

● Volumen I



Datalytics

MEJORES PRÁCTICAS SQL

EBOOK



datalytics.bi



datalyticsbi



Datalytics



Datalytics



○ INTRODUCCIÓN

SQL nace cuando –a partir del modelo relacional propuesto por Edgar Frank Codd en 1970– IBM crea SEQUEL en el año 1977. Pero habría que esperar dos años para que, recién en 1979, Oracle lo convirtiera en un producto comercial. En la actualidad SQL es el estándar de la inmensa mayoría de las bases de datos comerciales.

Su simplicidad, su estandarización y, sobre todo, su potencia, hacen que 40 años después, sigamos utilizando en entornos distribuidos, en cloud, en data warehouses, en data lakes e incluso en los nuevos lakehouses.

No importa en qué lenguaje desarrolles, SQL siempre te servirá para consultar una base de datos. Una buena sentencia SQL, te va a permitir hacer más eficiente tu proceso ya que traerá sólo los datos que necesites. Y ni hablemos si trabajas en áreas de BI o analítica, en ese caso, SQL será tu **mejor amigo**. Sabemos por qué lo decimos.

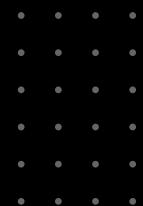


En estas páginas, queremos compartir algunos consejos sobre nuestras buenas prácticas usando SQL.

¡Esperamos que te sean de utilidad!

#MejorConDatos

ÍNDICE ÍNDICE ÍNDICE



- 
- NOMENCLATURAS
 - SINTAXIS
 - CLAVES SUBROGADAS
 - AGREGACIONES
 - FUNCIONES VENTANA
 - ÍNDICES
 - PARTICIONES
 - SUBCONSULTAS
 - ANÁLISIS DE PERFORMANCE (SELECT)
 - ANÁLISIS DE PERFORMANCE (WHERE)



I. NOMENCLATURAS

¿Para qué sirven?

A nadie le gusta leer una pared de texto plano. Cuando se trata de leer código, el problema es aún mayor. Saber usar bien las nomenclaturas nos permite: hacer menos esfuerzos para entender el código, mejorar su apariencia y hacer que el pase de proyectos a otros equipos sea más fluido.

Estas son algunas recomendaciones para trabajar con tablas y campos:

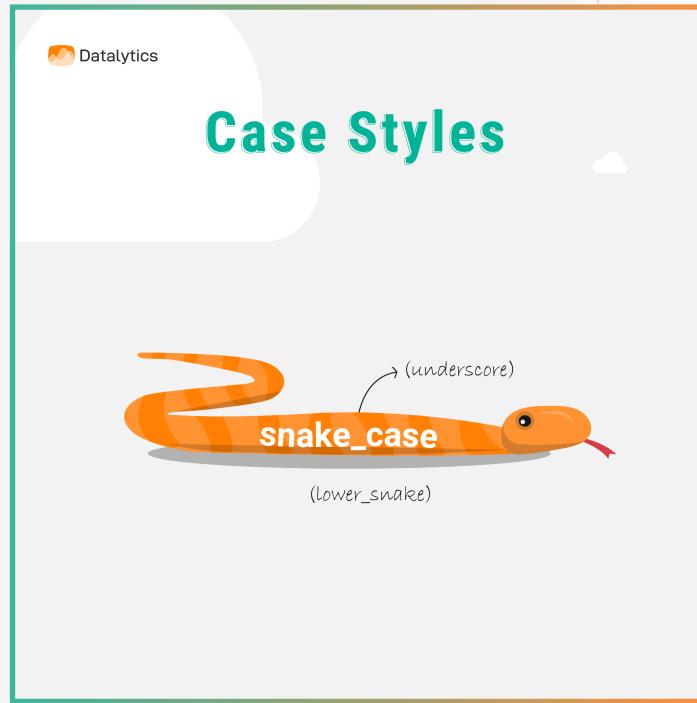
- Los nombres de las tablas son plurales, los campos singulares. 
- No tiene que haber espacios entre los nombres, la nomenclatura debe ser camelCase (lowerCamelCase).
- Si el nombre de la tabla tiene muchas palabras, sólo la última debería ser plural.
- Todas las tablas deben pertenecer a un esquema. Eviten usar los que son por defecto como “public” y recuerden separar las tablas por esquema.
- Si el motor no es case sensitive, utilicen underscore (snake_case).

Para las tablas:

- No usen prefijos innecesarios, por ejemplo tablaProductos.
- Usen nombres consistentes y bien definidos.
- La longitud del nombre de una tabla no debería superar los 30 caracteres.
- No usen tildes ni palabras reservadas.
- Eviten el uso de abreviaturas y, en caso de usarlas, tienen que ser claras.

Para los campos:

- Recuerden que los mismos lineamientos que se definieron para los nombres de las tablas aplican para los campos.
- Para los campos PK eviten el nombre id. Una buena práctica es combinar el nombre de la tabla con el prefijo id; por ejemplo idEmpleado.



II. SINTAXIS

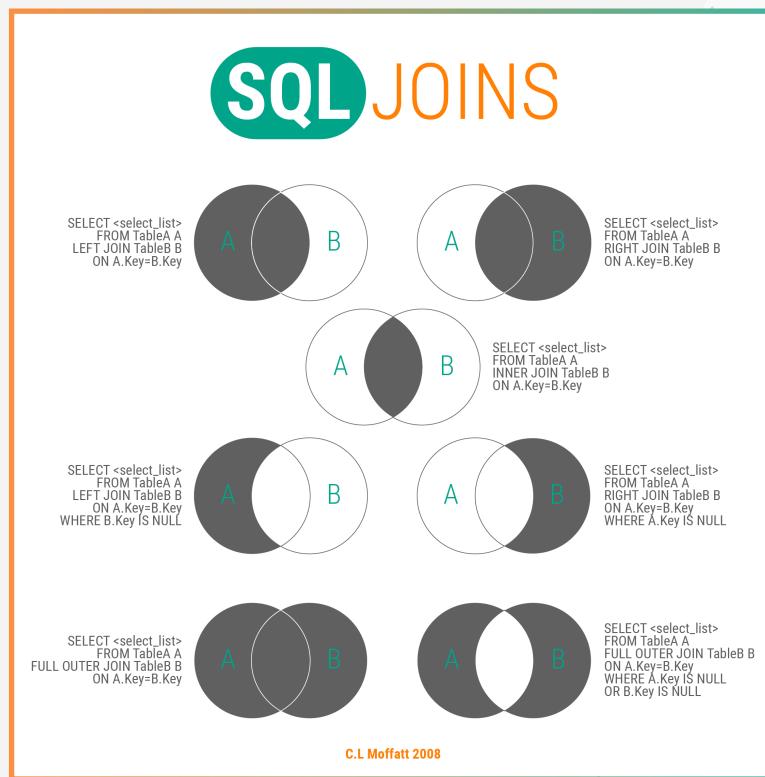
¿Para qué sirve?

La sintaxis permite autodocumentar los desarrollos de manera simple a partir de un código más fácil de leer y de entender. Además, en caso de tener errores, si nuestro código respeta estas buenas prácticas, va a ser más simple y rápido encontrarlos.

Algunos tips que no les pueden faltar:

- Colocar SOLO las columnas con las que van a trabajar. Eviten el uso de SELECT * 
- Existen joins explícitos y no implícitos. Si bien no hay diferencias en performance y resultados, les recomendamos siempre utilizar joins explícitos porque son más simples de leer, los otros tienden a confundir.
- Incluir AS al momento de utilizar alias, de esa manera, nuestras queries son más claras.
- Si bien es una buena práctica comentar las queries, hay que evitar que los comentarios sean muy extensos. En caso de incorporarlos, que sean siempre una línea nueva y no sobre la misma línea de nuestra query.
- Es muy aconsejable indentar el código, una línea por campo del SELECT, una para el FROM, para el JOIN, etc. Por si se pierden, abajo les dejamos un ejemplo de esto. 

También compartimos un recordatorio de los tipos de JOIN que les puede ayudar:



Sintaxis

QUERY EDITOR
QUERY HISTORY

```

1   SELECT p.PersonId,
2       p.FirstName,
3       p.LastName,
4       c.Name
5   FROM Person AS p
6   LEFT JOIN City AS c
7   ON p. CityId = c.CityId;

```

III. CLAVES SUBROGADAS

¿Qué hay que saber?

Lo más importante que tienen que saber es que nunca hay que confiar en las claves de origen, hay que manejarlas dentro del ETL o del data warehouse.

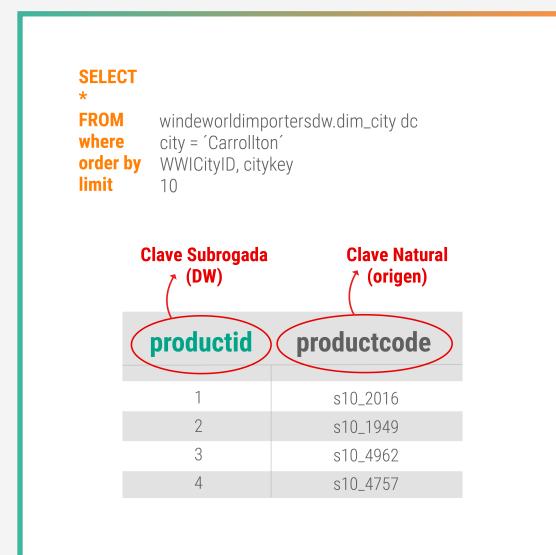
Ya sea en el data warehouse o en el data mart, hay que crear y usar claves subrogadas que se generen en forma automática desde el proceso de ETL o desde la base de datos.

Las subrogadas serán las claves primarias de las dimensiones y claves foráneas de las tablas de hechos.

Hacer esto es muy importante porque, básicamente, les va a ahorrar muchos problemas en el futuro. 

Al momento de trabajar con esto tengan en cuenta que:

- Las fuentes del data warehouse son heterogéneas y cada una tiene sus propias claves, por eso hay que generar una única clave subrogada desde el inicio.
- Puede haber cambios en las aplicaciones de origen.
- Es fundamental asegurar la integridad referencial en el ETL, tengan en cuenta que muchas de las tecnologías modernas de datawarehousing, no manejan el concepto de clave foránea.



```
SELECT
*
FROM
windeworldimportersdw.dim_city dc
where
city = 'Carrollton'
order by
WWICityID, citykey
limit
10
```

productid	productcode
1	s10_2016
2	s10_1949
3	s10_4962
4	s10_4757

IV. AGREGACIONES

¿De qué se tratan?

Es la posibilidad que nos da SQL de agrupar las filas como resultado de una consulta en conjuntos y de aplicar funciones sobre esos conjuntos de filas. Esto es muy importante porque, ¡es el primer paso para transformar los datos en información!

Estas son las funciones de agregación básicas:

- COUNT: devuelve el número total de filas seleccionadas por la consulta.
- MIN: devuelve el valor mínimo del campo que especifiquen.
- MAX: devuelve el valor máximo del campo que especifiquen.
- SUM: suma los valores del campo que especifiquen. Recuerden que sólo se puede utilizar en columnas numéricas. !
- AVG: devuelve el valor promedio del campo que especifiquen. También sólo se puede utilizar en columnas numéricas.

Además, tienen que saber que cada sistema ofrece su propio conjunto, más amplio, con otras funciones de agregación particulares.

Entonces, tengan en cuenta que:

- Todas estas funciones se aplican a una sola columna, que hay que especificar entre paréntesis.
- Excepto la función COUNT que se puede aplicar a una columna o indicar un “*”.

La diferencia entre uno y otro es que el primer caso no cuenta los valores nulos para dicha columna y en el segundo sí.



Por último, para poder hacer una operación sobre un conjunto de filas (y no sobre la totalidad), primero las tienen que agrupar:

- Utilicen la cláusula GROUP BY y tras ella, coloquen las columnas por las que van a agrupar.
- Esto se utiliza cuando seleccionan columnas múltiples desde una tabla y aparece al menos un operador aritmético en la instrucción SELECT.

Agregaciones

codProv	Provincia	ventas
17	Santa Fe	10
17	Santa Fe	20
18	Córdoba	30
18	Córdoba	40
18	Córdoba	40

Mostrar las ventas por provincia

Select codProv, Provincia,
sum(ventas) as ventas
From Ventas
Group by codProv, Provincia

codProv	Provincia	Ventas
17	Santa Fe	30
18	Córdoba	110

V. FUNCIONES VENTANA

¿Para qué sirven?

Evitan complejizar cualquier proceso de manipulación de datos ya que permiten resolver sumarizaciones con UNA sola línea de código.

- Permiten utilizar el resultado de otras filas.
- Se calculan POR FILA, NO son funciones de agregación (¡cuidado con esto!) 
- Son súper útiles para:
 - Calcular percentiles – NTILE.
 - Obtener el valor de filas anteriores o posteriores - LEAD y LAG.
 - Sumas acumulativas - SUM.
- ¿Cómo se usan? Tienen 3 partes:
 - Función a aplicar: NTILE, SUM, LEAD, LAG, etc.
 - Sobre qué aplicarlo: OVER (PARTITION BY x ORDER BY y).
 - Rango: RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING.
- Recuerden usar ORDER BY correctamente para evitar resultados erróneos.

Funciones ventana

```

WITH ventas_mes AS (
  SELECT
    dc.stateProvince,
    dd.calendarMonthNumber,
    dd.calendarMonthLabel,
    SUM(quantity*unitPrice) AS ventas
  FROM fact.Sale fs
  INNER JOIN Dimension.City dc ON (fs.cityKey = dc.cityKey)
  INNER JOIN Dimension.Date dd ON (fs.invoiceDateKey = dd.date)
  WHERE dd.calendarYear = 2016
  GROUP BY dc.stateProvince, dd.calendarMonthNumber, dd.calendarMonthLabel
)
  
```

SELECT

stateProvince **AS** estado,
 calendarMonthNumber **AS** mes,
 ventas **AS** ventaMes,

función ventana **divisiones** **orden**

función ventana **SUM(ventas) OVER (PARTITION BY stateProvince ORDER BY calendarMonthNumber)** ventaAcumulada,
función ventana **LAG(ventas 1) OVER (PARTITION BY stateProvince ORDER BY calendarMonthNumber)** ventaMesAnterior

FROM ventas_mes

estado	mes	ventaMes	ventaAcumulada	ventaMesAnterior
Alabama	1	123475.45	123475.45	NULL
Alabama	2	87047.10	210522.55	123475.45
Alabama	3	109683.00	320205.55	87047.10
Alabama	4	141620.20	461825.75	109683.00
Alabama	5	168068.40	629894.15	141620.20
Alaska	1	69987.30	69987.30	NULL
Alaska	2	68263.80	138251.10	69987.30
Alaska	3	80500.50	218751.60	68263.80
Alaska	4	50262.85	269014.45	80500.50
Alaska	5	97164.70	366179.15	50262.85

VI. ÍNDICES

¿Para qué se usan?

Es fácil, funcionan como los índices en un libro, se usan en bases relacionales y son estructuras ordenadas con referencias a los datos de la base.

Sirven para acelerar el tiempo de respuesta de las consultas y se definen usando una o más columnas de la tabla.

Les recomendamos usarlos cuando:

- Existe una primary key, en ese caso muchas veces se genera un índice implícito.
- Hay columnas que se usan en filtros, es decir dentro de un WHERE.
- Se utilizan agregaciones como max y min sobre una columna numérica.

Tengan en cuenta que, en contraposición a la mejora en el tiempo de respuesta, se penaliza el rendimiento en la base de datos al ejecutar INSERT, UPDATE y DELETE. 

Los índices no existen en el ámbito de data lakes, pero hay técnicas para emularlos. Si quieren saber cómo hacerlo acá hay varias pistas:

<https://bit.ly/2WJkwMW> 

Índices

SIN Índice

CON Índice

explain

```
select count(*)
from wideworldimportersdw.dim_city dc
where stateprovince = 'New York';
```

Result				Execution plan - 1			
Type	Name	Cost	Rows	Name	Value	Type	Name
✓ select		0	114.361	✓ General		✓ General	
table	dc (ALL)	0	114.361	Type	table	Type	table
				Name	dc (ALL)	Name	dc (ref)
				Cost	0	Cost	0
				Rows	114.361	Rows	4.951
				✓ Details		✓ Details	
				table_name	dc	table_name	dc
				access_type	ref	access_type	ref
				possible_keys	["dim_city_stateProvince_IDX"]	possible_keys	["dim_city_stateProvince_IDX"]
				key	dim_city_stateProvince_IDX	key	dim_city_stateProvince_IDX
				key_length	203	key_length	203
				used_key_parts	["stateProvince"]	used_key_parts	["stateProvince"]
				ref	["const"]	ref	["const"]
				row	4.951	row	4.951
				filtered	100	filtered	100
				attached_condition	dc.stateProvince = 'New York'	attached_condition	dc.stateProvince = 'New York'
				using_index	true	using_index	true

explain

```
select count(*)
from wideworldimportersdw.dim_city dc
where stateprovince = 'New York';
```

Statistics		Execution plan - 1		Name		Name	
Type	Name	Cost	Rows	Name	Value	Name	Value
✓ select		0	4.951	✓ General		✓ General	
table	dc (ref)	0	4.951	Type	table	Type	table
				Name	dc (ref)	Name	dc (ref)
				Cost	0	Cost	0
				Rows	4.951	Rows	4.951
				✓ Details		✓ Details	
				table_name	dc	table_name	dc
				access_type	ref	access_type	ref
				possible_keys	["dim_city_stateProvince_IDX"]	possible_keys	["dim_city_stateProvince_IDX"]
				key	dim_city_stateProvince_IDX	key	dim_city_stateProvince_IDX
				key_length	203	key_length	203
				used_key_parts	["stateProvince"]	used_key_parts	["stateProvince"]
				ref	["const"]	ref	["const"]
				row	4.951	row	4.951
				filtered	100	filtered	100
				attached_condition	dc.stateProvince = 'New York'	attached_condition	dc.stateProvince = 'New York'
				using_index	true	using_index	true

menos filas escaneadas

cambia tipo de acceso a la tabla usando el índice

VII. PARTICIONES

¿De qué se trata?

Partitionar es el proceso a partir del cual dividimos tablas muy grandes en múltiples partes más pequeñas. Con esto, las consultas son referenciadas a la tabla apropiada para aumentar la velocidad de recuperación. El objetivo principal es ayudar en el mantenimiento de tablas grandes y reducir el tiempo de respuesta.



Hay 3 tipos de particiones:

VERTICALES

- Son las divisiones de una tabla por columnas: un conjunto entra en un almacén de datos y otro conjunto lo hará en un data warehouse diferente.
- Los campos se dividen según su patrón de uso. Por ejemplo, podemos poner en una partición vertical aquellos a los que accedemos con mayor frecuencia y en otra, ubicar a los que utilizamos menos. De esta manera, también incrementamos el desempeño de las consultas.
- Además podemos utilizarlas para restringir el acceso a datos sensibles.

HORIZONTALES

- Son las divisiones de una tabla por filas: un conjunto de filas entra en un almacén de datos y otro conjunto lo hará en uno diferente.
- Dividen una tabla en múltiples tablas que contienen el mismo número de columnas (mismo schema en cada partición) pero menos filas.
- Permiten la referenciación de las consultas a la tabla apropiada, de esta manera limitan el acceso a lectura de volúmenes de datos muy grandes
- La clave de particionamiento seleccionada tiene que garantizar la creación de particiones uniformes para asegurar la eficiencia de la estructura creada. En general, es un tipo de dato fecha, aunque se pueden utilizar todos los tipos de datos válidos como índices.

Tengan en cuenta que los siguientes tipos de datos no pueden ser especificados: ntext, text, image, xml, varchar(max), nvarchar(max), o varbinary(max), columnas de tipo de datos de alias.



HÍBRIDAS

- Podemos combinar el particionamiento vertical y horizontal.

Les dejamos abajo tres ejemplos:

Particionamiento **vertical**

First Name	Last Name	Email	Thumbnail	Photo
David	Alexander	davida@contoso.com	3kb	3MB
Jarred	Carlson	jaredc@contoso.com	3kb	3MB
Sue	Charles	suec@contoso.com	3kb	3MB
Simon	Mitchel	simonm@contoso.com	3kb	3MB
Richard	Zeng	richard@contoso.com	3kb	3MB



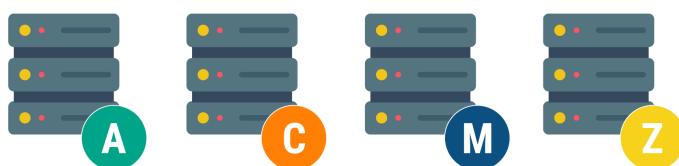
SQL DB



BLOBS

Particionamiento horizontal

First Name	Last Name	Email	Thumbnail	Photo
David	Alexander	davida@contoso.com	3kb	3MB
Jarred	Carlson	jaredc@contoso.com	3kb	3MB
Sue	Charles	suec@contoso.com	3kb	3MB
Simon	Mitchel	simonm@contoso.com	3kb	3MB
Richard	Zeng	richard@contoso.com	3kb	3MB



Particionamiento híbrido

First Name	Last Name	Email	Thumbnail	Photo
David	Alexander	davida@contoso.com	3kb	3MB
Jarred	Carlson	jaredc@contoso.com	3kb	3MB
Sue	Charles	suec@contoso.com	3kb	3MB
Simon	Mitchel	simonm@contoso.com	3kb	3MB
Richard	Zeng	richard@contoso.com	3kb	3MB



VIII. SUBCONSULTAS

¿Para qué sirven?

Sirven para obtener un resultado que se va a utilizar en una consulta principal. Además, en consultas complejas pueden utilizarse para ordenar el código y dividirlo en pasos lógicos.

Es importante que sepan cómo utilizarlas para evitar consultas con mala performance. Tengan en cuenta que las subconsultas se pueden aplicar en cualquier parte en la que se permita una expresión. !

Una subconsulta en SQL puede estar anidada en una instrucción SELECT, INSERT, UPDATE o DELETE, o bien en otra subconsulta.

Para que quede bien claro vamos a compartir algunos ejemplos de uso dentro de un SELECT:

Subconsultas en WHERE:

- El valor de comparación puede ser un valor único o un conjunto de valores. Tengan en cuenta este detalle ya que el tipo de operador a utilizar varía.
- Para valores únicos se puede utilizar un operador de comparación de carácter aritmético (=, <, >, etc.)
- Para valores múltiples uno de tipo lógico (IN).

Common Table Expressions (CTE):

- Permiten definir datasets a partir de subconsultas y utilizarlos en la consulta principal.
- Se usan para las consultas complejas que necesitan de varias subconsultas y también para organizar y darle legibilidad al código.

Subconsultas TABLAS TEMPORALES:

- Esta opción es para cuando es necesario reutilizar el resultado de una consulta como parte de otra más grande.
- Los datos persisten de forma temporal hasta que finaliza la sesión.
Pueden ser más performantes que una consulta con WITH y tengan cuidado con grandes volúmenes de información.

Les dejamos abajo algunos ejemplos de esto:

SubConsultas en WHERE

Valor Único

SELECT

```
CityKey, Quantity, UnitPrice  
FROM widerworldimportersdw.fact_sale  
Where Citykey = (  
    select max(CityKey) from widerworldsdw.dim_city dc  
    where dc.stateProvince = 'Texas' and dc.city= 'Port Alto'  
)
```

Subconsulta que obtiene el código de la ciudad Port Alto

_sale X	ECT CityKey, Quantity, UnitPrice FR	enter a SQL expression to filter results
123 CityKey ↴	123 Quantity ↴	123 UnitPrince ↴
83.491 ↴	6	13
83.491 ↴	5	13
83.491 ↴	5	13
83.491 ↴	2	13
83.491 ↴	10	13

SubConsultas en WHERE

Múltiples Valores

SELECT

```
CityKey, Quantity, UnitPrice
FROM widerworldimportersdw.fact_sale
Where Citykey in (
    select Citykey from widerworldsdw.dim_city dc
    where dc.stateProvince = 'Texas'
)
```

Subconsulta que obtiene los códigos de ciudades dentro del estado de Texas

_sale X		
ECT CityKey, Quantity, UnitPrice FR	23	enter a SQL expression to filter results
123 CityKey ↑	23 Quantity ↑	123 UnitPrice ↑
83.491 ↴	6	13
80.875 ↴	6	13
84.656 ↴	6	13
80.788 ↴	6	13
80.788 ↴	6	13

SubConsultas tablas temporales

```
CREATE TEMPORARY TABLE top_3_estados_mas_poblados as
select stateProvince, sum(latestRecordedPopulation)
from widerworldimportersdw.dim_city
group by 1 order by 2 desc
limit 3;
```

Al igual que el caso anterior, se obtienen los estados más poblados para calcular el promedio de ventas pero con una tabla temporal

-- promedio ventas en los 3 estados más poblados

```
SELECT
t3.stateProvince, avg(Quantity * UnitPrice)
FROM widerworldimportersdw.fact_sale fs
join widerworldimporets.dim_city dc on fs.CityKey = dc.cityKey
join top_3_estados_mas_poblados t3 on dc.stateProvince = t3.stateProvince
group by 1
```

_3_estados_mas_poblados X Statistics	
LECT t3.StateProvince,avg(Quantity)	Enter a SQL Expression to filter results (use Ctrl + Space)
ABC stateProvince ↑	123 avg(Quantity * UnitPrice) ↑
California	743,923044
New York	769,086424
Texas	756,820516

Common Table Expressions (CTE)

with

```
top_3_estados_mas_poblados as (
    select stateProvince, sum(latestRecordedPopulation)
    from wideworldimportersdw.dim_city
    group by 1 order by 2 desc
    limit 3
)
```

-- promedio ventas en los 3 estados más poblados

SELECT

```
t3.stateProvince,avg(Quantity * UnitePrice)
FROM wideworldimportersdw.fact_sale fs
join wideworldimportersdw.dim_city dc on fs.CityKey = dc.cityKey
join top_3_estados_mas_poblados t3 on dc.stateProvince = t3.stateProvince
group by 1
```

Obtiene los estados más poblados

Utiliza los estados más poblados para obtener el promedio de ventas

m_city X	
th top_3_Estados_mas_poblados as(Enter a SQL expression to filter results (use Ctrl+Space)
ABC StateProvince	123 avg(Quantity * UnitePrice)
California	743,923044
New York	769,086424
Texas	756,820516

IX. ANÁLISIS DE PERFORMANCE

1 / 2 (`select`)

¿Por qué es tan importante?

Detrás de todo lo que hacemos hay un motor que interpreta y procesa las consultas a partir de la calidad de la información que le damos, por lo tanto, cuanto mejor definamos las consultas, mejor será la performance.

Muchas veces les habrá pasado, o les va a pasar, que las personas propietarias de las aplicaciones piden aumentar los recursos del sistema (CPU y memoria) para mejorar el rendimiento. Estos recursos adicionales suelen tener un costo y lo más probable es que no sean del todo necesarios.

En la mayoría de los casos, estos problemas se resuelven con pequeñas mejoras que cambien el comportamiento de las consultas. ¡Por eso, es clave el uso de buenas prácticas para el desarrollo!

Si siguen los siguientes consejos, el Sistema Gestor de Bases de Datos (SGBD) va a responder mejor, más rápido y va a usar menos recursos, lo que se traduce en un menor costo. 

Les dejamos algunas recomendaciones para aplicar cuando escriban queries SQL.

En esta primera parte: SELECTs.

- No por básico menos importante: utilicen SELECT lista de columnas en lugar de SELECT *.

Supongamos que una tabla almacena datos para 290 personas empleadas y necesitamos recuperar la siguiente información: Employee National ID number, Birth Date, Gender, Hire date.

- Query ineficiente: si usan SELECT * statement, devolverá todos los registros de todas las columnas para las 290 personas. 
- Query eficiente: en lugar de *, definan las columnas específicas para la recuperación de los datos. 

Plan de ejecución: observar la diferencia en el tamaño de estimate row size para el mismo número de filas.

- No usen la cláusula INTO nombre_tabla (“SELECT... INTO”). Esto bloqueará las tablas del sistema mientras se ejecuta la consulta. En su lugar pueden crear primero las tablas y luego reescribir la sentencia como INSERT INTO nombre_tabla SELECT ...
!
- Promuevan el uso de EXISTS y NOT EXISTS, en lugar de IN y NOT IN. Más abajo les dejamos un ejemplo para que vean cómo se aplica.
!
- Si usan el operador UNION y tienen la seguridad que ambos select NO tienen registros duplicados, entonces es mejor usar UNION ALL. Esto es para evitar usar, en forma implícita, el operador DISTINCT que puede requerir almacenar todos los datos de salida en una tabla temporal para que luego se reordenen y se filtren los duplicados, esto aumenta considerablemente el costo de la consulta.
!
- Traten de usar lo menos posible el ORDER BY dentro de la consulta y dejarlo del lado del cliente o de la aplicación. Esta es una de las operaciones que más recursos consume en una consulta porque, una vez calculadas las filas resultantes, el SGBD debe comparar una por una para determinar su posición respecto al orden definido.
- Especificar siempre el alias de la tabla delante de cada campo definido en el select. Esto le ahorra al motor el tiempo de tener que buscar a qué tabla corresponde cada campo especificado.
- Siempre que sea posible, usen tablas derivadas ya que tienen un mejor desempeño. **Más abajo les dejamos un ejemplo.**
!

EXISTS y NOT EXISTS vs. IN y NOT IN

Primero tienen que encontrar el ID de producto de la tabla [Producción]. [Historial de transacciones] y luego buscar los registros correspondientes en la tabla [Producción]. [Producto].

```

1  Select * from [Production].[Product] p
2  Where Productid IN
3  (select productid from [AdventureWorks2019].[Production].[TransactionHistory]);
4  Go
    
```

En la siguiente query reemplazar el IN por la cláusula EXISTS.

```

1  Select * from [Production].[Product] p
2  Where EXISTS
3  (select productid from [AdventureWorks2019].[Production].[TransactionHistory])
    
```



EXISTS y NOT EXISTS vs. IN y NOT IN

Comparen las estadísticas después de ejecutar ambas consultas. La cláusula IN usa 504 escaneos, mientras que la cláusula EXISTS usa un escaneo para la tabla [Producción]. [TransactionHistory].

Row Num	Table	Scan Count	Logical Reads	Physical Reads	Read Ahead Reads	LOB Logical Reads	LOB Physical Reads	LOB REad-Ahead Reads	%Logical Reads of Total Reads
	TransactionHistory	504	1,101	0	0	0	0	0	98.656
	Product	1	15	0	0	0	0	0	1.344
	Total	505	1,116	0	0	0	0	0	

(1 row affected)

(504 rows affected)

Row Num	Table	Scan Count	Logical Reads	Physical Reads	Read Ahead Reads	LOB Logical Reads	LOB Physical Reads	LOB REad-Ahead Reads	%Logical Reads of Total Reads
	TransactionHistory	1	1,512	0	0	0	0	0	99.018
	Product	1	15	0	0	0	0	0	0.982
	Total	2	1,527	0	0	0	0	0	



Tablas derivadas

Menos performante

Considerando la siguiente consulta para encontrar el segundo salario más alto de la tabla de Empleados

```
SELECT MIN (Salary)  
FROM Employees  
Where EmpID IN (SELECT TOP 2 EmplD FROM Employees ORDER BY Salary DESC)
```

Más performante

```
SELECT MIN (A.Salary)  
FROM (SELECT TOP 2 Salary FROM Employees ORDER BY Salary DESC) AS A
```

La misma consulta puede ser reescrita usando una tabla derivada.
¡Y mejora el desempeño!

Tablas derivadas

Menos performante

```
SELECT MIN (Salary)  
FROM Employees  
Where EmpID IN (SELECT TOP 2 EmplD FROM Employees ORDER BY Salary DESC)
```

Más performante

```
SELECT MIN (A.Salary)  
FROM (SELECT TOP 2 Salary FROM Employees ORDER BY Salary DESC) AS A
```

La misma consulta puede ser reescrita usando una tabla derivada.

X. ANÁLISIS DE PERFORMANCE

2 / 2 (where)

Y para terminar, algunas sugerencias relacionadas al WHERE:



- Para unir tablas no usen un WHERE. Funciona, sí. Pero el motor hace el producto cartesiano de todas las tablas implicadas y después lo filtra. Les proponemos algo mejor:
- WHERE hace que las tablas se lean en su totalidad y después hace las relaciones.
- Háganse amigxs de INNER JOIN, LEFT JOIN o RIGHT JOIN. Si los usan, el SGBD lee las tablas y verifica qué relación se necesita entre cada una ellas, así lee menos registros y hace más eficiente la consulta. 😊
- Si usan LIKE en la cláusula WHERE traten de evitar el uso del operador "%" al principio de la cadena a buscar. Esto obligaría a leer todos los datos de la tabla para responder la consulta. También les recomendamos que existan, como mínimo, 3 caracteres antes del operador "%".
- Tengan en cuenta que SQL Server no es bueno y eficiente en la búsqueda de cadenas difusas. Cuando colocan un "%" al comienzo de una cadena, harán imposible el uso de cualquier índice ascendente. Pasa lo mismo cuando un "%" está al final de una cadena.
- Los operadores que usen dentro del filtro deben ser los de mejor rendimiento. Para que tengan a mano, el orden de rendimiento de mayor a menor es: = , >, >=, <, <=, LIKE, NOT IN, NOT LIKE ,IN y <> .
- Si dentro de una consulta tienen que utilizar el comando HAVING les recomendamos hacer todos los filtros posibles con el comando WHERE, de esta forma el trabajo involucrado al comando HAVING será el menor posible. Abajo, el ejemplo.
- Siempre filtren por campos que tengan índices, eligiendo los que formen parte de los índices definidos de la tabla, además de especificarlos en el mismo orden en el que estén definidos en la clave.

- Si quieren filtrar por campos pertenecientes a índices compuestos es mejor utilizar todos los campos de todos los índices. Abajo les dejamos un ejemplo.



- No usen funciones sobre campos que tengan índices definidos porque no funcionan cuando se los usa dentro de una fórmula. Miren el ejemplo de abajo.
- Eviten definir filtros mediante concatenación de cadenas.
- **Bonus track:** SQL Server siempre devuelve la cuenta del número de filas afectado por las consultas de INSERT, DELETE, UPDATE y SELECT. Usar la cláusula SET NOCOUNT ON va a evitar esto ahorrando memoria y tiempo. ¡Las consultas con muchos joins o subconsultas se lo van a agradecer!



Campos índices en funciones

En el siguiente caso existe un índice definido sobre el campo fecha, pero en la primera consulta queda desaprovechado ya que no tiene validez dentro de la función, por lo tanto podría redefinirse la consulta.

✗ --ANULARIAMOS EL INDICE SOBRE EL CAMPO FECHA
WHERE day(getdate()) - day(FECHA) = 7

✓ --SERIA MEJOR ESCRIBIRLO ALGO ASI
WHERE fecha = dateadd(day,-7,getdate())

Índices compuestos

Si tenemos **un índice** formado por el campo **NOMBRE** y el campo **APELLIDO** y **otro índice** formado por el campo **EDAD**.

La sentencia óptima sería:

- ✓ WHERE NOMBRE='Juan' AND APELLIDO Like '%' AND EDAD = 20
- ✗ WHERE NOMBRE = 'Juan' AND EDAD = 20

En el segundo caso, el SGBD no puede usar el primer índice y ambas sentencias son equivalentes porque la condición APELLIDO Like '%' devolvería todos los registros.



Filtros en WHERE vs filtros en HAVING

Filtros en WHERE

- Seleccionamos género, director y la suma del recaudo siempre y cuando el género sea 'Drama' (WHERE) y después agrupamos por género y director (GROUP BY)

SELECT genero, director, **SUM**(recaudo) **AS TOTAL** **FROM** peliculas
WHERE genero LIKE "%Drama%"
GROUP BY genero, director
HAVING **SUM**(recaudo) > 50;

Resultado de Consulta			
	GENERO	DIRECTOR	TOTAL
1	Drama	Roberto Benigni	99
2	Drama	Steven Spielberg	78

Filtros en HAVING

- Seleccionamos el género, director y hacer la suma del recaudo, sin importar si el género es o no 'Drama', luego agrupamos por género y director (GROUP BY). Por último seleccionamos solo los registros cuyo género sea 'Drama'.

- El resultado de la consulta hecha con HAVING se demora el doble de tiempo que la consulta hecha con WHERE!

SELECT genero, director, **SUM**(recaudo) **AS TOTAL** **FROM** peliculas
GROUP BY genero, director
HAVING **SUM**(recaudo) > 50
and genero LIKE "%Drama%";

Resultado de Consulta			
	GENERO	DIRECTOR	TOTAL
1	Drama	Roberto Benigni	99
2	Drama	Steven Spielberg	78

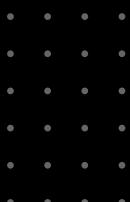


Si sabes de SQL y te interesaría trabajar con nosotros, puedes escribirnos a
rrhh@dataanalytics.com o rrhh_colombia@dataanalytics.com

FUENTES CONSULTADAS



- **SQL style guide by Simon Holywell**
- **24 Rules to the SQL Formatting Standard | LearnSQL.com**
- **Buenas prácticas en Transact SQL | tecnología y un poco de experiencia. (wordpress.com)**
- **Buenas prácticas para elaborar consultas SQL | @GabrielMorrisS (gabrielmorriasa.blogspot.com)**
- **Buenas prácticas para el diseño de base de datos – Videlcloud (wordpress.com)**
- **SQL Shack – articles about database auditing, server performance, data recovery, and more**
- **Mejores Prácticas SQL Server de Nicolás Nakasone**





¡MUCHAS GRACIAS!

