
ON O.[CustomerId] = C.[CustomerId]
WHERE C.[ContactName] = N'Maria Anders'

/* Although writing this query as a simple join is the best option, consider
what happens if a unique index is created on the ContactName column: */

CREATE UNIQUE INDEX [ContactName] ON [Customers] ([ContactName])

SELECT O.[OrderId]
FROM [Orders] O
WHERE O.[CustomerId] =
(
SELECT C.[CustomerId]
FROM [Customers] C
WHERE C.[ContactName] = N'Maria Anders'
)
DROP INDEX [Customers].[ContactName]

/* Because of the unique index, the optimizer knows that the subquery can
produce only one row, eliminates the now unnecessary stream aggregate and
assert operators, and converts the query into a join*/
```

```

## 7.2 Correlated scalar subqueries

It returns those orders in which the freight charge exceeds the average freight charge of all previously placed orders:

```

```sql
SELECT O1.[OrderId]
FROM [Orders] O1
WHERE O1.[Freight] >
(
SELECT AVG(O2.[Freight])
FROM [Orders] O2
WHERE O2.[OrderDate] < O1.[OrderDate]
)
```

```

SQL Server evaluated the subquery first and then executed the main query. This time SQL Server evaluates the main query first and then evaluates the subquery once for each row from the main query:

```

```sql
|--Filter(WHERE:([O1].[Freight]>[Expr1004]))
|--Nested Loops(Inner Join, OUTER REFERENCES:([O1].[OrderDate]))
```

```

```

|--Clustered Index Scan(OBJECT:([Orders].[PK_Orders] AS [O1]))
|--Index Spool(SEEK:([O1].[OrderDate]=[O1].[OrderDate]))
|--Compute Scalar(DEFINE:([Expr1004]=
CASE WHEN [Expr1011]=(0)
THEN NULL
ELSE [Expr1012]
/CONVERT_IMPLICIT(money,[Expr1011],0) END))

|--Stream Aggregate (DEFINE:([Expr1011]=
COUNT_BIG([O2].[Freight]),[Expr1012]=SUM([O2].[Freight])))
|--Index Spool(SEEK:([O2].[OrderDate] <[O1].[OrderDate]))
|--Clustered Index Scan
(OBJECT:([Orders].[PK_Orders] AS [O2]))
...

```

**\*\*The index spool immediately above the scan of [O2] is an eager index spool or index-on-the-fly spool. It builds a temporary index on the OrderDate column of the Orders table. It's called an eager spool because it "eagerly" loads its entire input set and builds the temporary index as soon as it's opened.\*\***

**\*\*The stream aggregate computes the average freight charge for each execution of the subquery. The index spool above the stream aggregate is a lazy index spool. It merely caches subquery results. If it encounters any OrderDate a second time, it returns the cached result rather than recomputing the subquery. It's called a lazy spool because it "lazily" loads results on demand only. Finally, the filter at the top of the plan compares the freight charge for each order to the subquery result ([Expr1004]) and returns those rows that qualify.\*\***

**\*\*We can determine the types of spools more easily from the complete SHOWPLAN\_ALL output or from the graphical plan.\*\***

Next, look at another example of a correlated scalar subquery. Suppose that we want to find those orders for which the freight charge exceeds the average freight charge for all orders placed by the same customer:

```

```sql
SELECT O1.[OrderId], O1.[Freight]
FROM [Orders] O1
WHERE O1.[Freight] >
(
SELECT AVG(O2.[Freight])
FROM [Orders] O2
WHERE O2.[CustomerId] = O1.[CustomerId]
)

```

/* This query is very similar to the previous one, yet a substantially different plan results: */

```

|--Nested Loops(Inner Join)
|--Table Spool
| |--Segment

```

```

| |--Sort(ORDER BY:([01].[CustomerID] ASC))
| |--Clustered Index Scan (OBJECT:([Orders].[PK_Orders] AS [01]),
  WHERE:([01].[CustomerID] IS NOT NULL))
|--Nested Loops(Inner Join, WHERE:([01].[Freight]>[Expr1004]))
|--Compute Scalar(DEFINE:([Expr1004]=
  CASE WHEN [Expr1012]=(0)
  THEN NULL
  ELSE [Expr1013]
  /CONVERT_IMPLICIT(money,[Expr1012],0)
  END))
|--Stream Aggregate(DEFINE:([Expr1012]=
COUNT_BIG([01].[Freight]),[Expr1013]=SUM([01].[Freight])))
| |--Table Spool
|--Table Spool
...

```

****The segment operator breaks the rows into groups (or segments) with the same value for the CustomerId column. Because the rows are sorted, sets of rows with the same CustomerId value will be consecutive. Next, the table spool—a segment spool—reads and saves one of these groups of rows that share the same CustomerId value.****

****When the spool finishes loading a group of rows, it returns a single row for the entire group. The topmost nested loops join executes its inner input. The two leaf level table spools replay the group of rows that the original segment spool saved.****

****The stream aggregate computes the average freight for each group of rows. The result of the stream aggregate is a single row. The inner of the two nested loops compares each spooled row against this average and returns those rows that qualify. Finally the segment spool truncates its work and repeats the process beginning with reading the next group with the next CustomerId value****

Finally, suppose that we want to compute the order with the maximum freight charge placed by each customer

```

```sql
SELECT O1.[OrderId], O1.[Freight]
FROM [Orders] O1
WHERE O1.[Freight] =
(
 SELECT MAX(O2.[Freight])
 FROM [Orders] O2
 WHERE O2.[CustomerId] = O1.[CustomerId]
)

|--Top(TOP EXPRESSION:((1)))
|--Segment
|--Sort(ORDER BY:([01].[CustomerID] DESC, [01].[Freight] DESC))
|--Clustered Index Scan
(OBJECT:([Orders].[PK_Orders] AS [01]),
WHERE:([01].[Freight] IS NOT NULL AND

```

```
[01].[CustomerID] IS NOT NULL))
```

```

****This plan sorts the Orders table by the CustomerId and Freight columns. As in the previous example, the segment operator breaks the rows into groups or segments with the same value for the CustomerId column. The top operator is a segment top. Unlike a typical top, which returns the top N rows for the entire input set, a segment top returns the top N rows for each group. The top is also a “top with ties.” A top with ties returns more than N rows if the Nth row has any duplicates or ties. In this query plan, because the sort ensures that the rows with the maximum freight charge are ordered first within each group, the top returns the row or rows with the maximum freight charge from each group. This plan is very efficient because it processes the Orders table only once.****

7.3 Removing correlations

Example. The following query uses a noncorrelated subquery to return orders placed by customers who live in London

```
```sql
SELECT O.[OrderId]
FROM [Orders] O
WHERE O.[CustomerId] IN
(
SELECT C.[CustomerId]
FROM [Customers] C
WHERE C.[City] = N'London'
)
```

/\* We can easily write the same query using a correlated subquery: \*/

```
SELECT O.[OrderId]
FROM [Orders] O
WHERE EXISTS
(
SELECT *
FROM [Customers] C
WHERE C.[CustomerId] = O.[CustomerId]
AND C.[City] = N'London'
)
```

/\* One query includes a noncorrelated subquery; the other includes a correlated subquery. However, the optimizer generates the same plan for both queries: \*/

```
--Nested Loops(Inner Join, OUTER REFERENCES:([C].[CustomerID]))
--Index Seek(OBJECT:([Customers].[City] AS [C]),
SEEK:([C].[City]=N'London'))
```

```

ORDERED FORWARD)
|--Index Seek(OBJECT:([Orders].[CustomerID] AS [O]),
SEEK:([O].[CustomerID]= [C].[CustomerID])
ORDERED FORWARD)
...

```

**\*\*In the second query, the optimizer removes the correlation from the subquery so that it can scan the Customers table first (on the outer side of the nested loops join). If the optimizer didn't remove the correlation, the plan would need to scan the Orders table first to generate a CustomerId value before it could scan the Customers table.\*\***

For more complex example of subquery decorrelation.

```

```sql
/* CHECK THE ANALYSIS THAT I DID WITH THE SCRIPTS */

SELECT 01.[OrderId], 01.[Freight],
(
SELECT AVG(02.[Freight])
FROM [Orders] 02
WHERE 02.[CustomerId] = 01.[CustomerId]
) Avg_Freight
FROM [Orders] 01

|--Compute Scalar(DEFINE:([Expr1004]=[Expr1004]))
|--Hash Match(Right Outer Join,HASH:([02].[CustomerId])=([01].
[CustomerId]),
RESIDUAL:(...))
|--Compute Scalar(DEFINE:([Expr1004]=CASE WHEN [Expr1013]=(0)
THEN NULL
ELSE [Expr1014]
/CONVERT_IMPLICIT(money,[Expr1013],0)
END))
| |--Stream Aggregate(GROUP BY:([02].[CustomerId])
DEFINE:([Expr1013]=COUNT_BIG([02].[Freight]),
[Expr1014]=SUM([02].[Freight])))
| |--Sort(ORDER BY:([02].[CustomerId] ASC))
| |--Clustered Index Scan
(OBJECT:([Orders].[PK_Orders] AS [02]))
|--Clustered Index Scan(OBJECT:([Orders].[PK_Orders] AS [01]))
...

```

7.4 Subqueries in CASE expressions

```

```sql
CREATE TABLE [MainTable] ([PK] int PRIMARY KEY, [Col1] int, [Col2] int,
[Col3] int)
CREATE TABLE [WhenTable] ([PK] int PRIMARY KEY, [Data] int)

```

```

CREATE TABLE [ThenTable] ([PK] int PRIMARY KEY, [Data] int)
CREATE TABLE [ElseTable] ([PK] int PRIMARY KEY, [Data] int)

INSERT [MainTable] VALUES (1, 11, 101, 1001)
INSERT [MainTable] VALUES (2, 12, 102, 1002)
INSERT [WhenTable] VALUES (11, NULL)
INSERT [ThenTable] VALUES (101, 901)
INSERT [ElseTable] VALUES (102, 902)

SELECT
 M.[PK],
 CASE
 WHEN EXISTS (SELECT * FROM [WhenTable] W WHERE W.[PK] = M.[Col1])
 THEN (SELECT T.[Data] FROM [ThenTable] T WHERE T.[PK] = M.[Col2])
 ELSE (SELECT E.[Data] FROM [ElseTable] E WHERE E.[PK] = M.[Col3])
 END AS Case_Expr
FROM [MainTable] M

DROP TABLE [MainTable], [WhenTable], [ThenTable], [ElseTable]
...

```

**\*\*As we might expect, this query plan begins by scanning MainTable, which returns two rows. Next, the plan executes the WHEN clause of the CASE expression. The plan implements the EXISTS subquery using a left semi-join with WhenTable. In this case, the query must return all rows from MainTable, regardless of whether these rows have a matching row in WhenTable.\*\***

## 8. Parallelism

SQL Server can execute queries using multiple CPUs simultaneously; this capability is referred to as parallel query execution.

Although parallelism can be used to reduce the response time of a single query, this speedup comes at a cost: It increases the overhead associated with executing a query. Although this overhead is relatively small, it does make parallelism inappropriate for small queries.

Parallelism is primarily useful on servers running a relatively small number of concurrent queries.

SQL Server parallelizes queries by horizontally partitioning the input data into approximately equal-sized sets, assigning one set to each CPU, and then performing the same operation (such as aggregate, join, and so on) on each set.

The query optimizer decides whether to execute a query in parallel. For the optimizer even to consider a parallel plan, the following criteria must be met.

- SQL Server must be running on a multiprocessor, multicore, or hyperthreaded machine.
- The process affinity configuration must allow SQL Server to use at least two processors.

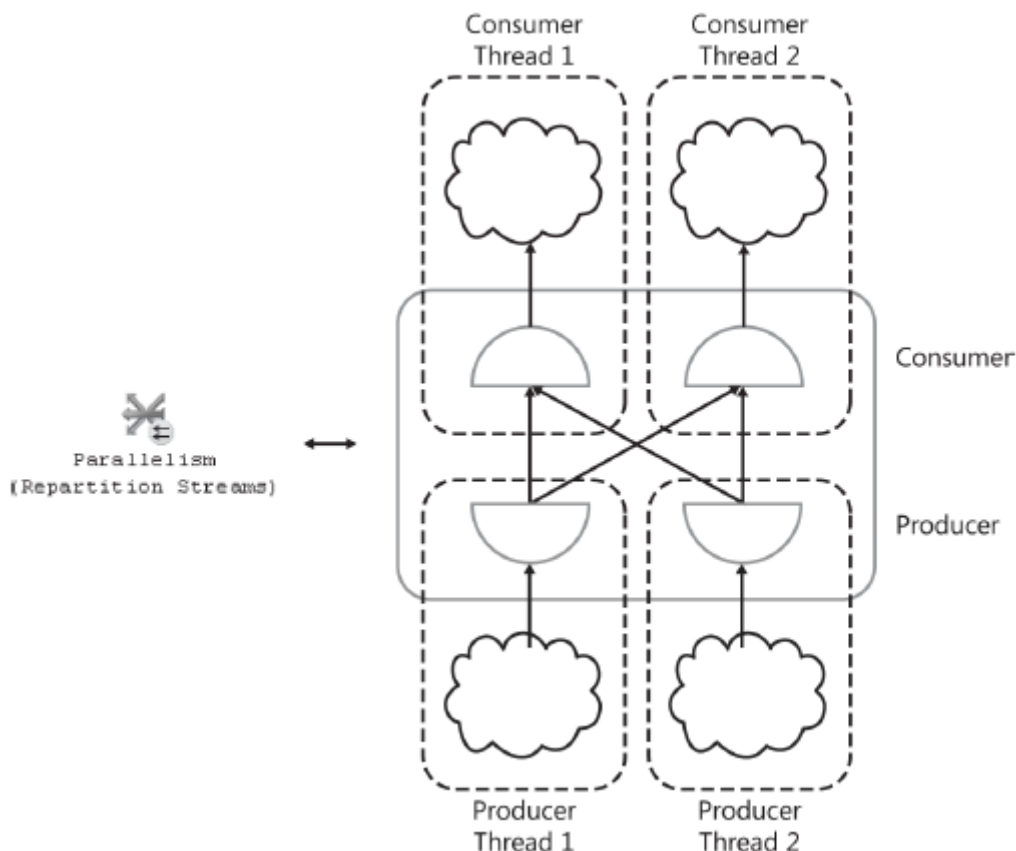
- The max degree of parallelism advanced configuration option must be set to zero (the default) or to more than one.
- The estimated cost to run a serial plan for a query is higher than the value set in cost threshold for parallelism.

#### ◦ Degree of parallelism (DOP)

SQL Server decides the DOP at the start of execution as follows:

1. If the query includes a MAXDOP N query hint, SQL Server sets the maximum DOP for the query to N or to the number of available processors if N is zero
  2. If the query doesn't include a MAXDOP N query hint, SQL Server sets the maximum DOP to the setting of the max degree of parallelism advanced configuration option. As with the MAXDOP N query hint, if this option is set to zero (the default), SQL Server sets the maximum DOP to the number of available processors
  3. SQL Server calculates the maximum number of concurrent threads that it needs to execute the query plan (as shown momentarily, this number can exceed the DOP) and compares this result to the number of available threads. If not enough threads are available, SQL Server reduces the DOP, as necessary. In the extreme case, SQL Server switches the parallel plan back to a serial plan.
- Parallelism operator (also known as exchange)

The exchange iterator is unique in that it's really two iterators: a producer and a consumer. For example, as Figure 10-22 illustrates, a repartition exchange running at DOP2 consists of two producers and two consumers.



Although the data flow between most iterators is pull-based (an iterator calls `GetRow` on its child when it's ready for another row), the data flow in between an exchange producer and consumer is push-based. That is, the producer fills a packet with rows and "pushes" it to the consumer. This model allows the producer and consumer threads to execute independently.

- Parallel scan

The scan operator is one of the few operators that is parallel "aware." The threads that compose a parallel scan work together to scan all rows in a table. Rows or pages aren't assigned beforehand to a particular thread. Instead, the storage engine dynamically hands out pages or ranges of rows to threads.

**At the beginning of a parallel scan, each thread requests a set of pages or a range of rows from the parallel page supplier. The threads then begin processing their assigned pages or rows and begin returning results. When a thread finishes with its assigned set of pages, it requests the next set of pages or the next range of rows from the parallel page supplier.**

Start with a simple example

```
CREATE TABLE [HugeTable1]
(
 [Key] int,
 [Data] int,
 [Pad] char(200),
 CONSTRAINT [PK1] PRIMARY KEY ([Key])
)
SET NOCOUNT ON
DECLARE @i int
BEGIN TRAN
SET @i = 0
WHILE @i < 250000
BEGIN
 INSERT [HugeTable1] VALUES(@i, @i, NULL)
 SET @i = @i + 1
 IF @i % 1000 = 0
 BEGIN
 COMMIT TRAN
 BEGIN TRAN
 END
END
COMMIT TRAN
SELECT [Key], [Data], [Pad]
INTO [HugeTable2]
FROM [HugeTable1]
ALTER TABLE [HugeTable2]
ADD CONSTRAINT [PK2] PRIMARY KEY ([Key])
Now try the simplest possible query:
SELECT [Key], [Data]
FROM [HugeTable1]

/* Despite the large table, this query results in a serial plan: */
|--Clustered Index Scan(OBJECT:([HugeTable1].[PK1]))
```



Now

```
SELECT [Key], [Data]
FROM [HugeTable1]
WHERE [Data] < 1000

/* This query results in a parallel plan: */

|--Parallelism(Gather Streams)
|--Clustered Index Scan(OBJECT:([HugeTable1].[PK1]),
 WHERE:([HugeTable1].[Data]<CONVERT_IMPLICIT(int,[@1],0)))
```

**Because the Data column doesn't have an index, SQL Server can't perform an index seek and must evaluate the predicate for each row. By running this query in parallel, SQL Server distributes the cost of evaluating the predicate across multiple CPUs.**

```
SELECT [Key], [Data]
FROM [HugeTable1]
WHERE [Data] < 1000
ORDER BY [Key]

/* The optimizer recognizes a clustered index that can return rows
sorted by the Key column. To exploit this index and avoid an explicit
sort, the optimizer adds a merging or order-preserving exchange to the
plan: */

|--Parallelism(Gather Streams, ORDER BY:([HugeTable1].[Key] ASC))
|--Clustered Index Scan(OBJECT:([HugeTable1].[PK1]),
 WHERE:([HugeTable1].[Data]<CONVERT_IMPLICIT(int,[@1],0)) ORDERED
 FORWARD)
```

- Load balancing

```
SELECT MIN([Data]) FROM [HugeTable1]
```

This query scans the entire table, but because of the aggregate it uses a parallel plan. The aggregate also ensures that the query returns only one row.

```
--Stream Aggregate(DEFINE:([Expr1003]=MIN([partialagg1004])))
|--Parallelism(Gather Streams)
 |--Stream Aggregate(DEFINE:([partialagg1004]=MIN([HugeTable1].
 [Data])))
 |--Clustered Index Scan(OBJECT:([HugeTable1].[PK1]))
```

Now repeat the experiment, but this time run an expensive serial query at the same time.

```
SELECT MIN(T1.[Key] + T2.[Key])
FROM [HugeTable1] T1 CROSS JOIN [HugeTable2] T2
OPTION (MAXDOP 1)
```

- Parallel nested loops join

SQL Server parallelizes a nested loops join by distributing the outer rows (that is, the rows from the first input) randomly among the nested loops join threads. For example, if two threads are running a nested loops join, SQL Server sends about half of the rows to each thread.

```
SELECT T1.[Key], T1.[Data], T2.[Data]
FROM [HugeTable1] T1 JOIN [HugeTable2] T2
ON T1.[Key] = T2.[Key]
WHERE T1.[Data] < 500

/* Now analyze the STATISTICS PROFILE output for this plan: */
Rows Executes
500 1 |--Parallelism(Gather Streams)
500 2 |--Nested Loops(Inner Join, OUTER REFERENCES:([T1].[Key],
[Expr1004])
WITH UNORDERED PREFETCH)
500 2 |--Clustered Index Scan (OBJECT:([HugeTable1].[PK1] AS [T1]),
WHERE:([T1].[Data]<(500)))
500 500 |--Clustered Index Seek (OBJECT:([HugeTable2].[PK2] AS [T2]),
SEEK:([T2].[Key]=[T1].[Key])
ORDERED FORWARD)
```

- Round-robin exchange

In some cases, SQL Server must add a round-robin exchange to distribute the rows.

```
SELECT T1_Top.[Key], T1_Top.[Data], T2.[Data]
FROM
(
SELECT TOP 100 T1.[Key], T1.[Data]
FROM [HugeTable1] T1
ORDER BY T1.[Data]
) T1_Top,
[HugeTable2] T2
WHERE T1_Top.[Key] = T2.[Key]
```

Here is the corresponding plan:

```
--Parallelism(Gather Streams)
--Nested Loops(Inner Join, OUTER REFERENCES:([T1].[Key],
```

```
[Expr1004]) WITH UNORDERED PREFETCH)
|--Parallelism(Distribute Streams, RoundRobin Partitioning)
| |--Top(TOP EXPRESSION:((100)))
| |--Parallelism(Gather Streams, ORDER BY:([T1].[Data] ASC))
| |--Sort(TOP 100, ORDER BY:([T1].[Data] ASC))
| |--Clustered Index Scan (OBJECT:([HugeTable1].[PK1] AS [T1]))
|--Clustered Index
```

#### [!WARNING]

Search for more info about it!

The top iterator can only be correctly evaluated in a serial plan thread. Thus, SQL Server must add a stop (that is, a distribute streams) exchange above the top iterator and can't use the parallel scan to distribute the rows among the join threads. Instead, SQL Server parallelizes the join by having the stop exchange use round-robin partitioning to distribute the rows among the join threads.

- Parallel nested loops join performance

The parallel scan has one major advantage over the round-robin exchange. A parallel scan automatically and dynamically balances the workload among the threads; a round-robin exchange doesn't.

- Inner-side parallel execution

This chapter noted earlier that with one exception, SQL Server always executes the inner side of a parallel nested loops join as a serial plan. The exception occurs when the optimizer knows that the outer input to the nested loops join is guaranteed to return only a single row and when the join has no correlated parameters.

```
SELECT T1.[Key], T1.[Data], T2.[Key]
FROM [HugeTable1] T1 JOIN [HugeTable2] T2
ON T1.[Data] = T2.[Data]
WHERE T1.[Key] = 0
```

- Parallel merge join

SQL Server parallelizes merge joins by distributing both sets of input rows among the individual merge join threads using hash partitioning. Unlike the parallel plan examples so far, merge join also requires that its input rows be sorted. In a parallel merge join plan, as in a serial merge join plan, SQL Server can use an index scan to deliver rows to the merge join in the correct sort order. However, in a parallel plan, SQL Server must also use a merging exchange to preserve the order of the input rows. Although, as discussed previously, the optimizer tends to favor plans that don't require a merging exchange; including an ORDER BY clause in a join query can encourage such a plan. Consider the following query:

```
SELECT T1.[Key], T1.[Data], T2.[Key]
FROM [HugeTable1] T1 JOIN [HugeTable2] T2
ON T1.[Key] = T2.[Data]
ORDER BY T1.[Key]
```

- Parallel hash join

SQL Server uses one of two different strategies to parallelize a hash join. One strategy uses hash partitioning just like a parallel merge join; the other strategy uses broadcast partitioning and is often called a broadcast hash join.

- Hash partitioning

The more common strategy for parallelizing a hash join involves distributing the build rows and the probe rows among the individual hash join threads using hash partitioning. After the data is hash partitioned among the threads, the hash join instances all run completely independently on their respective data sets. Unlike merge joins, hash joins don't require that input rows be delivered in any particular order and, as a result, don't require merging exchanges.

```
SELECT T1.[Key], T1.[Data], T2.[Key]
FROM [HugeTable1] T1
JOIN [HugeTable2] T2
ON T1.[Data] = T2.[Data]
```

- Broadcast partitioning

Consider what happens if SQL Server tries to parallelize a hash join using hash partitioning, but only a small number of rows exist on the build side of the hash join. Fewer rows than hash join threads means that some threads might receive no rows at all. In this case, those threads would have no work to do during the probe phase of the join and would remain idle. To reduce the risk of skew problems, when the optimizer estimates that the number of build rows is relatively small, it might choose to broadcast these rows to all the hash join threads.

```
SELECT T1.[Key], T1.[Data], T2.[Key]
FROM [HugeTable1] T1 JOIN [HugeTable2] T2
ON T1.[Data] = T2.[Data]
WHERE T1.[Key] < 100
```

Although broadcast hash joins do reduce the risk of skew problems, they aren't suitable for all scenarios. In particular, broadcast hash joins use more memory than their hash-partitioned counterparts.

- Bitmap filtering

```
SELECT T1.[Key], T1.[Data], T2.[Key]
FROM [HugeTable1] T1 JOIN [HugeTable2] T2
ON T1.[Data] = T2.[Data]
WHERE T1.[Key] < 10000
```

The predicate `T1.[Key] < 10000` eliminates 96 percent of the build rows from `HugeTable1`. It also indirectly eliminates 96 percent of the rows from `HugeTable2` because they no longer join with rows from `HugeTable1`.

```
--Parallelism(Gather Streams)
--Hash Match(Inner Join, HASH:([T1].[Data])=([T2].[Data]), RESIDUAL:
(...))
--Bitmap(HASH:([T1].[Data]), DEFINE:([Bitmap1004]))
| |--Parallelism(Repartition Streams, Hash Partitioning,
PARTITION COLUMNS:([T1].[Data]))
| |--Clustered Index Seek(OBJECT:([HugeTable1].[PK1] AS [T1]),
SEEK:([T1].[Key] < (10000)) ORDERED FORWARD)
--Parallelism(Repartition Streams, Hash Partitioning,
PARTITION COLUMNS:([T2].[Data]))
--Clustered Index Scan(OBJECT:([HugeTable2].[PK2] AS [T2]),
WHERE:(PROBE([Bitmap1004],[Northwind2].[dbo].[HugeTable2].[Data] as
[T2].
[Data],N'[IN ROW]'))))
```

As its name suggests, the bitmap operator builds a bitmap. Just like the hash join, the bitmap operator hashes each row of `HugeTable1` on the join key and sets the corresponding bit in the bitmap. When the scan of `HugeTable1` and the hash join build are complete, SQL Server transfers the bitmap to the probe side of the join, typically to the exchange operator.

- Inserts, updates, and deletes

Data modification plans consist of two sections: a “read cursor” and a “write cursor.” The read cursor determines which rows will be affected by the data modification statement. The write cursor executes the actual INSERTS, UPDATES, DELETES, or MERGES. Consider the following UPDATE statement. This statement is guaranteed to violate the foreign key constraint on the `ShipVia` column and will fail.

```
UPDATE [Orders]
SET [ShipVia] = 4
WHERE [ShipCity] = N'London'

/* This statement uses the following query plan: */

--Assert(WHERE:(CASE WHEN [Expr1023] IS NULL THEN (0) ELSE NULL END))
--Nested Loops(Left Semi Join, OUTER REFERENCES:([Orders].[ShipVia]),
DEFINE:([Expr1023] = [PROBE VALUE]))
--Clustered Index Update(OBJECT:([Orders].[PK_Orders]),
```

```

OBJECT:([Orders].[ShippersOrders]),
SET:([Orders].[ShipVia] = [Expr1019]))
| |--Compute Scalar(DEFINE:([Expr1021]=[Expr1021]))
| |--Compute Scalar(DEFINE:([Expr1021]=CASE WHEN [Expr1003]
THEN (1)
ELSE (0) END))
| |--Compute Scalar(DEFINE:([Expr1019]=(4)))
| |--Compute Scalar(DEFINE:([Expr1003]=
CASE WHEN [Orders].[ShipVia] = (4)
THEN (1) ELSE (0) END))
| |--Top(ROWCOUNT est 0)
| |--Clustered Index Scan
(OBJECT:([Orders].[PK_Orders]),
WHERE:([Orders].[ShipCity]=N'London') ORDERED)
|--Clustered Index Seek(OBJECT:([Shippers].[PK_Shippers]),
SEEK:([Shippers].[ShipperID]=[Orders].[ShipVia]) ORDERED FORWARD)

```

The read cursor for this plan consists solely of the clustered index scan of the Orders table, whereas the write cursor consists of the entire remainder of the plan.

- Understanding data warehouse

A data warehouse is a decision support system for business decision making, designed to execute queries from users as well as reporting and analytical applications. Because of these different purposes, both systems also have different workloads: A data warehouse usually must support complex and large queries, compared with the typically small transactions of an OLTP system.

Another main difference between OLTP databases and data warehouses is the degree of normalization found in them. An OLTP system uses normalized data, usually at a third normal form, while a data warehouse uses a denormalized dimensional model.

Dimensional data modeling on data warehouses relies on the use of fact and dimension tables. Fact tables contain facts or numerical measures of the business, which can participate in calculations, whereas dimension tables are the attributes or descriptions of the facts. Fact tables also usually have foreign keys to link them to the primary keys of the dimension tables.

Data warehouses also usually follow star and snowflake schema structures. A star schema contains a fact table and a single table for each dimension. Snowflake schemas are similar to star schemas to the extent that they also have a fact table, but dimension tables can also be normalized, and each dimension can have more than one table. Fact tables are typically huge and can store millions or billions of rows, compared to dimension tables, which are significantly smaller. The size of data warehouse databases tends to range from hundreds of gigabytes to terabytes.

Queries that join a fact table to dimension tables are called star join queries. SQL Server includes special optimizations for star join queries (which we'll look at shortly), can automatically detect star and snowflake schemas, and can reliably identify fact and dimension tables.

Regarding optimizations for star join queries, the use of Cartesian (or cross) products of the dimension tables with multicolumn index lookups on a fact table is interesting to consider.

In "Optimizing Star Join Queries for Data Warehousing in Microsoft SQL Server," defined three different approaches to optimizing star join queries based on the selectivity of the fact table, as shown next. As mentioned previously, selectivity is a measure of the number of records that are estimated to be returned by a query, with smaller numbers represent higher selectivity (fewer rows).

For highly selective queries that return up to 10 percent of the rows in the fact table, the query optimizer can produce a plan with nested loops joins, index seeks and bookmark lookups. For medium selectivity queries, which return anywhere from 10 to 75 percent of the records in the fact table, SQL Server might recommend hash joins with bitmap filters in combination with fact table scans or fact table range scans. Finally, for the least selective queries, processing more than 75 percent of the fact table, the query optimizer mostly will recommend regular hash joins with fact table scans.

Bitmap filtering is an optimization for star join queries that was introduced with SQL Server 2008 and is available only in the Enterprise, Developer, and Evaluation editions.

Optimized bitmap filtering works with hash joins that, as shown earlier, use two inputs, the smaller of which (the build table) is being completely read into memory.

Next is an example of optimized bitmap filtering. Run the following query:

```
USE AdventureWorksDW2012;
GO
SELECT *
FROM dbo.FactInternetSales AS f
JOIN dbo.DimProduct AS p ON f.ProductKey = p.ProductKey
JOIN dbo.DimCustomer AS c ON f.CustomerKey = c.CustomerKey
WHERE p.ListPrice > 50 AND c.Gender = 'M';
```

```
DBCC OPTIMIZER_WHATIF(CPUs, 8)
```

After running the previous statement, the query optimizer produces plans as in a system with eight processors. This command affects only your current session that you can reset to the state it was before by using the ResetAll parameter.

```
DBCC OPTIMIZER_WHATIF(ResetAll)
```

You can also use the Status parameter to see the current configuration; to see the output of this command you also need to run DBCC TRACEON(3604) first. Keep in mind that all these parameters are case sensitive.

```
DBCC OPTIMIZER_WHATIF(Status)
```

- Using columnstore indexes and batch processing

This section shows you the query processing aspect of the technology. Columnstore indexes are complemented with a new vector-based query execution capability with operators that can process batches of rows at a time. Batch processing alone provides performance benefits by reducing the overhead of data movement between operators along with the fact that these new processing algorithms are also optimized for the latest generation of processors. To take benefit of the columnstore indexes technology, you need only to create an index on fact tables and probably also large dimension tables (with more than 10 million rows); changing the queries or anything else on the application isn't needed.

Several existing operators can now run either in row mode or batch mode: hash join, hash aggregate, project, and filter. The same is true for the new columnstore index scan operator. A new operator, batch hash table build, can run in batch mode only.

Finally, a plan might switch from batch to row processing dynamically if the system doesn't have enough memory or threads, and sometimes this could be evidence of a performance problem.

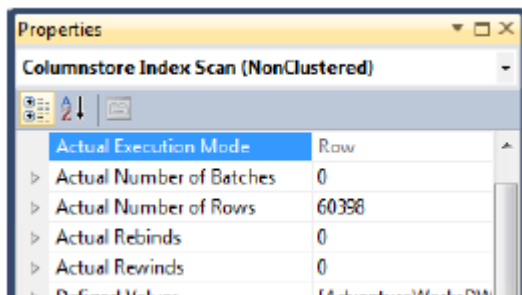
```
USE AdventureWorksDW2012;
GO
CREATE NONCLUSTERED COLUMNSTORE INDEX csi_FactInternetSales
ON dbo.FactInternetSales
(
 ProductKey,
 OrderDateKey,
 DueDateKey,
 ShipDateKey,
 CustomerKey,
 PromotionKey,
 CurrencyKey,
 SalesTerritoryKey,
 SalesOrderNumber,
 SalesOrderLineNumber,
 RevisionNumber,
 OrderQuantity,
 UnitPrice,
 ExtendedAmount,
 UnitPriceDiscountPct,
 DiscountAmount,
 ProductStandardCost,
 TotalProductCost,
 SalesAmount,
 TaxAmt,
 Freight,
 CarrierTrackingNumber,
 CustomerPONumber,
 OrderDate,
 DueDate,
 ShipDate
);
```



Then we can run a typical star join query. The following query is joining the FactInternetSale fact table with the DimDate dimension table, grouping on MonthNumberOfYear and aggregating data on the SalesAmount column to get the total of sales by month for the calendar year 2005

```
SELECT d.MonthNumberOfYear,
SUM(SalesAmount) AS TotalSales
FROM dbo.FactInternetSales AS f
JOIN dbo.DimDate AS d
ON f.OrderDateKey = d.DateKey
WHERE CalendarYear = 2005
GROUP BY d.MonthNumberOfYear;
```

It's really running in the row execution mode:



Properties	
Columnstore Index Scan (NonClustered)	
Actual Execution Mode	Row
Actual Number of Batches	0
Actual Number of Rows	60398
Actual Rebinds	0
Actual Rewinds	0
Defined Values	...

You can drop the columnstore index before continuing.

```
DROP INDEX csi_FactInternetSales ON dbo.FactInternetSales;
```

Now test with a bigger table with millions of records. By the way, if you don't have disk space for millions of records or don't want to wait to create those big tables, you can use a trick with the undocumented ROWCOUNT and PAGECOUNT options of the UPDATE STATISTICS statement. For example, you can run the following statement to create a copy of the FactInternetSales table:

```
SELECT * INTO dbo.FactInternetSalesCopy
FROM dbo.FactInternetSales;
```

Run the following statement:

```
UPDATE STATISTICS FactInternetSalesCopy WITH ROWCOUNT = 10000000, PAGECOUNT
= 1000000;
```

The UPDATE STATISTICS statement will update the page and row count in the catalog views and the query optimizer will use this information to generate a plan according to this data. You can drop it by running the following statement:

```
DROP TABLE dbo.FactInternetSalesCopy;
```

This exercise uses the table and columnstore index created in Chapter 7. The table name is `dbo.FactInternetSalesBig` and has more than 30 million rows. Update your query to use this table as shown here:

```
SELECT d.MonthNumberOfYear,
SUM(SalesAmount) AS TotalSales
FROM dbo.FactInternetSalesBig AS f
JOIN dbo.DimDate AS d
ON f.OrderDateKey = d.DateKey
WHERE CalendarYear = 2005
GROUP BY d.MonthNumberOfYear;
```

Running the star join query again will give you a totally different plan, this time running some operators in parallel and, more important, some operators in the batch mode.

- Adding new data

The most noticeable limitation of columnstore indexes, at least on the SQL Server 2012 release, is that tables containing these indexes are non-updatable. This means no INSERT, UPDATE, DELETE, or MERGE operations are allowed in the table as soon as a columnstore index is created.

Three common workarounds to this limitation, which Microsoft has said will go away in a future release of SQL Server, are to do the following.

- Drop/disable, create/rebuild the columnstore index.
- Use partition switching.
- Use UNION ALL.

- Hints

You can use the new `IGNORE_NONCLUSTERED_COLUMNSTORE_INDEX` hint to ask the query optimizer to avoid using any columnstore index, as shown next:

```
SELECT d.MonthNumberOfYear,
SUM(SalesAmount) AS TotalSales
FROM dbo.FactInternetSalesBig AS f
JOIN dbo.DimDate AS d
ON f.OrderDateKey = d.DateKey
WHERE CalendarYear = 2005
GROUP BY d.MonthNumberOfYear
OPTION (IGNORE_NONCLUSTERED_COLUMNSTORE_INDEX);
```

## 1.5. The Storage Engine

The SQL Server storage engine includes all components involved with the accessing and managing of data in your database. In SQL Server 2012, the storage engine is composed of three main areas: access methods, locking and transaction services, and utility commands.

**When SQL Server needs to locate data, it calls the access methods code**, which sets up and requests scans of data pages and index pages and prepares the OLE DB rowsets to return to the relational engine. Similarly, when data is to be inserted, the access methods code can receive an OLE DB rowset from the client. **The access methods code contains components to open a table, retrieve qualified data, and update data. It doesn't actually retrieve the pages; instead, it makes the request to the buffer manager, which ultimately serves up the page in its cache or reads it to cache from disk.** When the scan starts, a look-ahead mechanism qualifies the rows or index entries on a page. The retrieving of rows that meet specified criteria is known as a qualified retrieval. The access methods code is used not only for SELECT statements but also for qualified UPDATE and DELETE statements (for example, UPDATE with a WHERE clause) and for any data modification operations that need to modify index entries. The following sections discuss some types of access methods.

### 1.5.1. Access Method

To be Complete

### 1.5.2. Buffer Manager

To be Complete

### 1.5.3. Transaction Manager

A core feature of SQL Server is its ability to ensure that transactions are atomic—that is, all or nothing. Also, transactions must be durable, which means that if a transaction has been committed, it must be recoverable by SQL Server no matter what. **Write-ahead logging makes possible the ability to always roll back work in progress or roll forward committed work that hasn't yet been applied to the data pages. Write-ahead logging ensures that the record of each transaction's changes is captured on disk in the transaction log before a transaction is acknowledged as committed, and that the log records are always written to disk before the data pages where the changes were actually made are written. Writes to the transaction log are always synchronous—that is, SQL Server must wait for them to complete. Writes to the data pages can be asynchronous because all the effects can be reconstructed from the log if necessary. The transaction management component coordinates logging, recovery, and buffer management,**

RELEER ESTO PARA QUE ME QUDE 100% CLARO

The transaction management component delineates the boundaries of statements that must be grouped to form an operation. It handles transactions that cross databases within the same SQL Server instance and allows nested transaction sequences. For a distributed transaction to another SQL Server instance the transaction management component coordinates with the Microsoft Distributed Transaction Coordinator (MS DTC) service, using operating system remote procedure calls. The transaction management component marks save points that you designate within a transaction at which work can be partially rolled back or undone.

**The transaction management component also coordinates with the locking code regarding when locks can be released, based on the isolation level in effect. It also coordinates with the versioning code to**

**determine when old versions are no longer needed and can be removed from the version store.**

## 1.6. Plan Caching and Recompilation

Because query optimization can be complex and time-consuming, SQL Server frequently and beneficially reuses plans that have already been generated and saved in the plan cache, rather than produce a completely new plan. However, in some cases, using a previously created plan might not be ideal for the current query execution, and creating a new plan might result in better performance.

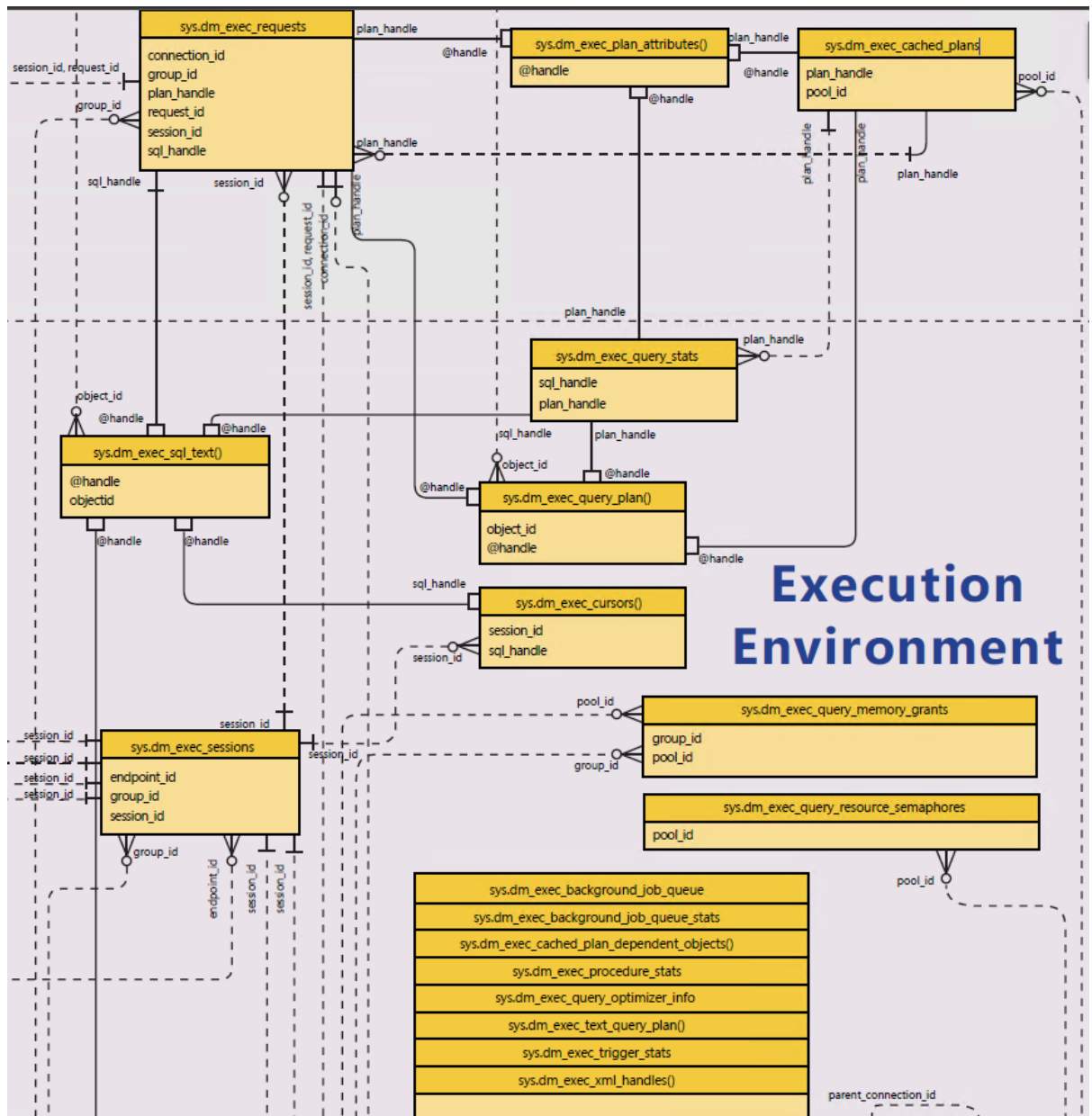
- The plan cache

**Understanding that the plan cache in SQL Server isn't actually a separate area of memory is important.**

- Plan Cache metadata

The view is **sys.dm\_exec\_cached\_plans**, which contains one row for each plan in cache, and we look at the columns **usecounts**, **cacheobjtype**, and **objtype** (the value in **usecounts** allow you to see how many times a plan has been reused). Also, the value in the column **plan\_handle** is used with the table value function **sys.dm\_exec\_sql\_text**

```
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
AND [text] NOT LIKE '%dm_exec_cached_plans%';
```



### ◦ Clearing Plan Cache

You can use any of the following commands.

1. **DBCC FREEPROCCACHE** This command removes all cached plans from memory. To remove a specific plan from cache, all plans with the same sql\_handle value, or all plans in a specific resource governor resource pool.
2. **DBCC FREESYSTEMCACHE** This command clears out all SQL Server memory caches, in addition to the plan caches.
3. **DBCC FLUSHPROCINDB ( )** This command allows you to specify a particular database ID, and then clears all plans from that particular database.

### • Caching Mechanisms

SQL Server can avoid compilations of previously executed queries by using four mechanisms to make plan caching accessible in a wide set of situations.

1. Ad hoc query caching

2. Autoparameterization
3. Prepared queries, using either `sp_executesql` or the `prepare` and `execute` method invoked through your API
4. Stored procedures or other compiled objects (triggers, TVFs, etc.)

To determine which mechanism is being used for each plan in cache, you should look at the values in the **cacheobjtype** and **objtype** columns in the **sys.dm\_exec\_cached\_plans** view.

The **cacheobjtype** column can have one of six possible values.

1. Compiled Plan
2. Compiled Plan Stub
3. Parse Tree
4. Extended Proc
5. CLR Compiled Func
6. CLR Compiled Proc

This section looks at only **Compiled Plan** and **Compiled Plan Stub**. Notice that the `usecount` query limits the results to row having one of these two values.

The **objtype** column has 11 different possible values:

1. Proc (stored procedure)
2. Prepared (prepared statement)
3. Adhoc (ad hoc query)
4. ReplProc (replication filter procedure)
5. Trigger
6. View
7. Default (default constraint or default object)
8. UsrTab (user table)
9. SysTab (system table)
10. Check (CHECK constraint)
11. Rule (rule object)
12. Ad hoc query caching

If the caching metadata indicates a **cacheobjtype** value of **Compiled Plan** and an **objtype** value of **Adhoc**, the plan is considered to be an **ad hoc plan**. When SQL Server caches the plan from an ad hoc query, the cached plan is reused only if a subsequent batch matches exactly.

```
/* 1 */
SELECT * FROM Person.Person WHERE LastName = 'Raheem';
SELECT * FROM Person.Person WHERE LastName = 'Garcia';
```

```

SELECT * FROM Person.Person WHERE LastName = 'Raheem';

/* 2 */
USE AdventureWorks2022

DBCC FREEPROCCACHE;
GO

SELECT * FROM Person.Person WHERE LastName = 'Raheem';
GO
SELECT * FROM Person.Person WHERE LastName = 'Garcia';
GO
SELECT * FROM Person.Person WHERE LastName = 'Raheem';
GO

SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
 AND [text] NOT LIKE '%dm_exec_cached_plans%';

/* 3 */
USE AdventureWorks2022;

DBCC FREEPROCCACHE;
GO

SELECT * FROM Person.Person WHERE LastName = 'Raheem';
GO
-- Try it again
SELECT * FROM Person.Person WHERE LastName = 'Raheem';
GO
SELECT * FROM Person.Person
WHERE LastName = 'Raheem';
GO
SELECT * FROM Person.Person WHERE lastname = 'Raheem';
GO
select * from Person.Person where LastName = 'Raheem';
GO

SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
AND [text] NOT LIKE '%dm_exec_cached_plans%';

```

A few special kinds of statements are always considered to be ad hoc.

A. A statement used with **EXEC**, as in:

```

EXEC('SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID =
6')

```

B. A statement submitted using **sp\_executesql**, if no parameters are supplied.

Queries that you submit via your application with `sp_prepare` and `sp_prepexec` aren't considered to be ad hoc.

### 1.1 Optimizing for ad hoc workloads

If most of your queries are ad hoc and never reused, caching their execution plans might seem like a waste of memory.

you can enable in those cases where you expect most of your queries to be ad hoc. When this option is enabled, SQL Server caches only a stub of the query plan the first time any ad hoc query is compiled, and only after a second compilation is the stub replaced with the full plan.

### 1.2 Controlling the optimize for ad hoc workloads setting

Enabling the Optimize for Ad Hoc Workloads option is very straightforward, as shown in the following code:

```
``sql
EXEC sp_configure 'optimize for ad hoc workloads', 1;
RECONFIGURE;
``
```

### 1.3 The compiled plan stub

The stub that SQL Server caches when Optimize for Ad Hoc Workloads is enabled is only about 300 bytes in size and doesn't contain any part of a query execution plan.

The usecounts value in the cache metadata is always 1 for compiled plan stubs because they are never reused.

When a query or batch that generated a compiled plan stub is recompiled, the stub is replaced with the full compiled plan.

```
``sql
EXEC sp_configure 'optimize for ad hoc workloads', 1;
RECONFIGURE;
GO
USE AdventureWorks2012;
DBCC FREEPROCCACHE;
GO
SELECT * FROM Person.Person WHERE LastName = 'Raheem';
GO
SELECT usecounts, cacheobjtype, objtype, size_in_bytes, [text]
FROM sys.dm_exec_cached_plans P
```



```

CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype LIKE 'Compiled Plan%'
AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
SELECT * FROM Person.Person WHERE LastName = 'Raheem';
GO
SELECT usecounts, cacheobjtype, objtype, size_in_bytes, [text]
FROM sys.dm_exec_cached_plans P
CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype LIKE 'Compiled Plan%'
AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
```

```

2. Parametrization

2.1 Simple parameterization

For certain queries, SQL Server can decide to treat one or more of the constants as parameters. When this happens, subsequent queries that follow the same basic template can use the same plan. For example, these two queries that run in the AdventureWorks2022 database can use the same plan:

```

```sql
USE AdventureWorks2022
GO
DBCC FREEPROCCACHE;
GO
SELECT FirstName, LastName, Title FROM Person.Person WHERE BusinessEntityID
= 6;
GO
SELECT FirstName, LastName, Title FROM Person.Person WHERE BusinessEntityID
= 2;
GO
SELECT usecounts, cacheobjtype, objtype, size_in_bytes, [text]
FROM sys.dm_exec_cached_plans P
CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
```

```

![alt text](image/cap12_2.png)

> [!WARNING]
 > On SQL 2022 the result is different.

> [!NOTE]
 > Don't confuse a shell query with a plan stub. A shell query contains the

complete text of the query and uses about 16 KB of memory.

> Shell queries are created only for those plans that SQL Server thinks are parameterizable. A plan stub, as mentioned previously, uses
 > about only 300 bytes of memory and is created only for unparameterizable, ad hoc queries, and only when the Optimize for Ad Hoc
 > Workloads option is set to 1.

By default, SQL Server is very conservative about deciding when to parameterize automatically. SQL Server automatically parameterizes queries only if the query template is considered to be safe. A template is safe if the plan selected doesn't change, even if the actual parameter values change.

- Drawbacks of simple parameterization

SQL Server makes its own decision as to the data type of the parameter. Looking at the Person.Person table, SQL Server chose to assume a parameter of type tinyint. If you rerun the batch and use a value that doesn't fit into the tinyint range SQL Server can't use the same autoparameterized query.

```

```sql
USE AdventureWorks2022
GO
DBCC FREEPROCCACHE;
GO
SELECT FirstName, LastName, Title FROM Person.Person WHERE
BusinessEntityID = 6;
GO
SELECT FirstName, LastName, Title FROM Person.Person WHERE
BusinessEntityID = 622;
GO

SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
```

```

****The only way to force SQL Server to use the same data type for both queries is to enable PARAMETERIZATION FORCED for the database****

```

```sql
USE AdventureWorks2022;
GO

ALTER DATABASE AdventureWorks2012 SET PARAMETERIZATION FORCED;
GO

```

```

SET STATISTICS IO ON;
GO

DBCC FREEPROCCACHE;
GO

SELECT * FROM Person.Person WHERE PersonType = 'EM';
GO
SELECT * FROM Person.Person WHERE PersonType = 'IN';
GO

SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO

ALTER DATABASE AdventureWorks2022 SET PARAMETERIZATION SIMPLE;
GO
```

```

When you run this code, you see that the first SELECT required 847 logical reads and the second required 56,642.

In this example, forcing SQL Server to treat the constant as a parameter isn't a good thing.

So what can you do if you have many queries that shouldn't be parameterized and many others that should be?

The SQL Server Performance Monitor includes an object called SQLServer:SQL Statistics that has several counters dealing with automatic parameterization. You can monitor these counters to determine whether many unsafe or failed automatic parameterization attempts have been made. If these numbers are high, you can inspect your applications for situations in which the application developers can take responsibility for explicitly marking the parameters.

2.2 Forced parametrization

If your application uses many similar queries that you know could benefit from the same plan but aren't autoparameterized, either because SQL Server doesn't consider the plans safe or because they use one of the disallowed constructs, SQL Server provides an alternative. A database option called ****PARAMETERIZATION FORCED**** can be enabled with the following command:

```

```sql
ALTER DATABASE <database_name> SET PARAMETERIZATION FORCED;
```

```

Be careful when setting this option on for the entire database because assuming that all constants should be treated as parameters during optimization and then reusing existing plans frequently can lead to very poor performance.

3. Prepared queries

1.7. Transactions and Concurrency

Summary

[!NOTE]

PARSER Summary What flow-of-control is?

OPTIMIZER Phases of OTIMIZER The Query Optimizer . Takes the query tree and prepares it for optimization. . DML can't be optimized and are compiled internally. . DML are optimized (SELECT, INSERT, UPDATE, DELETE, and MERGE) . The Query optimizar create an Execution Plan . The first step in producing such a plan is to normalize each query . The second step is to optimizes it, which means that it determines a plan for executing that query. Query optimization is cost-based, the Query Optimizer chooses the plan that it determines would cost the least based on internal metrics that include estimated memory requirements, CPU utilization, and number of required I/Os. . The Query Optimizer considers the type of statement requested, checks the amount of data in the various tables affected, looks at the indexes available for each table, and then looks at a sampling of the data values kept for each index or column referenced in the query. . Finally Based on the available information, the Query Optimizer considers the various access methods and processing strategies that it could use to resolve a query and chooses the most cost-effective plan. Summary QO functions: First step, name parsed, make a parsed of the query to tree representation. Second step, name binding, make some validations like columns and tables exists, the user can see them, etc Third step, start evaluating different query plans. Finding an optimal query plan is actually a much more difficult algorithmic problem for SQL Server. The storage of all possible query plans also becomes a problem. The Query Optimizer solves this problem using **heuristics** and **statistics** to guide those heuristics.

The QO uses a framework to search and compare many different possible plan alternative efficiently. Here you have: . The Rule . The Memo . The Properties . The Operators . The Apply . The Spools . The Exchange Fourd step, send the plan to te QE

1. Simplification The tree is normalized in the Simplification phase to convert the tree from a form linked closely to the user syntax into one that helps later processing
2. Trivial plan / auto-parameterization **TRIVIAL** >> The query is easy so SQL create only one obviously plan. If this happens the SQL also parameterize this query to be execute later of other similar query comes. **FULL** >> Means that the optimization process was performed. (query features that cause a query not to be considered trivial include Distributed Query, Bulk Insert, XPath queries, queries with joins or subqueries, queries with hints, some cursor queries, and queries over tables containing filtered indexes.) **forced parameterization**, to auto-parameterize queries more aggressively. This feature parameterizes all constants, ignoring cost-based considerations. The benefit of this feature is that it can reduce compilations, compilation time, and the number of plans in the procedure cache. All these things can improve system performance. On the other hand, this feature can reduce performance when different parameter

values would cause different plans to be selected. **The core structure of the QO is the Memo.** An optimization search pass is split into two parts. In the first part of the search, exploration rules match logical trees and generate new, equivalent alternative logical trees that are inserted into the Memo. Implementation rules run next, generating physical trees from the logical trees. After a physical tree is generated, it's evaluated by the costing component to determine the cost for this query tree. The resulting cost is stored in the Memo for that alternative. When all physical alternatives and their costs are generated for all groups in the Memo, the Query Optimizer finds the one query tree in the Memo that has the lowest cost and then copies that into a standalone tree. The selected physical tree is very close to the showplan form of the tree. This process can be repeated for the SQL if the plan selected is not good enough (Stage 0 / Stage 1 / Stage 2 ... etc).

Statistics The QO uses them to create the Plan. First check if the statistics for the columns exist; if they exist, use them; if not, create the stats. Auto-create and Auto-update statistics features are enabled by default. Although these settings are left enabled, some reasons for disabling the creation or update behavior of statistics include the following:

- The table is very large, and the time to update the statistics automatically is too high.
- The table has many unique values, and the sample rate used to generate statistics isn't high enough to capture all the statistical information needed to generate a good query plan.
- The DB app has a short query timeout defined and doesn't want automatic statistics to cause a query to require noticeably more time than average to compile because it could cause that timeout to abort the query. **For small tables, all pages are sampled. For large tables a smaller percentage of pages are sampled. So that the histogram remains a reasonable size, it's limited to 200 total steps.**

Density Information In addition to a histogram, the QO keeps track of the number of unique values for a set in columns. This information, called the **DENSITY INFORMATION**, is stored in the statistics objects. Density is calculated by the formula **1/frequency**, with frequency being the average number of duplicates for each value in a table. For multicolumn statistics, the statistics object stores density information for each combination of columns in the statistics object.

- **Assumption**
- **Independent:** When the QP tries to estimate the selectivity for each condition in an AND clause, it usually assumes that each condition is independent.
- **Uniformity :** This means that if a range of values is being considered but the values aren't known, they are assumed to be uniformly distributed over the range in which they exist.
- **Containment:** This says that if a range of values is being joined with another range of values, the default assumption is that the query is being asked because those ranges overlap and qualify rows. Without this assumption, many common queries would be underestimated and poor plans would be chosen.
- **String Statistics or trie trees:** SQL Server histograms can have up to 200 steps, or unique values, to store information about the overall distribution of a table. Although this works well for many numeric types, the string data types often have many more unique values. Trie trees were created to store efficiently a sample of the strings in a column.

Cardinality Estimation

- Selectivity, is the fraction of rows that are expected to be qualified by the predicate and then returned to the user. `SELECT col1, col2 FROM Table3 WHERE col3 < 10;` The distribu. on col3 is uniformly distributed from 0 to 49 (amount of steps in histogram), and 10/50 values are less than 10 (WHERE clause), or 20 percent (10/50) of the rows. Therefore, **selectivity** of this filter in the query is **0.2**, and the calculation of the number of rows resulting from the filter is as follows: (# rows in operator below) * (selectivity of this operator) $10000 * 0.2 = 2000$ rows
- Cardinality, **buscar bien esta definicion!!!!!!!**
- Density,
- Costing, is the process of determining how much time each potential plan choice will require to run, and it's done separately for each physical plan considered. The idea behind costing is actually quite simple. Using the cardinality estimates and some additional information about the average and maximum width of each column in a table, it can determine how many rows fit on each database page. This value is then translated into a number of disk reads that a query requires to complete. The total costs for each operator are then added to determine the total query cost, and the QO can select the fastest (lowest-cost) query plan from the set of considered plans during optimization.

Index Selection

- The basic idea behind index matching is to take predicates from a WHERE clause, JOIN condition, or other operation in a query and to convert that operation so that it can be performed against an index. Two basic operations can be performed against an index: **Seek** for a single value or a range of values on the index key and **Scan** the index forward or backward.
- The job of the QO is to figure out which predicates can be applied to the index to return rows as quickly as possible. Some predicates can be applied to an index whereas others can't.
 - Predicates that can be converted into an index operation are often called **sargable, or "search-ARGument-able."**
 - Predicates that can never match or don't match the selected index are called **non-sargable** predicates.

Filter Indexes

- Nevertheless, this feature exists for good reasons.
 - Indexed views are more expensive to use and maintain than filtered indexes.
 - The matching capability of the Indexed View feature isn't supported in all editions of SQL Server.
- When use Filter Index
 - If you are querying a table with a small number of distinct values and are using a multicolumn predicate in which some of the elements are fixed, you can create a FI to speed up this specific query.
 - The index can be used when an expensive query on a large table has a known query condition

Indexed Views

- NONindexed views have been used for goals such as simplifying SQL queries, abstracting data models from user models, and enforcing user security. From an optimization perspective, SQL

Server doesn't do much with these views because they are expanded, or in-lined, before optimization begins. [!IMPORTANT] **What "expanded or in-lined" means on this case**

- INDEXview create a materialized form of the query result. The resulting structure is a physically table with a Clustered Index (NCI are also supported) [!IMPORTANT] **What "materialized" means on this case. Means physical??**

Partitioned tables

- When comparing this to their nonpartitioned equivalents, the difference in the plan is often that the partitioned case requires iterating over a list of tables or a list of indexes to return all the rows.

Windowing functions Nothing

Data Warehousing

- Fact table on WH that nonclustered indexes are not an option
- SQL orders JOIN differently TO TRY TO PERFORM AS MANY LIMITING OPERATIONS AS CAN
- SQL 2012 introduce **Column Store** and **Batch Mode**. These two new feature on SQL 2012 improve significantly the queries start join pattern on DH.
- Columnstore indexes
 - Columnstores in SQL Server 2012 are nonclustered indexes that use less space than a traditional B-tree index in SQL Server. From the perspective of the QO, having a significantly smaller index reduces the I/O cost to read the fact table and process the query. **Search how the Column Store saved space?**
 - Columnstore limitations
 - One limitation is that tables must be marked as read-only as long as the columnstore exists. In other words, you can't perform INSERT, UPDATE, DELETE, or MERGE operations on the table while the columnstore index is active.
 - The other columnstore restrictions in SQL Server 2012 relate to data types. Some of the more complex data types, including varchar(max), nvarchar(max), CLR types, and other types not often found in fact tables are restricted from using columnstore indexes.
- Batch mode processing
 - This execution model improves CPU performance in multiple ways. First, it reduces the number of CPU instructions needed to process each row. Second, implements techniques that reduce the number of blocking memory references required to execute a query.
 - On this mode data is processed in groups of rows instead of one row at a time.
 - Like columnstore index, data within batches is allocated by column instead of by row. This allocation model allows some operations to be performed more quickly.
 - The third major difference in the batch model is that data is stored within memory using a probabilistic representation to further reduce the number of times the CPU core needs to access memory that isn't already in the CPU's internal caches.
- Logical database design best practices **Search these later**
- Plan shape **Find a better image**

Update

- Update include UPDATE, INSERT, DELETE and MERGE

- Update optimization also considers physical optimizations such as
- How many indexes need to be touched for each row
- Whether to process the updates one index at a time or all at once
- How to avoid unnecessary deadlocks while processing changes as quickly as possible.
- Every update query in SQL Server is composed of the same basic operations:
- It determines what rows are changed (inserted, updated, deleted, merged).
- It calculates the new values for any changed columns.
- It applies the change to the table and any nonclustered index structures.
- Halloween Protection
- This is a feature of relational databases that's used to provide correctness in update plans. If you do an UPDATE maybe the row move forward and you can read the same record twice. The typical protection against this problem is to scan all the rows into a buffer, and then process the rows from the buffer. In SQL Server, implemented a spool or a Sort operator.
- Split/Sort/Collpase I understood this. **But I'd like to search an explanation on chatgpt**
- Merge I understood this. **I HAVE TO PRACTISE MORE**
- Per-Index or wide update plans I understood this. **I HAVE TO PRACTISE MORE**
- Non-updating updates I understood this. **I HAVE TO PRACTISE MORE**
- Sparse column updates SQL Server provides a feature called sparse columns that supports creating more columns in a table than were previously supported, as well as creating rows that were technically greater than the size of a database page.
- Partitioned updates Updating partitioned tables is somewhat more complicated than nonpartitioned equivalents. Instead of a single physical table (heap) or B-tree, the query processor has to handle one heap or B-tree per partition.
- Locking **Read the corresponding book**
- Distributed Query Distributed Query is implemented within the Query Optimizer's plan-searching framework. Distributed queries initially are represented by using the same operators as regular queries. Each base table represented in the QO tree contains metadata collected from the remote source. The information collected is very similar to the information that the QP collects for local tables, including column data, index data, and statistics.
- Plan hinting
 - Determining when to use a hint requires an understanding of the workings of the QO and how it might not be making a correct decision, and then an understanding of how the hint might change the plan generation process to address the problem.
 - Here we explain how to identify **cardinality estimation errors** and then use hints to correct poor plan choices.
 - Hints
 - {HASH ! ORDER} GROUP // {MERGE | HASH | CONCAT} UNION // FORCE ORDER, {LOOP | MERGE | HASH} JOIN // FORCESEEK // FAST // MAXDOP // OPTIMIZE FOR // PARAMETERIZATION {SIMPLE ! FORCED} // NOEXPAND // USE PLAN **SEARCH INFO OF ALL OF THEM, WHEN TO USE IT, HOW TO USE IT, ETC**

NEXT STEPS LEER STATISTICS LEER EL LIBRO DE CONCURRENCY, LOCK, ETC