Engineer

Índice de base de datos

estructura que busca filas sin recorrer toda la tabla

Qué es

Un índice de base de datos es una estructura separada, almacenada junto a una tabla, que permite a la base de datos ir directamente a las filas que cumplen una condición en lugar de leer la tabla de la primera fila a la última. En el habla cotidiana y en la mayor parte de la documentación de PostgreSQL se llama simplemente índice. La documentación lo compara con el índice al final de un libro: un lector que busca un tema lo consulta allí y va a las páginas indicadas, en lugar de hojear todo el volumen.

Piense en una tabla de pedidos y en la condición `WHERE customer_id = 42` . Sin un índice sobre `customer_id` , PostgreSQL tiene una forma de responder: un recorrido secuencial, que lee todas las filas y conserva las que coinciden. Eso cuesta lo mismo tanto si el cliente tiene dos pedidos como si tiene dos mil, y el coste crece con la tabla. Con un índice en la columna, PostgreSQL recorre unos pocos niveles de un árbol de búsqueda hasta las entradas de 42 y recupera solo esas filas. Crear un índice requiere una sola sentencia, `DROP INDEX` vuelve a eliminarlo y ambas operaciones pueden realizarse en cualquier momento.

Una vez que existe un índice, PostgreSQL lo mantiene sincronizado con la tabla en cada escritura, y el planificador de consultas decide, consulta por consulta, si utilizarlo resulta más barato que un recorrido completo. Nada en el SQL nombra el índice, de modo que un mismo índice puede acelerar un `SELECT` , un `UPDATE` o un `DELETE` con una condición de búsqueda, así como una unión.

PostgreSQL incorpora varios tipos, cada uno construido para una clase distinta de pregunta:

- **B-tree** es el predeterminado y lo que construye `CREATE INDEX` salvo que se indique otra cosa. Responde a comparaciones de igualdad y rango (`<` , `<=` , `=` , `>=` , `>`), `BETWEEN` , `IN` , `IS NULL` y un `LIKE` cuyo patrón está anclado al inicio. Mantiene las entradas en orden, por lo que también puede devolver filas ya ordenadas.
- **Hash** almacena un código hash de 32 bits para cada valor y solo responde a igualdad.
- **GiST** y **SP-GiST** son marcos de trabajo para datos como la geometría, donde la pregunta es «se solapa», «contiene» o «es el más cercano a» en lugar de «es igual».
- **GIN** es un índice invertido para valores formados por muchas partes, como los arrays, y encuentra las filas cuyo valor contiene un elemento determinado.
- **BRIN** conserva únicamente un resumen por cada rango de bloques de tabla. Resulta adecuado para una columna cuyos valores siguen el orden físico de las filas, como suele ocurrir con una marca de tiempo de inserción.

Además de su tipo, un índice puede adaptarse a la pregunta. Un índice único rechaza valores duplicados. Un índice multicolumna abarca varias columnas, hasta 32. Un índice parcial tiene una cláusula `WHERE` y contiene solo las filas que la cumplen. Un índice de expresión almacena un valor calculado, como `lower(email)` . Y una lista `INCLUDE` incorpora columnas adicionales al índice sin utilizarlas para la búsqueda, de modo que una consulta pueda responderse solo desde el índice.

Utilice el concepto en una decisión

Aplique «¿Qué es un índice de base de datos?» a una decisión real. Podemos ayudarle a relacionar el concepto con el sistema que está construyendo o comprando. Detectar las consultas y los índices que faltan en un sistema cada año más pesado y corregir lo que realmente importa.

Explore nuestros servicios Póngase en contacto →

Compruebe los supuestos detrás del término

Utilice la explicación y los trabajos enlazados para formular preguntas sobre su proyecto. Podemos examinar los supuestos con usted antes de elegir una herramienta. 30 días de datos eléctricos en 180 ms — 180 ms para agregar un mes de lecturas de contadores

30 días de datos eléctricos en 180 ms Póngase en contacto →

Por qué importa

La diferencia que supone un índice crece con la tabla. En una tabla pequeña, un recorrido completo es barato y el planificador a menudo lo prefiere, con razón: leer unas pocas páginas en orden cuesta menos que recorrer un árbol para llegar a ellas. Con decenas de millones de filas, la diferencia entre un recorrido completo y una búsqueda por índice es la que hay entre una página que carga y

otra que agota el tiempo de espera. Una consulta que era rápida en el lanzamiento puede volverse lenta un año después sin cambiar una línea de código, simplemente porque la tabla que lee ha crecido.

Los índices no son gratuitos y la factura llega con las escrituras. Cada inserción añade una entrada a cada índice de la tabla. Una actualización solo puede evitarlo si es una actualización heap-only tuple (HOT), en la que la nueva versión de la fila permanece en la misma página y las entradas del índice que ya existen siguen conduciendo a ella. PostgreSQL solo lo permite cuando la actualización deja intactas todas las columnas que utilizan los índices de la tabla, excepto los índices BRIN, que solo contienen un resumen. Por tanto, cada índice reduce las posibilidades de seguir la vía barata; por eso la documentación aconseja eliminar los índices que se utilizan poco o nunca. Una tabla con una docena de índices paga una docena de veces en cada inserción.

Construir un índice también tiene un coste. Un `CREATE INDEX` corriente permite que otras sesiones lean la tabla, pero bloquea sus escrituras hasta terminar la construcción, y la documentación advierte de que en una tabla muy grande puede tardar muchas horas.

Por último, un índice solo sirve en la medida en que el planificador decida utilizarlo, y este trabaja a partir de estadísticas recogidas por `ANALYZE` . La pregunta no es «¿hay un índice?», sino «¿lo utilizan las consultas que importan?», y `EXPLAIN` permite averiguarlo.

Cómo se aplica en la práctica

La página de cuenta de una tienda en línea enumera los veinte pedidos más recientes de un cliente:

```
SELECT id, created_at, total
FROM   orders
WHERE  customer_id = 42
ORDER  BY created_at DESC
LIMIT  20;
```

Con solo una clave primaria en `id` , PostgreSQL tiene que leer `orders` entero, conservar las filas del cliente, ordenarlas y descartar todas menos veinte. `EXPLAIN` lo muestra: un `Seq Scan` sobre `orders` con la condición como `Filter`

y una ordenación por encima. `EXPLAIN ANALYZE` también ejecuta la consulta e imprime los recuentos reales de filas y los tiempos junto a las estimaciones.

El índice que encaja con esta pregunta sitúa primero la columna de igualdad y después la de ordenación:

```
CREATE INDEX CONCURRENTLY orders_customer_created_idx
ON orders (customer_id, created_at);
```

Al ejecutar `EXPLAIN` otra vez, el recorrido se convierte en un recorrido por índice que utiliza `orders_customer_created_idx`, y `customer_id = 42` aparece ahora como `Index Cond` en lugar de como filtro. La ordenación también desaparece. Un B-tree mantiene las entradas del cliente 42 ordenadas por `created_at`, PostgreSQL puede leerlas hacia atrás para obtener primero las más recientes y, con un `LIMIT`, se detiene después de veinte entradas en lugar de acceder al resto.

El orden de las columnas importa. El mismo índice sirve para una consulta solo sobre `customer_id`, su columna inicial, pero un informe sobre todos los pedidos realizados ayer, que filtre únicamente por `created_at`, obtiene mucho menos de él: en un B-tree multicolumna, la igualdad sobre las columnas iniciales más un rango sobre la siguiente es lo que reduce la parte del índice que se lee. PostgreSQL 18 puede realizar un skip scan, repitiendo la búsqueda una vez por cada valor inicial distinto, pero el planificador normalmente solo lo elige cuando la columna inicial tiene pocos valores distintos, y los identificadores de cliente tienen muchos. Si el informe importa, recibe su propio índice sobre `created_at`. Invertido, como `(created_at, customer_id)`, el índice dispersaría los pedidos del cliente 42 por todo su contenido.

La nota sobre optimizar la base de datos antes de fragmentarla entre servidores sitúa los índices en primer lugar entre las correcciones que ordena por su beneficio, una vez que `EXPLAIN (ANALYZE, BUFFERS)` ha mostrado qué recorridos ejecuta una consulta con mucha carga. Su propio ejemplo es un índice parcial, `units_unsold`, que contiene solo las unidades todavía a la venta, por lo que se reduce a medida que avanza una venta en lugar de crecer. La nota también cuenta el coste. En su tabla, las 2.000 operaciones de reserva que establecían `reserved`, `reserved_at` y `order_id` fueron todas actualizaciones HOT, mientras que, de 2.000 actualizaciones que establecían `sold`, una columna que lee el índice parcial, ninguna lo fue. Su consejo es eliminar los índices cuyo `idx_scan` en `pg_stat_user_indexes` permanezca en cero.

El caso práctico sobre una mejor elección de clave muestra por qué los valores que entran en un B-tree importan tanto como las columnas. Los UUID aleatorios dispersan las inserciones por el índice, lo que puede causar fragmentación. La plataforma estableció UUID v7 ordenados por tiempo como opción predeterminada para las claves primarias de las entidades, con una marca de tiempo en los bits iniciales para que los identificadores nuevos incorporen orden temporal. Reducir la fragmentación y acelerar las consultas eran objetivos de diseño; el resultado registrado es la convención de claves configurada, sin que el caso establezca una comparación de rendimiento de la carga antes y después.

Dónde surgen los problemas

Añadir un índice detiene las escrituras de la tabla. Un `CREATE INDEX` corriente sobre una tabla con mucha actividad bloquea inserciones, actualizaciones y eliminaciones durante toda la construcción, y un despliegue rutinario se convierte en una interrupción del servicio. Utilice `CREATE INDEX CONCURRENTLY` : recorre la tabla dos veces y espera a las transacciones que podrían utilizar el índice, por lo que tarda bastante más, pero las escrituras continúan. No puede ejecutarse dentro de un bloque de transacción, así que, si una herramienta de migración envuelve cada paso en uno de esos bloques, hay que indicarle que ejecute esta sentencia por separado. Solo puede ejecutarse una construcción concurrente por tabla a la vez.

Una construcción concurrente falla y deja un índice no válido. Un interbloqueo, o un valor duplicado al construir un índice único, hace que falle `CREATE INDEX CONCURRENTLY` , pero el índice que empezó permanece en el catálogo, marcado como no válido. Las consultas lo ignoran, pero todas las escrituras siguen pagando su mantenimiento, y un índice único no válido puede seguir imponiendo unicidad. La orden `\d` de psql lo enumera con `INVALID` después de su definición. Elimínelo y vuelva a ejecutar la construcción concurrente, o reconstrúyalo con `REINDEX INDEX CONCURRENTLY` .

La consulta oculta la columna dentro de una función. Un índice sobre `email` no sirve para `WHERE lower(email) = 'ana@example.com'` , porque el índice contiene `email` , no `lower(email)` . Indexe la expresión que utiliza la consulta:

```
CREATE INDEX users_lower_email_idx ON users (lower(email));
```

La expresión se calcula en cada inserción y cada actualización que no sea HOT; por eso la documentación recomienda índices de expresión cuando la velocidad de recuperación importa más que la de escritura.

Se acumulan índices que nadie utiliza. Cada uno que se añade «por si acaso» cuesta en todas las escrituras y puede impedir actualizaciones HOT. `pg_stat_user_indexes` cuenta los recorridos iniciados sobre cada índice en `idx_scan` y registra el momento del último en `last_idx_scan`:

```
SELECT relname, indexrelname, idx_scan, last_idx_scan
FROM   pg_stat_user_indexes
WHERE  idx_scan = 0
ORDER  BY relname;
```

Compruebe antes de eliminar. Un índice que respalda una clave primaria o una restricción de unicidad cumple su función aunque ninguna consulta lo lea nunca, y un recuento de cero significa poco en un servidor que acaba de empezar a recibir tráfico.

El planificador ignora un buen índice. Después de una carga masiva, las estadísticas se quedan por detrás de los datos, las estimaciones son erróneas y un recorrido completo gana sobre el papel. Ejecute `ANALYZE`; la documentación dice que evaluar el uso de índices sin hacerlo es «una causa perdida». Pruebe también con volúmenes realistas: con cien filas que caben en una página de disco, ningún índice ofrece ventaja frente a leer esa página. Para saber si un índice puede utilizarse en absoluto, establezca `enable_seqscan = off` en una sesión de prueba y ejecute `EXPLAIN` de nuevo. Si el plan sigue haciendo un recorrido completo, la condición no coincide con el índice.

Lecturas adicionales

- Índices: introducción — para qué sirve un índice, cómo se crea y elimina y la sobrecarga que añade a cada escritura.
- Tipos de índice — B-tree, Hash, GiST, SP-GiST, GIN y BRIN, y los operadores a los que puede responder cada uno.
- Índices multicolumna — cómo determina el orden de las columnas qué parte puede acotar un B-tree y cuándo se aplica skip scan.
- CREATE INDEX — la sintaxis completa, incluidos `CONCURRENTLY`, `INCLUDE`, índices parciales y recuperación tras una construcción no válida.

- Uso de EXPLAIN — lectura de un plan: tipos de recorrido, costes estimados y cifras reales de `EXPLAIN ANALYZE` .
- Análisis del uso de índices — por qué `ANALYZE` y los datos reales van primero y cómo comprobar si se puede utilizar un índice.

Glosario

Elija según sus restricciones reales

Traiga un conjunto de datos representativo, las preguntas que debe responder y el cuello de botella actual. Podemos acordar cómo medir la calidad, la capacidad de procesamiento y la recuperación.

Vea cuánto cuesta Póngase en contacto →

Haga práctica la terminología

Pregunte cómo se aplica «¿Qué es un índice de base de datos?» a su organización. Basta un ejemplo concreto de su sistema actual para iniciar la conversación.

Explore nuestros servicios

welcome@engineer.company +45 50 10 61 84 Copenhague, Dinamarca

Llévese esta página

Tarjeta de contacto

Creado con esmero y cariño por D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Derechos de autor © 2025 – actualidad · Engineer ApS · Todos los derechos reservados.

Impreso desde https://engineer.company/es/glossary/database-index/