I'll provide you with a couple of reasons to use both and some of the costs associated with each and thereby avoid the religious argument. An identity column is usually an `INT` or a `BIGINT`, which makes it narrow and easy to index, improving performance. Also, separating the value of the primary key from any business knowledge is considered good design in some circles. Sometimes globally unique identifiers (GUIDs) may be used as a primary key. They work fine but are difficult to read, so it impacts troubleshooting, and they can lead to greater index fragmentation. The breadth of the key can also cause a negative impact to performance. One of the drawbacks of artificial keys is that the numbers sometimes acquire business meaning, which should never happen. Another thing to keep in mind is that you have to create a unique constraint for the alternate keys to prevent the creation of multiple rows where none should exist. This increases the amount of information you have to store and maintain. Natural keys provide a clear, human-readable, primary key that has true business meaning. They tend to be wider fields—sometimes very wide—making them less efficient inside indexes. Also, sometimes the data may change, which has a profound trickle-down effect within your database because you will have to update every single place that key value is in use instead of simply one place with an artificial key. With the introduction of compliance like the General Data Protection Regulation (GDPR) in the European Union, natural keys become more problematic when worrying about your ability to modify data without removing it.

Let me just reiterate that either approach can work well and that each provides plenty of opportunities for tuning. Either approach, properly applied and maintained, will protect the integrity of your data.

Besides maintaining data integrity, unique indexes—the primary vehicle for entity-integrity constraints—help the optimizer generate efficient execution plans. SQL Server can often search through a unique index faster than it can search through a nonunique index. This is because each row in a unique index is unique; and, once a row is found, SQL Server does not have to look any further for other matching rows (the optimizer is aware of this fact). If a column is used in sort (or `GROUP BY` or `DISTINCT`) operations, consider defining a unique constraint on the column (using a unique index) because columns with a unique constraint generally sort faster than ones with no unique constraint. Also, a unique constraint adds additional information for the optimizer's cardinality estimation. Even an "unused" or "disused" index may still be helpful for optimization because of the effects on cardinality estimation.

To understand the performance benefit of entity-integrity or unique constraints, consider an example. Assume you want to modify the existing unique index on the `Production.Product` table.

```
CREATE NONCLUSTERED INDEX AK_Product_Name
ON Production.Product
(
    Name ASC
)
WITH (DROP_EXISTING = ON)
ON [PRIMARY];
GO
```

The nonclustered index does not include the `UNIQUE` constraint. Therefore, although the `[Name]` column contains unique values, the absence of the `UNIQUE` constraint from the nonclustered index does not provide this information to the optimizer in advance. Now, let's consider the performance impact of the `UNIQUE` constraint (or a missing `UNIQUE` constraint) on the following `SELECT` statement:

```
SELECT DISTINCT
       (p.Name)
FROM Production.Product AS p;
```

Figure 28-1 shows the execution plan of this `SELECT` statement.

Figure 28-1. *An execution plan with no UNIQUE constraint on the [Name] column*

From the execution plan, you can see that the nonclustered `AK_ProductName` index is used to retrieve the data, and then a `Stream Aggregate` operation is performed on the data to group the data on the `[Name]` column so that the duplicate `[Name]` values can be removed from the final result set. Note that the `Stream Aggregate` operation would not have been required if the optimizer had been told in advance about the uniqueness of

the [Name] column. You can accomplish this by defining the nonclustered index with a UNIQUE constraint, as follows:

```
CREATE UNIQUE NONCLUSTERED INDEX [AK_Product_Name]
ON [Production].[Product]([Name] ASC)
WITH (
DROP_EXISTING = ON)
ON    [PRIMARY];
GO
```

Figure 28-2 shows the new execution plan of the SELECT statement.

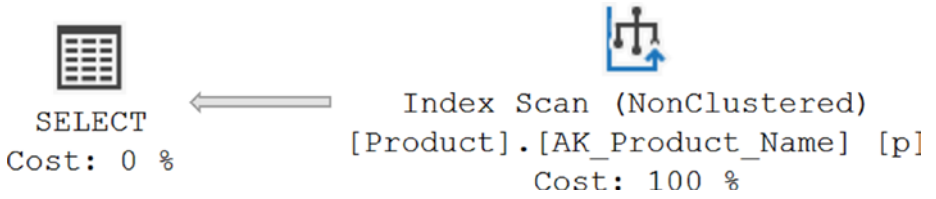

Figure 28-2. *An execution plan with a UNIQUE constraint on the [Name] column*

In general, the entity-integrity constraints (in other words, primary keys and unique constraints) provide useful information to the optimizer about the expected results, assisting the optimizer in generating efficient execution plans. Of note is the fact that sys.dm_db_index_usage_stats doesn't show when a constraint check has been run against the index that defines the unique constraint.

Maintain Domain and Referential Integrity Constraints

The other two important components of data integrity are *domain integrity* and *referential integrity*. Domain integrity for a column can be enforced by restricting the data type of the column, defining the format of the input data, and limiting the range of acceptable values for the column. Referential integrity is enforced by the use of foreign key constraints defined between tables. SQL Server provides the following features to implement the domain and referential integrity: data types, FOREIGN KEY constraints, CHECK constraints, DEFAULT definitions, and NOT NULL definitions. If an application requires that the values for a data column be restricted to a range of values, then this business rule can be implemented either in the application code or in the database schema. Implementing such a business rule in the database using domain constraints (such as the CHECK constraint) can help the optimizer generate efficient execution plans.

To understand the performance benefit of domain integrity, consider this example:

```
DROP TABLE IF EXISTS dbo.Test1;
GO
CREATE TABLE dbo.Test1 (
     C1 INT,
     C2 INT CHECK (C2 BETWEEN 10 AND 20)
    ) ;
INSERT  INTO dbo.Test1
VALUES  (11, 12);
GO
DROP TABLE IF EXISTS dbo.Test2;
GO
CREATE TABLE dbo.Test2 (C1 INT, C2 INT);
INSERT  INTO dbo.Test2
VALUES  (101, 102);
```

Now execute the following two SELECT statements:

```
SELECT T1.C1,
       T1.C2,
       T2.C2
FROM dbo.Test1 AS T1
    JOIN dbo.Test2 AS T2
        ON T1.C1 = T2.C2
           AND T1.C2 = 20;
GO
SELECT T1.C1,
       T1.C2,
       T2.C2
FROM dbo.Test1 AS T1
    JOIN dbo.Test2 AS T2
        ON T1.C1 = T2.C2
           AND T1.C2 = 30;
```

The two SELECT statements appear to be the same, except for the predicate values (20 in the first statement and 30 in the second). Although the two SELECT statements have the same form, the optimizer treats them differently because of the CHECK constraint on the T1.C2 column, as shown in the execution plan in Figure 28-3.

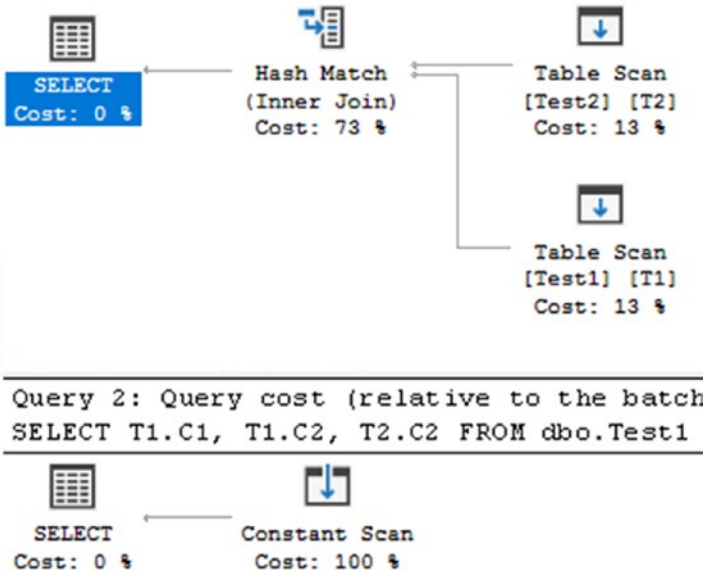

Figure 28-3. *Execution plans with predicate values within and outside the CHECK constraint boundaries*

From the execution plan, you can see that, for the first query (with T1.C2 = 20), the optimizer accesses the data from both tables. For the second query (with T1.C2 = 30), the optimizer understands from the corresponding CHECK constraint on the column T1.C2 that the column can't contain any value outside the range of 10 to 20. Thus, the optimizer doesn't even access the data from either table. Consequently, the relative estimated cost, and the actual performance measurement of doing almost nothing, of the second query is 0 percent.

I explained the performance advantage of referential integrity in detail in the "Declarative Referential Integrity" section of Chapter 19.

Therefore, you should use domain and referential constraints not only to implement data integrity but also to facilitate the optimizer in generating efficient query plans. Make sure that your foreign key constraints are created using the WITH CHECK option, or the optimizer will ignore them. To understand other performance benefits of domain and referential integrity, please refer to the "Using Domain and Referential Integrity" section of Chapter 19.

Adopt Index-Design Best Practices

The most common optimization recommendation—and frequently one of the biggest contributors to good performance—is to implement the correct indexes for the database workload. Indexes are unlike tables, which are used to store data and can be designed even without knowing the queries thoroughly (as long as the tables properly represent the business entities). Instead, indexes must be designed by reviewing the database queries thoroughly. Except in common and obvious cases, such as primary keys and unique indexes, please don't fall into the trap of designing indexes without knowing the queries. Even for primary keys and unique indexes, I advise you to validate the applicability of those indexes as you start designing the database queries. Considering the importance of indexes for database performance, you must be careful when designing indexes.

Although the performance aspect of indexes is explained in detail in Chapters 8, 9, 12, and 13, I'll reiterate a short list of recommendations for easy reference here:

- Choose narrow columns for index keys.
- Ensure that the selectivity of the data in the candidate column is very high (that is, the column must have a low number of candidate values returned).
- Prefer columns with the integer data type (or variants of the integer data type). Also, avoid indexes on columns with string data types such as `VARCHAR`.
- Consider listing columns having higher selectivity first in a multicolumn index.
- Use the `INCLUDE` list in an index as a way to make an index cover the index key structure without changing that structure. Do this by adding columns to the key, which enables you to avoid expensive lookup operations.
- When deciding which columns to index, pay extra attention to the queries' `WHERE` clauses and `JOIN` criteria columns and `HAVING` clause. These can serve as the entry points into the tables, especially if a `WHERE` clause criterion on a column filters the data on a highly selective value or constant. Such a clause can make the column a prime candidate for an index.

- When choosing the type of an index (clustered or nonclustered, columnstore or rowstore), keep in mind the advantages and disadvantages of the various index types.

Be extra careful when designing a clustered index because every nonclustered index on the table depends on the clustered index. Therefore, follow these recommendations when designing and implementing clustered indexes:

- Keep the clustered indexes as narrow as possible. You don't want to widen all your nonclustered indexes by having a wide clustered index.
- Create the clustered index first and then create the nonclustered indexes on the table.
- If required, rebuild a clustered index in a single step using the `DROP_EXISTING = {ON|OFF}` command in the `CREATE INDEX` command. You don't want to rebuild all the nonclustered indexes on the table twice: once when the clustered index is dropped and again when the clustered index is re-created.
- Do not create a clustered index on a frequently updated column. If you do so, the nonclustered indexes on the table will create additional load by remaining in sync with the clustered index key values.

To keep track of the indexes you've created and determine others that you need to create, you should take advantage of the dynamic management views that SQL Server 2017 and Azure SQL Database make available to you. By checking the data in `sys.dm_db_index_usage_stats` on a regular basis—say once a week or so—you can determine which of your indexes are actually being used and which are redundant. Indexes that are not contributing to your queries to help you improve performance are just a drain on the system. They require both more disk space and additional I/O to maintain the data inside the index as the data in the table changes. On the other hand, querying `sys.dm_db_missing_indexes_details` will show potential indexes deemed missing by the system and even suggest `INCLUDE` columns. You can access the DMV `sys.dm_db_missing_indexes_groups_stats` to see aggregate information about the number of times queries are called that could have benefited from a particular group of indexes. Just remember to test these suggestions thoroughly and don't assume that they will be correct. All these suggestions are just that: suggestions. All these tips can be combined to give you an optimal method for maintaining the indexes in your system over the long term.

Avoid the Use of the sp_Prefix for Stored Procedure Names

As a rule, don't use the `sp_` prefix for user stored procedures since SQL Server assumes that stored procedures with the `sp_` prefix are system stored procedures, and these are supposed to be in the master database. Using `sp` or `usp` as the prefix for user stored procedures is quite common. This is neither a major performance hit nor a major problem, but why court trouble? The performance hit of the `sp_` prefix is explained in detail in the "Be Careful Naming Stored Procedures" section of Chapter 20. Getting rid of prefixes entirely is a fine way to go. You have plenty of space for descriptive object names. There is no need for odd abbreviations that don't add to the functional definition of the queries.

Minimize the Use of Triggers

Triggers provide an attractive method for automating behavior within the database. Since they fire as data is manipulated by other processes (regardless of the processes), triggers can be used to ensure certain functions are run as the data changes. That same functionality makes them dangerous since they are not immediately visible to the developer or DBA working on a system. They must be taken into account when designing queries and when troubleshooting performance problems. Because they carry a somewhat hidden cost, triggers should be considered carefully. Before using a trigger, make sure that the only way to solve the problem presented is with a trigger. If you do use a trigger, document that fact in as many places as you can to ensure that the existence of the trigger is taken into account by other developers and DBAs.

Put Tables into In-Memory Storage

While there are a large number of limitations on in-memory storage mechanisms, the performance benefits are high. If you have a high-volume OLTP system and you're seeing lots of contention on I/O, especially around latches, the in-memory storage is a viable option. You may also want to explore using in-memory storage for table variables to help enhance their performance. If you have data that doesn't have to persist, you can even create the table in-memory using the `SCHEMA_ONLY` durability option. The general approach is to use the in-memory objects to help with high-throughput OLTP where you may have concurrency issues, as opposed to greater degrees of scans, and

so on, experienced in a data warehouse situation. All these methods lead to significant performance benefits. But remember, you must have the memory available to support these options. There's nothing magic here. You're enhancing performance by throwing significant amounts of memory, and therefore money, at the problem.

Use Columnstore Indexes

If you're designing and building a data warehouse, the use of columnstore indexes is almost automatic. Most of your queries are likely to involve aggregations across large groups of data, so the columnstore index is a natural performance enhancer. However, don't forget about putting the nonclustered columnstore index to work in your OLTP systems where you get frequent analytical style queries. There is additional maintenance overhead, and it will increase the size of your databases, but the benefits are enormous. You can create a rowstore table, as defined by the clustered index, that can use columnstore indexes. You can also create a columnstore table, as defined by its clustered index, that can use rowstore indexes. Using both these mechanisms, you can ensure that you meet the most common query style, analytical versus OLTP, while still supporting the other.

Configuration Settings

Here's a checklist of the server and database configurations settings that have a big impact on database performance:

- Memory configuration options
- Cost threshold for parallelism
- Max degree of parallelism
- Optimize for ad hoc workloads
- Blocked process threshold
- Database file layout
- Database compression

I cover these settings in more detail in the sections that follow.

Memory Configuration Options

As explained in the "SQL Server Memory Management" section of Chapter 2, it is strongly recommended that the `max server memory` setting be configured to a nondefault value determined by the system configuration. These memory configurations of SQL Server are explained in detail in the "Memory Bottleneck Analysis" and "Memory Bottleneck Resolutions" sections of Chapter 2.

Cost Threshold for Parallelism

On systems with multiple processors, the parallel execution of queries is possible. The default value for parallelism is 5. This represents a cost estimate by the optimizer of a five-second execution on the query. In most circumstances, I've found this value to be radically too low; in other words, a higher threshold for parallelism results in better performance. Testing on your system will help you determine the appropriate value. Suggesting a value for this can be considered somewhat dangerous, but I'm going to do it anyway. I'd begin testing with a value of 35 and see where things go from there. Even better, use the data from the Query Store to determine the average cost of all your queries and then go two to three standard deviations above that average for the value of the Cost Threshold for Parallelism. In that way, you're looking at 95 percent to 98 percent of your queries will not go parallel, but the ones that really need it will. Finally, remember what type of system you're running. An OLTP system is much more likely to benefit from a lot of queries using a minimal amount of CPU each, while an analytical system is much more likely to benefit from more of the queries using more CPU.

Max Degree of Parallelism

When a system has multiple processors available, by default SQL Server will use all of them during parallel executions. To better control the load on the machine, you may find it useful to limit the number of processors used by parallel executions. Further, you may need to set the affinity so that certain processors are reserved for the operating system and other services running alongside SQL Server. OLTP systems may receive a benefit from disabling parallelism entirely, although that's a questionable choice. First try increasing the cost threshold for parallelism because, even in OLTP systems, there are queries that will benefit from parallel execution, especially maintenance jobs. You may also explore the possibility of using the Resource Governor to control some workloads.

Optimize for Ad Hoc Workloads

If the primary calls being made to your system come in as ad hoc or dynamic T-SQL instead of through well-defined stored procedures or parameterized queries, such as you might find in some of the implementations of object-relational mapping (ORM) software, then turning on the `optimize for ad hoc workloads` setting will reduce the consumption of procedure cache because plan stubs are created for initial query calls instead of full execution plans. This is covered in detail in Chapter 18.

Blocked Process Threshold

The `blocked process threshold` setting defines in seconds when a blocked process report is fired. When a query runs and exceeds the threshold, the report is fired. An alert, which can be used to send an e-mail or a text message, is also fired. Testing an individual system determines what value to set this to. You can monitor for this using events within Extended Events.

Database File Layout

For easy reference, the following are the best practices you should consider when laying out database files:

- Place the data and transaction log files of a user database on different I/O paths. This allows the transaction log disk head to progress sequentially without being moved randomly by the nonsequential I/Os commonly used for the data files.
- Placing the transaction log on a dedicated disk also enhances data protection. If a database disk fails, you will be able to save the completed transactions until the point of failure by performing a backup of the transaction log. By using this last transaction log backup during the recovery process, you will be able to recover the database up to the point of failure. This is known as *point-in-time recovery*.
- Avoid RAID 5 for transaction logs because, for every write request, RAID 5 disk arrays incur twice the number of disk I/Os compared to RAID 1 or 10.

- You may choose RAID 5 for data files since even in a heavy OLTP system, the number of read requests is usually seven to eight times the number of write requests. Also, for read requests the performance of RAID 5 is similar to that of RAID 1 and RAID 10 with an equal number of total disks.
- Look into moving to a more modern disk subsystem like SSD or FusionIO.
- Have multiple files for tempdb. The general rule would be half or one-quarter the files for the number of logical processor cores. All allocations in tempdb now use uniform extents. You'll also see the files will automatically grow at the same size now.

For a detailed understanding of database file layout and RAID subsystems, please refer to the "Disk Bottleneck Resolutions" section of Chapter 3.

Database Compression

SQL Server has supplied data compression since 2008 with the Enterprise and Developer editions of the product. This can provide a great benefit in space used and in performance as more data gets stored on a page. These benefits come at the cost of added overhead in the CPU and memory of the system; however, the benefits usually far outweigh the costs. Take this into account as you implement compression.

Database Administration

For your reference, here is a short list of the performance-related database administrative activities that you should perform on a regular basis as part of the process of managing your database server:

- Keep the statistics up-to-date.
- Maintain a minimum amount of index defragmentation.
- Avoid automatic database functions such as `AUTOCLOSE` or `AUTOSHRINK`.

In the following sections, I cover the preceding activities in more detail.

Note For a detailed explanation of SQL Server 2017 administration needs and methods, please refer to the Microsoft SQL Server Books Online article "Database Engine Features and Tasks" (http://bit.ly/SIlz8d).

Keep the Statistics Up-to-Date

The performance impact of database statistics is explained in detail in Chapter 13 (and in various places throughout the book); however, this short list will serve as a quick and easy reference for keeping your statistics up-to-date:

- Allow SQL Server to automatically maintain the statistics of the data distribution in the tables by using the default settings for the configuration parameters `AUTO_CREATE_ STATISTICS` and `AUTO_UPDATE_STATISTICS`.
- As a proactive measure, you can programmatically update the statistics of every database object on a regular basis as you determine it is needed and supported within your system. This practice partly protects your database from having outdated statistics in case the auto update statistics feature fails to provide a satisfactory result. In Chapter 13, I illustrate how to set up a SQL Server job to programmatically update the statistics on a regular basis.
- Remember that you also have the ability to update the statistics in an asynchronous fashion. This reduces the contention on stats as they're being updated; thus, if you have a system with fairly constant access, you can use this method to update the statistics more frequently. Async is more likely to be helpful if you're seeing waits on statistics updates.

Note Please ensure that the statistics update job is scheduled before the completion of the index defragmentation job, as explained later in this chapter.

Maintain a Minimum Amount of Index Defragmentation

The following best practices will help you maintain a minimum amount of index defragmentation:

- Defragment a database on a regular basis during nonpeak hours.
- On a regular basis, determine the level of fragmentation on your indexes; then, based on that fragmentation, either rebuild the index or defrag the index by executing the defragmentation queries outlined in Chapter 14.
- Remember that very small tables don't need to be defragmented at all.
- Different rules may apply for very large databases when it comes to defragmenting indexes.
- If you have indexes that are only ever used for single seek operations, then fragmentation doesn't impact performance.
- In Azure SQL Database, it's much more important to only rebuild indexes if you really need to. Index rebuilds can use up a lot of I/O bandwidth and can lead to throttling.

Also remember that index fragmentation is much less of a problem than most people make it out to be. Some experts are even suggesting that defragmenting of indexes is a waste of time. While I still think there are benefits, that is on a situational basis, so be sure you're monitoring and measuring your performance metrics carefully so that you can tell whether defragmentation is a benefit.

Avoid Database Functions Such As AUTO_CLOSE or AUTO_SHRINK

`AUTO_CLOSE` cleanly shuts down a database and frees all its resources when the last user connection is closed. This means all data and queries in the cache are automatically flushed. When the next connection comes in, not only does the database have to restart but all the data has to be reloaded into the cache. Also, stored procedures and the other queries have to be recompiled. That's an extremely expensive operation for most database systems. Leave `AUTO_CLOSE` set to the default of `OFF`.

AUTO_SHRINK periodically shrinks the size of the database. It can shrink the data files and, when in Simple Recovery mode, the log files. While doing this, it can block other processes, seriously slowing down your system. More often than not, file growth is also set to occur automatically on systems with AUTO_SHRINK enabled, so your system will be slowed down yet again when the data or log files have to grow. Further, you're going to see the physical file storage get fragmented at the operating system level, seriously impacting performance. Set your database sizes to an appropriate size, and monitor them for growth needs. If you must grow them automatically, do so by physical increments, not by percentages.

Database Backup

Database backup is a broad topic and can't be given due justice in this query optimization book. Nevertheless, I suggest that when it comes to database performance, you be attentive to the following aspects of your database backup process:

- Differential and transaction log backup frequency
- Backup distribution
- Backup compression

The next sections go into more detail on these suggestions.

Incremental and Transaction Log Backup Frequency

For an OLTP database, it is mandatory that the database be backed up regularly so that, in case of a failure, the database can be restored on a different server. For large databases, the full database backup usually takes a long time, so full backups cannot be performed often. Consequently, full backups are performed at widespread time intervals, with incremental backups and transaction log backups scheduled more frequently between two consecutive full backups. With the frequent incremental and transaction log backups set in place, if a database fails completely, the database can be restored up to a point in time.

Differential backups can be used to reduce the overhead of a full backup by backing up only the data that has changed since the last full backup. Because this is potentially much faster, it will cause less of a slowdown on the production system. Each situation is unique, so you need to find the method that works best for you. As a general rule, I recommend taking a weekly full backup and then daily differential backups. From there, you can determine the needs of your transaction log backups.

Frequently backing up of the transaction log adds a small amount of overhead to the server, especially during peak hours.

For most businesses, the acceptable amount of data loss (in terms of time) usually takes precedence over conserving the log-disk space or providing ideal database performance. Therefore, you must take into account the acceptable amount of data loss when scheduling the transaction log backup, as opposed to randomly setting the backup schedule to a low-time interval.

Backup Scheduling Distribution

When multiple databases need to be backed up, you must ensure that all full backups are not scheduled at the same time so that the hardware resources are not hit at the same time. If the backup process involves backing up the databases to a central SAN disk array, then the full backups from all the database servers must be distributed across the backup time window so that the central backup infrastructure doesn't get slammed by too many backup requests at the same time. Flooding the central infrastructure with a great deal of backup requests at the same time forces the components of the infrastructure to spend a significant part of their resources just managing the excessive number of requests. This mismanaged use of the resources increases the backup durations significantly, causing the full backups to continue during peak hours and thus affecting the performance of the user requests.

To minimize the impact of the full backup process on database performance, you must first determine the nonpeak hours when full backups can be scheduled and then distribute the full backups across the nonpeak time window, as follows:

1. Identify the number of databases that must be backed up.
2. Prioritize the databases in order of their importance to the business.
3. Determine the nonpeak hours when the full database backups can be scheduled.
4. Calculate the time interval between two consecutive full backups as follows: Time interval = (Total backup time window) / (Number of full backups).

5. Schedule the full backups in order of the database priorities, with the first backup starting at the start time of the backup window and subsequent backups spread uniformly at the time intervals calculated in the preceding equation.

This uniform distribution of the full backups will ensure that the backup infrastructure is not flooded with too many backup requests at the same time, thereby reducing the impact of the full backups on the database performance.

Backup Compression

For relatively large databases, the backup durations and backup file sizes usually become an issue. Long backup durations make it difficult to complete the backups within the administrative time windows and thus start affecting the end user's experience. The large size of the backup files makes space management for the backup files quite challenging, and it increases the pressure on the network when the backups are performed across the network to a central backup infrastructure. Compression also acts to speed up the backup process since fewer writes to the disk are needed.

The recommended way to optimize the backup duration, the backup file size, and the resultant network pressure is to use *backup compression*.

Query Design

Here's a list of the performance-related best practices you should follow when designing the database queries:

- Use the command `SET NOCOUNT ON`.
- Explicitly define the owner of an object.
- Avoid *nonsargable* search conditions.
- Avoid arithmetic operators and functions on `WHERE` clause columns.
- Avoid optimizer hints.
- Stay away from nesting views.
- Ensure there are no implicit data type conversions.
- Minimize logging overhead.

- Adopt best practices for reusing execution plans.
- Adopt best practices for database transactions.
- Eliminate or reduce the overhead of database cursors.
- Use natively compile stored procedures.
- Take advantage of columnstore indexes for analytical queries

I further detail each best practice in the following sections.

Use the Command SET NOCOUNT ON

As a rule, always use the command `SET NOCOUNT ON` as the first statement in stored procedures, triggers, and other batch queries. This enables you to avoid the network overhead associated with the return of the number of rows affected after every execution of a SQL statement. The command `SET NOCOUNT` is explained in detail in the "Use SET NOCOUNT" section of Chapter 20.

Explicitly Define the Owner of an Object

As a performance best practice, always qualify a database object with its owner to avoid the runtime cost required to verify the owner of the object. The performance benefit of explicitly qualifying the owner of a database object is explained in detail in the "Do Not Allow Implicit Resolution of Objects in Queries" section of Chapter 16.

Avoid Nonsargable Search Conditions

Be vigilant when defining the search conditions in your query. If the search condition on a column used in the `WHERE` clause prevents the optimizer from effectively using the index on that column, then the execution cost for the query will be high in spite of the presence of the correct index. The performance impact of nonsargable search conditions is explained in detail in the corresponding section of Chapter 19.

Additionally, please be careful about providing too much flexibility on search capabilities. If you define an application feature such as "retrieve all products with product name ending in caps," then you will have queries scanning the complete table (or the clustered index). As you know, scanning a multimillion-row table will hurt your database performance. Unless you use an index hint, you won't be able to benefit from

the index on that column. However, using an index hint overrides the decisions of the query optimizer, so it's generally not recommended that you use index hints either (see Chapter 19 for more information). To understand the performance impact of such a business rule, consider the following SELECT statement:

```
SELECT p.*
FROM Production.Product AS p
WHERE p.Name LIKE '%Caps';
```

In Figure 28-4, you can see that the execution plan used the index on the [Name] column, but it had to perform a scan instead of a seek. Since an index on a column with character data types (such as CHAR and VARCHAR) sorts the data values for the column on the leading-end characters, using a leading % in the LIKE condition doesn't allow a seek operation into the index. The matching rows may be distributed throughout the index rows, making the index ineffective for the search condition and thereby hurting the performance of the query.

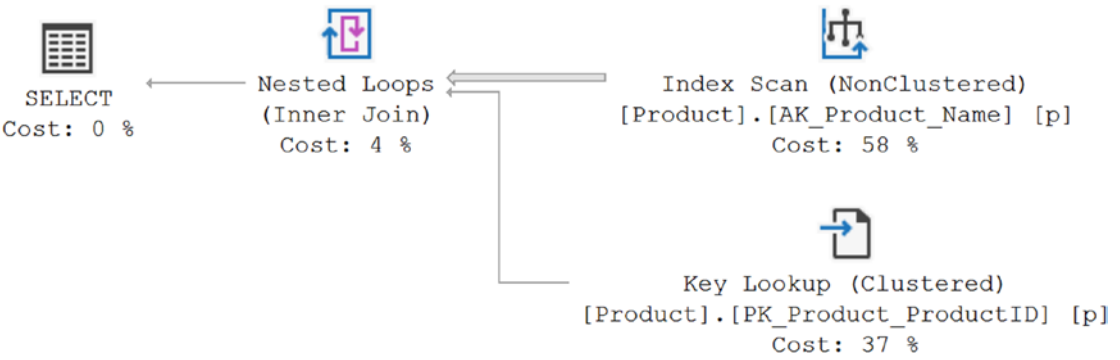

Figure 28-4. *An execution plan showing a clustered index scan caused by a nonsargable LIKE clause*

Avoid Arithmetic Expressions on the WHERE Clause Column

Always try to avoid using arithmetic operators and functions on columns in the WHERE and JOIN clauses. Using operators and functions on columns prevents the use of indexes on those columns. The performance impact of using arithmetic operators on WHERE clause columns is explained in detail in the "Avoid Arithmetic Operators on the WHERE Clause Column" section of Chapter 18, and the impact of using functions is explained in detail in the "Avoid Functions on the WHERE Clause Column" section of the same chapter.

To see this in action, consider the following queries:

```
SELECT  soh.SalesOrderNumber
FROM    Sales.SalesOrderHeader AS soh
WHERE   'SO5' = LEFT(SalesOrderNumber, 3);

SELECT  soh.SalesOrderNumber
FROM    Sales.SalesOrderHeader AS soh
WHERE   SalesOrderNumber LIKE 'SO5%';
```

These queries basically implement the same logic: they check `SalesOrderNumber` to see whether it is equal to `SO5`. However, the first query performs a function on the `SalesOrderNumber` column, while the second uses a `LIKE` clause to check for the same data. Figure 28-5 shows the resulting execution plans.

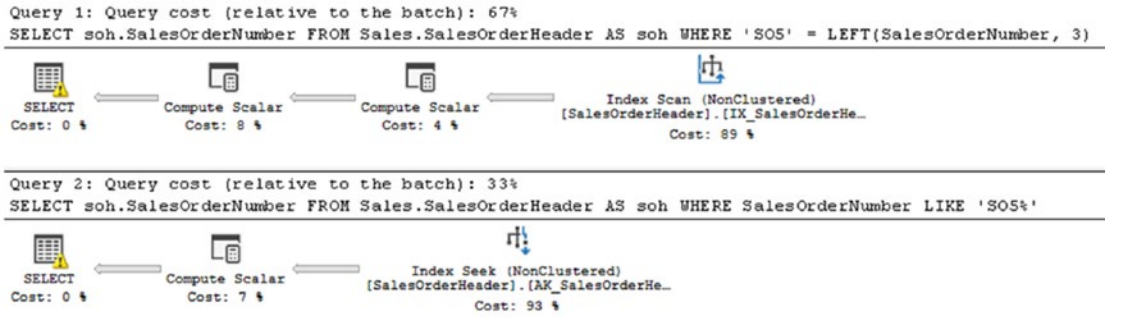

Figure 28-5. *Execution plans showing a function that prevents index use*

As you can see in Figure 28-5, the first query forces an `Index Scan` operation, while the second is able to perform a nice, clean `Index Seek`. These examples demonstrate clearly why you should avoid functions and operators on `WHERE` clause columns.

The warning you see in the plans relates to the implicit conversion occurring within the calculated columns in the `SalesOrderHeader` table.

Avoid Optimizer Hints

As a rule, avoid the use of optimizer hints, such as index hints and join hints, because they overrule the decision-making process of the optimizer. In most cases, the optimizer is smart enough to generate efficient execution plans, and it works best without any optimizer hint imposed on it. For a detailed understanding of the performance impact of optimizer hints, please refer to the "Avoiding Optimizer Hints" section of Chapter 19.