

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.

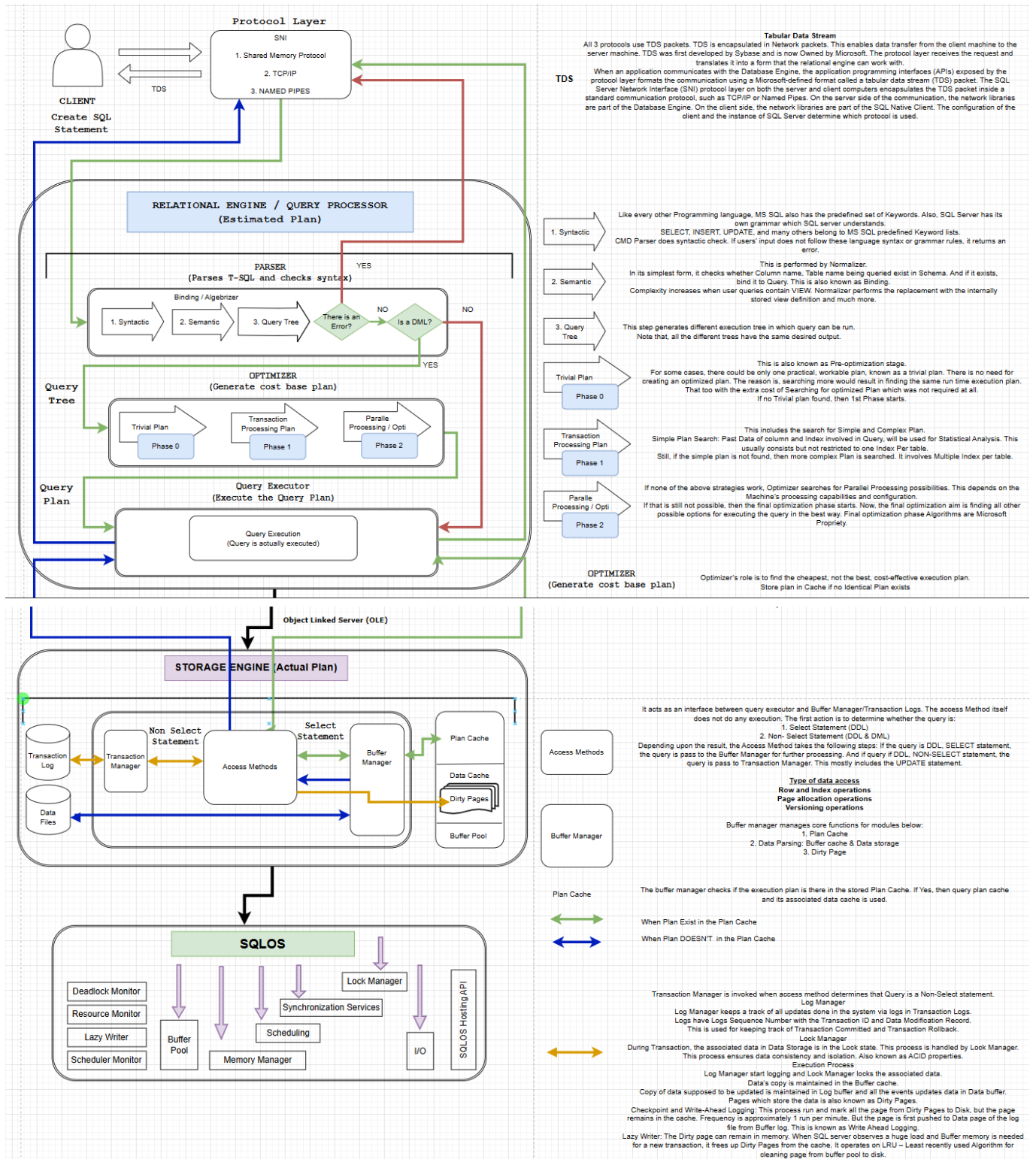
Index

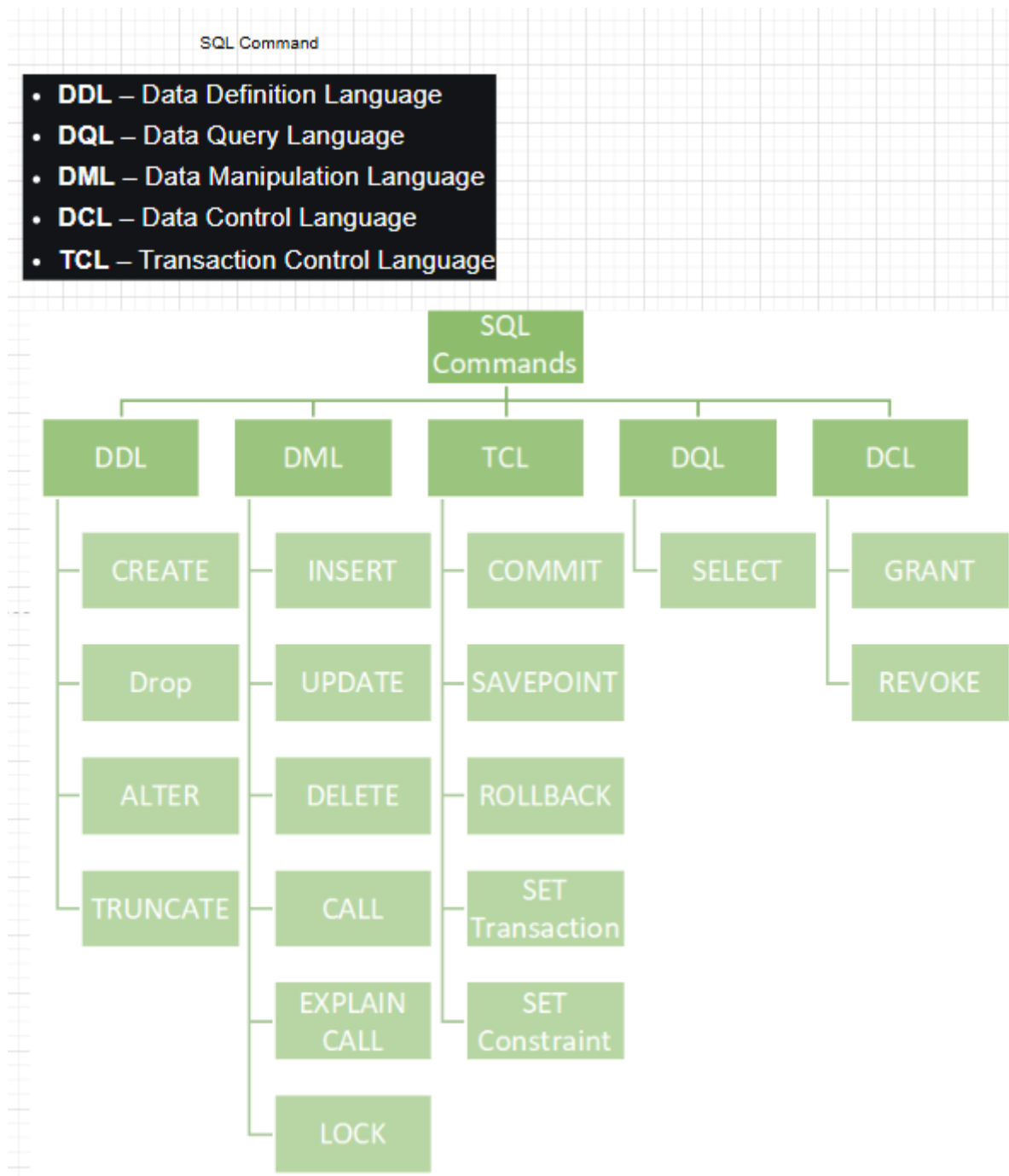
- [SQL Server 2012 architecture and configuration](#)
 - [1. Architecture](#)
 - [1.1. MSSQL Server Physical CLIENT-SERVER Architecture Diagram](#)
 - [1.2. Components of the SQL Server Engine](#)
 - [1.3. The Protocol Layer](#)
 - [1.4. The Query Processor](#)
 - [1.5. The Storage Engine](#)
 - [1.5.1. Access Method](#)
 - [1.5.2. Buffer Manager](#)
 - [1.5.3. Transaction Manager](#)

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.

[!NOTE] Summary 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 optimizer 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.

- Overview

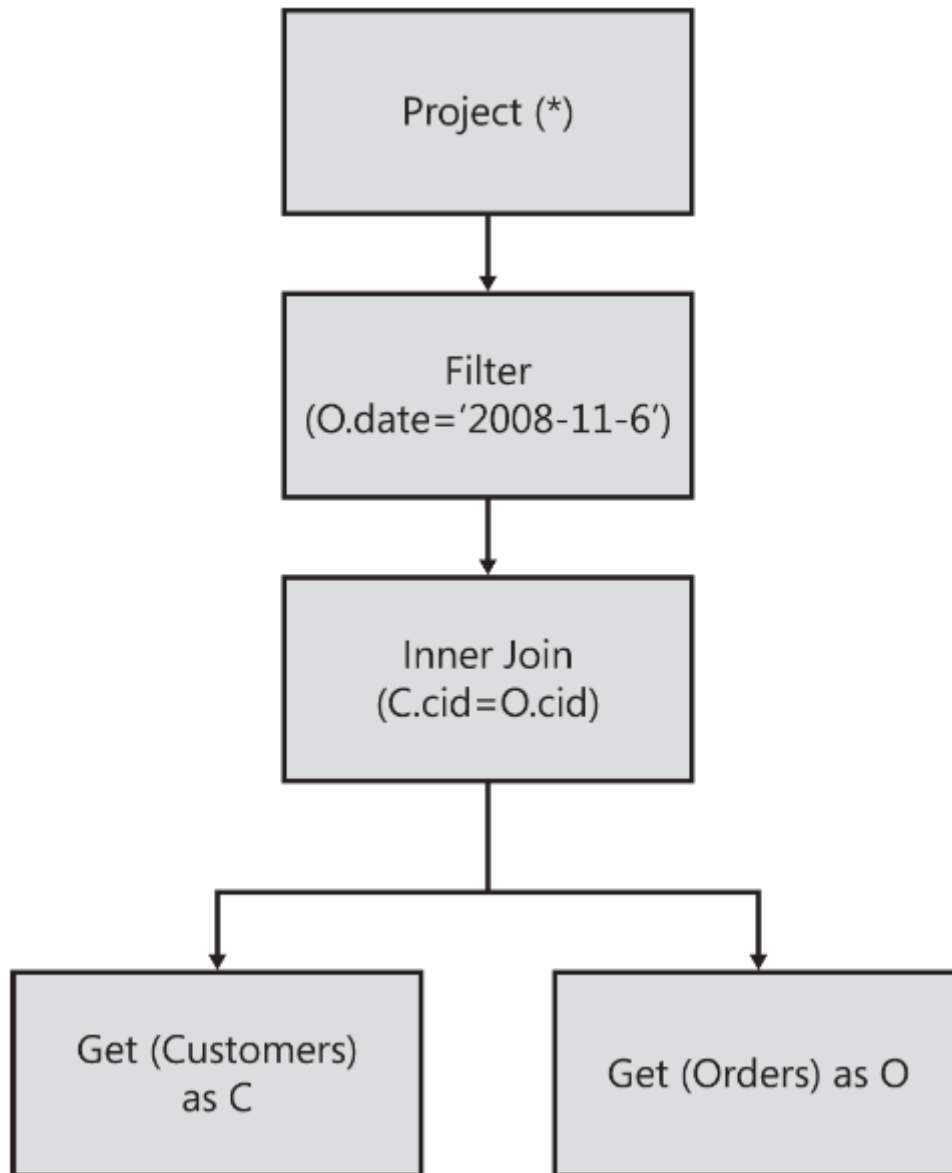
When a query is compiled, the SQL Statement is first parsed into an equivalent tree representation. For queries with valid SQL syntax, the next stage performs a series of validation steps on the query, generally called binding, where the columns and tables in the tree are compared to databases metadata 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 start evaluating different possible query plans. The QO perform 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 spacefic query tree from the current state to a different. In the framework used in SQL Server, the transformations are done via RULES, wich 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.

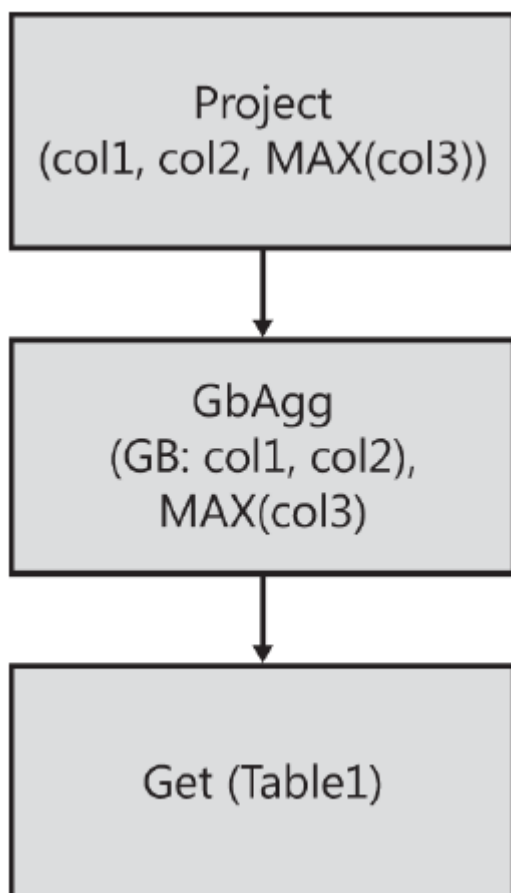
The best of these generated physical alternatives from implementation rules is eventually output by the QO as the final query execution plan.

- 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;
```

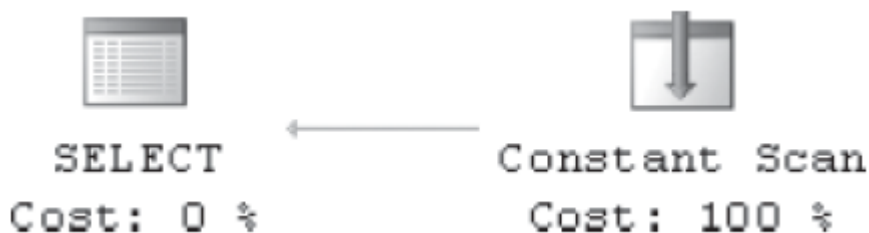


By looking at the final query plan you can see that the QO performs no grouping operation, even though the query uses a GROUP BY. The properties collected during optimization enable this rule to perform a trees transformation to make the resulting query plan complete more quickly.

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:

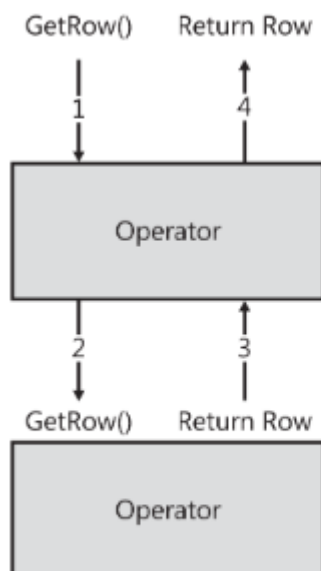


FIGURE 11-6 SQL Server operator data flow model.

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

-- exercice
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 );
```

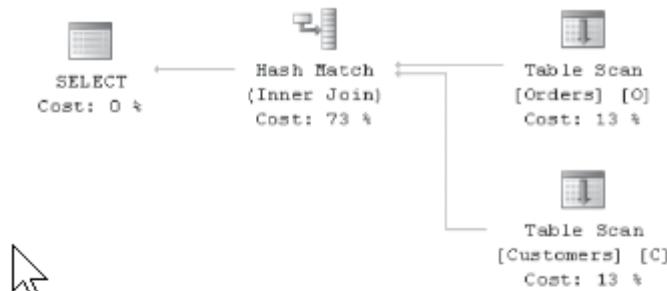


FIGURE 11-7 Query plan using an *INNER JOIN* instead of a subquery (the section commented with "Semantically wrong way to ask the question - returns duplicate names").



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

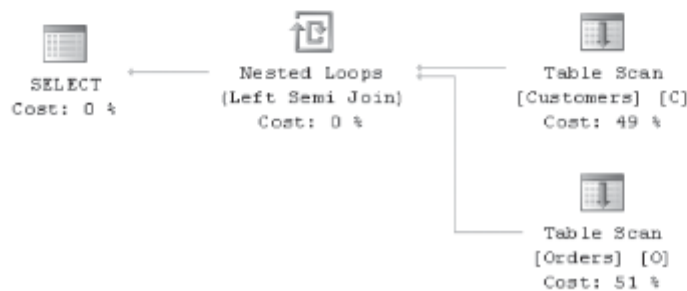


FIGURE 11-9 Query plan using subquery.

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;
```

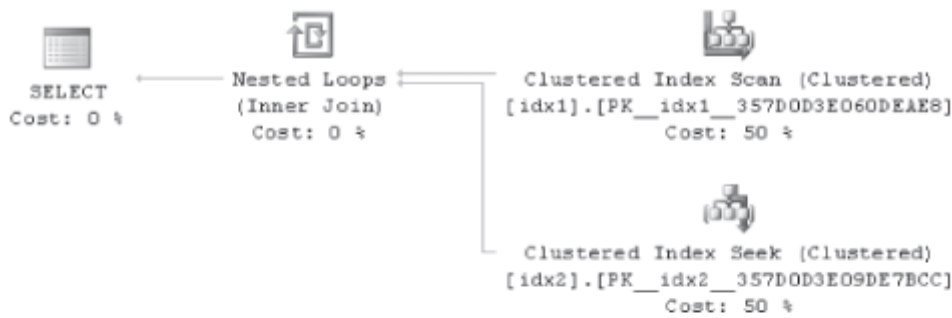
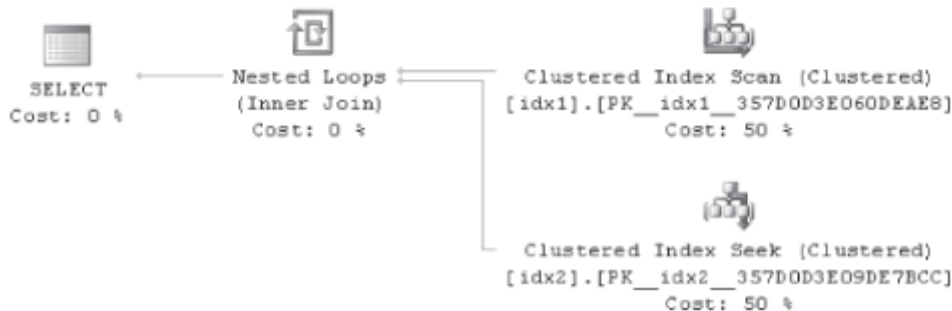


FIGURE 11-10 *APPLY* query plan.

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

```

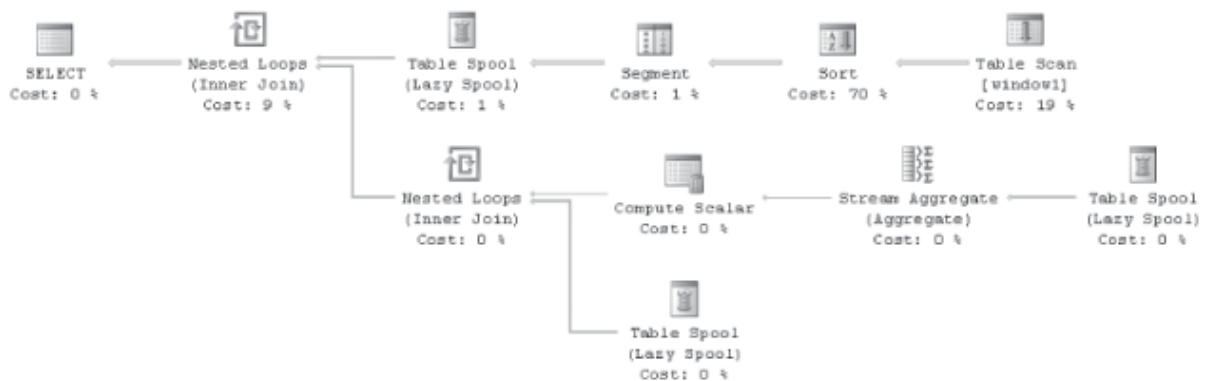


FIGURE 11-12 Query plan containing common subexpression spool.

- 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.

[!TIP] 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 Fourth step, send the plan to te QE

- 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

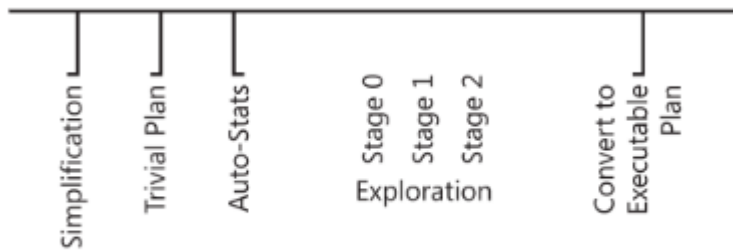


FIGURE 11-14 Query Optimizer pipeline.

- Before optimization The SQL Server query processor 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. For example, the QO detects semantic contradictions in the query and removes them by rewriting the query into a simpler form. 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 uses 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 query processor actually takes this concept one step further. When simple queries are compiled and optimized, the query processor 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 Query Optimizer 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 Query Optimizer 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 memor 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 Query Optimizer to generate plans efficiently for a wide range of workloads. By the end of the search, the Query Optimizer 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 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 sta-tistics update operation to be performed on a background thread in a different transaction context. The benefit to this model is that it avoids the repeating rollback issue. The original query continues and uses out-of-date statistical information to compile the query and return it to be executed.

- Statistics design Statistics are stored in the sustem 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 tatistics on computed column. The following code creates statisitics 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 if columns. this information, called the **DENSITY INFORMATION**, is stored in the statistics objects. Dentisy 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 objetct stores density information for each combination of columns in the statitisc 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 1/0.000 = 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].[MULTIDENS...	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. For example, 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 Taurus 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. For example, 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. For example 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	String 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, as shown in Figure 11-32. 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 tress. 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 Query Optimizer 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 Query Optimizer 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 query processor 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. For example, if you are using an application that shows you pages of rows onscreen 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 INNER JOIN A as A2 ON A1.col1=A2.col1; SELECT
```

```
A1.* FROM A as A1 INNER JOIN A as A2 ON A1.col1=A2.col1 OPTION (FAST
1);
```

- Index selection Index selection is one of the most important aspects of query optimization. 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. Otherwise, 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 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 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.

- 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.
- As shown in Listing 11-7, 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 Query Optimizer 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 table1(id INT PRIMARY KEY, submitdate DATETIME, comment
NVARCHAR(200));
CREATE TABLE table2(id INT PRIMARY KEY IDENTITY, commentid INT, product
NVARCHAR(200));
GO
-- submit some data into each table
INSERT INTO table1(id, submitdate, comment) VALUES (1, '2008-08-21',
'Conor Loves Indexed Views');
INSERT INTO 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
INNER 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
INNER 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
INNER JOIN dbo.table1 t1
ON t2.commentid=t1.id;
```

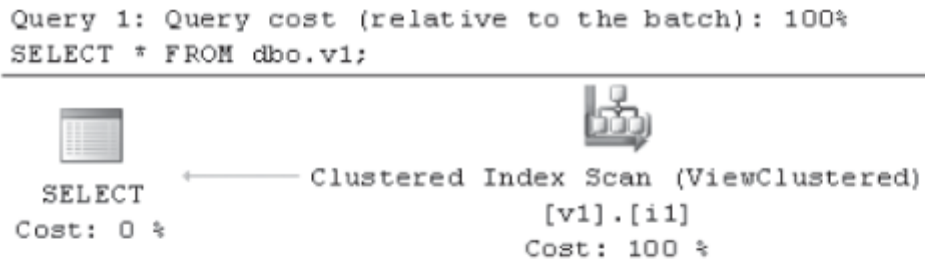


FIGURE 11-38 A direct reference match of an indexed view.

Microsoft SQL Server 2012 Internals

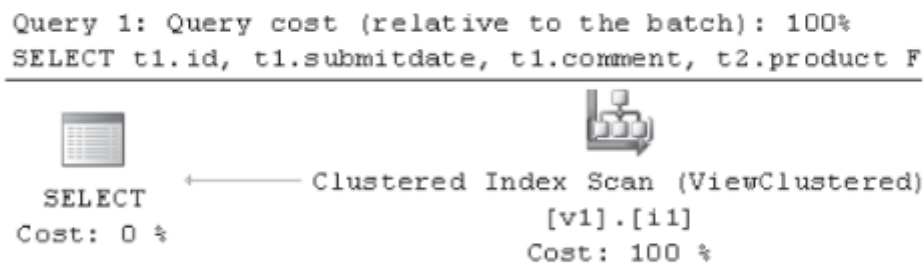
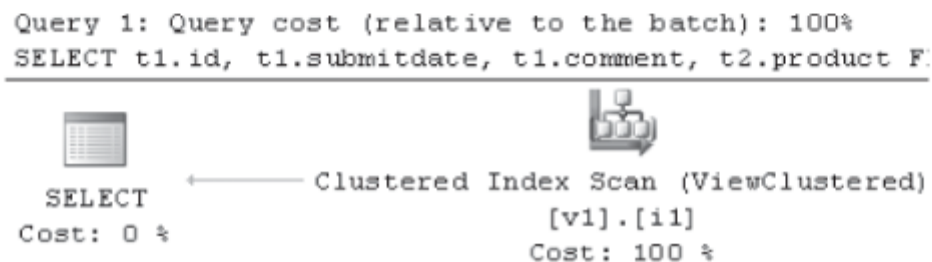


FIGURE 11-39 An indexed view match when the query is a match to the view definition.



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

```
CREATE TABLE table3(col1 INT PRIMARY KEY IDENTITY, col2 INT);
INSERT INTO table3(col2) VALUES (10);
INSERT INTO table3(col2) VALUES (20);
INSERT INTO table3(col2) VALUES (30);
GO


-- create a view that returns values of col2 > 20
CREATE VIEW dbo.v2 WITH SCHEMABINDING AS
SELECT t3.col1, t3.col2
FROM dbo.table3 t3
WHERE t3.col2 > 20;
GO

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

```
-- now 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)
```

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

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



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.

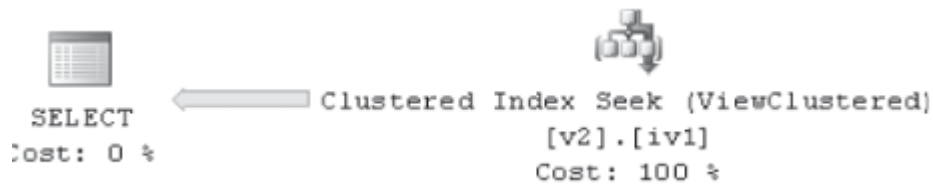
```
-- base table
CREATE TABLE basetbl1 (col1 INT, col2 INT, col3 BINARY(4000));
CREATE UNIQUE CLUSTERED INDEX i1 ON basetbl1(col1);
GO

-- 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

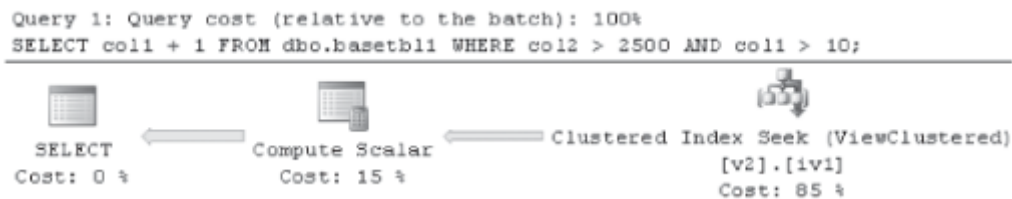
-- create a view over the 2 integer columns
CREATE VIEW dbo.v2 WITH SCHEMABINDING AS
SELECT col1, col2 FROM dbo.basetbl1;
GO

-- index that on col2 (base table is only indexed on col1)
CREATE UNIQUE CLUSTERED INDEX iv1 on dbo.v2(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
```



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 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 pf2008(date) AS RANGE RIGHT
FOR VALUES ('2012-10-01', '2012-11-01', '2012-12-01');

CREATE PARTITION SCHEME ps2012 AS PARTITION pf2012 ALL TO ([PRIMARY]);

CREATE TABLE ptnsales(saledate DATE, salesperson INT, amount MONEY) ON
ps2012(saledate);
INSERT INTO ptnsales (saledate, salesperson, amount) VALUES ('2012-10-20',
1, 250.00);
INSERT INTO ptnsales (saledate, salesperson, amount) VALUES ('2012-11-05',
2, 129.00);
INSERT INTO ptnsales (saledate, salesperson, amount) VALUES ('2012-12-23',
2, 98.00);
INSERT INTO ptnsales (saledate, salesperson, amount) VALUES ('2012-10-3', 1,
450.00);

SELECT * FROM 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);

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;

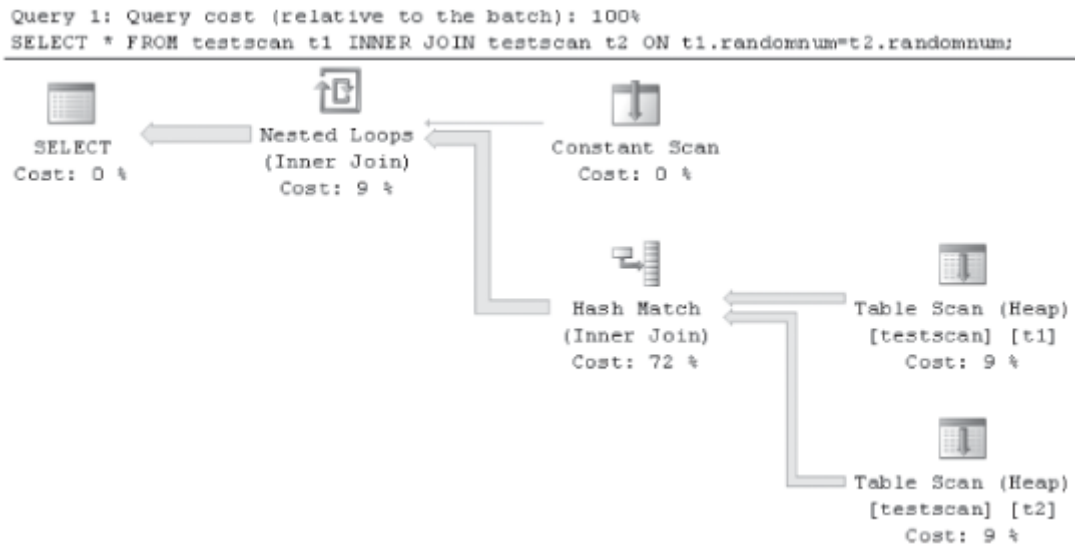
    WHILE @i < 100000
    BEGIN
        INSERT INTO testscan(randomnum, value)
        VALUES (rand()*400, @i);

        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
INNER 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 Query Optimizer, 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.

- 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 nullableint 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.

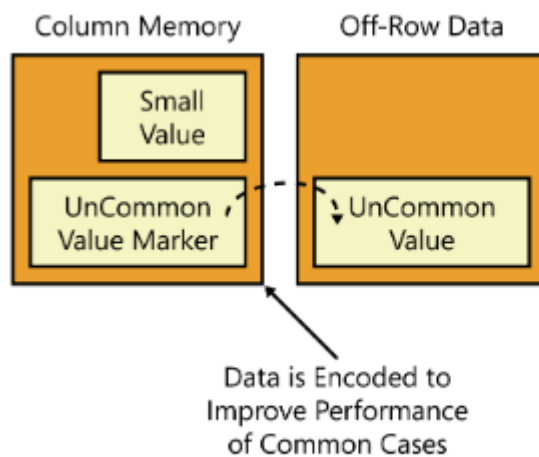


FIGURE 11-56 Uncommon data stored outside of the main batch.

- 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. Generally, this precludes many types that don't fit within a CPU register or that require functionality that can't be optimized, such as CLR 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 11-59 shows the common query plan shape for batch execution plans in this scenario.

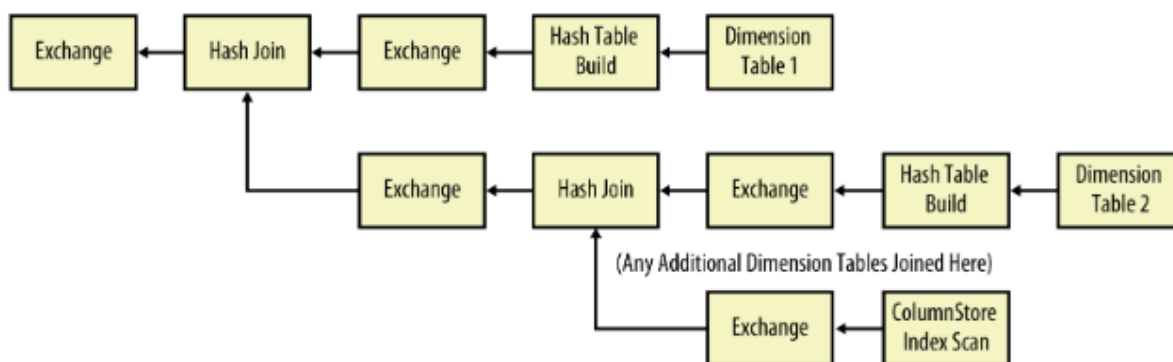


FIGURE 11-59 Typical columnstore plan shape.

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.

- 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;

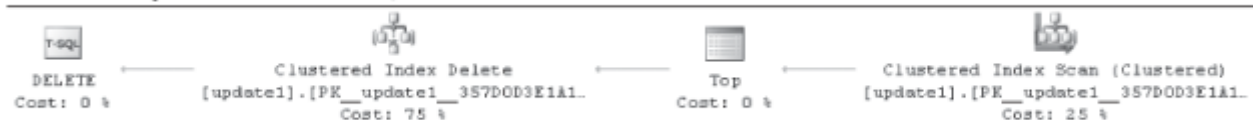


FIGURE 11-62 *DELETE* query plan.

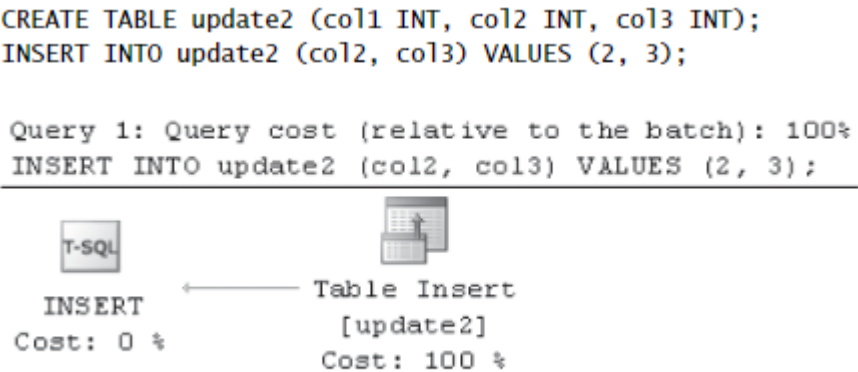


FIGURE 11-63 Simple INSERT query plan.

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.

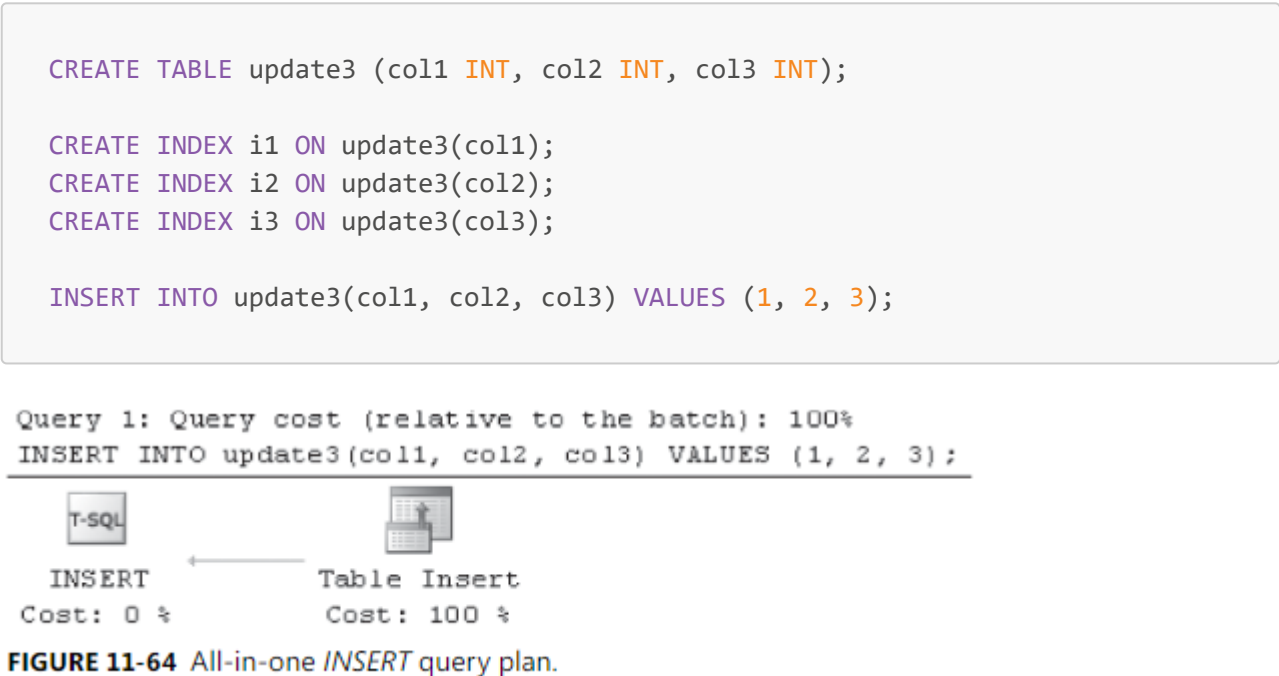


FIGURE 11-64 All-in-one INSERT query plan.

This query needs to update all the indexes because a new row has been created. However, Figure 11-64 shows that the plan has only the one operator. If you look at the properties for this operator in Management Studio, as shown in Figure 11-65, 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 11-66 shows the resulting query plan

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

Query 1: Query cost (relative to the batch): 100%
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/Collapse** 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 11-1.

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. In Listing 11-12, 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. Figure 11-69 shows the resulting MERGE query plan.

```
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);

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:

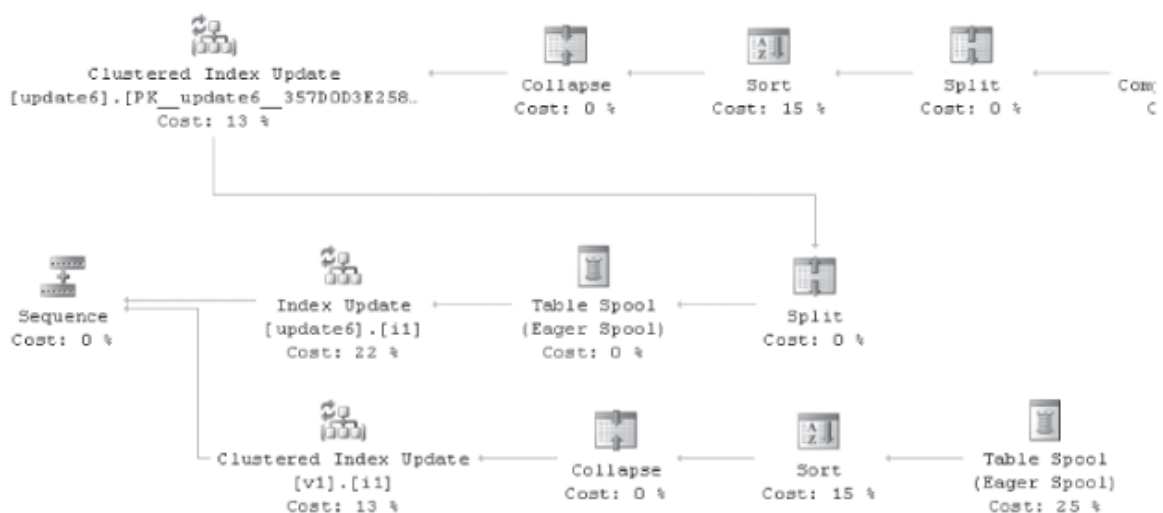
- Check for whether a row exists.
 - If it doesn't exist, INSERT
- Wide update plans SQL Server also has special optimization logic to speed the execution of large batch changes to a table. If a query is changing a large percentage of a table, SQL Server can create a plan that avoids modifying each B-tree with many individual updates. Instead, it can generate a per-ndex 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 11-75 shows the plan.

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

CREATE PARTITION SCHEME psinsert AS PARTITION pfininsert 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 quer 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 follow-ing example, a large number of rows are inserted into the partitioned table, and the Query Optimizer chooses to sort on the virtual partition ID column before inserting to reduce the number of partition switches at run time. Figure 11-80 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. Thi 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 11-82 shows the query plan used in this example, and Figure 11-83 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;
```

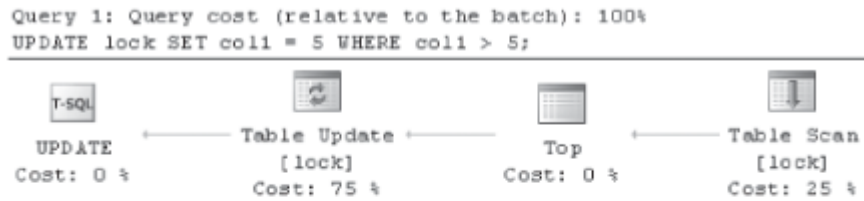


FIGURE 11-82 The update plan used in the locking example.

	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 11-84 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 Query Optimizer tree contains metadata collected from the remote source. The information collected is very similar to the information that the query processor 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 remotized 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 Query Optimizer 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 Query Optimizer has used bad information to make a decision. If the Query Optimizer makes a poor decision even with up-to-date statistics, this might mean that the Query Optimizer 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.

PONER ESTO EN IA Y QUE TIRE EJEMPLOS

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.

PONER ESTO EN IA Y QUE TIRE EJEMPLOS

- 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 data-warehouse systems 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.

PONER ESTO EN IA Y QUE TIRE EJEMPLOS

- INDEX = | Is very effective in forcing the QO to use a specific index when compiling a plan.
- FORCESEEK It tells the Query Optimizer that it needs to generate a seek predicate when using an index. In a few cases, the Query Optimizer can determine that an index scan is better than a seek when compiling the query. EARLIER VERSION OF 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 Query Optimizer 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 Query Optimizer 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 Query Optimizer.

```
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 query processor expands view definitions when parsing and binding the query tree. The **NOEXPAND** hint causes the query processor 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 Query Optimizer 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 Query Optimizer. 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 Query Optimizer 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 '<ShowPlanXML**

```
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 query executor runs the execution plan that the Query Optimizer 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.”

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.