



# Microsoft Build

May 6–8, 2019





# **Built for Speed: SQL Server Database Application Design for Performance**

Pam Lahoud, Sr. Program Manager  
Pedro Lopes, Sr. Program Manager

## Meet the speakers!



@SQLGoddess



@SQLPedro

# Performance tuning – it's not just for DBAs!

What you can do as a developer to help generate efficient SQL Server code

- Application Design Patterns

- To ORM or not to ORM
- Are you cloud-ready?
  - Technical debt
  - DB Compatibility Certification

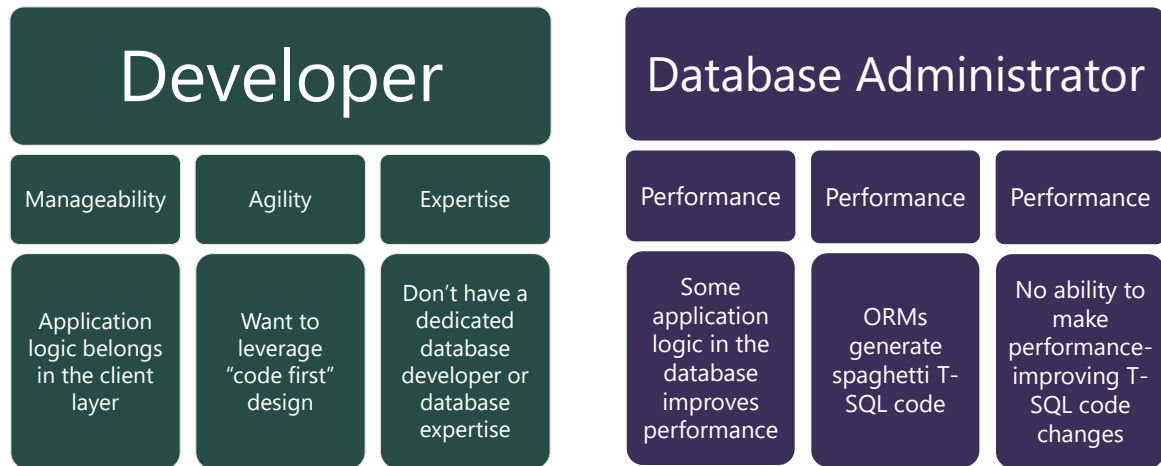
- Writing Efficient T-SQL

- Cardinality
- SARGability
- Common T-SQL Anti-Patterns

- Application Design Patterns
  - To ORM or not to ORM –Pam
  - Are you cloud-ready? - Pam
    - Death by paper cut – the perils of the chatty application - Pam
    - DB Compatibility Certification - Pedro
- Writing Efficient T-SQL
  - Cardinality - Pedro
  - SARGability - Pedro
  - Common T-SQL Anti-Patterns Pam/Pedro – decide division later

# Application Design Patterns

## To ORM or not to ORM

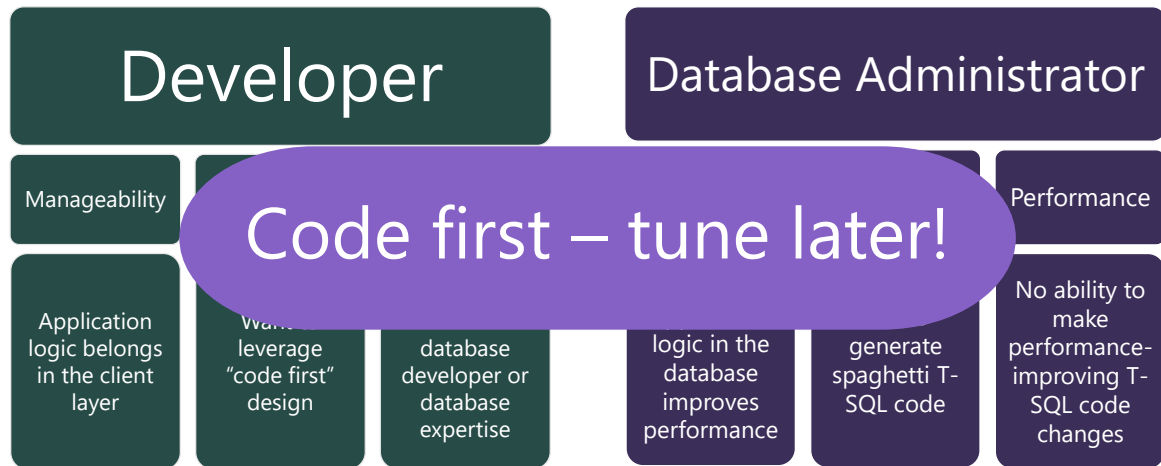


<https://aka.ms/EFPerf>

Object-relational mapping allows developers to quickly and easily design applications that rely on relational databases without having to have intimate knowledge of database and query design. It allows developers to create platform-agnostic code that can easily be ported to multiple systems, or in the case of an ISV (independent software vendor), allow end users a choice of platform. It also enables developers to choose a code-first design approach, which negates the need for spending a large amount of time designing a database schema.

All of these benefits help developers adhere to an Agile development methodology, but while this may improve the ease and speed of code development, as the database grows and the application scales, database performance can become an issue.

## To ORM or not to ORM



<https://aka.ms/EFPerf>

Object-relational mapping allows developers to quickly and easily design applications that rely on relational databases without having to have intimate knowledge of database and query design. It allows developers to create platform-agnostic code that can easily be ported to multiple systems, or in the case of an ISV (independent software vendor), allow end users a choice of platform. It also enables developers to choose a code-first design approach, which negates the need for spending a large amount of time designing a database schema.

All of these benefits help developers adhere to an Agile development methodology, but while this may improve the ease and speed of code development, as the database grows and the application scales, database performance can become an issue.

## Are you cloud-ready?

What can you do today to make moving to the cloud tomorrow easier?





# Pay down technical debt

Or don't accrue it in the first place

## Proximity considerations

- Caching
- Use of Stored Procedures
- Avoid looping logic and cursors outside of the database
- Return only the data you need at the time you need it

## Cost of Goods Sold (COGS)

- "Throwing hardware at the problem" is no longer a one-time cost
- Tune queries to reduce CPU and I/O
- Remove unnecessary tables/indexes/data
- Implement an archiving strategy

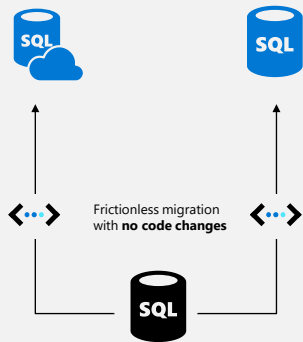
## Containment

- Is your application contained in a single database, or is there sprawl?
- What does your security model look like?
- Are there any features being used that would add extra cost or make cloud-migration challenging?




# Upgrade & modernize your SQL Server database on-premises and in the cloud with compatible certification

Stop worrying about certifying to Azure, on-premises, or named SQL Server versions.  
Compatibility-based certification allows you to certify with focus on continuous application lifecycle

Upgrade to the latest SQL Server Database Engine without changing your critical applications



## Compatibility certification benefits

- |   |  |   |
|---|--|---|
|  | <b>Simplified application certification</b>          | Applications tested and certified on a given SQL Server version are also implicitly tested and certified on that SQL Server version native database compatibility level |
|  | <b>Reduce upgrade risks</b>                          | Separate application and platform layer upgrade cycles for less disruption  |
|  | <b>Upgrade to latest SQL Database Engine version</b> | Upgrade your SQL Server Database Engine or move instances to the cloud with no code changes   |

## Database Compatibility Level protection with Microsoft

Microsoft provides an ecosystem of tools and services to test whether Database Compatibility Level certification is right for you and protect you as you upgrade



### Maintain backwards compatibility

Applications running on a newer SQL Server Database Engine while using an older database compatibility level can still leverage server-level enhancements without application changes

Database Compatibility Level settings affect behaviors for a specified database, not the entire server



### Query plan shape protection

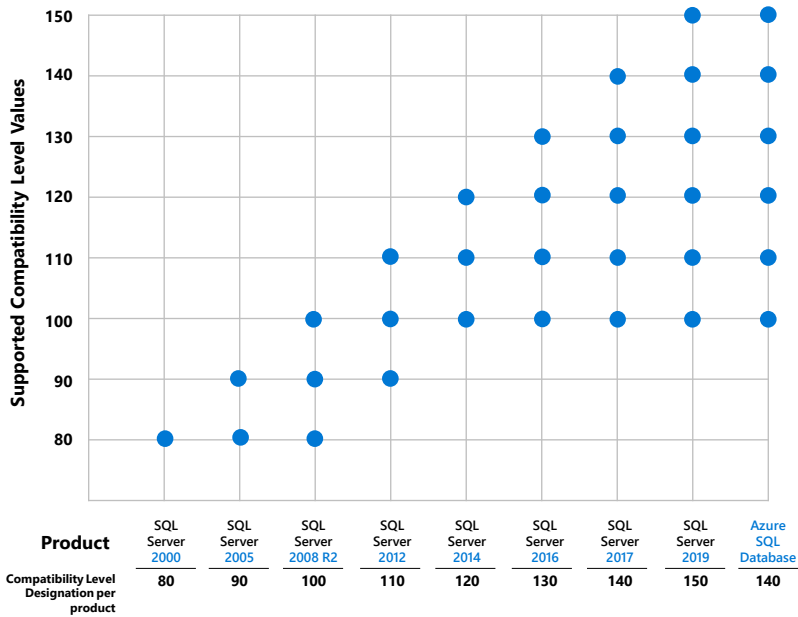
Microsoft gates query plan changes behind Database Compatibility Level to upgrade without issue once validation testing is completed

Hardware target and source tests should be run separately

Learn more here: <http://aka.ms/dbcompat>

## Changes needed pre-upgrade

Changing your database compatibility level changes the database feature set. Any discontinued functionality, code or features in a given SQL Server version may not be protected. Using tools like DMA can help assess your ability to leverage the enhanced security and scalability of the new database engine in SQL Server and Azure SQL Database without any required application changes.



## Explore your Database Compatibility Level supported values

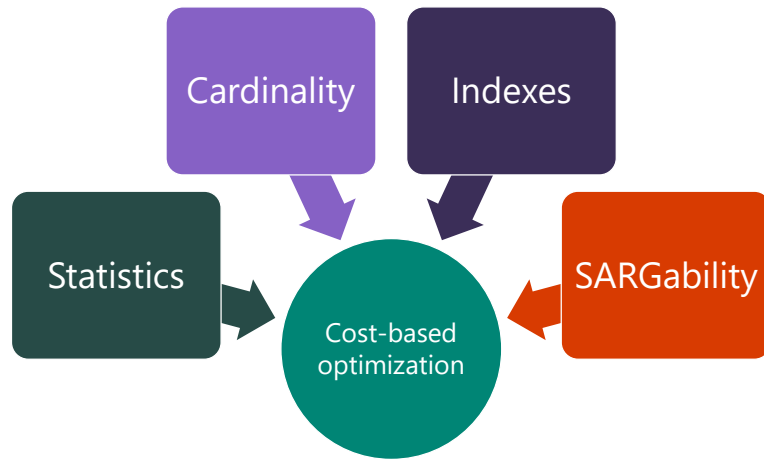
Upgrade from any earlier version of SQL Server and the database retains its existing compatibility level if it is at least minimum allowed for that instance of SQL Server

For example, SQL Server 2008 databases have supported compatibility up to SQL Server 2019 and Azure SQL Database

# Writing Efficient T-SQL

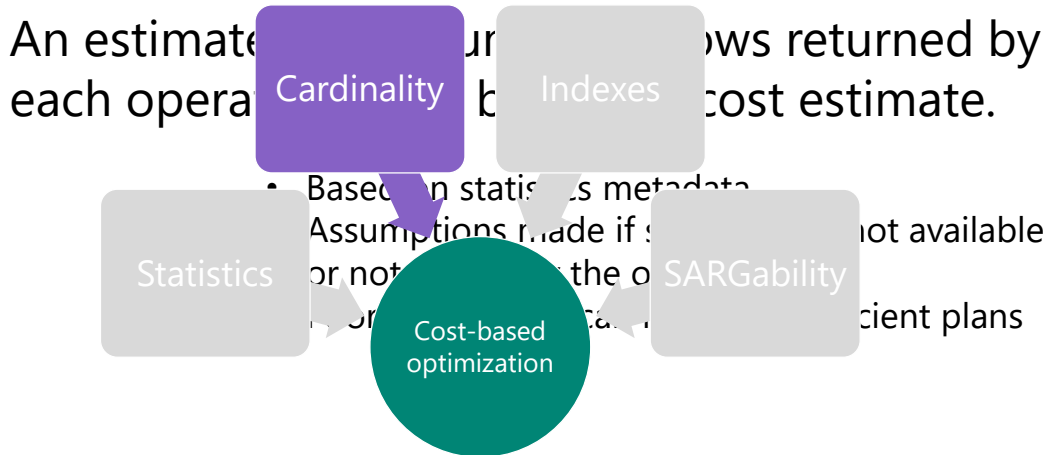
Time check – should be at 20 minutes.

## Why did SQL Server pick this plan?



SQL Server uses a cost-based optimizer which means that all the decisions that are made when generating a query plan are based on their estimated cost. The goal of the optimizer is to return the results of your query in the cheapest (i.e. the fastest) way. The optimizer is quite sophisticated and does a very good job of executing queries quickly and efficiently, provided that it has the right tools and information to do its job. There are many aspects to cost-based optimization, but essentially the optimizer uses the available statistics and other information about your data such as constraints to estimate the cardinality of the query. Based on this cardinality estimate, the optimizer will choose methods to access the data such as seeks, scans and lookups using the indexes that are in place, provided that the predicates in the query are SARGable. In the first part of our session, we are going to focus on cardinality and SARGability, since these are often impacted by the code that is written against the database.

# What is Cardinality?



Cardinality is basically a fancy word for the number of rows returned by a query. In the case of query optimization, SQL Server needs to understand the cardinality of each piece of the query – each table accessed, each predicate in the WHERE clause, each join condition – in order to choose the correct data access method. Take the following query for example:

```
SELECT *  
FROM Users  
WHERE IsActive = 1
```

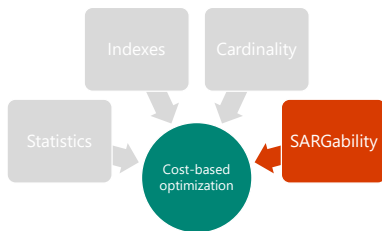
Let's say we have a non-clustered index on IsActive. Does it make sense for SQL Server to use this index to return the rows for this query? In order to answer that question, SQL Server needs to have an idea of how many rows have a value of 1 for IsActive in the Users table. If the entire table has IsActive = 1, it would make more sense for SQL Server to scan the whole table rather than use the non-clustered index. SQL Server uses statistics to estimate the number of rows returned, and this in turn allows SQL Server to choose the access method that best suits the query. Of course, data will change over time, so this is one of the reasons why it is very important to make sure your statistics are maintained on a regular basis.

Healthy statistics are essential for good query optimization, but the way you write the query can also have an impact on how SQL Server estimates cardinality. This is what we will focus on in this session.

# What is SARGability?

The extent to which a predicate can be used as a **Search ARGument** for an index seek

- Non-SARGable expressions cannot seek, they must scan a table or an index
- Non-SARGable expressions can significantly slow down queries



SARGability is a word that implies whether or not SQL Server will be able to use an index to locate rows that meet a predicate in a WHERE clause. In an index seek, the filter condition is used to limit the data that needs to be searched in order to locate the matching rows. In a scan, all the data that is part of the table or index needs to be searched for the rows that match the filter criteria.

Think of a cook book with a recipe name index in the back. If you are looking for a recipe for chocolate chip cookies, you can use the index in the back of the book to look up chocolate chip cookies and will find the page number for the recipe. You then turn to that page and you are done. Your filter condition "WHERE RecipeName = 'Chocolate Chip Cookies'" is SARGable. But what if you wanted to find all the recipes that have the word 'Chip' somewhere in the recipe name? You would not be able to use the recipe name index to find those recipes directly, so you would have to scan through all the names in the index to find the ones that have 'Chip' in the name. Once you had those recipe names, you'd have to keep track of all the page numbers and then flip through the book to each page number on your list to look at the recipes. That's a lot of work, it would probably be easier to flip through all the pages of the book looking for the recipes with 'Chip' in the name. That's a table scan, and that's usually what happens when you have a non-SARGable predicate in your WHERE clause.



## Common T-SQL Anti-patterns

Certain types of expressions can limit SQL Server's ability to correctly estimate cardinality and/or use an index to evaluate a predicate

Implicit  
Conversions

Functions in  
the Predicate

LIKE with a  
Leading  
Wildcard

OR in the  
WHERE Clause

Composable  
Logic

Table-valued  
Functions

There are several code practices which can lead to issues with both cardinality estimates and SARGability. This list is not comprehensive, but these are some of the most common issues that we encounter with SQL Server applications in the field.

# Implicit Conversions

ProductID is defined as VARCHAR(8)

```
SELECT *  
FROM Product  
WHERE ProductID = 7;
```



Implicit conversions are essentially functions in the WHERE clause, but because they happen automatically in the background, they are more difficult to detect. In the example on the slide, if 7 had been sent as a string instead of an integer, there would be no issue with the query and the predicate would be SARGable. Unfortunately, because of the rules of data type precedence, SQL Server needs to convert the varchar(8) ProductID to an integer in order to make this comparison, it will not convert the 7 to a varchar(8).

Another common cause of implicit conversions is in string types being sent from .NET. String values coming from .NET will be Unicode by default. If the strings in the database are defined as varchar rather than nvarchar, this will lead to implicit conversions. Again, because of the rules of data type precedence, varchar needs to be converted to nvarchar which will make all string comparisons in the application non-SARGable and also cause SQL Server to make assumptions about cardinality.

If you are using Entity Framework, it is critical to ensure all your data types are properly mapped in the mapping file. String types that are not mapped properly will be assumed to be nvarchar(4000).

## Functions in the Predicate

```
SELECT *  
FROM Person.Person  
WHERE SUBSTRING(FirstName, 1, 1) = 'B';
```

```
SELECT *  
FROM Sales.SalesOrderHeader  
WHERE YEAR(OrderDate) = 2008;
```

Functions in the WHERE clause (when executed against a column in a table, not a literal value) will always be non-SARGable because SQL Server needs to evaluate the function for every row in the table/index before it can compare the results of this function to the value in the predicate. This results in a scan, but also will impact the cardinality estimate. SQL Server has statistics for actual values in the table, not for the results of the function, so it will have to make an assumption. This can lead to problems later in the query plan with other predicates or joins that are not directly related to the function.

Consider this problem when designing your initial data model. Using a case-sensitive collation for example might necessitate using the UPPER function for string comparisons. Using some kind of non-date data type for storing dates may also lead to functions to translate the value into a date. Another common issue is using a date/time field but wanting to compare only the date portion. Rather than using a function that cuts off the time portion of the date/time field, use ranges instead as these will be SARGable. Often with functions the predicate can be re-written as a range scan which is SARGable.

If the predicate cannot be re-written, consider using a computed column to pre-evaluate the function. Creating an index on the computed column will make any predicate that uses the function SARGable and allow SQL Server to calculate statistics for the results of the function, improving cardinality estimates.

## Functions in the Predicate

```
SELECT *  
FROM Person.Person  
WHERE FirstName LIKE 'B%';
```

```
SELECT *  
FROM Sales.SalesOrderHeader  
WHERE OrderDate BETWEEN '1/1/2008'  
AND '12/31/2008';
```



Functions in the WHERE clause (when executed against a column in a table, not a literal value) will always be non-SARGable because SQL Server needs to evaluate the function for every row in the table/index before it can compare the results of this function to the value in the predicate. This results in a scan, but also will impact the cardinality estimate. SQL Server has statistics for actual values in the table, not for the results of the function, so it will have to make an assumption. This can lead to problems later in the query plan with other predicates or joins that are not directly related to the function.

Consider this problem when designing your initial data model. Using a case-sensitive collation for example might necessitate using the UPPER function for string comparisons. Using some kind of non-date data type for storing dates may also lead to functions to translate the value into a date. Another common issue is using a date/time field but wanting to compare only the date portion. Rather than using a function that cuts off the time portion of the date/time field, use ranges instead as these will be SARGable. Often with functions the predicate can be re-written as a range scan which is SARGable.

If the predicate cannot be re-written, consider using a computed column to pre-evaluate the function. Creating an index on the computed column will make any predicate that uses the function SARGable and allow SQL Server to calculate statistics for the results of the function, improving cardinality estimates.

## LIKE with a leading wildcard

### Non-SARGable

```
SELECT *  
FROM Person.Person  
WHERE FirstName LIKE '%B%';
```

### SARGable

```
SELECT *  
FROM Person.Person  
WHERE FirstName LIKE 'B%';
```



LIKE without a leading wildcard can be executed as a range scan, so this is SARGable. With a leading wildcard, SQL Server needs to scan every value. Computed columns can be used in this case to pre-evaluate the LIKE condition, but in most cases using a full-text index will work best.

## OR in the WHERE clause

```
SELECT CustomerID, OrderDate,  
       ShipDate, [Status]  
FROM Sales.SalesOrderHeader  
WHERE SalesPersonID = 277  
OR CustomerID = 29523;
```

An OR in the WHERE clause can also impact cardinality estimates and SARGability if the OR condition connects predicates on two different fields in the table or in different tables. Often rewriting the query as two separate queries with a UNION operator (or UNION ALL if there's no possibility of duplicates) can be much more efficient.

## OR in the WHERE clause

```
SELECT CustomerID, OrderDate,  
       ShipDate, [Status]  
FROM Sales.SalesOrderHeader  
WHERE SalesPersonID = 277  
UNION  
SELECT CustomerID, OrderDate,  
       ShipDate, [Status]  
FROM Sales.SalesOrderHeader  
WHERE CustomerID = 29523;
```



An OR in the WHERE clause can also impact cardinality estimates and SARGability if the OR condition connects predicates on two different fields in the table or in different tables. Often rewriting the query as two separate queries with a UNION operator (or UNION ALL if there's no possibility of duplicates) can be much more efficient.

## Composable Logic – The All-Purpose Query

```
CREATE PROCEDURE usp_GetSalesPersonOrders @SalesPerson  
INT NULL AS  
SELECT SalesOrderID,  
       p.FirstName AS SalesFirstName,  
       p.LastName AS SalesLastName  
FROM Sales.SalesOrderHeader AS soh  
LEFT JOIN Person.Person AS p  
       ON soh.SalesPersonID = p.BusinessEntityID  
WHERE @SalesPerson IS NULL  
OR SalesPersonID = @SalesPerson;
```



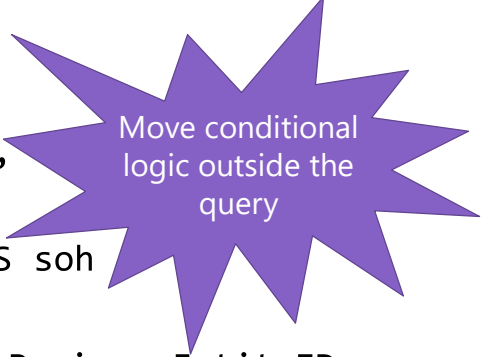
## Composable Logic – The All-Purpose Query

```
CREATE PROCEDURE usp_GetSalesPersonOrders @SalesPerson  
INT NULL AS
```

```
IF @SalesPerson IS NULL
```

```
    SELECT SalesOrderID,  
    p.FirstName AS SalesFirstName,  
    p.LastName AS SalesLastName  
    FROM Sales.SalesOrderHeader AS soh  
    LEFT JOIN Person.Person AS p  
        ON soh.SalesPersonID = p.BusinessEntityID;
```


```
ELSE ...
```



Move conditional  
logic outside the  
query

## Composable Logic – The All-Purpose Query

```
DECLARE @sql nvarchar(max);  
SET @sql = 'SELECT SalesOrderID, p.FirstName AS  
SalesFirstName, p.LastName AS SalesLastName  
FROM Sales.SalesOrderHeader AS soh  
LEFT JOIN Person.Person AS p  
    ON soh.SalesPersonID = p.BusinessEntityID';  
IF @SalesPerson IS NOT NULL  
    SET @sql = @sql + 'WHERE SalesPersonID = @p1';  
EXEC sp_executesql @stmt = @sql, @params = N'@p1 INT',  
@p1 = @SalesPerson;
```



Or use  
dynamic  
SQL

# Table-valued Functions

Works like a parameterized view

```
SELECT EmployeeID,  
       FirstName,  
       LastName,  
       JobTitle,  
       RecursionLevel  
FROM dbo.ufn_FindReports(25);
```

A table-valued function is a function which returns a table as a result. You can use table-valued functions to encapsulate query logic (similar to a view) and to simplify queries, but if used in the wrong way, they can cause inefficient query plans to be created. An inline TVF is a function that has a single SELECT statement and returns a resultset directly to the caller. Another way to write a TVF would be to have more complicated logic that inserts values into a table variable and returns that table to the caller as a return value. Functions like this can cause problems when used in a query because SQL Server cannot accurately estimate the cardinality. Joining to an atomic TVF like this requires SQL Server to execute the entire function first, put the results into tempdb and then join them with the rest of the query plan. Not only can this be expensive, it's also difficult to determine the true cost of the query because the cost of the function will not be included in the total cost of the overall query and also won't be reflected in the query plan when you review it. With inline functions, SQL Server can treat them like a view. They can be expanded, simplified and joined into the overall query tree, and thus their cost will be accurately estimated and reflected in the query plan. If you choose to use table-valued functions, be sure to use inline functions if possible.

*Inline User-Defined Functions*

[http://technet.microsoft.com/en-us/library/ms189294\(v=sql.105\).aspx](http://technet.microsoft.com/en-us/library/ms189294(v=sql.105).aspx)

# Table-valued Functions

Can be a multi-statement TVF (MSTVF)

```
CREATE FUNCTION dbo.ufn_FindReports (@InEmpID INT)
RETURNS @retFindReports TABLE (
    EmployeeID int primary key NOT NULL,
    FirstName nvarchar(255) NOT NULL,
    LastName nvarchar(255) NOT NULL,
    JobTitle nvarchar(50) NOT NULL,
    RecursionLevel int NOT NULL ) AS
BEGIN [multiple statements]
```


```
CREATE OR ALTER FUNCTION dbo.ufn_FindReports (@InEmpID INT)
RETURNS @retFindReports TABLE
(
    EmployeeID int primary key NOT NULL,
    FirstName nvarchar(255) NOT NULL,
    LastName nvarchar(255) NOT NULL,
    JobTitle nvarchar(50) NOT NULL,
    RecursionLevel int NOT NULL
)
--Returns a result set that lists all the employees who report to the
--specific employee directly or indirectly.*/
AS
BEGIN
WITH EMP_cte(EmployeeID, OrganizationNode, FirstName, LastName, JobTitle, RecursionLevel) -- CTE
name and columns
AS (
    -- Get the initial list of Employees for Manager n
    SELECT e.BusinessEntityID, e.OrganizationNode, p.FirstName, p.LastName, e.JobTitle, 0
    FROM HumanResources.Employee e
    INNER JOIN Person.Person p
    ON p.BusinessEntityID = e.BusinessEntityID
    WHERE e.BusinessEntityID = @InEmpID
    UNION ALL
    -- Join recursive member to anchor
    SELECT e.BusinessEntityID, e.OrganizationNode, p.FirstName, p.LastName, e.JobTitle, RecursionLevel +
```

```
1
    FROM HumanResources.Employee e
        INNER JOIN EMP_cte
            ON e.OrganizationNode.GetAncestor(1) = EMP_cte.OrganizationNode
    INNER JOIN Person.Person p
        ON p.BusinessEntityID = e.BusinessEntityID
    )
-- copy the required columns to the result of the function
INSERT @retFindReports
SELECT EmployeeID, FirstName, LastName, JobTitle, RecursionLevel
FROM EMP_cte
RETURN
END;
GO
```

## Table-valued Functions

Can be an inline TVF

```
CREATE FUNCTION  
dbo.ufn_FindReports (@InEmpID int)  
RETURNS TABLE AS  
RETURN  
[single query]
```



Always use Inline  
TVFs if possible

But SQL Server 2019 and Azure SQL will inline MSTVF automatically

## Demo

Detecting Anti-Patterns in a Query Plan

See <https://github.com/Microsoft/tigertoolbox/tree/master/Sessions/Build-2019> for demo files.

Continue learning  
with our new book!

<https://aka.ms/LearnTSQLQuerying>

Check out other  
great data-related  
demos here:

<https://aka.ms/DataDemos>

<https://aka.ms/IQPDemos>

# Learn T-SQL Querying

A guide to developing efficient and elegant T-SQL code



Pedro Lopes and Pam Lahoud

**Packt**  
www.packt.com





# Free Technical Webinar

## Modernizing Your Data Platform: May 9, 11:00 EST

The end of support for SQL Server 2008 is almost here, join leading data experts including Buck Woody and Bob Ward to walk through the steps to modernization. Hosted by PASS, a global community of data professionals and presented by Microsoft and Intel®, sessions include:

- Azure Data Estate Modernization
- Microsoft SQL Server 2019 Big Data Clusters Architecture
- Intelligent Query Processing in SQL Server 2019
- Modernizing with Intel Technologies
- Experience SQL Server 2019 on Linux and Containers

Register Now

<https://www.pass.org/Modernization.aspx>



Thank you!



@SQLGoddess

THANKS



@SQLPedro



© Copyright Microsoft Corporation. All rights reserved.