Contenido

[OPTIMIZACION Y BUENAS PRACTICAS SOBRE CONSULTAS 2](#_Toc79478968)

[1. Utilice WHERE en lugar de HAVING para definir filtros 2](#_Toc79478969)

[2. Utilice comodines (%) solo al final de una frase 2](#_Toc79478970)

[3. Utilice LIMIT o TOP para muestrear los resultados de la consulta 3](#_Toc79478971)

[4. Se debe evitar la utilización de “SELECT \*” 3](#_Toc79478972)

[5. Utilización de NOLOCK 3](#_Toc79478973)

[6. Ejecute su consulta durante las horas de menor actividad. 3](#_Toc79478974)

[7. Evite utilizar el prefijo “sp\_” en los procedimientos almacenados 4](#_Toc79478975)

[8. Implemente “set nocount on” en sus consultas 4](#_Toc79478976)

[9. Utilice sp\_executesql en lugar execute 5](#_Toc79478977)

[10. Siempre que sea posible se debe evitar el uso de cursores 5](#_Toc79478978)

[11. Características de las variables tipo tabla, tablas temporales & expresiones de tabla comunes 5](#_Toc79478979)

[ **Variables de tabla** 6](#_Toc79478980)

[ **Tabla temporal Local** 6](#_Toc79478981)

[ **Tabla temporal global** 6](#_Toc79478982)

[ **Características generales de las tablas temporales** 6](#_Toc79478983)

[ **Expresiones de tabla comunes (CTE)** 6](#_Toc79478984)

[12. Sugerencias para aplicar índices de forma optimizada 7](#_Toc79478985)

[ **Tipos de índices son soportados en SQL Server** 7](#_Toc79478986)

[13. No se recomienda el uso Subconsulta 8](#_Toc79478987)

[14. Tipos de join en sql 8](#_Toc79478988)

[15. Plan de ejecución 10](#_Toc79478989)

[ **¿Qué es un plan de ejecución de consultas de SQL Server?** 10](#_Toc79478990)

[ **Definir objetivos de rendimiento** 11](#_Toc79478991)

[ **¿Cómo recompilar planes de consulta en SQL Server?** 11](#_Toc79478992)

[16. Operaciones básicas del plan de ejecución de SQL Server 11](#_Toc79478993)

[ **Operaciones de acceso a tabla y índice:** 12](#_Toc79478994)

[ **Operaciones de unión (join)** 12](#_Toc79478995)

[ **Ordenar y agrupar** 12](#_Toc79478996)

[ **Sentencias Top-N** 12](#_Toc79478997)

[17. RECOMENDACIONES ADICIONALES 12](#_Toc79478998)

# OPTIMIZACION Y BUENAS PRACTICAS SOBRE CONSULTAS

## Utilice WHERE en lugar de HAVING para definir filtros

las instrucciones HAVING se calculan después de las instrucciones WHERE. Si la intención es filtrar una consulta según una condicion, una declaración WHERE es más eficiente.

Por ejemplo: supongamos que se han realizado 200 ventas en el año 2016 y queremos consultar el número de ventas por cliente en 2016.

SELECT Customers.CustomerID, Customers.Name, Count(Sales.SalesID)  
FROM Customers  
   INNER JOIN Sales  
   ON Customers.CustomerID = Sales.CustomerID  
GROUP BY Customers.CustomerID, Customers.Name  
HAVING Sales.LastSaleDate BETWEEN ‘1/1/2016’ AND ‘12/31/2016’

Esta consulta extraería 1,000 registros de ventas de la tabla Sales, luego filtraría los 200 registros generados en el año 2016 y finalmente contabilizaría los registros en el conjunto de datos.

En comparación, las cláusulas WHERE limitan el número de registros extraídos:

SELECT Customers.CustomerID, Customers.Name, Count(Sales.SalesID)  
FROM Customers  
  INNER JOIN Sales  
  ON Customers.CustomerID = Sales.CustomerID  
WHERE Sales.LastSaleDate BETWEEN ‘1/1/2016’ AND ‘12/31/2016’  
GROUP BY Customers.CustomerID, Customers.Name

Esta consulta extraería los 200 registros del año 2016 y luego contabilizaría los registros en el conjunto de datos. El primer paso de la cláusula HAVING se ha eliminado por completo.

HAVING solo debe usarse cuando se filtra en un campo agregado. En la consulta anterior, también podríamos filtrar por clientes con más de 5 ventas usando una declaración HAVING.

SELECT Customers.CustomerID, Customers.Name, Count(Sales.SalesID)  
FROM Customers  
  INNER JOIN Sales  
  ON Customers.CustomerID = Sales.CustomerID  
WHERE Sales.LastSaleDate BETWEEN ‘1/1/2016’ AND ‘12/31/2016’  
GROUP BY Customers.CustomerID, Customers.Name

HAVING Count(Sales.SalesID) > 5

## Utilice comodines (%) solo al final de una frase

Cuando se utiliza un comodín inicial en combinación con un comodín final, la base de datos tiene la tarea de buscar en todos los registros una coincidencia en cualquier lugar dentro del campo seleccionado creando una búsqueda lo más amplia posible. Sin embargo, la búsqueda más amplia es también puede ser la búsqueda más ineficiente.

Por ejemplo, necesito extraer toda las ciudades que comienzan con 'Char'

SELECT City

FROM Customers  
WHERE City LIKE ‘%Char%’

Esta consulta traerá todos los resultados que comiencen con 'Char’ Sin embargo, también obtendrá resultados inesperados como los resultados que finalicen o contenga la sigla 'Char' ; Una consulta más eficiente sería:

SELECT City

FROM Customers  
WHERE City LIKE ‘Char%’

## Utilice LIMIT o TOP para muestrear los resultados de la consulta

Antes de ejecutar una consulta por primera vez, asegúrese de que los resultados sean deseables y significativos utilizando una declaración LIMIT . (En algunos sistemas DBMS, se utiliza la sentencia TOP ). La instrucción LIMIT devuelve solo el número de registros especificado y evita a la base de datos tenga que realizar una gran consulta y nos sirve para realizar un muestreo sobre un conjunto de datos específico.

## Se debe evitar la utilización de “SELECT \*”

Al ejecutar consultas exploratorias, muchos desarrolladores suelen utilizar **SELECT \*** para traer todos los campos, esto debe omitirse ya que impide el uso efectivo de los índices, aparte, si una tabla tiene muchos campos y muchas filas, esta práctica consume los recursos de la base de datos al consultar una gran cantidad de datos innecesarios.

En lugar de consultar la totalidad de los datos o campos, se deben especificar solamente los campos que serán necesario traerse. En caso de tener que utilizar **todos los campos de la tabla**, se debe realizar ALT+F1 sobre el nombre de la tabla seleccionada, esta acción traerá la estructura, luego copiar, pegar los campos y añadir una “,” al final del campo

Estructura optimizada para verificar la existencia de un dato :

IF EXISTS (SELECT 1 FROM  Tabla WHERE Campo=Condicionante)

BEGIN

--INSTRUCCIONES

END

## Utilización de NOLOCK

El uso de **NOLOCK** puede mejorar considerablemente la velocidad de algunas consultas. Al usar **NOLOCK** en las consultas se establece que la lectura no “Bloquea” la tabla y esta puede ser leída al mismo tiempo por otros usuarios.

Sin embargo, hay que tener en cuenta que el conjunto de datos mostrados es considerado como una “dirty read” (significa que los datos mostrados podrían encontrarse en medio de alguna transacción “DELETE, UPDATE o INSERT”)

Aun así, para el caso de tablas estáticas como pueden ser tablas catálogos, informacion histórica o que no se verá afectada en el momento, debiera ser una mejora aceptable sobre todo si son consultas frecuentes en un ambiente multiusuario, pero se debe evaluar cada escenario en particular.

**NOLOCK** se usa al momento de efectuar una consulta y solo aplica a una tabla. Su estructura seria la siguiente:

SET NOCOUNT ON

SELECT Campo1, Campo2,… CampoN

FROM Tabla1  T1(NOLOCK)

INNER JOIN Tabla2 T2(NOLOCK) ON T2.CampoX=T1.CampoX

INNER JOIN Tabla3 T3(NOLOCK) ON T3.CampoY=T2.CampoY

WHERE <Condicionantes>

SET NOCOUNT OFF

## Ejecute su consulta durante las horas de menor actividad.

Para minimizar el impacto de sus consultas en la base de datos de producción, hable con un DBA sobre la programación de la consulta para que se ejecute en horas de menor actividad.

Por lo general la consulta que consumen gran cantidad de recursos se debe ejecutar cuando los usuarios concurrentes sean bajos. Alguno de los criterios para identificar si una consulta debe ejecutar en horario de menor recurrencia pueden ser

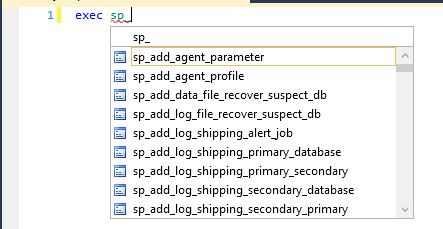
* Selección de tablas grandes (> 1,000,000 de registros)
* Uniones cartesianas o uniones cruzadas
* Utilización de Cursores
* Si tiene Subconsultas anidadas con gran cantidad de datos
* Búsquedas con comodines en campos de texto extensos o notas
* Consulta de múltiples esquemas
* Utilización de tablas temporales con gran volumen de datos

## Evite utilizar el prefijo “sp\_” en los procedimientos almacenados

Se debe evitar el prefijo “***sp\_***” para etiquetar procedimientos almacenados, ya que **SQL Server** reconoce el prefijo como “**System Stored Procedure**”, es decir, un procedimiento almacenado de Sistema.

Cuando se ejecuta el **Stored Procedure** el motor de SQL Server asume que se trata de un procedimiento almacenado de sistema, y por lo tanto Intenta primeramente localizar el procedimiento en la base de datos **MASTER**. Al no encontrarlo continua su búsqueda la base de datos activa, provocando con esto una caída del rendimiento, si bien puede parecer insignificante en un ambiente de transacciones pequeño podría influir significantemente en ambiente mucho más grande en donde se ejecuten miles de transacciones por minuto.

Ejemplo de **System Stored Procedure**



## Implemente “set nocount on” en sus consultas

Al ejecutar sentencias en un procedimiento almacenado, el servidor SQL regresa un mensaje indicando el número de filas afectadas por el proceso. Aunque esta información puede ser útil para depurar el código, resulta obsoleta si no se está depurando; Al establecer el uso de **SET NOCOUNT ON**, se desactiva la función de retorno de esta información adicional.

En procedimientos almacenados formados por varias instrucciones o declaraciones de sentencias SQL (Querys), habilitar **SET NOCOUNT****ON** puede ayudarnos a conseguir un aumento de rendimiento considerable, además, si tomamos en cuenta que una base de datos puede tener decenas de Procedimientos Almacenados (SP) un detalle tan sencillo como este puede aportar una diferencia notable.

CREATE PROC dbo.StoreProcedureExample

(@variable\_1 INT, @variable\_2 VARCHAR(10), @variable\_n BIT)

AS BEGIN

SET NOCOUNT ON;

--INSTRUCCIÓN

SET NOCOUNT OFF;

END

## Utilice sp\_executesql en lugar execute

Se recomienda el uso de **sp\_executesql,** en lugar de la instrucción **EXECUTE o EXEC**al **ejecutar código dinámico**. Esto debido a que **sp\_executesql** admite la sustitución de parámetros, es mucho más versátil que **EXECUTE**y además genera planes de ejecución. Aparte **sp\_executesql** ofrece las siguientes ventajas adicionales:

* Como el texto real de la instrucción Transact-SQL no cambia entre una ejecución y otra, el optimizador de consultas debe comparar la instrucción Transact-SQL de la segunda ejecución con el plan de ejecución generado para la primera instrucción. De este modo, SQL Server no tiene que compilar la segunda instrucción.
* El archivo de datos tiene formato nativo. No es necesaria la conversión a Unicode
* Genera planes de ejecución con más probabilidades de que SQL Server los vuelva a utilizar (es más eficaz)
* Se utiliza para la creación de SQL dinámicos

Codigo de ejemplo

DECLARE @SqlStatment AS NVARCHAR(1000)

DECLARE @ColNames AS NVARCHAR(100)

SET @ColNames = N' Campo\_1, Campo\_2 , Campo\_3';

SET @SqlStatment = 'SELECT ' + @ColNames + ' FROM [Tabla] WHERE [Condicion]=@PerType'

EXECUTE sp\_executesql @SqlStatment , N'@PerType nchar(2)', @PerType='EM'

## Siempre que sea posible se debe evitar el uso de cursores

Los cursores son una herramienta usada para acceder y modificar el resultado de una cláusula SELECT fila por fila. El problema es que su uso consume una enorme cantidad de recursos, especialmente de memoria, esto ocurre porque los sistemas de administración de bases de datos relacionales (RDBMS), están optimizado para operaciones basadas en conjuntos.

**Siempre que sea posible, se debe omitir el uso de cursores o** minimizar su uso. Algunas alternativas y sustitutos al uso de cursores pueden ser:

* Usar ciclos WHILE.
* Uso de tablas derivadas
* Uso de subqueries correlacionados
* Uso de CASE.
* Uso de múltiples consultas.
* La combinación de todas las anteriores.

## Características de las variables tipo tabla, tablas temporales & expresiones de tabla comunes

### **Variables de tabla**

* Se crean como cualquier otra variable, utilizando la instrucción DECLARE.
* Su contenido no siempre está en memoria. En caso de que se inserte una cantidad grande de registros esta se almacena en TEMPDB.
* Solo son accesibles dentro de la sesión que las creó.
* Solo se puede acceder a la variable dentro de la consulta actual. No son visibles fuera de la consulta en ejecución.
* En lo que respecta al rendimiento son útiles con pequeñas cantidades de datos.
* Su uso en procedimientos almacenados provoca menos re-compilaciones.
* El contenido de la variable no es afectado por el comando ROLLBACK.
* No usan paralelismo (multiple threads) en su plan de ejecución.
* Su uso para tablas de concurrencia alta o con una gran cantidad de datos puede afectar su desempeño
* No se les puede agregar índices.
* No se les pueden modificar ni truncar una vez creadas.

### **Tabla temporal Local**

* Las tablas temporales locales de SQL Server se crean utilizando el símbolo hashtag (#) seguido del nombre de la tabla (**#Table\_name**) y solo es visible para la sesión actual. No puede ser vista y utilizada por procesos o consultas fuera de la sesión en la que esta se declara.

### **Tabla temporal global**

* Son muy útiles cuando desea que el conjunto de resultados sea visible para todas las demás sesiones; Al igual que las tablas temporales de Local SQL Server, se eliminan una vez que la sesión se desconecta y ya no hay más referencias a la tabla. Para crearla simplemente use los dos símbolos **##Global\_Table\_Name**.

### **Características generales de las tablas temporales**

* Se crea y almacena automáticamente en la base de datos TEMPDB.
* Su creación puede ocasionar bloqueos en las tablas sysobjects, sysindexes y afectar a todo el servidor.
* Permite el uso de índices.
* Su uso en procedimientos almacenados puede provocar una re-compilación continua.
* Pueden tratarse como una tabla normal (Con excepciones)
* Solo es visible para la sesión actual. No puede ser visto y tampoco utilizado por procesos o consultas fuera de la sesión en la que esta se declara y se descarta o destruye una vez que la sesión se desconecta
* No es necesario tratar de configurar los permisos
* Cualquiera puede insertar valores, modificar o recuperar registros de la tabla o eliminar la tabla

### **Expresiones de tabla comunes (CTE)**

* Una expresión CTE debe ir seguida de una única instrucción SELECT, INSERT, UPDATE o DELETE que haga referencia a una parte o a la totalidad de sus columnas
* Se puede hacer referencia a tablas de servidores remotos.
* Se pueden especificar varias definiciones de consulta de CTE en una CTE no recursiva.Deben combinarse mediante uno de estos operadores de conjuntos: UNION ALL, UNION, INTERSECT o EXCEPT.
* Se conservan los índices de la tabla.
* Una expresión CTE puede hacer referencia a ella misma y a otras expresiones CTE previamente definidas en la misma cláusula WITH.

Algunas restricciones comunes de las CTE son:

* No se puede utilizar ORDER BY (excepto cuando se especifica una cláusula TOP)
* No se puede utilizar INTO
* No se puede utilizar Cláusula OPTION con sugerencias de consulta
* No se permite especificar más de una cláusula WITH en una expresión CTE o anidada que defina otra expresión CTE.

## Sugerencias para aplicar índices de forma optimizada

La declaración y el uso de índices optimiza las consultas, ya que le indican al motor de la base de datos que esa columna o conjunto de columnas se van a utilizar con mayor frecuencia haciendo que el acceso a los datos sea más eficiente. Es aconsejable que los índices se formen sobre los campos que:

* + Son **PRIMARY KEY** o **FOREIGN KEY**.
  + Se usan frecuentemente para enlazar tablas con **JOIN**.
  + Se usan de forma habitual dentro de los procedimientos almacenados o en consultas emparejados con alguno de los siguientes comandos: **BETWEEN**, **ORDER BY**, **GROUP BY, TOP, DISTINCT**
  + Utilizados para el filtrados en la clausura **WHERE**.

Pese a que los índices generen una mejora en tiempos de respuesta, tampoco se puede abusar indiscriminadamente de ellos. En contraposición a su mejora de tiempo de respuesta cada índice ocupa un espacio equivalente al número de registros en la tabla, penalizando el rendimiento en la base de datos al ejecutar **INSERT, UPDATES y DELETE**.

Otra recomendación seria Omitir la creación de índices en campos que contienen pocos valores, como aquellos que son de tipo **BIT** o **CHAR** (de uno a cuatro).

### **Tipos de índices son soportados en SQL Server**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Categoría** | **Tipo** | **Versión SQL** | **Unique** | **Include** | **Filtered** | **Compressed** |
| **Basado en Filas** | Clustered | – | Disponible | – | – | Disponible |
| Non-Clustered | – | Disponible | Disponible | Disponible | Disponible |
| **Basado en Columnas** | ColumStore Clustered | +SQL 2014 | – | – | – | Automático |
| ColumnStore Non-Clustered | +SQL 2012 | – | – | Disponible | Automático |
| **Memoria** | Hash (In Memory) | +SQL 2014 | – | – | – | – |
| Non-Clustered Hash | +SQL 2014 | – | Disponible | Disponible | – |
| In-Memory ColumnStore | +SQL 2016 | – | – | – | – |
| **Especiales** | Spatial | +SQL 2008 | – | – | – | – |
| XML | +SQL 2008 | – | – | – | – |
| Full-Text |  | – | – | – | – |

## No se recomienda el uso Subconsulta

Las subconsultas crean una combinación cartesiana, también denominada producto cartesiano o combinación cruzada. Creando todas las combinaciones posibles de las variables.

En este ejemplo, si tuviéramos 1,000 clientes con 1,000 ventas totales, la consulta generaría primero 1,000,000 de resultados, luego filtraría por los 1,000 registros donde CustomerID está correctamente unido.

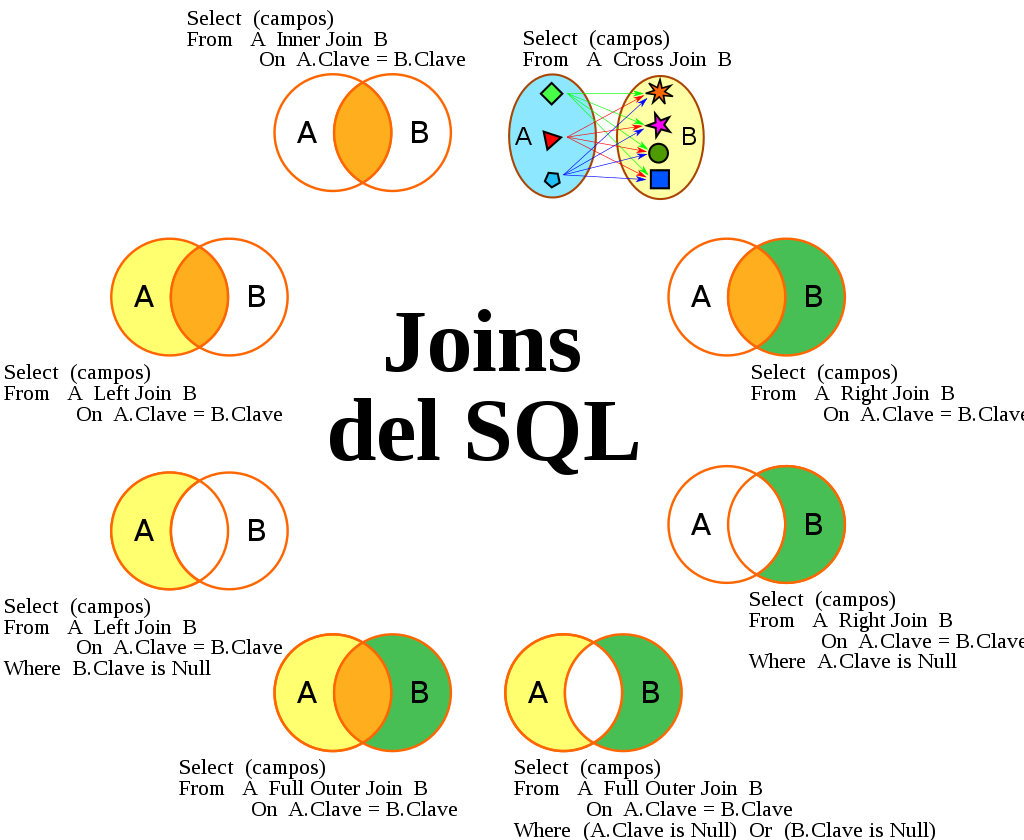
SELECT Customers.CustomerID, Customers.Name, Sales.LastSaleDate  
FROM Customers, Sales  
WHERE Customers.CustomerID = Sales.CustomerID

Este es un uso ineficiente de los recursos de la base de datos, ya que la base de datos ha realizado 100 veces más trabajo del necesario.

Las combinaciones cartesianas son especialmente problemáticas en las bases de datos a gran escala, porque una combinación cartesiana de dos tablas grandes podría generar miles de millones o billones de resultados. Para evitar la creación de una combinación cartesiana, utilice INNER JOIN en su lugar

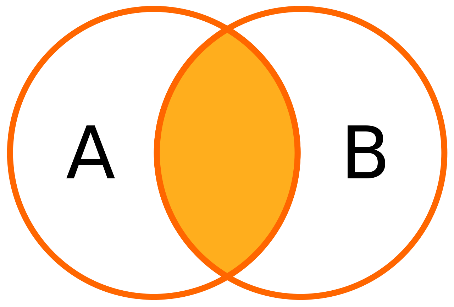
SELECT Customers.CustomerID, Customers.Name, Sales.LastSaleDate  
FROM Customers  
   INNER JOIN Sales  
   ON Customers.CustomerID = Sales.CustomerID

## Tipos de join en sql



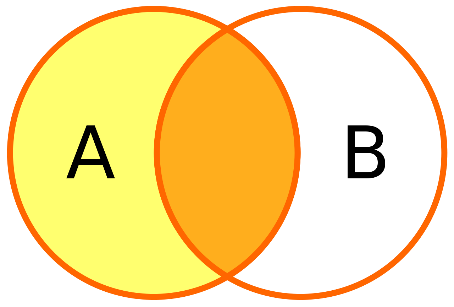
* **INNER JOIN:**

Con esta operación cada registro en la tabla A es combinado con los correspondientes de la tabla B que satisfagan las condiciones que se especifiquen en el predicado del JOIN. Cualquier registro de la tabla A o de la tabla B que no tenga uno correspondiente en la otra tabla es excluido.



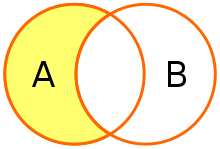
* **LEFT JOIN**

Retorna la pareja de todos los valores de la tabla izquierda con los valores de la tabla de la derecha correspondientes, si los hay, o retorna un valor nulo [NULL](https://es.wikipedia.org/wiki/NULL_(SQL)) en los campos de la tabla derecha cuando no haya correspondencia.



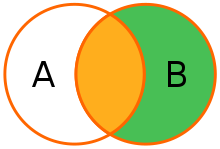
* **LEFT JOIN** (excluyendo la intersección):

Si se quieren mostrar solo los registros de la primera tabla que no tengan correspondientes en la segunda, se puede agregar la condición adecuada en la cláusula WHERE

[](https://commons.wikimedia.org/wiki/File:SQL_Join_-_02_A_Left_Join_B_Where_B.key_%3D_null.svg)

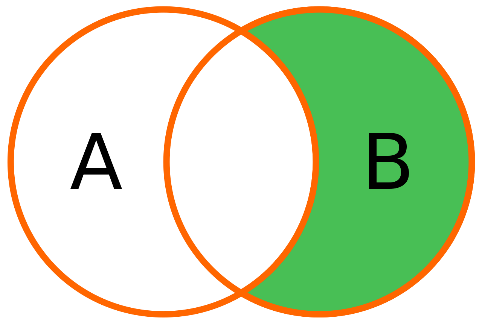
* + **RIGHT JOIN**

retorna todos los valores de la tabla derecha con los valores de la tabla de la izquierda correspondientes, si los hay, o retorna un valor nulo NULL en los campos de la tabla izquierda cuando no haya correspondencia.



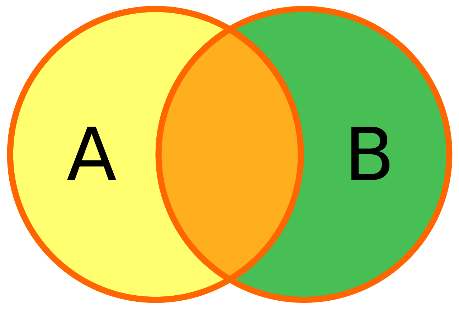
* **RIGHT JOIN** (*excluyendo la intersección)*

Si se quieren mostrar solo los registros de la tabla de Departamento que no tengan correspondientes en la tabla de Empleado, se puede agregar la condición adecuada en la cláusula WHERE.



* **FULL OUTER JOIN** (Combinación completa)

Esta operación presenta los resultados de tabla izquierda y tabla derecha, aunque alguna no tenga correspondencia en la otra tabla. La tabla combinada contendrá, entonces, todos los registros de ambas tablas y presentará valores nulos NULL para registros sin combinación.



## Plan de ejecución

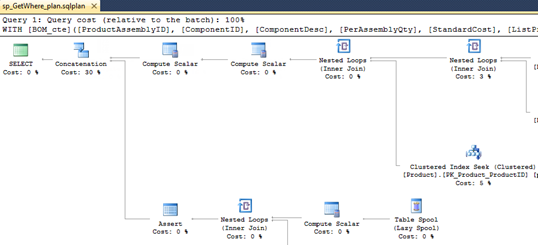
### **¿Qué es un plan de ejecución de consultas de SQL Server?**

El plan de ejecución es un algoritmo que muestra cómo se ejecutó o cómo se ejecutará una consulta, Cuando se visualiza se puede observar la cantidad de objetos involucrados de la base de datos, uniones, índices y su disponibilidad, la cantidad de columnas de salida y más, por lo tanto, se puede utilizar para solucionar o detectar problemas de rendimiento.

Cuando se ejecuta una nueva consulta, El Query Optimizer evalúa el plan de consulta, lo optimiza, lo compila y lo almacena en la caché. La caché es una parte del búfer del SQL Server donde los datos y los planes de consulta se almacenan pudiendo ser reutilizar más tarde.

Una característica muy útil de los planes es la siguiente, cuando se ejecuta un procedimiento almacenado (SP), el plan de ejecución se crea para el nombre del SP y se reutilizará siempre que se ejecute, Cuando se trata de consultas ad hoc, los planes de consulta se crean en base al código desarrollado, por lo que diferentes parámetros o cualquier cambio en el código evitarán la reutilización del plan existente.

Una buena práctica (siempre que sea posible) indica que aquellas consultas ad hoc recurrente se encapsulen en SP o funciones de esta forma los planes de consulta existentes se reutilizarán y, por lo tanto, el código se ejecute más rápido.

[](https://www.sqlshack.com/wp-content/uploads/2014/03/QueryPlan.png)

### **Definir objetivos de rendimiento**

La supervisión del rendimiento depende de los objetivos planteados, pero todos tienen una cosa en común: **garantizar el rendimiento óptimo** (a veces hay que compensar las necesidades y el costo para lograr este objetivo) algunas sugerencias son

* Analizar los recursos de hardware disponibles y su rendimiento, ya que pueden ralentizar significativamente todo el sistema.
* Definir el rendimiento óptimo de su servidor y crear tendencias basadas en la información de métricas capturadas. Si el rendimiento no es satisfactorio, es necesario identificar los problemas y encontrar su origen. Esto implica comprobar cómo las diferentes consultas afectan al rendimiento para estas tarea, el plan de ejecución del SQL es una excelente herramienta que puede ayudarlo a diagnosticar el problema.

### **¿Cómo recompilar planes de consulta en SQL Server?**

Los planes se vuelven a compilar automáticamente en caché, pero deja de ser válido u obsoleto si hay un cambio en la base de datos.

Las causas más comunes son cambios en los objetos utilizados, modificaciones del procedimiento almacenado, Modificaciones en la tabla o vista referenciada o índices utilizados por el plan, estadísticas, cambios significativos en los datos clave, etc.

Si detecta lentitud o falta de optimización en los tiempos de consultas se puede volver a compilar explícitamente el plan, ejecutando la cláusula WITH RECOMPILE o directamente se ejecuta el procedimiento almacenado sp\_recompile.

## Operaciones básicas del plan de ejecución de SQL Server

Los planes de ejecución se leen de derecha a izquierda, de arriba hacia abajo, y que es importante poder leer las operaciones que realizan y de ser necesario mejorarlas.

* **Operaciones de acceso a tabla y índice:**

las operaciones “Scan” leen el índice o la tabla entera mientras las operaciones “Seek” usan el B-tree o la dirección física para tener acceso a una parte específica del índice o de la tabla.

* **Index Seek, Clustered Index Seek** realiza un recorrido del B-tree y lee los nodos hoja para encontrar todas las entradas que coinciden.
* **Index Scan, Clustered Index Scan** Lee el índice entero (todas las filas) en el orden del índice. Dependiendo de consulta o SP la base de datos podría realizar esta operación si se necesitan todas las filas en el orden del índice, por ejemplo, debido a la cláusula del **order by.**
* **Key Lookup (Clustered**): Recupera una sola fila desde una agrupación de índice.
* **RID Lookup (Heap):** Recupera una sola fila desde una tabla.
* **Table Scan** Lee la tabla entera (todas las filas y las columnas) tal y como se almacenan sobre el disco es una de las operaciones más costosas, como el Table Scan debe leer todas las filas de la tabla puede consumir una grande cantidad de tiempo CPU.
* **Operaciones de unión (join)**

Por lo general, las operaciones de unión procesan sólo dos tablas a la vez. En el caso de que la sentencia tenga que unir más tablas, las operaciones se ejecutan de forma secuencial: primero las dos tablas y después el resultado intermedio con la siguiente tabla.

* **Nested Loops:** Unir dos tablas es buscar el resultado desde una tabla y después seleccionar la otra tabla por cada fila de la primera. SQL Server también utiliza la operación “nested loops” para recuperar datos de la tabla después de tener un acceso al índice
* **Hash Match:** La coincidencia hash de la unión carga los registros candidatos de un lado de la unión dentro de la tabla hash y después se comparan cada fila con el otro lado de la unión.
* **Merge Join:** Es la combinación de dos listas ordenadas.
* **Ordenar y agrupar**
* **Sort :** Ordenar el resultado de acuerdo con la cláusula order by. Esta operación necesita una gran cantidad de memoria para materializar el resultado intermedio
* **Stream Aggregate:** Agrega un conjunto preordenado de acuerdo con la cláusula group by. Esta operación no pone en memoria el resultado intermedio.
* **Hash Match (Aggregate):** Agrupa el resultado usando una tabla hash. Esta operación necesita gran cantidad de memoria para materializar el resultado intermedio
* **Sentencias Top-N**
* **Top:** Aborta las operaciones subyacentes cuando se alcanza el número de filas deseado. La eficiencia de la sentencia top-N depende del modo de ejecución de las operaciones subyacentes.

## RECOMENDACIONES ADICIONALES

* Si el procedimiento almacenado contiene demasiadas sentencias **IF-ELSE** para ejecutar distintos procesos acordes a ciertos parámetros, resulta más eficiente separar cada bloque y encapsularlos en procedimientos almacenados diferentes; Esto para evitar que el plan de ejecución cambie acorde al valor del resultado de la expresión **IF**, desperdiciando con esto la ventaja de el plan de ejecución en memoria y la pre-compilación.
* Evite el uso de la instrucción **SELECT-INTO**. Al ejecutarse se bloquean las tablas involucradas. Aplique en su lugar la sentencia **INSERT INTO-SELECT**.
* Trate de usar siempre la instrucción **JOIN** antes que cualquier **subconsulta.**
* Use la instrucción **BETWEEN** en lugar de **IN** siempre que sea posible.
* Evite en la medida de lo posible el uso de DISTINCT
* En caso de usar la instrucción **UNION** y existiera la seguridad de que en los SELECT involucrados no se obtendrán registros duplicados, entonces lo recomendable en este escenario es sustituir **UNION** por **UNION ALL** para evitar que se haga uso implícito de la instrucción **DISTINCT**, ya que esta aumenta el consumo de recursos.
* Evada siempre que sea posible el uso de **ORDER BY**. Al igual que **DISTINCT** consume una elevada cantidad de recursos. Considere si es realmente necesario usarlo o, si por otro lado se puede dejar el ordenamiento de los resultados a la aplicación que recibirá los datos.
* Use **SQL Profiler** y levante una traza para estudiar el rendimiento de los procedimientos almacenados de mayor concurrencia y vea opciones para refactorizar el proceso. Otorgue especial atención a las columnas **CPU**, **Duraction**,**reads** y **writes** para optimizar el uso de memoria y el I/O generado.
* Mantenga las transacciones lo más cortas posibles dentro de un procedimiento almacenado. Esto favorece a la reducción del número de bloqueos, promoviendo a acelerar el rendimiento general de la base de datos.
* Utilice el **Plan de Ejecución** para revisar sus consultas, entienda como **SQL Server** está ejecutando sus consultas y evalué sus sugerencias.
* Si requiere almacenar volúmenes de texto muy grandes, pero son menores a 8000 caracteres, use el tipo de dato **VARCHAR** en lugar de **TEXT**.
* Evalué cuidadosamente el uso de **CHAR** y **VARCHAR** dependiendo si el campo en el que se va a usar varía mucho o no de tamaño. Use **CHAR** para columnas de poca variación en longitud y **VARCHAR** para aquellas que no tienen una longitud estable o promedio.
* No use columnas con tipos de datos **FLOAT**, **REAL o DATETIME** como **FOREIGN KEY**.