Transacciones

Tiempo estimado: 15min

A continuación, se presenta uno de los aspectos más importantes de las bases de datos de hoy en día: las transacciones. Generalmente, los sistemas de gestión de bases de datos son ACID, siendo SQL su principal abanderado. Las bases de datos NoSQL no suelen proporcionar transacciones ACID, aunque ArangoDB sí lo hace. Lo que favorece su uso en proyectos más interesantes que cuando no se soporta ACID al completo.

Comenzamos la lección introduciendo los conceptos de transacción y ACID, así como los tipos de transacción más extendidos en los motores de bases de datos. A continuación, se muestra cómo ejecutar transacciones mediante el *shell* de ArangoDB. Y finalmente, se presenta el concepto de transacción anidada.

Al finalizar la lección, el estudiante sabrá:

- Qué es una transacción.
- Qué es ACID.
- Qué tipos de transacciones hay y cuáles soporta ArangoDB.
- Cómo ejecutar transacciones mediante el *shell* de ArangoDB.

Introducción

Una transacción (*transaction*) es una unidad de trabajo indivisible, tratada de manera independiente de otras transacciones ejecutadas concurrentemente. El objetivo de una transacción es realizar una secuencia de operaciones de lectura y/o escritura de manera completamente independiente y cuyos cambios el motor de bases de datos debe garantizar se realizarán todos ellos o ninguno, dejando en cualquier caso la base de datos en un estado consistente. Si alguna de las operaciones de la transacción falla, el motor deshará automáticamente todos los cambios realizados hasta ese momento, dejando los datos modificados como estaban antes de iniciarse la transacción.

ArangoDB, al igual que SQL, garantiza que las transacciones cumplen las propiedades ACID, asegurando así que se procesan adecuadamente:

- Atomicidad (*atomicity*). Las operaciones que forman la transacción se ejecutarán completamente. No se permite que al finalizar la transacción haya aplicado unos cambios y otros no. O todos o ninguno, nada de medias tintas.

  Así pues, si la transacción finaliza con éxito, el motor de bases de datos garantiza que todos sus cambios se habrán aplicado. En cambio, si finaliza en fallo, garantiza que cualquier cambio que se haya realizado durante la transacción se deshará dejando los datos modificados como estaban antes de comenzar la transacción. Es más, si la instancia cae, el motor garantiza que deshará cualquier cambio realizado por cualquier transacción que estuviera sin finalizar antes de la caída, devolviendo así la base de datos a un estado consistente.

  La atomicidad se consigue básicamente mediante dos conceptos: la demarcación de la transacción y el registro de transacciones. Con la demarcación de la transacción, se indica cuándo comienza y cuándo acaba. Mientras que el registro de transacciones se utiliza para registrar cada operación realizada por la transacción, de tal manera que si la transacción finaliza en fallo, utilizará el registro para deshacer los cambios, esto es, devolver los datos a sus valores iniciales antes del comienzo de la transacción, tal como se guardó en el registro.

- Consistencia (*consistency*). Cuando la transacción finalice, el sistema quedará en estado consistente, cumpliéndose las restricciones de integridad de la base de datos. Durante la ejecución de la transacción, se puede relajar el cumplimiento de las restricciones, pero una vez finalizada la transacción todos los cambios cumplirán, sin excepciones, las restricciones de integridad, dejando así la base de datos en un estado consistente.

- Aislamiento (*isolation*). La ejecución de varias transacciones concurrentemente tendrá el mismo resultado que si sus operaciones se aplicaran secuencialmente, esto es, primero todas las de una transacción y después las de la otra.

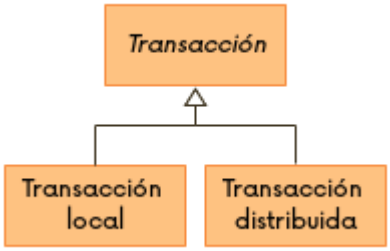
  Está relacionado con los datos que la transacción puede ver y lo que otras pueden ver acerca de los cambios realizados por la transacción.

  El aislamiento se consigue mediante bloqueos y el nivel de aislamiento. Para garantizar el aislamiento de la transacción con respecto a las demás, el motor bloquea los datos accedidos por la transacción; cuantos más bloqueos aplique, menor será el nivel de concurrencia en la base de datos.
- Durabilidad (*durability*). Una vez una transacción ha finalizado y confirmado que se han aplicado sus cambios, si se produce un fallo en la instancia, no habrá, bajo ningún concepto, pérdida de datos. Esto quiere decir que el motor de bases de datos garantiza que los cambios realizados por una transacción finalizada en éxito, no se perderán pase lo que pase.

  La durabilidad se consigue mediante el uso de un registro de transacciones.

Transacciones locales

Atendiendo a la ubicación de los recursos involucrados en una transacción, distinguimos entre transacciones locales y distribuidas.

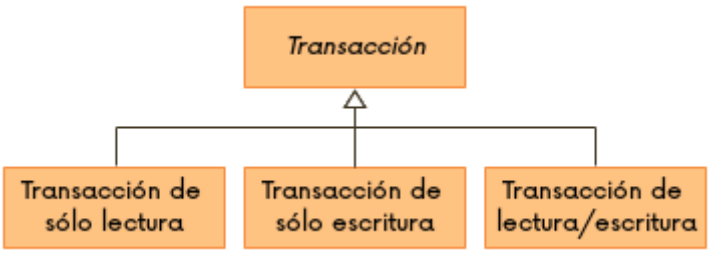


Una transacción local (*local transaction*) es aquella que afecta o actualiza una única base de datos. En cambio, una transacción distribuida (*distributed transaction*) es aquella que afecta o actualiza varias bases de datos, de la misma u otra instancia.

Actualmente, ArangoDB sólo soporta transacciones locales.

Transacciones de lectura/escritura

Atendiendo a lo que la transacción puede hacer con los datos, se distingue entre transacción de sólo lectura, de sólo escritura o de lectura/escritura.



Una transacción de sólo lectura (*read-only transaction*) es aquella que sólo puede realizar operaciones de lectura de datos. En cambio, una transacción de sólo escritura (*write-only transaction*) sólo de escritura. Mientras que una transacción de lectura/escritura (*read/write transaction*) puede realizar ambos tipos de operaciones.

Atendiendo al tipo de transacción, el motor realizará unos bloqueos u otros, los cuales reducirán en mayor o menor medida la concurrencia del sistema. Las operaciones de escritura se llevan a cabo mediante bloqueos más estrictos que reducen lo que se puede hacer con los recursos bloqueados.

ArangoDB permite trabajar con los tres tipos de transacción.

Demarcación de transacciones

La demarcación (*demarcation*) fija los límites de una transacción. Donde un límite (*boundary*) marca el inicio o fin de la transacción.

El inicio de una transacción (*transaction begin*) indica el comienzo de la transacción. Todas las operaciones que se lleven a cabo desde el inicio hasta el final de la transacción formarán la unidad indivisible. En SQL, por lo general, se marca mediante el comando BEGIN TRANSACTION.

El fin de una transacción (*transaction end*), por su parte, marca el término de la transacción. Se distingue entre confirmación y aborto. La confirmación (*commit*) hace efectivos todos los cambios realizados desde el inicio de la transacción, asegurando que todos los cambios se vuelcan a la base de datos. En cambio, el aborto (*rollback*) interrumpe la ejecución normal de la transacción, indicando su fracaso y deshaciendo cualquier cambio que se haya volcado a la base de datos, asegurando dejarla como estaba antes del inicio de la transacción. En SQL, las confirmaciones se marcan con el comando COMMIT y el aborto con ROLLBACK.

Delimitación explícita de transacción

En ArangoDB, una transacción se representa mediante una función JavaScript que contiene el código de la transacción. Si la función propaga un error, el motor considerará que se ha producido un error y la transacción debe abortarse. En cambio, si finaliza sin fallos, la transacción se confirmará.

Para ejecutar una transacción, hay que tener claro las siguientes cosas:

- Qué colecciones serán accedidas en la transacción y el modo en que se accederán.
- Qué operaciones de lectura y/o escritura realizará la transacción.
- Qué datos deben pasarse como parámetros a la transacción.

Una vez lo tenemos claro, lo siguiente es pasar a implementar la transacción. Por un lado, hay que saber cómo invocar transacciones mediante el *shell* de ArangoDB. Para este fin, se utiliza el método _executeTransaction() del objeto db:

```
_executeTransaction(def)
```

Parámetro	Tipo de datos	Descripción

def	object	Propiedades de definición de la transacción:
		• collections (object). Colecciones accedidas y modo en el que serán accedidas:
		  ◦ read (string o string[]). Nombre de las colecciones accedidas en modo lectura.
		  ◦ write (string o string[]). Nombre de las colecciones accedidas en modo escritura.
		• action (function). El código de la transacción: fn(params).
		• params (object). Parámetros a pasar a la función action.
		• waitForSync (boolean). ¿Transacción síncrona?
		• lockTimeout (number). Tiempo máximo que puede esperar la transacción a que se libere un bloqueo.

Veamos un ejemplo ilustrativo:

```
db._executeTransaction({
  collections: {
    read: "bands",
    write: "artists"
  },
  action: function() {
    const db = require("@arangodb").db;

    for (let b of db.bands.all()) {
      db.artists.insert(b);
    }
  }
})
```


Nivel de aislamiento

Un nivel de aislamiento (*isolation level*) fija a qué datos, accedidos por la transacción, el motor debe aplicar bloqueos para impedir su acceso a otras transacciones. Cuanto mayor es el nivel de aislamiento, mayor número de objetos serán bloqueados, cayendo así el nivel de concurrencia de la instancia.

Cuanto más bajo es el nivel de aislamiento, mayor cantidad de datos se puede consultar y, por lo tanto, incrementa el nivel de concurrencia de las transacciones, pero huelga decir que incrementa también la posibilidad de efectos colaterales como, por ejemplo, las lecturas sucias o las actualizaciones perdidas. Digamos que menos probabilidades hay de que dos o más transacciones se bloqueen entre sí. En cambio, cuanto mayor es el nivel de aislamiento, mayores son los recursos invertidos por la instancia para conseguirlo e incrementa el número de bloqueos y, por ende, reduce el nivel de concurrencia de la instancia, porque las transacciones tendrán que esperar a que los datos que desean acceder se desbloqueen para así poder usarlos.

Las colecciones que pueden ser accedidas por la transacción se especifican mediante la propiedad collections. Atendiendo a cómo es accedida cada una de ellas, se indicará en collections.read o collections.write. Si se intenta acceder a una colección en un modo que no sea el indicado, se propagará un error y la transacción será abortada. Los modos se utilizan para aplicar el mejor bloqueo a las colecciones al comienzo de la transacción. No se aplica el mismo bloqueo a una colección que es accedida sólo en lectura que en escritura.

Función de transacción

El código de la transacción se proporciona mediante una función en la propiedad action. Esta función la invoca el motor pasándole los parámetros indicados en la propiedad params. No se recomienda utilizar clausuras (*closures*). No olvide esto: si hay que pasar datos a la transacción, hay que pasarlos mediante la propiedad params y, de ahí, se pasarán al parámetro params de la función. Veámoslo mediante un ejemplo:

```
var key = "la que sea";

db._executeTransaction({
  collections: {
    read: "bands",
    write: "artists"
  },
  action: function(params) {
    const db = require("@arangodb").db;
    db.artists.insert(db.bands.document(params.key));
  },
  params: {
    key: key
  }
})
```

Para acceder al objeto de base de datos, hay que utilizar la propiedad db del módulo @arangodb. A partir del objeto db, podemos acceder a las colecciones, tanto mediante la API simple como con AQL. En la función de la transacción no se puede crear, ni suprimir objetos de tipo colección o índice.

Aborto explícito de transacción

Para abortar la transacción, por la razón que sea, basta con propagar una excepción en la función action. Al capturarla la instancia, se dará por terminada la transacción deshaciendo todos los cambios realizados hasta el momento.

Delimitación implícita de transacción

Hemos visto que las transacciones están formadas por una secuencia de operaciones. También es posible que sólo estén formadas por una única operación. Para facilitar este tipo de transacciones, se dispone del modo de autoconfirmación (*autocommit mode*), el cual indica que cada operación ejecutada lleva implícita una confirmación de la transacción al finalizarse. Se trata de transacciones con una única operación. Desde este modo, se puede abrir, en cualquier momento, una transacción delimitada explícitamente mediante el método db._executeTransaction(). Al finalizarse la transacción delimitada, la sesión vuelve al modo autoconfirmación.

En ArangoDB, cualquier operación ejecutada fuera de una función de transacción se ejecuta en modo autoconfirmación. Cada vez que ejecutamos un insert(), un update(), un remove() o cualquier otra operación DML de manera aislada, sin encontrarse en una función de transacción, se estará ejecutando bajo una transacción implícita debido al modo de autoconfirmación.

Transacciones anidadas

Una transacción anidada (*nested transaction*) es aquella que se define dentro de otra transacción. La transacción dentro de la cual se define la anidada, se conoce formalmente como transacción externa (*outer transaction*), siendo la transacción raíz (*root transaction*) la madre de la jerarquía de transacciones.

Generalmente, cada motor de bases de datos implementa las transacciones de manera distinta, pero todas tienen una cosa en común: los cambios realizados por la transacciones anidadas no serán visibles hasta que la transacción raíz finalice.

En ArangoDB, *no* se permite el anidamiento de transacciones.

