Engineer Datos

Optimice la base de datos antes de fragmentarla

14 de octubre de 2026

Cuando una base de datos se ralentiza bajo carga, la primera propuesta suele ser añadir máquinas. A veces es lo correcto. Más a menudo, la parte lenta es un puñado de sentencias que nadie ha leído todavía, y repartir los datos entre servidores reparte esas sentencias entre más hardware sin abaratar ninguna. Un problema que parece irresoluble suele estar a un plan de consulta de una respuesta sencilla. A veces se elige escalar horizontalmente precisamente para que nadie tenga que leer la base de datos que ya tiene.

Este es nuestro orden antes de dividir nada: observar, ajustar, sacar las lecturas del primario, particionar las tablas que rodean el punto conflictivo y fragmentar solo cuando una máquina esté llena de trabajo útil. La venta relámpago, en la que todos los compradores escriben en una fila, se desarrolla en Vender la última entrada exactamente una vez. Esta nota explica el método que la rodea, y su último apartado examina la propuesta opuesta: vender desde todos los servidores y disculparse por las unidades vendidas dos veces.

Aplique la ingeniería a su sistema

La ingeniería detrás de «Optimice la base de datos antes de fragmentarla» puede orientar su próximo cambio. Podemos investigar comportamientos similares en su software, datos o infraestructura. Una estructura que refleja cómo funciona el negocio para poder responder también a las preguntas que surjan más adelante.

Explore nuestros servicios Póngase en contacto →

Siga el razonamiento y sus límites

Lea el razonamiento y sus límites en esta nota. Podemos comprobar si el mismo diagnóstico se sostiene en su entorno. Diseño y operación de 6 flujos ETL/ELT — 6 flujos ETL en tres motores de bases de datos

Diseño y operación de 6 flujos ETL/ELT Póngase en contacto →

Observar primero

PostgreSQL ya registra casi todo lo que hace falta saber. Antes de cambiar nada, hágale cuatro preguntas.

¿Qué sentencias cuestan más en total? `pg_stat_statements` , una extensión cargada mediante `shared_preload_libraries` , conserva cada sentencia normalizada con su número de llamadas y su tiempo. Ordene por el total, no por la media: una consulta de 2 ms ejecutada un millón de veces por hora cuesta más que un informe de dos segundos ejecutado dos veces.

```
SELECT calls,
       round(total_exec_time) AS total_ms,
       round(mean_exec_time::numeric, 1)
         AS mean_ms,
       left(query, 60) AS query
  FROM pg_stat_statements
 ORDER BY total_exec_time DESC
 LIMIT 10;
```

¿Qué hace realmente la primera? `EXPLAIN (ANALYZE, BUFFERS)` ejecuta la sentencia y muestra el plan elegido, las filas que devolvió realmente cada paso frente a las esperadas y cuántas páginas leyó. Un recorrido de un millón de filas para devolver diez, o una estimación errónea por un factor de cien, suele explicar todo el problema. `ANALYZE` ejecuta lo que explica, así que envuelva una escritura en `BEGIN` y `ROLLBACK` .

¿Quién espera y a quién?

```
SELECT pid, wait_event_type, wait_event,
       pg_blocking_pids(pid) AS blocked_by,
       left(query, 60) AS query
  FROM pg_stat_activity
 WHERE wait_event_type = 'Lock';
```

Las filas que aparezcan aquí en un minuto de mucha actividad indican que las transacciones hacen cola a la espera de otras, no del hardware. Una máquina más potente no elimina la necesidad de actualizar una misma fila compartida de

forma secuencial. Más servidores de aplicaciones pueden añadir a esa cola clientes que realizan escrituras.

¿El servidor trabaja o hace malabares? Compare las sesiones activas de `pg_stat_activity` con los núcleos. Cuando hay muchas veces más sesiones activas que núcleos, el servidor dedica el tiempo a alternar entre ellas. Un pool de conexiones lo corrige antes que cualquier hardware. Después lea `n_dead_tup` en `pg_stat_user_tables` : una tabla con más filas muertas que vivas se recorre entre sus propios residuos. Es autovacuum quedándose atrás, no una base de datos demasiado pequeña.

Anote lo que encuentre. Será la referencia frente a la que se mida cada paso posterior, y a menudo ya contiene la respuesta honesta a si hace falta fragmentar.

Ajustar lo que ya existe

Aproximadamente en orden de rendimiento obtenido por esfuerzo invertido.

Índices que correspondan a las preguntas. Los planes anteriores identifican los recorridos. Un índice sobre las columnas por las que filtra una consulta frecuente convierte un recorrido en una búsqueda directa. Un índice parcial conserva solo las filas que puede interesar consultar —por ejemplo, unidades aún a la venta— y se reduce a medida que avanza la venta en vez de crecer:

```
CREATE INDEX units_unsold
    ON units (item_id)
 WHERE NOT sold;
```

Cada índice también se paga en cada escritura. Elimine los que mantengan `idx_scan` a cero en `pg_stat_user_indexes` .

Tipos que digan qué es el valor. Los esquemas escritos deprisa, por una persona o por un modelo, recurren a `TEXT` donde bastaría `SMALLINT` o `BOOLEAN` . Un `BOOLEAN` ocupa un byte, un `SMALLINT` dos y `'yes'` almacenado como texto

cuatro. PostgreSQL también rellena cada columna de ancho fijo hasta su alineación, así que un orden accidental de columnas cuesta bytes. Una unidad de existencias escrita deprisa:

```
CREATE TABLE units (
  sold        text,  -- 'yes' or 'no'
  id          bigint,
  reserved    text,  -- 'yes' or 'no'
  reserved_at timestamptz,
  item_id     bigint,
  order_id    bigint
);
```

Y la misma unidad con tipos que dicen qué son los valores, los más anchos primero:

```
CREATE TABLE units (
  id          bigint PRIMARY KEY,
  reserved_at timestamptz,
  order_id    bigint,
  item_id     integer NOT NULL,
  sold        boolean NOT NULL DEFAULT false,
  reserved    boolean NOT NULL DEFAULT false
);
```

En PostgreSQL 18, la primera fila mide 72 bytes y la segunda 54, una cuarta parte menos, de modo que cabe más tabla en memoria. Los tipos también rechazan valores como `'maybe'`, que de otro modo tendría que vigilar la aplicación. Una restricción `CHECK` hace lo mismo con los rangos: `quantity smallint CHECK (quantity > 0)`.

Menos viajes de ida y vuelta, y el trabajo donde están los datos. Cada sentencia que envía la aplicación cuesta un viaje de ida y vuelta y un análisis sintáctico. Un bucle que lee cien filas y las actualiza una a una hace ciento un viajes donde bastaría una sentencia basada en conjuntos. Una página que recupera todas las unidades de un artículo para contar las libres envía miles de filas para calcular una cifra. Nuestra regla: lo que necesite ver los datos va a la base de datos —filtrar, contar, una escritura que dependa de lo que ya hay— y las reglas de negocio permanecen en la aplicación, porque la CPU de la base de datos es la más difícil de ampliar del sistema.

Actualizaciones que permanezcan en su página. Una fila que cambia muchas veces por minuto deja una versión nueva cada vez. Si una actualización no modifica ninguna columna indexada y queda espacio en la página, PostgreSQL conserva allí la nueva versión sin tocar ningún índice: una tupla solo de heap, o actualización HOT. Mantenga fuera de los índices las columnas que cambian con frecuencia y deje espacio en las páginas:

```
ALTER TABLE units SET (fillfactor = 90);
```

La configuración se aplica a las páginas escritas a partir de entonces, así que establézcala antes de cargar las filas. En la tabla anterior, 2.000 reservas que establecían `reserved`, `reserved_at` y `order_id` fueron todas HOT. De 2.000 actualizaciones que establecían `sold`, leído por el índice parcial, ninguna fue HOT. La venta ocurre una vez por unidad; la reserva puede ocurrir muchas. La relación entre `n_tup_hot_upd` y `n_tup_upd` en `pg_stat_user_tables` muestra la proporción en un sistema activo.



Una fila dice quién llegó primero

La unidad corresponde al primer comprador cuya sentencia la alcanza, y la reserva caduca según el reloj de la base de datos. La representan dos columnas: `reserved`, un booleano, y `reserved_at`, el momento en que se estableció.

```
UPDATE units
   SET reserved = true,
       reserved_at = now(),
       order_id = :order
 WHERE id = :unit
   AND NOT sold
   AND (NOT reserved
        OR reserved_at
           < now() - interval '10 minutes')
RETURNING id;
```

Una fila devuelta: este comprador la tiene. Ninguna: alguien llegó antes o ya está vendida. Dos reservas de la misma unidad hacen cola en su bloqueo de fila. En READ COMMITTED, el nivel predeterminado de PostgreSQL, la segunda vuelve a

evaluar `WHERE` sobre la fila confirmada por la primera, así que solo una recibe una fila. Una reserva de más de diez minutos cuenta como libre; no hace falta limpiarla antes para que la respuesta sea correcta.

Por eso el booleano por sí solo es una respuesta incompleta. `reserved` sigue siendo verdadero tras caducar la reserva, y solo ambas columnas juntas indican si la unidad está retenida. Toda consulta debe comprobar las dos. Un barrido periódico que restablezca las reservas caducadas mantiene la tabla coherente para quien lea `reserved` a solas, pero la corrección nunca espera a ese barrido. Una restricción mantiene coherente el par:

```
ALTER TABLE units
  ADD CONSTRAINT reserved_has_time
  CHECK (NOT reserved
         OR reserved_at IS NOT NULL);
```

La nota de las entradas almacena la fecha límite, `held_until`, en vez del comienzo; es el mismo diseño. Cuando el comprador elige una unidad —un asiento, una impresión numerada—, la sentencia la identifica, como aquí. Si sirve cualquier unidad, la variante `SKIP LOCKED` de esa nota elige una que nadie más esté reteniendo.



Sacar las lecturas del primario

La mayor parte del tráfico de una tienda son lecturas: la página del producto, el aviso «quedan 3», el carrito, el historial de pedidos. Nada de eso necesita el primario, la máquina que debe quedar libre para escribir. Una réplica en streaming asume esas lecturas. Añadirla es configuración, mientras que un fragmento es un proyecto.

Hay dos precauciones. Una réplica va algo por detrás: un comprador que acaba de reservar una unidad y después lee su carrito desde ella puede no ver la reserva. Lea del primario lo que un usuario acaba de escribir o diríjalo al primario unos segundos después de cada escritura. Además, un informe largo en una réplica puede cancelarse si retrasa la aplicación de los cambios replicados. La documentación de PostgreSQL los llama conflictos de consulta y enumera los ajustes que los contrapesan con el retraso de replicación.

La lectura más barata es la que nunca llega a una base de datos. Cuando todas las unidades de un artículo están reservadas, nadie puede comprar hasta que caduque o se libere una reserva. La base de datos puede indicar exactamente cuándo:

```
SELECT count(*) FILTER (
         WHERE NOT reserved
            OR reserved_at
               < now() - interval '10 minutes')
         AS free,
       min(reserved_at) + interval '10 minutes'
         AS until
  FROM units
 WHERE item_id = :item
   AND NOT sold;
```

Cuando `free` es cero, `until` es el instante en que caduca la primera reserva: almacene en caché «todo reservado» con esa caducidad, nunca mayor. Hasta entonces, la caché rechaza a cada visitante en lugar de la base de datos. Si `free` es cero y `until` está vacío, todas las unidades están vendidas y la respuesta es definitiva. Una reserva liberada antes —una tarjeta rechazada, un carrito abandonado— elimina la entrada de caché para poner la unidad a la venta de inmediato.

La caché solo dice no. El sí siempre procede del `UPDATE` condicional anterior, de modo que una caché desactualizada puede costar una venta durante unos segundos, pero nunca vender una unidad dos veces. Es el «agotado» en caché de la nota de las entradas, con una fecha límite.

Particionar las tablas que rodean la fila conflictiva

El particionamiento divide una tabla en varias por una clave —un mes, una organización cliente— bajo un único nombre. Conviene decir claramente lo que no hace: no acelera una fila disputada. Esa fila sigue en una partición, en una máquina, detrás de un bloqueo.

Lo que hace es mantener pequeñas las tablas que rodean el punto conflictivo. Los pedidos, el historial de reservas y los eventos crecen sin parar, y la mayoría de las consultas quiere la parte reciente:

```
CREATE TABLE orders (
  id          bigint NOT NULL,
  created_at  timestamptz NOT NULL
              DEFAULT now(),
  item_id     integer NOT NULL,
  quantity    smallint NOT NULL
              CHECK (quantity > 0),
  paid        boolean NOT NULL DEFAULT false,
  PRIMARY KEY (id, created_at)
) PARTITION BY RANGE (created_at);

CREATE TABLE orders_2026_10
  PARTITION OF orders
  FOR VALUES FROM ('2026-10-01')
              TO ('2026-11-01');
```

Una consulta con un intervalo de fechas lee solo las particiones incluidas, y los índices del mes siguen siendo lo bastante pequeños para caber en memoria. La retención deja de ser un `DELETE` de millones de filas, con el crecimiento innecesario y la limpieza posteriores, y pasa a ser una sentencia:

```
ALTER TABLE orders
  DETACH PARTITION orders_2025_10;
```

La clave primaria debe incluir la de partición; por eso es `(id, created_at)`. Una consulta que no indique la clave lee todas las particiones, así que particione por aquello que ya filtran las consultas.



Cuándo es razonable el sharding

Hay un límite que el sharding no puede mover: cada artículo tiene un único responsable. Una unidad no puede reservarse en dos servidores a la vez, así que se mueven artículos completos, nunca la mitad de uno. Cuando un artículo supera la capacidad de su máquina, los buckets de la nota de las entradas dividen sus existencias en partes menores, cada una de nuevo con un único responsable.

Dentro de ese marco, consideramos razonable el sharding cuando se cumple una de estas condiciones, y no antes:

- **Una máquina está llena de trabajo útil.** Las sentencias ya son ligeras — una escritura condicional cada una, nada externo dentro de una transacción, actualizaciones que permanecen en su página— y aun así la CPU o el registro de escritura anticipada del primario se saturan en la mayor máquina que merece la pena comprar, con muchos artículos y no solo uno. Un artículo muy solicitado es un problema de buckets, no de fragmentos.
- **Los datos han superado su capacidad de mantenimiento.** Una copia de seguridad no puede restaurarse dentro del tiempo prometido al negocio, un índice no puede construirse o una tabla limpiarse en horas tranquilas, o el conjunto de trabajo ya no cabe en memoria.
- **El aislamiento es el requisito.** Una organización cliente cuyo contrato o regulador exige separar sus datos, o cuya venta no debe compartir disco con otra.

El coste recae sobre todo lo demás. Las migraciones, las copias de seguridad, las restauraciones y los informes deben tener en cuenta los distintos fragmentos; un join entre fragmentos puede pasar al código de la aplicación, y una consulta que desconoce su clave de fragmentación puede tener que preguntar a todos. Esa es la factura que los apartados anteriores intentan evitar, y una vez pagada cuesta volver atrás.

Cuando una disculpa cuesta menos que un bloqueo

Existe un argumento serio a favor de ir en sentido contrario. Dejar que cada servidor venda del stock que ve, aceptar que alguna unidad vaya a dos compradores y permitir que atención al cliente reembolse a uno con una disculpa. La lógica de negocio pasa a ser de consistencia eventual, diseñada para el 99,9 % de solicitudes que no colisionan. Un bucle asíncrono encuentra después las colisiones y las compensa. A cambio, la tienda nunca tiene que decir «inténtelo de nuevo». Helland y Campbell presentan el argumento con mayor fuerza en *Building on Quicksand*: un sistema que confirma trabajo antes de que su copia de seguridad lo tenga convierte cada promesa en probabilística, y las empresas siempre han resuelto con disculpas las promesas que no pudieron cumplir.

Estamos de acuerdo con buena parte. Discrepamos en dónde se paga ese coste.

La mayor parte de una tienda debería tener consistencia eventual. El catálogo, el índice de búsqueda, el aviso «quedan 3», el carrito, las recomendaciones, los correos, la exportación al almacén y los informes pueden ir segundos por detrás. Hacerlos esperar al primario cuesta velocidad y no aporta nada. Dynamo se creó justamente para esto: el carrito de Amazon nunca olvida ni rechaza un «añadir al carrito», incluso con máquinas averiadas, a cambio de que una fusión pueda recuperar un artículo que el comprador había quitado. Tras la reserva, el checkout de la nota de las entradas ya es asíncrono: una saga de pasos locales, mensajes entregados al menos una vez y consumidores idempotentes identificados por el pedido. Un bucle de conciliación pertenece a cualquier diseño que hable con un proveedor de pagos, porque los webhooks se pierden.

La reserva es el único punto donde no vale. «Raro» supone colisiones raras, y en una venta relámpago las colisiones son la venta. Un servidor que vende desde su propia vista sigue vendiendo hasta enterarse de lo vendido por los demás. Por eso la sobreventa equivale aproximadamente a la tasa de intentos por el retraso de replicación. En una simulación sencilla, dos regiones se reparten 2.000 intentos por segundo, cada una vende según su contador y conoce las ventas de la otra 200 milisegundos después. En cinco ejecuciones de una venta de 10.000 unidades, cada ejecución vendió entre 191 y 205 unidades inexistentes. La aritmética predice 200: tasa de intentos por retraso por proporción de ventas de la otra región. Con un único responsable por artículo, el mismo modelo vendió exactamente el stock. Ocurre en cada agotamiento de existencias, haya fallado algo o no, y en el momento culminante, cuando más compradores miran.

La ganancia de disponibilidad es menor de lo que parece. Gilbert y Lynch demostraron el compromiso de la consistencia eventual: durante una partición de red, un sistema puede responder o ser consistente, pero no ambas cosas. PACELC de Abadi añade la otra mitad. Sin partición, el compromiso es entre latencia y consistencia, y un responsable por artículo con sentencias ligeras ya es rápido. Ningún diseño alcanza el 100 %: el proveedor de pagos, DNS y las redes intermedias sufren sus propios fallos. Lo que cede el diseño estricto es limitado. Si no se puede llegar al responsable del stock de un artículo, ese artículo deja de venderse durante los segundos de la conmutación por error. El apartado sobre pausar el artículo de la nota de las entradas limita la pausa a los artículos de la máquina caída. El resto de la tienda sigue vendiendo.

Una disculpa no es un reembolso. El reembolso es la parte barata. Al comprador se le dijo que sí, y quizá ya reservó el tren, pidió el día libre y avisó a sus amigos. Oír «agotado» un segundo antes es una decepción. Oír que se retira un pedido confirmado es una promesa rota. Atención al cliente tiene personal para un día normal y las sobreventas llegan juntas, en el minuto de mayor actividad del año. Antes de planear retirar pedidos confirmados, consulte con un abogado a qué compromete una confirmación a la tienda.

Conserve la parte asíncrona y muévala antes del sí. El acierto de la propuesta es que el comprador no necesita esperar a un bloqueo. Acepte cada clic en un registro duradero y replicado, una partición por artículo, y comunique que la respuesta está en camino. Aceptar un clic depende entonces solo del registro, y uno replicado en tres máquinas sigue admitiendo escrituras si falla una. Un responsable por artículo lee el registro en orden, decide a velocidad de memoria y devuelve la respuesta como mensaje en unos segundos. Cuando ese responsable conmuta por error, las solicitudes esperan en el registro en vez de rechazarse o venderse dos veces. El sí es una reserva y el pago viene después, como en el checkout de la nota de las entradas, de modo que nadie paga por una unidad que no obtuvo. La conciliación sigue existiendo, pero antes de la promesa, no después.

Reparta el stock, no el contador. Si el negocio necesita que las regiones vendan sin consultarse, asigne a cada una una cuota de existencias y deje que venda de ella sin preguntar. La región que agota su cuota pide más, y al final de la venta las cuotas vuelven a concentrarse en un responsable. Bailis y sus colegas analizan qué reglas pueden mantener los servidores sin comunicarse. Un stock que no debe bajar de cero no es una de ellas. Las cuotas no eliminan la conversación: la sacan de la ruta crítica y la llevan al momento infrecuente de reponer una cuota. Ese es el diseño de consistencia eventual que mantiene la invariante. Su único coste es el tramo final, en el que un comprador es rechazado en una región mientras otra conserva stock.

Cuándo la sobreventa es la política correcta. Algunos bienes pueden producirse de nuevo: una impresión bajo demanda, una camiseta que aún tiene el proveedor, una clave de licencia. Vender más allá del contador es entonces un pedido pendiente con una fecha honesta, y la consistencia eventual es el diseño adecuado. Las aerolíneas sobrevenden a propósito. Conocen las tasas de pasajeros que no se presentan, ponen precio al riesgo y compensan según una regla escrita de antemano. Esa es la prueba. La sobreventa es aceptable cuando el negocio eligió su magnitud y su remedio antes de vender. Es un defecto cuando recae sobre quien hizo clic el último por casualidad.

Si elige las disculpas, constrúyalas como los pagos. El bucle descrito es un proceso de reembolso y necesita lo mismo que todo lo que mueve dinero:

- **Un identificador por venta**, para que un reintento sea la misma venta, y un libro de ventas que solo permita añadir registros. El stock se deriva del libro, nunca se modifica directamente.
- **Una regla sobre quién conserva la unidad** que pueda explicarse al comprador que la perdió. Ordene por el registro, no por los relojes de distintos servidores, donde unos milisegundos de desfase decidirían quién recibe el reembolso.
- **Compensaciones idempotentes y registradas:** cancelar, anular o reembolsar, avisar al comprador. Son los pasos de deshacer de la saga, ejecutados por una máquina.
- **Un plazo.** El bucle debe encontrar cada sobreventa antes de enviar nada y antes de que la descubra el comprador.
- **Un presupuesto medido.** Sobreventas por venta frente a la cifra aprobada por el negocio, junto a la invariante en el mismo panel, con una alerta al agotar el presupuesto.

Nuestra respuesta es consistencia eventual alrededor de la reserva, consistencia estricta en ella y asincronía al llevar la respuesta al comprador. Diseñe para el camino habitual, pero en una venta relámpago la colisión es el camino habitual. Si el negocio quiere algunas sobreventas a cambio de algo, que sea una cifra elegida por él, con un remedio escrito antes de la venta, y mídala.



Qué le diríamos a la siguiente persona

- **Lea antes de comprar.** `pg_stat_statements` y un `EXPLAIN` responden a más preguntas de capacidad que un servidor nuevo.
- **Haga que los tipos digan qué son los datos.** Un booleano almacenado como booleano ocupa menos y rechaza absurdos.
- **Envíe a la base de datos lo que necesita ver los datos y nada más.** Filtrar y contar van ahí. Las reglas de negocio, no.
- **La caché dice no; solo la base de datos dice sí.** Así una caché desactualizada cuesta segundos, nunca una unidad.
- **El particionamiento es mantenimiento.** Mantiene pequeñas las tablas alrededor de una fila conflictiva y no hace nada por la fila.
- **Fragmente cuando una máquina esté llena de trabajo útil**, no cuando esté llena de esperas.

- **Concilie antes del sí, no después.** Todo lo que rodea la reserva puede tener consistencia eventual. Aplicada a la propia reserva, produce sobreventa según la tasa de intentos por el retraso, en cada venta.



Fuentes

- PostgreSQL: pg_stat_statements
- PostgreSQL: uso de EXPLAIN
- PostgreSQL: el sistema de estadísticas acumuladas
- PostgreSQL: tipos numéricos
- PostgreSQL: tipos de caracteres
- PostgreSQL: tipo booleano
- PostgreSQL: índices parciales
- PostgreSQL: tuplas solo de heap (HOT)
- PostgreSQL: particionamiento de tablas
- PostgreSQL: hot standby y conflictos de consulta
- Pat Helland y Dave Campbell, Building on Quicksand
- Seth Gilbert y Nancy Lynch, Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services
- Daniel Abadi, Consistency Tradeoffs in Modern Distributed Database System Design
- DeCandia et al., Dynamo: Amazon’s Highly Available Key-value Store
- Hector Garcia-Molina y Kenneth Salem, Sagas
- Bailis et al., Coordination Avoidance in Database Systems



Todas las entradas del blog

Mida antes de cambiar

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 →

Hable de sus observaciones

Hablemos de «Optimice la base de datos antes de fragmentarla» en el contexto de su proyecto. Cuéntenos qué observó y cómo sería un resultado útil.

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://data.engineer.company/es/notes/optimise-before-you-shard/