UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CAMPUS D'ALCOI

DSIC

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Sistemas de almacenamiento y procesado distribuido

Laboratorio 1

Javier Esparza Peidro – jesparza@dsic.upv.es

Contenido

1. Introducción

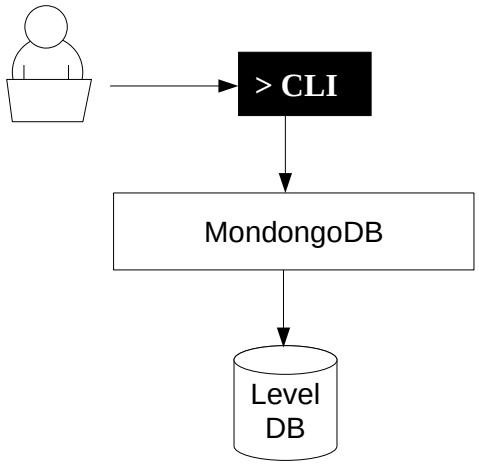
En este laboratorio se inicia el 1er proyecto de la asignatura. El objetivo global del proyecto consiste en diseñar un almacén de datos NoSQL replicado, completamente funcional. El proyecto será desarrollado a lo largo de varios laboratorios.

En este laboratorio se pondrán en práctica algunos de los conceptos teóricos estudiados en clase. En concreto, se persigue implementar un almacén de documentos similar a MongoDB, que denominaremos MondongoDB. Para ello, se pretende recubrir el almacén de tipo clave-valor LevelDB con una capa de software adicional (middleware). Esta capa adicional proporcionará una API orientada a documentos. Sin embargo, al final toda la información se almacenará en el almacén NoSQL de tipo clave-valor LevelDB.

En la sección 2 se plantea la arquitectura del sistema. En el resto de secciones se proporcionan las directrices necesarias para implementar cada uno de los componentes incluidos en el sistema.

2. Arquitectura del sistema

El sistema se compone de los componentes incluidos en la siguiente figura, y que se describen a continuación.



- LevelDB: almacén clave-valor que registra toda la información del sistema de manera persistente, en base a pares clave-valor.
- MondongoDB: almacén orientado a documentos. Publica una API de alto nivel en la que el usuario podrá gestionar colecciones y documentos. Internamente, almacena toda la información en LevelDB. Debe por tanto efectuar una traducción de las operaciones de alto nivel publicadas en su API a las operaciones de bajo nivel disponibles en LevelDB.
- CLI: herramienta Command Line Interpreter (CLI) que permite al usuario interactuar con MondongoDB, enviando consultas y recibiendo los resultados en línea de comandos.

Comenzaremos con una estructura de proyecto muy simple:

```
lab1/
  |- mondongo.py
  |- cli.py
```

3. LevelDB

Como ya hemos visto en clase, LevelDB es un almacén NoSQL de tipo clave-valor muy eficiente. Se trata de un almacén serverless, es decir, que no requiere de una instalación típica cliente-servidor. Para empezar a trabajar con él basta con instalar su driver.

```
> python3 -m venv venv
> source venv/bin/activate
(venv) > pip install plyvel
```

Este almacén posee las siguiente peculiaridades:

- Tanto las claves como los valores deben de ser Strings.
- Las claves se almacenan en orden.
- Soporta consultas de tipo rango, en base al orden de sus claves.

La API del driver está disponible en el siguiente enlace:

https://github.com/wbolster/plyvel

Para comprenderla, a continuación se muestra un fragmento de código que permite trabajar con este almacén utilizando Python:

```
import plyvel
db = plyvel.DB('./db', create_if_missing=True)
db.put(b'test', b'Hello wolrd!')
db.get(b'test')
db.delete(b'test')
for key,value in db: print(key)
for key,value in db.iterator(start=b'start', stop=b'end'): print(key)
db.close()
```

4. MondongoDB

El almacén de documentos se implementará en el módulo *mondongo.py*, apoyándose en el almacén clave-valor LevelDB presentado en la sección anterior. A continuación se presenta primero su API y después se propone una estrategia que permite mapear documentos a un esquema (clave,valor).

4.1 La API

El módulo *mondongo.py* publica una única operación *connect()* que permite crear una nueva base de datos, o bien conectarse a una base de datos ya creada. Si la operación tiene éxito, devuelve un objeto de la clase *DocStore*, que representa la conexión contra la base de datos. Este objeto publica las operaciones típicas de un almacén de documentos. A continuación se listan las operaciones que como mínimo debería de incluir el módulo *mondongo.py*, siguiendo las convenciones clásicas de la documentación Python:

`mondongo.connect(path:str, opts:dict={})`

Conecta contra una base de datos.

El argumento *path* apunta a un directorio que contiene toda la información de la base de datos. Si el directorio existe, la base de datos se carga. Si el directorio no existe, la base de datos se crea y se carga.

El argumento *opts* contiene opciones adicionales.

Si la base de datos se carga con éxito, se devuelve un objeto de la clase mondongo*.DocStore*

`clase mondongo.DocStore`

Representa una conexión abierta contra la base de datos.

`mondongo.DocStore.create(col:str, schema:dict)`

Crea una nueva colección de documentos con nombre *col*.

El argumento *schema* contiene un diccionario que especifica las propiedades que poseen todos los documentos de la colección. En cada entrada del diccionario la clave determina el nombre de la propiedad y el valor su tipo de datos. Al menos se deben soportar los tipos de datos simples “str”, “int”, “bool”. Si la propiedad debe ser indexada entonces el nombre de la propiedad posee como prefijo un “*”. Si la propiedad está indexada, entonces se podrán definir condiciones de búsqueda sobre ésta en las consultas.

`mondongo.DocStore.destroy(col:str)`

Elimina la colección especificada.

`mondongo.DocStore.describe()`

Describe el contenido del almacén, con información sobre todas las colecciones.

Devuelve una lista de diccionarios. Cada diccionario contiene información sobre una colección almacenada. Este diccionario poseerá la entrada “name”, con el nombre de la colección, “schema”, con el esquema de la colección y “count”, con el número de documentos almacenados en la colección.

`mondongo.DocStore.insert(col:str, doc:dict)`

Inserta el documento *doc* (es un diccionario) en la colección *col.* Si la colección no existe o el documento posee las propiedades equivocadas, se lanza una excepción.

Cuando se inserta un nuevo documento, automáticamente se añade la propiedad “._id” que contiene el identificador único del documento.

Si tiene éxito, esta operación devuelve el documento insertado.

`mondongo.DocStore.search(col:str, query:dict)`

Efectúa una búsqueda de los documentos de la colección *col* que verifican la consulta *query.*

La consulta *query* es un diccionario que contiene condiciones de igualdad y/o operadores de consulta simples, como en MongoDB. Los operadores de consulta que reconocerá el motor son al menos $eq, $in, $gt, $gte, $lt y $lte. Se recomienda acudir a la documentación de MongoDB para comprender su significado.

Para poder incluir una condición sobre la propiedad de una colección en la consulta, debe existir un índice sobre ésta (definido en la operación `.create()`). La propiedad “._id” está siempre indexada de manera automática.

Si la operación tiene éxito, se devuelve una lista con todos los documentos que verifican la consulta.

`mondongo.DocStore.update(col:str, query:dict, data:dict)`

Se modifican los documentos de la colección *col* que verifican la consulta *query*.

Sobre *query* idénticas consideraciones a las explicadas en la operación `.search()`.

Únicamente se actualizan las propiedades especificadas en *data.* El valor de la propiedad es reemplazado por el nuevo valor especificado en *data*.

Si la operación tiene éxito, se devuelve una lista con los documentos actualizados.

`mondongo.DocStore.delete(col:str, query:dict)`

Se eliminan los documentos de la colección *col* que verifican la consulta *query*.

Sobre *query* idénticas consideraciones a las explicadas en la operación `.search()`.

`mondongo.DocStore.close()`

Cierra la conexión contra la base de datos.

A continuación se muestra a modo de ejemplo un fragmento de código que permitiría trabajar con este almacén de datos.

```
import mondongo
db = mondongo.connect("test")
db.create("users", {"*email":"str", "name":"str", "*age":"int"})
db.insert("users", {"email":"a","name":"a","age":18})
db.insert("users", {"email":"b","name":"b","age":18})
docs = db.search("users", {"age":18})
docs = db.search("users", {"age": {"$gte": 18}})
for doc in docs: print(doc)
db.update("users", {"email":"a"}, {"age":19})
db.delete("users", {"email":"b"})
db.close()
```

4.2 Mapeo de documentos a clave-valor

El almacén de documentos se construye a partir de un almacén clave-valor. En realidad, un almacén clave-valor se puede considerar como una generalización de un almacén de documentos, donde la clave contiene el identificador del documento y el valor contiene todo el documento. En esta sección se plantea un mapeo un poco más sofisticado, que utiliza un índice primario sobre la propiedad "_id", y múltiples índices secundarios, y que permite efectuar consultas más complejas sobre cualquiera de ellos.

Para empezar, cuando se crea una nueva colección con la operación `.create(<col>,<esquema>)` se registran en el almacén clave-valor las entradas:

```
/<col>  → <esquema>
collections  → [<col>]
```

En la primera entrada se almacena el esquema de la colección. En la segunda entrada se almacenan los nombres de todas las colecciones existentes.

Cuando se inserta un nuevo documento en una colección, se añade automáticamente un identificador único bajo la propiedad "_id". Este identificador único podría obtenerse fácilmente utilizando *time.time_ns()* o bien *uuid.uuid4()*. Entonces, se registra en el almacén clave-valor la entrada:

/<col>/_id/<id> → <doc>

Este mapero representa un índice primario sobre la propiedad "_id". De este modo, resulta muy sencillo recuperar los documentos de una colección por su identificador único. Por ejemplo:

```
db.search("users", {"_id": 16334246435878665O4})
```

Se traduciría en una búsqueda de la clave `"/users/_id/1633424643587866504"`, que devolvería el documento en cuestión de manera muy eficiente.

Además, el almacén permite búsquedas sobre cualquier otra propiedad que haya sido previamente indexada, son los índices secundarios. Para ello, al crear una colección, el usuario marca las propiedades que desea indexar con el prefijo “*”. Por ejemplo:

```
db.create("users", {"*email":"str", "name":"str", "*age":"int"})
```

Se indexarían las propiedades “email” y “age”. Cuando el usuario inserta un documento con propiedades indexadas, es necesario insertar nuevas entradas en el almacén clave-valor. Por ejemplo:

```
db.insert("users", {"email":"a","name":"a","age":18})
```
Aquí, el almacén registraría las siguientes entradas:

```
/users/_id/1 → {"_id":1,"email":"a","name":"a","age":18}
/users/email/a → [1]
/users/age/18 → [1]
```

La primera entrada se corresponde con el índice primario. Las dos últimas entradas implementan dos índices secundarios sobre las propiedades “email” y “age” respectivamente. En el ejemplo, la entrada con clave “/users/age/18” contiene una lista con los identificadores de todos los usuarios que poseen en su propiedad “age” el valor 18. Si a continuación se inserta un nuevo usuario:

```
db.insert("users", {"email":"b","name":"b","age":18})
```

Se añadirán las siguientes entradas:

```
/users/_id/2 → {"_id":2,"email":"b","name":"b","age":18}
/users/email/b → [2]
/users/age/18 → [1,2]
```

Como se puede observar, la entrada “/users/age/18” se sobreescribe, al existir ahora dos usuarios con la misma edad.

Gracias a estos índices secundarios, es posible efectuar la siguiente consulta de manera eficiente:

```
docs = db.search("users", {"age":18})
```

4.3 Consultas sobre rangos

Una característica interesante de LevelDB es que las claves se almacenan de manera ordenada. Esto permite que se puedan efectuar consultas sobre rangos. Esto significa que es posible especificar una clave de comienzo (incluída) y una clave de fin (excluída), y LevelDB devolverá todas las entradas comprendidas entre ambas claves.

Podemos aprovechar esta característica para recuperar colecciones de documentos en las consultas especificadas en nuestro almacén de documentos. Por ejemplo, la siguiente consulta debería de devolver todos los documentos que pertenecen a la colección “users”.

```
docs = db.search("users", {})
```

Para ello, habría que efectuar una consulta sobre rangos a LevelDB, especificando como clave de comienzo la clave más pequeña existente en la colección, y como clave de fin la clave más grande existente en la colección. ¿Pero cuál es la clave más pequeña y la más grande? Sabemos que deben de empezar por “/users/”, pero a priori desconocemos cuál es la primera y la última. Para

ello, podemos usar el carácter más pequeño que hay “\u0000” y el carácter más grande que hay “\ uffff”. Así, cuando se desee hacer una búsqueda de todos los usuarios:

```
docs = db.search("users", {})
```

Habrá que recuperar todas las entradas comprendidas en el rango abierto:
]"/users/_id/\u0000", "/users/_id/\uffff"[

Un mecanismo similar se usaría para recuperar por ejemplo todos los usuarios cuya edad es superior a 10:

```
docs = db.search("users", {"age": {$gt: 10}})
```

Aquí habría que buscar las entradas comprendidas en el siguiente rango:

["/users/age/10", "/users/age/\uffff"[

4.4 Actualizaciones

Como hemos visto, los índices aceleran las consultas sobre los datos. Sin embargo, mantener estos índices va a resultar costoso, ya que cada vez que se inserte un nuevo documento, se elimine o se actualice una propiedad indexada, será necesario modificar varias claves. Por ejemplo:

```
db.update("users", {"email":"a"}, {"age":19})
```

Esta operación implica modificar las siguientes claves:

```
/users/_id/1 → {"_id":1,"email":"a","name":"a","age":19}
/users/age/18 → [2]
/users/age/19 → [1]
```

Si se elimina el documento:

```
db.delete("users", {"age":19})
```

Será necesario eliminar la clave:

```
/users/_id/1
```

Y modificar las claves:

```
/users/email/a → []
/users/age/19 → []
```

Si no existen referencias lo lógico sería eliminar también estas entradas.

5. CLI

Este componente permite utilizar el almacén de documentos desde línea de comandos. Se implementará en el módulo *cli.py.* A continuación se muestra una traza de su posible funcionamiento:

```
> python3 cli.py help
Usage: python3 cli.py <command>
Available commands:
  create <path>: create a new store
  delete <path>: delete a store
  open <path>: open a connection against the specified path and open a prompt
Available commands within the prompt
  addcol <name> {key=val,}+: add a new collection with the specified schema
  rmcol <name>: remove de specified collection
  desc: describe the store

  adddoc <col> {key=val,}+: add a new document with the specified attributes
  search <col> <query>: obtain all the documents which meet the specified query
  update <col> <query> {key=val,}+: update the documents meeting the query
  remove <col> <query>: remove the documents which meet the specified query
  exit: close the current connection

> python3 cli.py create db
> python3 cli.py open db
db > addcol users *email=str,name=str,*age=int
Col added.
db > add users email=a name=a age=18
Doc added with _id 2dc694b9-ead6-4805-9cfb-14fb00c47f1a.
db > add users email=b name=b age=18
Doc added with _id a2d3ae6c-44af-4fb5-8197-3a3f590ed8b1.
db > search users _id=2dc694b9-ead6-4805-9cfb-14fb00c47f1a
Results:
- id: 2dc694b9-ead6-4805-9cfb-14fb00c47f1a
  email: a
  name: a
  age: 18
db > search users age>10 age<20
Results:
- id: 2dc694b9-ead6-4805-9cfb-14fb00c47f1a
  email: a
  name: a
  age: 18
- id: a2d3ae6c-44af-4fb5-8197-3a3f590ed8b1
  email: b
  name: b
  age: 18
db > update users _id=2dc694b9-ead6-4805-9cfb-14fb00c47f1a email=c,age=19
Doc updated.
db > exit
Bye
```