Оглавление

[Основы индексов в SQL Server 2](#_Toc128854995)

[Структура индекса 2](#_Toc128854996)

[Кластеризованный индекс 3](#_Toc128854997)

[Некластеризованный индекс 3](#_Toc128854998)

[Колонный (Columnstore) индекс 4](#_Toc128854999)

[Alter index Reorganize vs Rebuild 4](#_Toc128855000)

[Как сделать UNION для различного количества столбцов? 4](#_Toc128855001)

[Function vs Stored procedure 4](#_Toc128855002)

[CTE (Common table expression) 5](#_Toc128855003)

[Use cases 6](#_Toc128855004)

[Recursive CTE 7](#_Toc128855005)

[Views 8](#_Toc128855006)

[Benefits of views 8](#_Toc128855007)

[VIEWS vs WITH 8](#_Toc128855008)

[Types of Triggers 8](#_Toc128855009)

[1. **DDL Triggers** 9](#_Toc128855010)

[2. **DML Triggers** 9](#_Toc128855011)

[1. **After Trigger (using FOR/AFTER CLAUSE)** 9](#_Toc128855012)

[2. **Instead of Trigger (using INSTEAD OF CLAUSE)** 9](#_Toc128855013)

[3. **CLR Triggers** 9](#_Toc128855014)

[4. **Logon Triggers** 10](#_Toc128855015)

[Temp tables 10](#_Toc128855016)

[Usage: 10](#_Toc128855017)

[Temp tables vs Views 11](#_Toc128855018)

[Variable tables 11](#_Toc128855019)

[Temp vs Var tables 12](#_Toc128855020)

[Global tables 12](#_Toc128855021)

[Performance 13](#_Toc128855022)

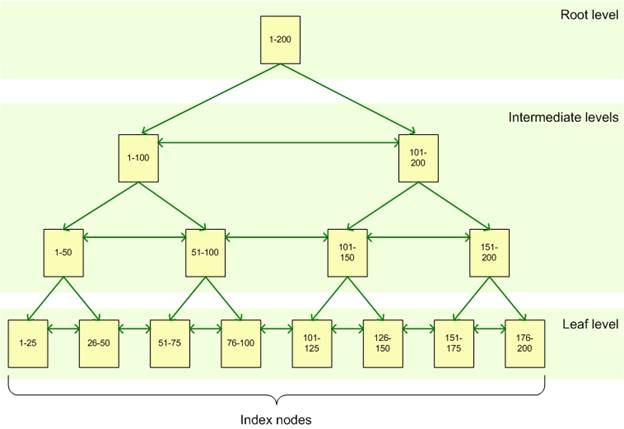
[When “NOT NULL” can be NULL 13](#_Toc128855023)

[Truncate vs Delete 13](#_Toc128855024)

Основы индексов в SQL Server

Структура индекса

Индексы создаются для столбцов таблиц и представлений. Индексы предоставляют путь для быстрого поиска данных на основе значений в этих столбцах. Например, если вы создадите индекс по первичному ключу, а затем будете искать строку с данными, используя значения первичного ключа, то *SQL Server* сначала найдет значение индекса, а затем использует индекс для быстрого нахождения всей строки с данными. Без индекса будет выполнен полный просмотр (сканирование) всех строк таблицы, что может оказать значительное влияние на производительность.  
Вы можете создать индекс на большинстве столбцов таблицы или представления. Исключением, преимущественно, являются столбцы с типами данных для хранения больших объектов (*LOB*), таких как *image*, *text*или *varchar(max)*. Вы также можете создать индексы на столбцах, предназначенных для хранения данных в формате *XML*, но эти индексы устроены немного иначе, чем стандартные. *Сolumnstore*индексы – так же отдельная история.

Индекс состоит из набора страниц, узлов индекса, которые организованы в виде древовидной структуры — *сбалансированного дерева*. Эта структура является иерархической по своей природе и начинается с корневого узла на вершине иерархии и конечных узлов, листьев, в нижней части, как показано на рисунке:  


Когда вы формируете запрос на индексированный столбец, подсистема запросов начинает идти сверху от корневого узла и постепенно двигается вниз через промежуточные узлы, при этом каждый слой промежуточного уровня содержит более детальную информацию о данных. Подсистема запросов продолжает двигаться по узлам индекса до тех пор, пока не достигнет нижнего уровня с листьями индекса. К примеру, если вы ищете значение 123 в индексированном столбе, то подсистема запросов сначала на корневом уровне определит страницу на первом промежуточном (intermediate) уровне. В данном случае первой страница указывает на значение от 1 до 100, а вторая от 101 до 200, таким образом подсистема запросов обратится ко второй странице этого промежуточного уровня. Далее будет выяснено, что следует обратиться к третьей странице следующего промежуточного уровня. Отсюда подсистема запросов прочитает на нижнем уровне значение самого индекса. Листья индекса могут содержать как сами данные таблицы, так и просто указатель на строки с данными в таблице, в зависимости от типа индекса: кластеризованный индекс или некластеризованный.

Кластеризованный индекс

Кластеризованный индекс хранит реальные строки данных в листьях индекса. Возвращаясь к предыдущему примеру, это означает что строка данных, связанная со значение ключа, равного 123 будет храниться в самом индексе. Важной характеристикой кластеризованного индекса является то, что все значения отсортированы в определенном порядке либо возрастания, либо убывания. Таким образом, таблица или представление может иметь только один кластеризованный индекс. В дополнение следует отметить, что данные в таблице хранятся в отсортированном виде только в случае если создан кластеризованный индекс у этой таблицы.  
Таблица не имеющая кластеризованного индекса называется кучей.

Некластеризованный индекс

В отличие от кластеризованного индекса, листья некластеризованного индекса содержат только те столбцы (*ключевые*), по которым определен данный индекс, а также содержит указатель на строки с реальными данными в таблице. Это означает, что системе подзапросов необходима дополнительная операция для обнаружения и получения требуемых данных. Содержание указателя на данные зависит от способа хранения данных: кластеризованная таблица или куча. Если указатель ссылается на кластеризованную таблицу, то он ведет к кластеризованному индексу, используя который можно найти реальные данные. Если указатель ссылается на кучу, то он ведет к конкретному идентификатору строки с данными. Некластеризованные индексы не могут быть отсортированы в отличие от кластеризованных, однако вы можете создать более одного некластеризованного индекса на таблице или представлении, вплоть до 999. Это не означает, что вы должны создавать как можно больше индексов. Индексы могут как улучшить, так и ухудшить производительность системы. В дополнение к возможности создать несколько некластеризованных индексов, вы можете также включить дополнительные столбцы (*included column*) в свой индекс: на листьях индекса будет храниться не только значение самих индексированных столбцов, но и значения этих не индексированных дополнительных столбцов. Этот подход позволит вам обойти некоторые ограничения, наложенные на индекс. К примеру, вы можете включить неидексируемый столбец или обойти ограничение на длину индекса (900 байт в большинстве случаев).

Колонный (Columnstore) индекс

Индекс применяется для больших хранилищ данных. Он разбивает данные на группы колонок одного типа и делает индекс по каждой из колонок внутри группы.   
После создания НЕКЛАСТЕРИЗОВАННОГО колонного индекса на таблицу, вставка в нее невозможна.  
КЛАСТЕРИЗОВАННЫЙ колонный индекс позволяет проводить вставку, но другие индексы создать нельзя.   
(ограничения до SQL Server 2014)  
Дает преимущество в скорости поиска.

Покрывающий (covering) индекс

In SQL Server, a covering index is an index that contains all of the columns needed to satisfy a query, so that the query can be fully resolved using only the index, without having to access the table itself.

When a query is executed against a table, SQL Server uses indexes to speed up the process of finding the relevant data. If a query needs to access only a few columns of a table, SQL Server can use a narrow index to quickly find the relevant rows and then retrieve only the necessary columns from the table. However, if the query needs to access many columns or perform a calculation, SQL Server may need to retrieve the data from the table, which can be slow if the table is large.

A covering index can help to speed up these types of queries by including all of the columns needed by the query in the index itself, so that SQL Server can find and return the necessary data without having to access the table. This can be especially useful for queries that involve large tables or complex calculations.

To create a covering index in T-SQL, you can use the CREATE INDEX statement with the INCLUDE clause to specify the additional columns to include in the index. For example:

CREATE NONCLUSTERED INDEX IX\_CoveringIndex ON MyTable (Column1, Column2) INCLUDE (Column3, Column4);

This creates a nonclustered index called IX\_CoveringIndex on the MyTable table, with Column1 and Column2 as the key columns, and Column3 and Column4 as the included columns. When a query is executed that needs to access these columns, SQL Server can use this index as a covering index to speed up the query.

Alter index Reorganize vs Rebuild

REORGANIZE is an online operation that defragments leaf pages in a clustered or non-clustered index page by page using little extra working space.

REBUILD is an online operation in Enterprise editions, offline in other editions, and uses as much extra working space again as the index size. It creates a new copy of the index and then drops the old one, thus getting rid of fragmentation. Statistics are recomputed by default as part of this operation, but that can be disabled.

Cross Join «другими словами»

SELECT ColumnName\_1,

ColumnName\_2,

ColumnName\_N

FROM [Table\_1]

CROSS JOIN [Table\_2]

==

SELECT ColumnName\_1,

ColumnName\_2,

ColumnName\_N

FROM [Table\_1],[Table\_2]

Как сделать UNION для различного количества столбцов?

You would want to select columns as NULL to take up for the empty space in certain tables.

Table A: (id, column1)

Table B: (id, column1, column2)

Select id, column1, null as column2 from tableA

UNION

Select id, column1, column2 from tableB

Function vs Stored procedure

Functions are calculated values that cannot make lasting modifications to SQL Server's environment (i.e., no INSERT or UPDATE statements allowed).  
  
If a function provides a scalar value, it may be used inline in SQL queries; if it returns a result set, it can be joined on.  
  
Functions must return a value and cannot change the data they receive as parameters, as defined by computer science (the arguments). Functions can't modify anything and must have at least one parameter. They also have to return a result. Stored procedures don't need a parameter, may modify database objects, and don't have to return a result.  
  
Stored procedures are used to connect SQL queries in a transaction and to communicate with the outside world. ADO.NET and other frameworks can't call a function directly, but they can call a stored procedure.

CTE (Common table expression)

A common table expression (CTE) in SQL is a named temporary result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. CTEs are similar to derived tables or subqueries, but they are defined prior to the execution of the main query and can be referred to multiple times within the main query.

Here's an example of how you might use a CTE in T-SQL:

WITH temp\_table(id, name, parent\_id) AS

(

SELECT id, name, parent\_id

FROM Employee

WHERE parent\_id IS NULL

)

SELECT \* FROM temp\_table

UNION ALL

SELECT e.id, e.name, e.parent\_id

FROM Employee e

JOIN temp\_table t ON e.parent\_id = t.id

In this example, the CTE named "SalesCTE" is defined as the result of a query that selects the product and the sum of sales from the SalesTable, grouped by product. The main query then selects the product and sales from the CTE where sales are greater than 1000.

Another example:

WITH RecursiveCTE (id, name, parent\_id) AS

(

SELECT id, name, parent\_id

FROM Employee

WHERE parent\_id IS NULL

UNION ALL

SELECT e.id, e.name, e.parent\_id

FROM Employee e

JOIN RecursiveCTE rcte ON e.parent\_id = rcte.id

)

SELECT \* FROM RecursiveCTE;

In this example, the CTE named "RecursiveCTE" is defined as the result of a recursive query that selects the id, name and parent\_id from Employee table where parent\_id is null and then it's joined with itself to get all the child and grandchild etc. under the parent\_id.

What is the point of using CTE in the second example? How would it look like without CTE?

In the second example, the CTE named "RecursiveCTE" is used to recursively query data from the Employee table. The CTE is used to organize and simplify the query by breaking it down into a series of smaller, more manageable steps. Without the CTE, the query would likely be longer, more complex, and harder to read.

Here's an example of how the query would look like without CTE:

WITH temp\_table(id, name, parent\_id) AS

(

SELECT id, name, parent\_id

FROM Employee

WHERE parent\_id IS NULL

)

SELECT \* FROM temp\_table

UNION ALL

SELECT e.id, e.name, e.parent\_id

FROM Employee e

JOIN temp\_table t ON e.parent\_id = t.id

In this example, a temporary table is created and a UNION statement is used to combine the data in the temporary table with the data in the Employee table where the parent\_id matches. The query becomes more complex, less readable and harder to understand as the recursion depth increases.

It's worth noting that the CTEs are only visible to the query in which they are defined, once the query is finished, they are discarded and don't affect the database. They are also memory-bound and not stored on disk, so they are useful for queries that return a small number of rows.

## Use cases

1. Break down a complex query into smaller, more manageable steps. CTEs allow you to organize a complex query into a series of smaller, simpler queries, making the overall query easier to read and understand.
2. Reference the same subquery multiple times within a single query. CTEs can be referred to multiple times within the main query, avoiding repeating the same subquery multiple times.
3. Use recursion in your query. CTEs can be defined recursively, making it easy to query hierarchical or recursive data.
4. Improve the performance of your query. CTEs can improve query performance by allowing the query optimizer to work with smaller, simpler subqueries, resulting in faster query execution.
5. Improve readability and maintainability of your code. By breaking down complex queries into smaller, more manageable steps, CTEs make it easier for developers to understand and maintain the code.

It's worth noting that CTEs are not suitable for large datasets, since they are memory-bound and not stored on disk. In those cases, other alternatives like temporary tables, table variables, or subqueries should be considered.

The key words for defining a common table expression (CTE) in SQL are:

1. **WITH**: This keyword is used at the beginning of a CTE definition to indicate that a CTE is being defined.
2. **AS**: This keyword is used to separate the CTE name and definition.
3. **(CTE\_name)**: This is the name of the CTE. It should be unique within the query and can be used to reference the CTE in the main query.
4. **SELECT**: This keyword is used to define the query that composes the CTE.
5. **UNION**, **UNION ALL**, **INTERSECT** and **EXCEPT** : These are used to combine the results from multiple CTEs and can be used to create advanced queries.

## Recursive CTE

In a recursive common table expression (CTE) in SQL, you can have one or more **UNION** and/or **UNION ALL** statements. However, the recursive query, which defines how to move from one level of the hierarchy to the next, should use only **UNION ALL** instead of **UNION**.

The reason for this is that a **UNION** statement removes duplicates from the result set, which can cause the recursion to stop prematurely. On the other hand, a **UNION ALL** statement includes all rows in the result set, regardless of whether they are duplicates. This is important when recursing through a hierarchy, as it ensures that all levels of the hierarchy are included in the final result set.

Here's an example of how you might use multiple **UNION ALL** statements in a recursive CTE:

WITH EmployeeCTE (id, name, manager\_id) AS

(

SELECT id, name, manager\_id

FROM Employee

WHERE manager\_id IS NULL

UNION ALL

SELECT e.id, e.name, e.manager\_id

FROM Employee e

JOIN EmployeeCTE ec ON e.manager\_id = ec.id

UNION ALL

SELECT e.id, e.name, e.manager\_id

FROM Employee e

JOIN EmployeeCTE ec ON e.manager\_id = ec.id

WHERE e.status = 'active'

)

SELECT \* FROM EmployeeCTE;

In this example, the CTE named "EmployeeCTE" is defined as the result of a recursive query that starts by selecting the id, name, and manager\_id of the employees who have no manager (manager\_id is null) and then it's joined with itself to get all the employee and their manager and so on under the manager\_id and also it's joined with itself again to get all the employees whose status is active.

It's important to note that using multiple **UNION ALL** statements in a recursive CTE can make the query more complex and harder to understand, so you should use them carefully and only when necessary.

Views

A view in a database is a virtual table that represents data from one or more underlying tables. Both the SELECT statement and the WITH statement can be used to query views.

The SELECT statement is used to retrieve data from one or more tables in a database. The results of a SELECT statement can be used in subsequent operations or stored in a new table.

## Benefits of views

Using views in a database can provide several benefits:

1. Abstraction: Views can provide a level of abstraction over the underlying tables, allowing for a simpler interface to be presented to the end-users or other systems accessing the database.
2. Simplification of complex queries: Complex database queries can be simplified by encapsulating them within a view. This can make it easier to work with the data and improve maintainability.
3. Security: Views can be used to restrict access to specific columns or rows of data, providing a level of security and protecting sensitive information.
4. Performance: Views can be optimized and used as a means of improving query performance by reducing the amount of data returned or aggregating data.

Overall, views can improve the functionality, security, and performance of a database and simplify the process of working with data.

## VIEWS vs WITH

The WITH statement, also known as a Common Table Expression (CTE), is used to define a temporary, named result set that can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement. Unlike a view, a CTE is only valid for the duration of a single query and does not persist as a named object in the database.

In summary, views provide a permanent, named representation of a subset of data in a database that can be used in multiple SELECT statements. CTEs are a way to define a temporary result set within a single query and are not stored in the database.

# Types of Triggers

In SQL Server we can create four types of triggers Data Definition Language (DDL) triggers, Data Manipulation Language (DML) triggers, CLR triggers, and Logon triggers.

## **DDL Triggers**

In SQL Server we can create triggers on DDL statements (like CREATE, ALTER, and DROP) and certain system-defined stored procedures that perform DDL-like operations.

Example: If you are going to execute the CREATE LOGIN statement or the sp\_addlogin stored procedure to create login user, then both these can execute/fire a DDL trigger that you can create on CREATE\_LOGIN event of SQL Server.

We can use only FOR/AFTER clause in DDL triggers not INSTEAD OF clause means we can make only After Trigger on DDL statements.

DDL trigger can be used to observe and control actions performed on the server, and to audit these operations. DDL triggers can be used to manage administrative tasks such as auditing and regulating database operations.

## **DML Triggers**

In SQL Server we can create triggers on DML statements (like INSERT, UPDATE, and DELETE) and stored procedures that perform DML-like operations. DML Triggers are of two types

## **After Trigger (using FOR/AFTER CLAUSE)**

This type of trigger fires after SQL Server finishes the execution of the action successfully that fired it.

Example: If you insert record/row in a table then the trigger related/associated with the insert event on this table will fire only after the row passes all the constraints, like as primary key constraint, and some rules. If the record/row insertion fails, SQL Server will not fire the After Trigger.

## **Instead of Trigger (using INSTEAD OF CLAUSE)**

This type of trigger fires before SQL Server starts the execution of the action that fired it. This differs from the AFTER trigger, which fires after the action that caused it to fire. We can have an INSTEAD OF insert/update/delete trigger on a table that successfully executed but does not include the actual insert/update/delete to the table.

Example: If you insert record/row in a table then the trigger related/associated with the insert event on this table will fire before the row passes all the constraints, such as primary key constraint and some rules. If the record/row insertion fails, SQL Server will fire the Instead of Trigger.

## **CLR Triggers**

CLR triggers are a special type of triggers based on the CLR (Common Language Runtime) in .net framework.

We coded the objects (like trigger) in the CLR that have heavy computations or need references to objects outside the SQL Server. We can write code for both DDL and DML triggers, using a supported CLR language like C#, Visual Basic and F#. I will discuss CLR trigger later.

## **Logon Triggers**

Logon triggers are a special type of trigger that fire when LOGON event of SQL Server is raised. This event is raised when a user session is being established with SQL Server that is made after the authentication phase finishes, but before the user session is actually established. Hence, all messages that we define in the trigger such as error messages, will be redirected to the SQL Server error log. Logon triggers do not fire if authentication fails. We can use these triggers to audit and control server sessions, such as to track login activity or limit the number of sessions for a specific login.

Syntax for Logon Trigger

CREATE TRIGGER trigger\_name

ON ALL SERVER

[WITH ENCRYPTION]

{FOR|AFTER} LOGON

AS

sql\_statement [1...n ]

Temp tables

created in the tempdb database and work similarly to regular tables.

Use # before name to create temp table

They can have a defined schema, constraints and can be queried, updated and manipulated using T-SQL. Only available to the session that created them and are automatically dropped when the session ends. (A session in SQL Server refers to a series of interactions between a client and the database server. It starts when a client connects to the database and ends when the client disconnects.)

SELECT

product\_name,

list\_price

INTO #trek\_products

FROM

production.products

WHERE

brand\_id = 9;

or

CREATE TABLE #haro\_products (

product\_name VARCHAR(MAX),

list\_price DEC(10,2)

);

## Usage:

1. Storing intermediate results of complex queries: Temporary tables can be used to store the intermediate results of a complex query and then used to join with other tables to get the final result.
2. Simplifying complex queries: Temporary tables can be used to simplify complex queries by breaking them down into smaller and more manageable parts.
3. Improving performance: Temporary tables can be used to improve the performance of complex queries by reducing the amount of data that needs to be processed.
4. Implementing session-level logic: Temporary tables can be used to store data that is specific to a single user session and is not needed once the session ends.
5. Storing temporary data for batch processes: Temporary tables can be used to store temporary data for batch processes, such as data that is being processed in stages, or data that is being used to populate other tables.
6. Implementing pagination: Temporary tables can be used to implement pagination by storing the result set of a query and then retrieving a specified number of rows at a time.

## Temp tables vs Views

A view, on the other hand, is a virtual table that doesn't physically exist but rather represents the result of a SELECT statement. Views do not store data, they simply reference the underlying tables and the data is retrieved dynamically when the view is queried. Views can be used to simplify complex queries and provide a level of abstraction over the underlying tables. Unlike temporary tables, views are persistent and are available to multiple sessions as long as they have the necessary permissions.

In some cases, it is possible to use views instead of temporary tables to achieve the same result. For example, views can simplify complex queries by breaking them down into smaller and more manageable parts, just like temporary tables. However, views are not stored in the database and are instead constructed dynamically each time they are queried. This can result in longer execution times for complex views compared to temporary tables.

In terms of performance, temporary tables can provide a performance boost in certain situations. For example, when processing large amounts of data, creating a temporary table can allow the database engine to cache the data, which can improve query performance compared to repeatedly executing the same complex query. However, this is dependent on the specifics of each scenario and the performance benefits of using temporary tables over views can vary.

In summary, whether to use views or temporary tables depends on the specific requirements of the task at hand, including the size and complexity of the data, the required performance, and the need for persistence. Both views and temporary tables have their own strengths and limitations, and the best choice will depend on the specific use case.

Variable tables

A way of storing temporary data in the form of a table. Used to store intermediate results or data that needs to be used across multiple statements in a single batch or stored procedure.

stored in memory

accessible within the scope in which they are declared, and their data is lost when the scope is exited, for example, when the stored procedure or batch in which they are declared is completed.

Variable tables are created using the DECLARE statement and can be used in the same way as a regular table. For example:

DECLARE @temp\_table TABLE (

column1 INT,

column2 VARCHAR(50)

);

INSERT INTO @temp\_table (column1, column2)

VALUES (1, 'A'), (2, 'B'), (3, 'C');

SELECT \* FROM @temp\_table;

In this example, a variable table named @temp\_table is created with two columns, column1 and column2. Data is then inserted into the table and the contents can be selected and used just like a regular table.

## Temp vs Var tables

1. Persistence: Temporary tables are only available to the session that created them and are automatically dropped when the session ends, while table variables are only accessible within the context of a single query and are automatically deleted when the query execution is completed.
2. Performance: Table variables are typically faster than temporary tables, especially for smaller datasets, as they are stored in memory, while temporary tables are stored on disk. However, as the data size of a table variable increases, its performance degrades relative to that of a temporary table.
3. Indexing: Temporary tables can be indexed, while table variables cannot be indexed, which can affect query performance.

In general, you should use a table variable when the data you need to store is small and will not be used by multiple queries, and use a temporary table when you need to store a large amount of data that will be used within a single session.

Global tables

Global tables in SQL Server refer to a table that is replicated across multiple databases in different instances of SQL Server. These tables are used to maintain data consistency and availability in a multi-database environment.

Here are the steps to create and use global tables in SQL Server:

1. Create a table in a database: Create a table with the desired columns and data types in the source database.
2. Configure replication: Set up replication between the source database and the target databases using either SQL Server Management Studio or Transact-SQL scripts.
3. Create a Global Table: In the source database, create a Global Table from the existing table by using the CREATE GLOBAL TEMPORARY TABLE statement.
4. Insert data into the Global Table: Insert data into the Global Table in the source database, which will be replicated to the target databases.
5. Query data from the Global Table: Query data from the Global Table in any of the target databases.

It's important to note that global tables are only accessible within the same instance of SQL Server and cannot be used to share data between instances. Additionally, data consistency and availability should be monitored and managed to ensure that all replicas are up to date.

## Performance

Using global tables in SQL Server can impact performance in several ways:

1. Network Latency: Replicating data between databases in different instances can result in increased network latency, which can impact the overall performance of the system.
2. Increased Resource Utilization: The replication process itself can consume system resources, such as CPU, memory, and I/O, which can impact the performance of other database operations.
3. Data Consistency: Global tables rely on transactional replication to maintain data consistency across replicas, which can result in increased lock contention and deadlocks, especially if the tables are frequently updated.
4. Query Performance: Querying global tables can be slower compared to querying local tables due to the overhead of communicating with the source database.

To mitigate the performance impact of using global tables, it's important to properly tune the replication process and monitor its performance, as well as optimizing query performance by using appropriate indexes and query optimization techniques. Additionally, careful consideration should be given to the data design, to minimize the amount of data that needs to be replicated and ensure that only the necessary data is included in the global tables.

When “NOT NULL” can be NULL

A column with a "NOT NULL" constraint is designed to contain a non-NULL value. However, there are some specific cases in Microsoft SQL Server where a column with a "NOT NULL" constraint can still contain NULL values:

1. System generated columns: System generated columns, such as a computed column or an identity column, can contain NULL values before their values are generated.
2. Table variables: Table variables in SQL Server can contain NULL values, even if the corresponding column in a table has a "NOT NULL" constraint.
3. Data import: When importing data into a table with a "NOT NULL" constraint, the data import process may result in NULL values in columns that are supposed to have a "NOT NULL" constraint.
4. Out-of-band updates: Out-of-band updates, such as direct updates to the underlying table using T-SQL, can result in NULL values in columns with a "NOT NULL" constraint.
5. Circular references: Circular references between tables can result in NULL values in columns with a "NOT NULL" constraint.
6. Database replication: In a replicated database environment, changes made to data on one server may not be immediately propagated to other servers, resulting in temporary NULL values in columns with a "NOT NULL" constraint.

Truncate vs Delete

TRUNCATE and DELETE are two T-SQL commands used to remove data from a database table.

TRUNCATE:

* Truncates all data from a table, including all rows and all data in the columns.
* Is a DDL (Data Definition Language) statement.
* Faster than DELETE because it only deallocates data pages and does not log individual row deletions.
* Cannot be rolled back.
* Automatically resets identity column values.
* Does not fire triggers.

DELETE:

* Deletes selected rows or all rows from a table.
* Is a DML (Data Manipulation Language) statement.
* Slower than TRUNCATE because it logs each row deletion, which makes it slower in large data sets.
* Can be rolled back.
* Does not reset identity column values.
* Fires triggers.

Use cases:

TRUNCATE: Use when you want to remove all data from a table and reset the table to its initial state.

DELETE: Use when you want to remove specific rows from a table.

TRUNCATE removes all data from a table regardless of constraints such as foreign key constraints or unique constraints. However, if the table being truncated is referenced by a foreign key constraint in another table, the truncate operation will fail unless the foreign key constraint is first disabled or dropped.

TRANSACTION

CREATE PROCEDURE p\_name

@Param nvarchar(20)

AS

BEGIN TRY

BEGIN TRANSACTION

<SELECT, INSERT, UPDATE...>

COMMIT TRANSACTION

END TRY

BEGIN CATCH

ROLLBACK TRANSACTION

END CATCH

SELECT \* FROM t\_name

WHERE id = @Param

CURSOR

DECLARE db\_cursor CURSOR FOR

SELECT name

FROM MASTER.dbo.sysdatabases

WHERE name NOT IN ('master','model','msdb','tempdb')

OPEN db\_cursor

FETCH NEXT FROM db\_cursor INTO @name

WHILE @@FETCH\_STATUS = 0

BEGIN

SET @fileName = @path + @name + '\_' + @fileDate + '.BAK'

BACKUP DATABASE @name TO DISK = @fileName

FETCH NEXT FROM db\_cursor INTO @name

END

CLOSE db\_cursor

DEALLOCATE db\_cursor