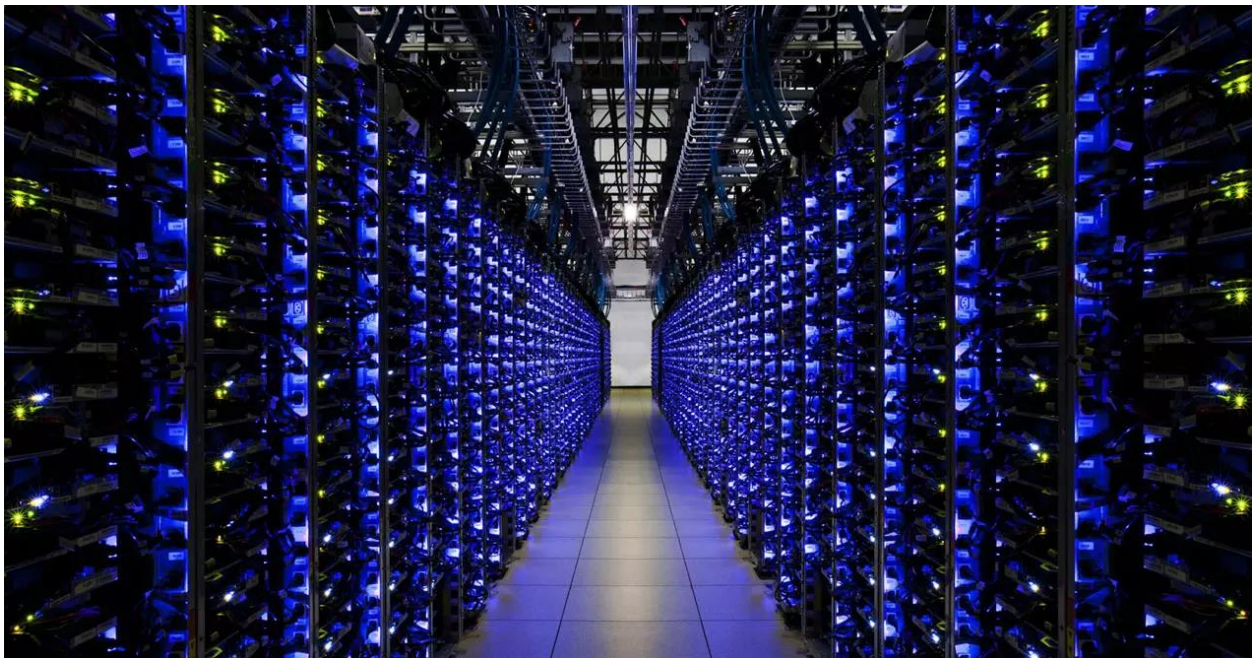


Optimización de Bases de Datos

Por Sebastian Machado y Daira Orlandini



La optimización de bases de datos es esencial para garantizar un rendimiento eficiente en la gestión de datos. En este documento se explorarán conceptos, técnicas y mejores prácticas para lograr un rendimiento óptimo en bases de datos y maximizar su potencial.


¿Qué es la optimización?

La optimización de bases de datos se refiere a la mejora del rendimiento de una base de datos, con el objetivo de aumentar la velocidad de las consultas, conseguir la mayor eficiencia posible en el almacenamiento de datos y conseguir la menor carga del sistema. En general, se busca que la base de datos tenga la capacidad de manejar grandes volúmenes de datos y usuarios concurrentes (aquellos que tienen la posibilidad de actuar en simultáneo con otros).

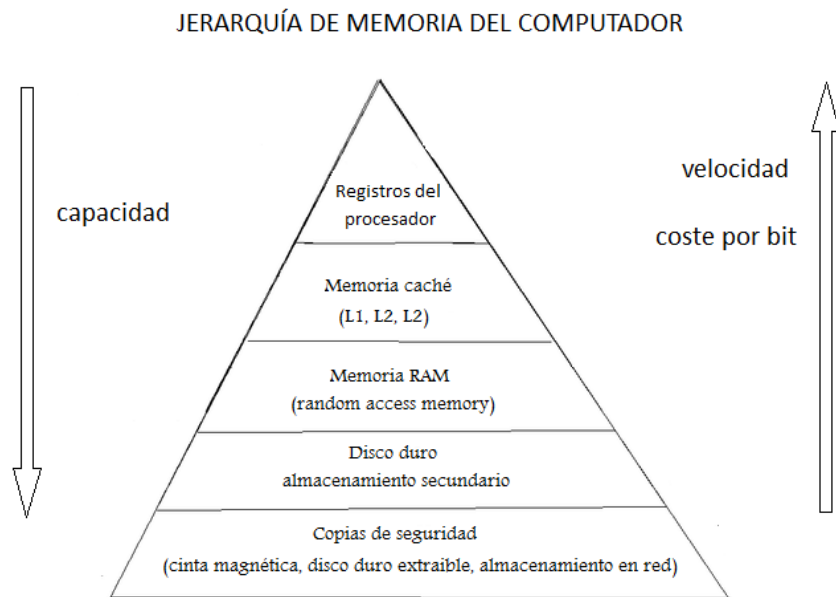
¿Cómo optimizar una base de datos?

1. El diseño de la base

Se deben tomar las decisiones correctas en cuanto a una serie de aspectos según el proyecto que se vaya a encarar. Esto implica, el tipo de base de datos, las tecnologías a utilizar, el diseño



propiamente dicho de la base, las restricciones y el hardware que dará soporte, entre otras. Recordemos que hay memorias más efectivas que otras.



Dependiendo del proyecto, podría elegir distintos tipos de base de datos.

Algunos ejemplos son:

- Una base de datos relacional para un proyecto con una estructura de datos bien definida y estable, como un sistema de gestión o de inventario.
- Una base de datos no relacional para un proyecto con grandes cantidades de datos no estructurados y variables, como redes sociales o sistemas de sensores.
- Una base de datos orientada a objetos para una aplicación en la cual no solo necesite guardar datos, sino también métodos.

2. Gestion de Indices

Un índice SQL se utiliza para recuperar datos de una base de datos rápidamente. Indexar una tabla o vista es, sin duda, una de las mejores formas de mejorar el rendimiento de consultas y aplicaciones.

Es una **tabla de búsqueda rápida** para encontrar registros que los usuarios necesitan buscar con frecuencia. Un índice es pequeño, rápido y está optimizado para búsquedas rápidas. Es muy útil para conectar las tablas relacionales y buscar tablas grandes.

Los índices SQL son principalmente una herramienta de rendimiento, por lo que realmente se aplican si una base de datos crece. SQL Server admite varios tipos de índices, pero uno de los tipos más comunes es el índice agrupado. Este tipo de índice se crea automáticamente con una **clave principal**.

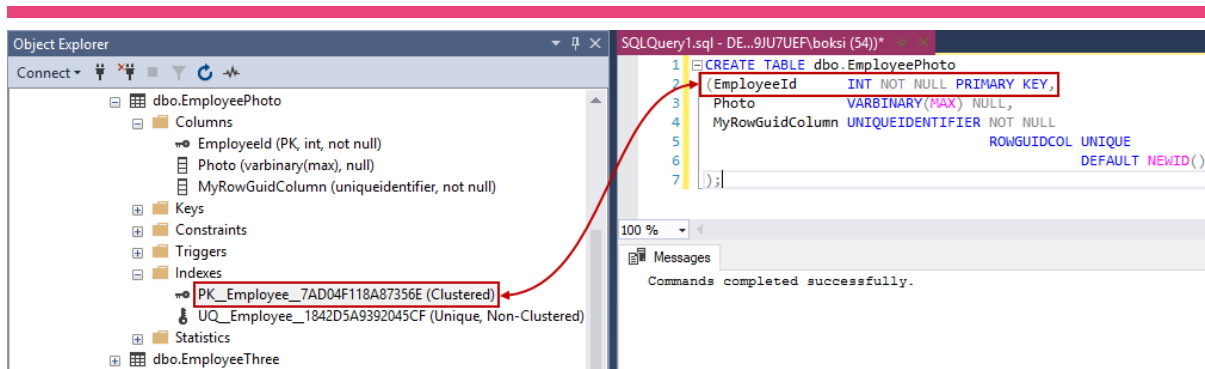
Para aclarar el punto, el siguiente ejemplo crea una tabla que tiene una clave principal en la columna "EmployeeId":

```
CREATE TABLE dbo.EmployeePhoto

(EmployeeId          INT NOT NULL PRIMARY KEY,

Photo              VARBINARY(MAX) NULL,
MyRowGuidColumn    UNIQUEIDENTIFIER NOT NULL
                    ROWGUIDCOL UNIQUE
                    DEFAULT NEWID());
```

En la tabla "EmployeePhoto", la clave principal al final de la definición de columna "EmployeeId". Esto crea un índice SQL que está especialmente optimizado para usarse mucho. Cuando se ejecuta la consulta, SQL Server creará automáticamente un índice agrupado en la columna especificada y podemos verificar esto desde el **Explorador de objetos** si navegamos a la tabla recién creada y luego a la carpeta **Índices** :



Se pueden crear índices adicionales utilizando la palabra clave Index en la definición de la tabla. Esto puede ser útil cuando hay más de una columna en la tabla que se buscará con frecuencia. El siguiente ejemplo crea índices dentro de la instrucción Create table:

```
CREATE TABLE Bookstore2
( ISBN_NO      VARCHAR(15) NOT NULL PRIMARY KEY,
  SHORT_DESC   VARCHAR(100) ,
  AUTHOR       VARCHAR(40) ,
  PUBLISHER    VARCHAR(40) ,
  PRICE        FLOAT,
  INDEX SHORT_DESC_IND (SHORT_DESC, PUBLISHER)
);
```

Estrategias para indexar:

- Evite indexar tablas/columnas muy utilizadas: cuantos más índices haya en una tabla, mayor será el efecto en el rendimiento de las instrucciones Insertar, Actualizar, Eliminar y Merge porque todos los índices deben modificarse adecuadamente. Esto significa que SQL Server tendrá que dividir páginas, mover datos y tendrá que hacer de las eso para todos los índices afectados por esas declaraciones DML.
- Use claves de índice estrechas siempre que sea posible: mantenga los índices estrechos, es decir, con la menor cantidad de columnas posible. Las claves numéricas

exactas son las claves de índice SQL más eficientes (p. ej., números enteros). Estas claves requieren menos espacio en disco y gastos generales de mantenimiento.

- Use índices agrupados en columnas únicas : considere las columnas que son únicas o contienen muchos valores distintos y evítelas para las columnas que sufren cambios frecuentes
- Índices no agrupados en columnas que se buscan o unen con frecuencia : asegúrese de que los índices no agrupados se coloquen en foreign keys y columnas que se usan con frecuencia en las condiciones de búsqueda, como la cláusula Where que devuelve coincidencias exactas.
- Cubra los índices de SQL para obtener grandes ganancias de rendimiento : las mejoras se logran cuando el índice contiene todas las columnas de la consulta.

3. Optimización de consultas

Al consultar una base de datos, la optimización es clave. Una consulta ineficiente agotará los recursos de la base de datos y provocará un rendimiento lento o no disponibilidad. Es vital la optimización de consultas para lograr un impacto mínimo en el rendimiento de la base de datos.


Puede ocurrir por dos vías:

→ **Manual:**

Es aquella que realiza el programador, y será ampliada en una sección posterior.

En caso de que sea una base de datos relacional, es importante el conocimiento de álgebra relacional, del su diseño y la relación entre las tablas de la base.

En caso de que no lo sea, es importante conocer los algoritmos de búsqueda que se pueden utilizar y elegir los más óptimos.



Dos consultas distintas pueden traer los mismos resultados, pero tener eficiencias distintas, y utilizar una cantidad mayor o menor de recursos, además de demandar distintas cantidades de tiempo de procesamiento.


→ **Parte de la base de datos:**

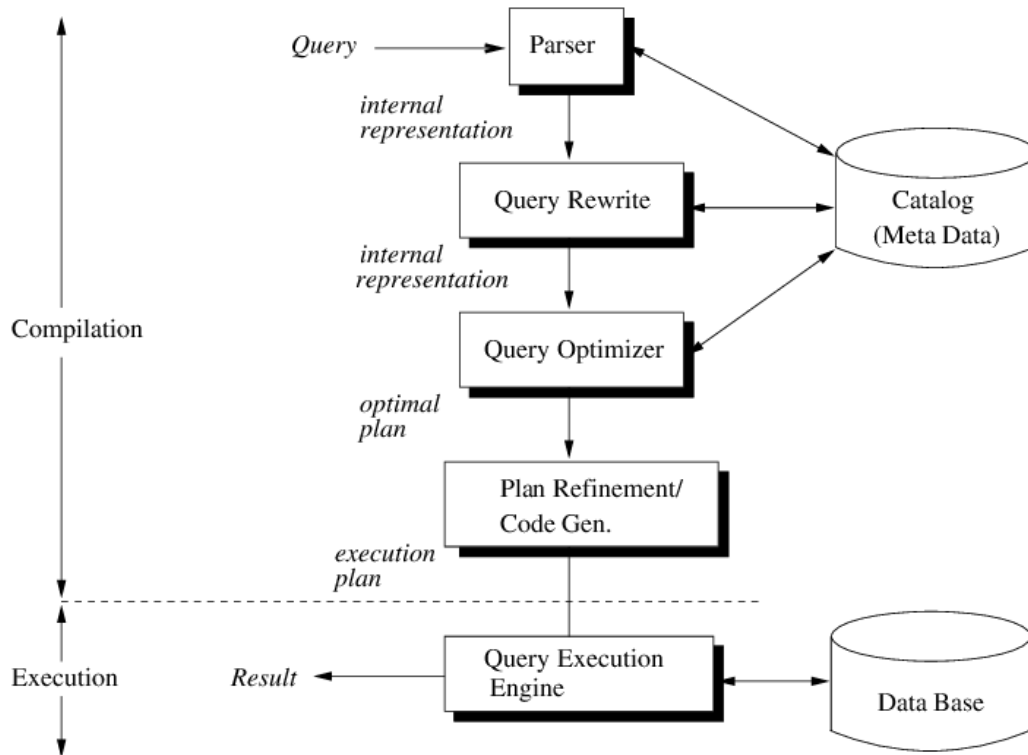
Puede haber estructuras intrínsecas de los sistemas de gestión de bases de datos, como el procesador de consultas, que tengan un impacto directo en el rendimiento.

El procesador de consultas es el encargado de traducir la consulta al plan de consultas más eficiente.

Un plan de consultas es la secuencia de operaciones a realizar sobre los datos.

Este tiene 3 partes:

- 1) **Query phaser:** Analiza la consulta y construye una estructura de árbol a partir del texto de la consulta.
 - 2) **Query processor:** Procesa a consulta y realiza su comprobación semántica (verifica la existencia de lo solicitado) para poder descartar la consulta en caso de ser negativa.
 - 3) **Query optimizer:** Transforma el plan inicial en la mejor secuencia de operaciones disponible.
- 



Seleccionar campos necesarios en lugar de usar SELECT *

Consultar únicamente los campos que requerimos, es decir, si nuestra consulta tiene como objetivo obtener los datos de envío de un cliente.

Consulta ineficiente:

```
SELECT *  
FROM Customers
```

La forma correcta de realizar la consulta sería:

```
SELECT FirstName, LastName, Address, City, State, Zip  
FROM Customer
```

Es una consulta más limpia y solo trae los datos que requerimos.

Evitar SELECT DISTINCT

Si bien sirve para eliminar registros duplicados, internamente agrupando todos los campos de la consulta, SELECT DISTINCT funciona agrupando todos los campos de la consulta para crear resultados distintos y esto requiere un gran poder de procesamiento.

Ineficiente:


```
SELECT DISTINCT FirstName, LastName, State  
FROM Customers
```

En bases de datos grandes, una gran cantidad de Juan Perez harán que esta consulta se ejecute lentamente.

Eficiente:

```
SELECT FirstName, LastName, Address, City, State, Zip  
FROM Customers
```

Añadiendo más campos, los campos no duplicados serán devueltos sin necesidad de un DISTINCT, por lo que la base no tendrá que agrupar ningún campo.



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

En una unión cartesiana(cross join), se crean 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 los 1,000 registros donde CustomerID está correctamente unido. 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 requerido. Las uniones cartesianas son especialmente problemáticas en bases de datos a gran escala, porque una unión cartesiana de dos tablas grandes podría generar miles de millones o billones de resultados.

En lugar de usar una unión cartesiana, usar un INNER JOIN:

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

Algunos sistemas DBMS pueden reconocer las uniones WHERE y ejecutarlas automáticamente como INNER JOIN en su lugar. En esos sistemas DBMS, no habrá diferencia en el rendimiento entre un WHERE join y un INNER JOIN.

Usar WHERE en lugar de HAVING para definir los filtros




HAVING se calcula después de las declaraciones WHERE. Si lo que se requiere es filtrar una consulta según las condiciones, una declaración WHERE es más eficiente.

Por ejemplo, supongamos que se realizaron 200 ventas en el año 2016 y queremos consultar la cantidad 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 1000 registros de ventas de la tabla Ventas, luego filtraría los 200 registros generados en el año 2016 y finalmente contaría los registros en el conjunto de datos.

```
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 contaría los registros en el conjunto de datos.


HAVING solo debe usarse cuando se filtra en un campo agregado (COUNT). Podríamos filtrar por clientes con más de 5 ventas utilizando 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
```

Limitar la cantidad de datos recuperados

Utiliza la paginación o límites de resultados en las consultas para reducir la cantidad de datos que se recuperan y se transmiten.

```
SELECT name
FROM Player
WHERE point>100;
```



```
SELECT name
FROM Player
WHERE point>100
LIMIT 10;
```

Usar WILDCARDS solo al final de la frase.

Al buscar datos de texto sin formato, como ciudades o nombres, los wildcard crean la búsqueda más amplia posible. La búsqueda más amplia es también la más ineficiente.

Cuando se usa un wildcard inicial, especialmente en combinación con un wildcard al final, la base de datos tiene la tarea de buscar en todos los registros una coincidencia en cualquier lugar dentro del campo seleccionado.

Esta consulta para extraer ciudades que comienzan con 'Char':

```
SELECT City FROM Customers
WHERE City LIKE '%Char%'
```

Esta query va retornar registros como **Charleston**, **Charlotte**, and **Charlton**. However, it will also pull unexpected results, such as Cape **Charles**, Crab **Orchard**, and **Richardson**.

En lugar, consultar:

```
SELECT City FROM Customers
WHERE City LIKE 'Char%'
```

Traerá resultados como **Charleston**, **Charlotte**, and **Charlton**.



Correr consultas complejas durante un horario no pico

Para minimizar el impacto de sus consultas analíticas en la base de datos la programación de la consulta para que se ejecute en un horario de menor actividad. La consulta debe ejecutarse cuando los usuarios simultáneos están en su número más bajo, que suele ser en medio de la noche (3 a 5 a. m.).

- Selección de tablas grandes (>1,000,000 registros)
- Uniones cartesianas o CROSS JOIN
- Declaraciones en bucle
- SELECT DISTINCT
- Subconsultas anidadas
- Búsquedas con wildcards
- Consultas de varios esquemas

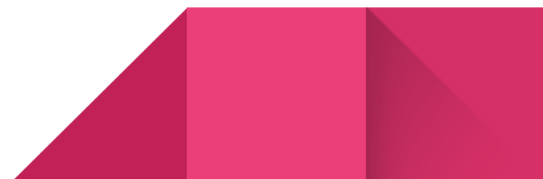
Evitar subconsultas innecesarias


Las subconsultas pueden ser costosas en términos de rendimiento. Evita anidar demasiadas subconsultas o utilizarlas de manera innecesaria. En su lugar, utiliza joins o técnicas de combinación de datos para obtener los resultados necesarios de manera más eficiente.

4. Mantenimiento

El mantenimiento para la mejora del rendimiento de una base de datos implica una serie de actividades destinadas a optimizar su funcionamiento con el fin de lograr un mejor rendimiento en términos de velocidad de procesamiento, eficiencia y utilización de recursos.

Algunas de las actividades comunes que se pueden realizar para mejorar el rendimiento de una base de datos incluyen:



-
- **Reorganización de índices:** Los índices son estructuras utilizadas para acelerar la búsqueda y recuperación de datos en una base de datos. Con el tiempo, los índices pueden volverse fragmentados y desorganizados, lo que puede afectar negativamente el rendimiento de las consultas. La reorganización de índices implica reorganizar físicamente los índices para eliminar la fragmentación y mejorar la eficiencia de las operaciones de búsqueda y recuperación de datos.
 - **Actualización de estadísticas:** Las estadísticas son utilizadas por el sistema de gestión de bases de datos (SGBD) para tomar decisiones sobre cómo ejecutar consultas y optimizar el rendimiento. Es importante mantener actualizadas las estadísticas de la base de datos para asegurarse de que el SGBD tenga la información más reciente sobre la distribución de datos y pueda tomar decisiones de optimización adecuadas.
 - **Ajuste de la configuración del SGBD:** Los sistemas de gestión de bases de datos suelen tener configuraciones que afectan su rendimiento. Revisar y ajustar la configuración del SGBD, como el tamaño de memoria asignada, la cantidad de conexiones permitidas, la configuración de caché y otros parámetros de rendimiento, puede tener un impacto en el rendimiento general de la base de datos.
 - **Compactación de tablas:** Las tablas pueden volverse fragmentadas y ocupar más espacio del necesario en disco, lo que puede afectar negativamente el rendimiento de la base de datos. La compactación de tablas implica reorganizar físicamente los datos en las tablas para reducir la fragmentación y el espacio ocupado en disco, lo que puede mejorar la eficiencia en la búsqueda y recuperación de datos.
 - **Gestión de transacciones:** Las transacciones son operaciones que se realizan en la base de datos y pueden afectar el rendimiento si no se gestionan adecuadamente. Es importante asegurarse de que las transacciones se manejen de manera eficiente, con un compromiso adecuado entre la duración de las transacciones y la cantidad de recursos utilizados, para evitar cuellos de botella y optimizar el rendimiento.
- 

- **Actualización del software del SGBD:** Mantener actualizado el software del SGBD con las últimas actualizaciones y parches puede mejorar el rendimiento y la seguridad de la base de datos. Las actualizaciones del software suelen incluir mejoras de rendimiento y correcciones de errores, por lo que es importante mantener la base de datos actualizada con las últimas versiones

5. Escalabilidad

La escalabilidad de una base de datos se refiere a su capacidad para manejar eficientemente un crecimiento en la cantidad de datos y la carga de trabajo sin perder rendimiento o eficiencia. Una base de datos escalable es capaz de adaptarse y crecer con las necesidades de almacenamiento y procesamiento de datos a medida que estas aumentan con el tiempo.

Existen dos tipos principales de escalabilidad en una base de datos:

Escalabilidad vertical:

Implica agregar más recursos a un servidor o máquina para mejorar el rendimiento de la base de datos. Esto puede incluir agregar más CPU, memoria RAM o almacenamiento en disco al servidor que aloja la base de datos. La escalabilidad vertical suele ser más sencilla de implementar, pero tiene un límite físico en cuanto a la cantidad de recursos que se pueden agregar a un solo servidor.

Escalabilidad horizontal:

Implica agregar más servidores a un sistema distribuido para manejar la carga de trabajo. Esto implica dividir los datos y las tareas de procesamiento entre varios servidores que trabajan en paralelo para mejorar el rendimiento y la capacidad de manejo de la base de datos.



Se configuran en forma de una red de servidores denominada clúster. Esta tiene la finalidad de dividir eficientemente la demanda de trabajo entre todos los nodos que conforman la red de servidores.

Es importante tener en cuenta que lograr una verdadera escalabilidad en una base de datos puede requerir una planificación y diseño adecuados desde el inicio, como el uso de técnicas de diseño de bases de datos distribuidas, particionamiento de datos, uso de clústeres y técnicas de replicación, entre otros. Además, el monitoreo y la optimización constantes son esenciales para asegurar un rendimiento óptimo en una base de datos escalable a medida que crece y se enfrenta a mayores demandas de carga de trabajo.

