UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CAMPUS D'ALCOI

DSIC

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Sistemas de almacenamiento y procesado distribuido

Laboratorio 2

Javier Esparza Peidro – jesparza@dsic.upv.es

Contenido

1. Introducción

En este laboratorio ampliamos las capacidades del almacén virtual que comenzamos a diseñar en el laboratorio 1. Se propone añadir las siguientes funcionalidades:

- Operaciones de definición de datos.
- Operaciones de manipulación de datos.
- Restricciones de integridad.

En los siguientes apartados se tratarán estos aspectos.

2. Operaciones DDL

Las operaciones DDL (Data Definition Language) permiten definir la estructura de la base de datos. En nuestro caso, esta estructura queda reflejada en el fichero del catálogo. Sin embargo, hasta el momento, hemos asumido que dicha estructura existía previamente en las fuentes de datos.

Ahora pretendemos crear la estructura necesaria de manera explícita. Para ello, será necesario añadir nuevas operaciones a la Driver API.

`driver.Connection.create(collection, fields)`

Crea una nueva colección en la fuente de datos. El parámetro *collection* contiene el nombre de la colección y el parámetro *fields* contiene una lista con los detalles de todos los campos (procedente del catálogo).

En un almacén clave-valor a priori no sería necesario hacer nada. Sin embargo, es importante recordar los campos que posee una determinada colección, para otras operaciones. Por tanto, sería conveniente almacenar la información sobre campos, por ejemplo bajo la entrada /database/collection.

En un almacén relacional implica la creación de una tabla. Hay que tener en cuenta el orden de los campos, y recordarlos para posteriores inserciones. Por tanto, sería conveniente almacenar la información sobre campos, por ejemplo en una tabla cfg, o bien en un fichero collection.table o similar.

En un almacén de documentos se podría argumentar algo parecido a los casos anteriores.

`driver.Connection.destroy(collection)`

Elimina una colección.

En un almacén clave-valor, implica eliminar todas las entradas cuya clave comience por el prefijo /database/collection.

En un almacén relacional habría que eliminar la tabla asociada a la colección.

En un almacén de documentos habría que eliminar la colección.

Una vez se dispone de estas operaciones en la Driver API, se propone incluir nuevas operaciones en el almacén virtual implementado en *virtualdb.py*.

`virtualdb.create():`

Crea la infraestructura necesaria para implementar la base de datos virtual en las fuentes de datos. Para ello, será necesario recorrer el catálogo y crear cada colección en cada fuente de datos, utilizando para ello el driver adecuado.

`virtualdb.destroy():`

Destruye la infraestructura de la base de datos virtual en las fuentes de datos. Para ello, será necesario recorrer el catálogo y destruir cada colección en cada fuente de datos, utilizando para ello el driver adecuado.

Por último, sería necesario añadir dos operaciones nuevas al CLI, por ejemplo:

```
$ python3 cli.py catalog.yaml
virtual> .help
Available commands within the prompt
    <sql>: execute SQL query or statement
    .help: print help
    .create: create the virtual database
    .destroy: destroy the virtual database
    .describe: print virtual schema
    .exit: close the current connection
virtual> .create
virtual database created.
virtual> .destroy
virtual database destroyed.
virtual>
```

3. Operaciones DML

Una vez se ha creado la infraestructura necesaria para albergar la base de datos virtual, es posible manipular su contenido con sentencias DML (Data Manipulation Language). Consideramos en esta sección las sentencias INSERT, UPDATE y DELETE, que verifiquen la siguiente sintaxis:

```
<DML> ::= <INSERT> | <UPDATE> | <DELETE>
<INSERT> ::= INSERT INTO <ID> VALUES (<CONST> {, <CONST>})
<UPDATE> ::= UPDATE <ID> SET <ID>=<CONST> {,<ID>=<CONST>}
             WHERE <COND> {AND <COND>}
<DELETE> ::= DELETE FROM <ID> WHERE <COND> {AND <COND>}
<ID> ::= ([a-z] | [A-Z]) {[0-9] | [a-z] | [A-Z]}
<COND> ::= <ID> <OP> <CONST>
<OP> ::= <> | = | <= | < | >= | >
<CONST> ::= <STR> | <FLOAT> | <INT>
<STR> ::= ‘{[0-1] | [a-z] | [A-Z]}’
<FLOAT> ::= <INT> [.{[0-9]}]
<INT> ::= 0 | [1-9] {[0-9]}
```

Para implementar estas operaciones en el almacén virtual, primero deberemos añadir nuevas operaciones de manipulación de agregados a la Driver API. Se proponen las siguientes:

`driver.Connection.insert(collection, doc)`

Inserta el agregado *doc* a la colección *collection*.

En un almacén clave-valor será necesario obtener primero su clave primaria, porque se utiliza para construir la clave /database/collection/id. A continuación se transformará el documento a JSON y se insertará la nueva entrada.

En un almacén relacional será necesario construir la sentencia SQL INSERT adecuada, teniendo en cuenta el orden de los campos.

En un almacén de documentos la inserción sería directa.

`driver.Connection.update(collection, query, data)`

Actualiza los agregados de la colección *collection* que se obtienen tras aplicar el filtro *query.* El parámetro *data* especifica los campos que se deben modificar en el agregado.

En un almacén clave-valor será necesario obtener primero los agregados que verifican la *query*. Después habría que modificar sus atributos y finalmente sería necesario volver a escribir las entradas asociadas.

En un almacén relacional será necesario construir la sentencia SQL UPDATE apropiada.

En un almacén de documentos será necesario actualizar los documentos que verifiquen la *query*.

`driver.Connection.delete(collection, query)`

Elimina los agregados de la colección *collection* que verifican el filtro *query.*

En un almacén clave-valor será necesario obtener todos los agregados que veirifican la *query* y posteriormente eliminar dichas entradas.

En un almacén relacional será necesario construir la sentencia SQL DELETE adecuada.

En un almacén de documentos será necesario eliminar los documentos que verifiquen la *query*.

Una vez la Driver API dispone de las operaciones DML, es posible propagarlas al almacén virtual implementado en *virtualdb.py*.

`virtualdb.execute(sql):`

Ejecuta una sentencia DML. Internamente, deberá de identificar qué tipo de operación es, y redirigir la operación al driver adecuado.

Por último, el CLI deberá reconocer la nueva sintaxis, por ejemplo:

```
$ python3 cli.py catalog.yaml
virtual> .help
Available commands within the prompt
    <sql>: execute SQL query or statement
    .help: print help
    .create: create the virtual database
    .destroy: destroy the virtual database
    .describe: print virtual schema
    .exit: close the current connection
virtual> .create
virtual> insert into users values(1,'a','a')
done.
virtual> insert into users values(1,'b','b')
done.
virtual> select * from users
1, 'a', 'a'
2, 'b', 'b'
virtual> update users set name='c',email='c'
done.
virtual> select * from users
1, 'c', 'c'
2, 'b', 'b'
virtual> delete from users where id=1
done.
virtual> select * from users
2, 'b', 'b'
```

4. Restricciones de integridad

Hasta el momento, se ha comentado la posibilidad de incluir restricciones de integridad en el catálogo, como por ejemplo *primary* o *unique*, pero no se utilizan, ni se comprueban. En este apartado proponemos implementar las restricciones de integridad típicas de una base de datos relacional: clave primaria, valor nulo, unicidad, clave ajena.

4.1 Clave primaria

La restricción de clave primaria se puede especificar sobre un único campo en una tabla, e implica las restricciones de valor no nulo y unicidad sobre dicho campo. A efectos prácticos, constituye un identificador único para la fila, siempre debe de tener un valor y no puede haber dos filas que tengan el mismo valor. Ya se indicó en el laboratorio anterior que TODAS las tablas deben tener una clave primaria.

Lo especificamos en el catálogo del siguiente modo.

```
users:
  fields:
  - name: id
    type: int
    primary: true
  ...
```

En un almacén clave-valor será necesario comprobar en cada inserción/actualización que no se viola esta restricción.

En un almacén relacional, es posible delegar esta responsabilidad al almacén, utilizando para ello la restricción PRIMARY KEY en el momento de creación de la tabla.

En un almacén de documentos es posible definir índices únicos sobre un campo. Por ejemplo, en MongoDB se puede usar: db.collection.createIndex()

4.2 Valor no nulo

Hasta el momento hemos asumido que todos los campos en una fila tienen valor. Podrían también ser nulos NULL. Si se soporta esta condición, tendrá implicaciones tanto en las consultas como en las sentencias DML, y será necesario modificar el código desarrollado.

En las consultas, si un campo es nulo entonces no se pueden evaluar condiciones sobre él, y por tanto la fila asociada no aparecería en el resultado. En las inserciones, si un campo es nulo, habrá que hacer traducciones a según qué fuente de datos destino.

En este contexto, será posible definir restricciones de valor no nullo del siguiente modo:

```
users:
  fields:
  ...
  - name: name
    type: str
    notnull: true
  ...
```

En un almacén clave-valor será necesario comprobar en cada inserción/actualización que no se viola esta restricción.

En un almacén relacional, es posible delegar esta responsabilidad al almacén, utilizando para ello la restricción NOT NULL en el momento de creación de la tabla.

En un almacén de documentos será necesario comprobar en cada inserción/actualización que no se viola esta restricción.

4.3 Unicidad

Esta restricción fuerza a que los valores asignados a una determinada columna sean únicos. Por tanto, dos filas no pueden tener el mismo valor asignado.

Lo especificamos en el catálogo del siguiente modo.

```
users:
  fields:
  ...
  - name: email
    type: str
    unique: true
  ...
```

En un almacén clave-valor será necesario comprobar en cada inserción/actualización que no se viola esta restricción.

En un almacén relacional, es posible delegar esta responsabilidad al almacén, utilizando para ello la restricción UNIQUE en el momento de creación de la tabla.

En un almacén de documentos es posible definir índices únicos sobre un campo. Por ejemplo, en MongoDB se puede usar: db.collection.createIndex()

4.4 Clave ajena

Es la restricción más compleja. En este caso se indica que una columna de una tabla mantiene referencias a la columna de otra tabla. Para mantener la integridad, el almacén debe garantizar que estas referencias no se rompen. Existen diversas estrategias para ello, restringir las operaciones de modificación, propagar los cambios, reiniciar las referencias a nulo.

Para simplificar, soportaremos claves ajenas sencillas, que únicamente referenciarán la clave primaria de otra tabla. Además, restringiremos las operaciones que violen esta restricción. Es decir, si se efectúa una operación de modificación sobre una fila y el resultado es una referencia rota, entonces se rechazará dicha operación de modificación.

Lo especificamos en el catálogo del siguiente modo.

```
users:
  fields:
  - name: id
    type: int
    primary: true
  ...
contacts:
  fields:
  ...
  - name: user
    type: int
    foreign: users
  ...
```

Implementar esta restricción será un tanto complicado, ya que afecta a dos tablas, y estas dos tablas podrían mapearse a distintas fuentes de datos. Por este motivo, será necesario implementarla a nivel del almacén virtual en el fichero *virtualdb.py*. Para ello, se deberá hacer uso de al menos dos drivers, el driver de la fuente de datos a la que pertenece la fila que referencia, y el driver de la fuente de datos a la que pertenece la fila referenciada.