Engineer

Transacción

Un grupo de cambios de base de datos de todo o nada

Qué es

Una transacción es un grupo de cambios que una base de datos aplica como una unidad: todos surten efecto o ninguno lo hace. Se especifica transacción de base de datos cuando hay que distinguirla de un pago, que comparte la palabra. El tutorial de PostgreSQL la describe como algo que «agrupa varios pasos en una sola operación de todo o nada», sin que ningún estado intermedio sea visible para los demás.

En PostgreSQL una transacción se abre con `BEGIN` y termina con `COMMIT`, que hace permanentes sus cambios y los vuelve visibles para las demás sesiones, o con `ROLLBACK`, que los descarta. Una sentencia enviada por sí sola también se ejecuta dentro de una: PostgreSQL la envuelve en un `BEGIN` implícito y, si tiene éxito, un `COMMIT` implícito. Una transacción larga puede establecer un `SAVEPOINT` y volver a él después con `ROLLBACK TO`, descartando el trabajo posterior a ese punto y conservando todo lo anterior.

Las garantías de una transacción se resumen en ACID. La atomicidad significa que se completa como una unidad o no se completa, y que ningún resultado a medias sobrevive a una caída. La consistencia significa que las restricciones declaradas sobre los datos se cumplen al confirmar, y que una transacción que dejaría alguna incumplida se revierte. El aislamiento significa que sus cambios permanecen invisibles para las transacciones simultáneas hasta la confirmación. La durabilidad significa que un cambio confirmado sobrevive a una caída del servidor.

El aislamiento tiene distintos niveles, y el estándar SQL define cuatro: Read Uncommitted, Read Committed, Repeatable Read y Serializable. PostgreSQL inicia cada transacción en Read Committed salvo que se indique otra cosa, y su nivel Read Uncommitted se comporta exactamente como Read Committed. En Read Committed cada sentencia ve los datos confirmados antes de empezar esa

sentencia, así que dos consultas de una misma transacción pueden ver cifras distintas. En Repeatable Read toda la transacción trabaja sobre una instantánea tomada cuando se ejecuta su primera sentencia que no sea de control de transacciones. Serializable va más allá y hace que las transacciones simultáneas produzcan el mismo resultado que si se hubieran ejecutado una tras otra.

Utilice el concepto en una decisión

Aplique «¿Qué es una transacción?» a una decisión real. Podemos ayudarle a relacionar el concepto con el sistema que está construyendo o comprando. 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 →

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. 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 →

Por qué importa

Casi toda acción que merece registrarse afecta a más de una fila. Un pedido supone una fila en una tabla, una reducción en otra y un asiento en un libro de movimientos. Sin una transacción, una caída, un tiempo de espera agotado o una restricción incumplida entre la segunda y la tercera sentencia dejan un pedido cuyo stock nunca se descontó o stock descontado para un pedido inexistente. Después alguien lo descubre en un informe y lo corrige a mano. Con una transacción, ese estado a medias nunca existe a la vista de nadie, y es la base de datos, no la aplicación, la que lo garantiza.

La segunda razón es la concurrencia. Dos solicitudes que llegan juntas se ejecutan en paralelo, y el nivel de aislamiento decide qué ve cada una mientras los bloqueos deciden cuál espera. Aquí es donde más suelen sorprenderse los equipos: envolver código en `BEGIN` y `COMMIT` no impide por sí solo que dos solicitudes lean el mismo saldo y ambas lo gasten. En Read Committed cada sentencia lee lo que estaba confirmado cuando empezó, de modo que entre una comprobación realizada en una sentencia y una escritura realizada en la siguiente puede interponerse la confirmación de otra sesión. La comprobación debe bloquear primero la fila o formar parte de la escritura.

Las transacciones también tienen un coste, que se paga en tiempo. Los bloqueos de fila adquiridos por `UPDATE`, `DELETE` y `SELECT ... FOR UPDATE` suelen durar hasta que termina la transacción; volver a un punto de guardado libera los adquiridos después de él. Una sesión que quiera un bloqueo incompatible debe

esperar, salvo que el vencimiento del tiempo de espera, una cancelación o la resolución de un interbloqueo interrumpan la espera. Una transacción abierta también impide que `VACUUM` elimine versiones antiguas de filas que todavía podría necesitar ver, por lo que dejarla abierta durante horas permite que las tablas acumulen espacio innecesario. La decisión de diseño nunca es solo si utilizar una transacción: es dónde poner sus límites y cuánto reducir lo que ocurre entre ellos.

Por último, una transacción local ordinaria de base de datos no incluye por sí sola los efectos externos. Una llamada a un proveedor de pagos, un correo o un mensaje enviado a una cola separada no se deshacen al revertir la base de datos. Coordinar otros sistemas requiere un diseño adicional; envolverlos en un `BEGIN` no basta. Decidir qué queda dentro del límite y cómo se entera el mundo exterior de una confirmación constituye la mayor parte del trabajo de diseñar una transacción.

Cómo se aplica en la práctica

Un miembro de un programa de fidelización gasta 250 puntos en un vale de cine. Tres cosas deben ocurrir juntas: el monedero pierde los puntos, el libro de movimientos registra el motivo y se crea el canje. Como una sola transacción, se escribe así:

```
BEGIN;

UPDATE wallet
SET    points = points - 250
WHERE  member_id = 42
AND    points >= 250
RETURNING points;
-- no row back: not enough points, so ROLLBACK and stop here

INSERT INTO points_entry (member_id, delta, reason)
VALUES (42, -250, 'redeemed: cinema voucher');

INSERT INTO redemption (member_id, reward_id)
VALUES (42, 7);

COMMIT;
```

El `UPDATE` adquiere un bloqueo de fila sobre el monedero del miembro 42. Hasta `COMMIT`, cualquier otra sesión que lea el monedero sigue viendo el saldo anterior, porque los bloqueos de fila no bloquean las lecturas y los cambios sin confirmar son invisibles. Un segundo canje del mismo miembro, enviado desde otra pestaña un instante después, llega a su propio `UPDATE` y espera. Cuando se confirma la primera transacción, PostgreSQL en Read Committed vuelve a evaluar la cláusula `WHERE` de la segunda sobre la fila tal como ha quedado. Si solo quedan 100 puntos, la condición falla, no se devuelve ninguna fila y la segunda solicitud revierte sin haber escrito nada. Si, en cambio, falla el último `INSERT`, por ejemplo porque se ha retirado la recompensa 7 y una clave foránea lo rechaza, el cliente revierte y los puntos quedan exactamente donde estaban.

Esa es la estructura del monedero de fidelización con ligas e insignias. Cada modificación de un monedero adquiere un bloqueo de fila antes de leerlo, de modo que una segunda solicitud espera a que termine la primera en lugar de competir con ella, y, como el monedero está en la misma base de datos que el resto, el saldo y lo que ha pagado se confirman juntos o no se confirma ninguno. La página reconoce el precio: bloquear filas es la solución más lenta, y los miembros esperan su turno en lugar de gastar dos veces los mismos puntos.

La nota sobre vender la última entrada exactamente una vez aplica la misma herramienta a una venta relámpago. Su solución convierte la comprobación de stock y la reducción en un solo `UPDATE` condicional, porque un bloqueo mantenido mientras la aplicación decide entre un `SELECT` y un `UPDATE` hace esperar al siguiente comprador cada uno de esos milisegundos. La primera medida de su lista para el resto del trabajo es mantener fuera de la transacción todo lo externo: una llamada de pago, una solicitud HTTP o la publicación de un mensaje deben ir antes o después.

Dónde surgen los problemas

Una sesión queda inactiva dentro de una transacción y todas las que vienen detrás esperan. Una aplicación abre una transacción, actualiza una fila y después espera a una API lenta, al siguiente clic del usuario o a un proceso de trabajo que se cayó sin enviar nunca `COMMIT`. Sus bloqueos permanecen y, con los tiempos de espera desactivados, PostgreSQL hace que una sesión que necesita un bloqueo incompatible espere indefinidamente salvo que detecte un interbloqueo o se cancele la espera. La documentación considera una mala idea mantener una transacción abierta mientras se espera una entrada del usuario. `pg_stat_activity` muestra esas sesiones con el estado `idle in transaction`, y `xact_start` indica cuándo empezó la transacción:

```
SELECT pid, now() - xact_start AS open_for, state, query
FROM pg_stat_activity
WHERE state LIKE 'idle in transaction%'
ORDER BY xact_start;
```

Saque el trabajo lento de la transacción y dé al servidor un límite para que una transacción olvidada no mantenga bloqueos durante horas. `idle_in_transaction_session_timeout` termina cualquier sesión que permanezca inactiva dentro de una transacción abierta más tiempo que su valor. Su valor predeterminado es 0, lo que lo desactiva, así que hay que configurarlo, por ejemplo para el rol de la aplicación con `ALTER ROLE app SET idle_in_transaction_session_timeout = '30s';` .

Dos solicitudes gastan el mismo saldo. El código lee un valor en una sentencia, decide en la aplicación y escribe el resultado en otra. Dos sesiones leen la misma cifra, ambas superan la comprobación y ambas escriben, de modo que un cambio sobrescribe silenciosamente al otro. Un `BEGIN` alrededor no lo impide en Read Committed. Incorpore la comprobación a la escritura mediante una condición en el `UPDATE` , como en el ejemplo anterior, o bloquee la fila antes de leerla con `SELECT ... FOR UPDATE` , que hace esperar a la segunda sesión hasta que termine la primera.

Aparecen errores bajo carga en Repeatable Read o Serializable. En estos niveles PostgreSQL rechaza una transacción que incumpliría las garantías del nivel y notifica un fallo de serialización, SQLSTATE `40001` , con mensajes como `could not serialize access due to concurrent update` . Ese fallo es el comportamiento previsto del diseño, no un defecto, y la documentación establece que las aplicaciones que utilizan estos niveles deben estar preparadas para reintentar. Capture `40001` y vuelva a ejecutar toda la transacción desde `BEGIN` , incluidas sus lecturas, porque su decisión se basó en datos que ya han cambiado. Reintentar solo la sentencia que falló repite el error.

Dos transacciones se bloquean mutuamente. Una bloquea el pedido 1 y después el 2; otra bloquea el pedido 2 y después el 1, y cada una espera a la otra. PostgreSQL detecta el ciclo y aborta una de las transacciones para que la otra pueda terminar, y es difícil predecir cuál elegirá. La defensa documentada consiste en que todas las rutas de código adquieran los bloqueos en el mismo orden, por ejemplo actualizando las filas en orden ascendente de su clave primaria, y en reintentar una transacción abortada por un interbloqueo.

Se envía un correo para un pedido que se revirtió. La aplicación envía una confirmación o publica un evento desde dentro de la transacción y después falla una sentencia posterior. La reversión elimina el pedido, pero nada puede retirar el correo ya enviado. Registre el mensaje en una tabla outbox dentro de la misma transacción que el cambio que describe y deje que un proceso separado lo envíe una vez confirmada. El mensaje existirá entonces exactamente cuando exista el pedido.

Lecturas adicionales

- Transacciones — capítulo del tutorial de PostgreSQL: comportamiento de todo o nada, transacciones implícitas y puntos de guardado, con una transferencia bancaria como ejemplo.
- Aislamiento de transacciones — los cuatro niveles estándar, qué ve cada uno en PostgreSQL y los fallos de serialización que requieren reintentar.
- Bloqueo explícito — bloqueos de filas y tablas, cuánto duran y cómo se detectan y evitan los interbloqueos.
- Valores predeterminados de las conexiones de cliente — `default_transaction_isolation`, `idle_in_transaction_session_timeout`, `lock_timeout` y los demás límites de una sesión.
- Limpieza rutinaria con vacuum — por qué se conservan las versiones antiguas de las filas mientras alguna transacción pueda verlas y cómo afectan a la limpieza las transacciones abiertas durante mucho tiempo.

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 una transacción?» 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/transaction/