Projekt z Programowania Obiektowego

Jakub Gałecki - 324384

Krótki Opis

Celem tego projektu jest zaimplementowanie prostej ***bazy danych key-value z użyciem B-Drzew***. W dużym skrócie *B-Drzewa* to drzewiasta struktura danych , która przechowuje dane w uporządkowany sposób i pozwala na szybkie wykonywanie operacji na drzewie. Dla wyszukiwania, dodawania oraz usuwania wartości w drzewie ma złożoność czasową: **O(log n)**. Jest ona często używana w bazach danych, właśnie ze względu na jej niską złożoność czasową. Projekt jest stworzony głównie w celach naukowych. aby zbadać, jak mniej więcej takie bazy danych działają. Wstępnie będą dwa sposoby komunikowania się z bazą:

1. Przez REPL'a
2. Poprzez specjalną metode, która umożliwia wykonywanie zapytań do bazy w DenoJS.

W przyszłości chciałbym jeszcze wprowadzić serwer, który byłby odpalony lokalnie i umożliwiał wysyłanie zapytań przez HTTP.

Technologie wykorzystane w projekcie: TypeScript oraz DenoJS.

Instalacja oraz Wymagania

Operacje poniżej wykonywane są pod Debianem.

Aby korzystać z bazy danych musimy zainstalować `DenoJS` (*link do instalacji: https://deno.land/#installation*).

Po pobraniu `DenoJS` należy w terminalu udać się do folderu `nextdb` i uruchomić `deno run --unstable --allow-read --allow-write mod.ts -r true` . To powinno pobrać wszystkie potrzebne biblioteki zewnętrzne. Możemy chwilowo wyłączyć program poprzez `CTRL + C`. Możemy odpalić testy, aby sprawdzić czy wszystko poprawnie się zainstalowało `deno test --unstable --allow-write --allow-read`, wszystkie testy powinny przejść, jeśli dostaniemy informację, że brakuje jakiegoś modułu to możemy go zainstalować za pomocą `deno info <link_do_modułu>` .

Korzystanie z bazy danych

Na początek będąc w głównym folderze, czyli `nextdb` możemy odpalić testy: `deno test --unstable --allow-write --allow-read`. Wszystkie powinny przejść, jeśli się tak nie stanie to może być kwestia błędnej instalacji lub jakiegoś błędu w kodzie. Taka sytuacja nie powinna wystąpić.

Tak jak napisałem wyżej, istnieją dwa sposoby komunikowania się z bazą danych:

1. REPL

   Aby uruchomić REPL'a należy w głównym folderze `nextdb` wpisać w terminalu następującą komendę:

   `deno run --unstable --allow-read --allow-write mod.ts -r true`

   Flagi `--allow-read`, `--unstable` oraz `--allow-write` są wymagane przez Deno, aby nadać programowi uprawnienia do czytania oraz tworzenia plików.

Po uruchomieniu powyższej komendy powinniśmy już móc porozumiewać się z bazą danych. Znak `$` sygnalizuje początek linii i to po nim powinniśmy wpisywać zapytania do bazy.

Możliwe zapytania: `createDb`, `deleteDb`, `insert`, `delete`, `search`, `update`. Zapytanie do bazy powinno wyglądać w następujący sposób:

`<nazwa_bazy>.<komenda>`. Użycie powyższych komend w celu stworzenia zapytania do bazy:

- `<nazwa_bazy>.createDb`
- `<nazwa_bazy>.insert(key, value)`
- `<nazwa_bazy>.search(key)`
- `<nazwa_bazy>.update(key, newValue)`
- `<nazwa_bazy>.delete(key)`
- `<nazwa_bazy>.deleteDb`

Jako odpowiedź otrzymamy obiekt `Response`, którego struktura wygląda następująco:

```
Response {
    status: "OK" // lub ERRROR
    found: "" // będzie uzupełnione tylko w przypadku metody search, kiedy
wartość zostanie znaleziona wpp. jest puste.
}
```

2. Przez specjalną metodę.

Aby korzystać z tej klasy musimy stworzyć nowy plik TypeScript oraz zaimportować naszą klasę na samej górze pliku:

`import {query} from "./<scieżka_do_folderu>/mod.ts"`

Teraz możemy wysyłać zapytania do bazy z użyciem metody `query`. Funkcja `query` przyjmuje jeden argument, którym jest obiekt mający następująca strukturę:

```
{
    dbname: "<nazwa_bazy>",
    command: "<komenda>", // createDb,deleteDb, insert, delete, search ,
update
    key: "<klucz>",
    value: "<wartość>"
}
```

Jako odpowiedź dostaniemy obiekt `Promise<Response>`, gdzie `Request` ma taką samą strukturę jak wcześniej.

Przykładowa komunikacja z bazą:

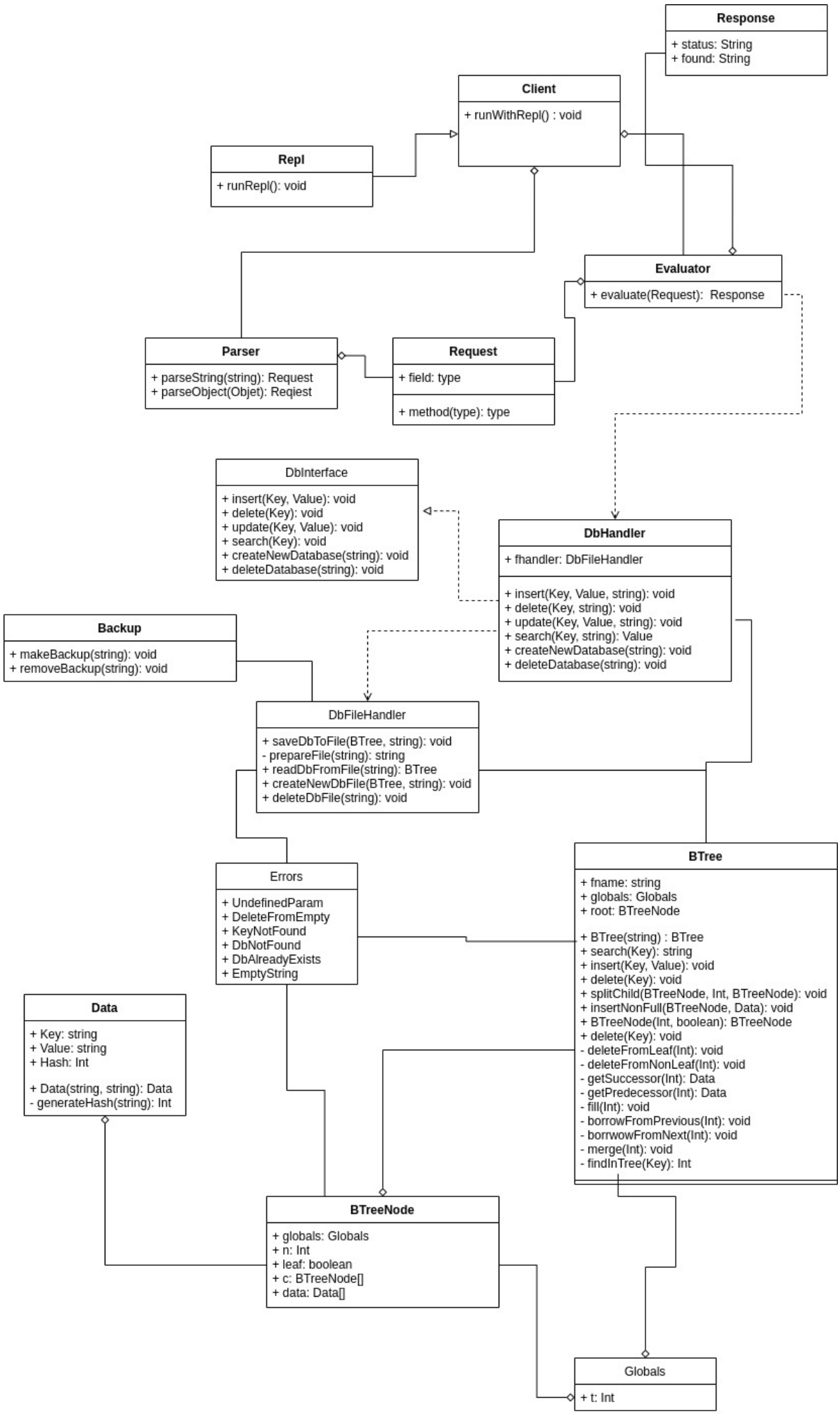
```
import {query} from "./mod.ts

const qCreateDb: Object = {
    dname: "test",
    command: "createDb",
    key: "",
    value: ""
}
const qInsert: Object = {
    dname: "test",
    command: "insert",
```

```
    key: "kkk",
    value: "value"
}

const req1 = await query(qCreateDb);
if (req1.status === "ERROR"){
    throw new Error();
}
const req2 = await query(qInsertDb);
if (req2.status === "ERROR"){
    throw new Error();
}
const reqSearch = await query({
    dname: "test",
    command: "search",
    key: "kkk",
    value: ""
});

if(reqSearch.status !=== "ERROR"){
    console.log(req.found);
}
```

Ważne: po każdej operacji dodania nowego klucza do bazy tworzona jest kopia zapasowa, stanu bazy przed wykonaniem tej operacji.

Opis i diagram klas

1. Backup - Klasa odpowiedzialna za tworzenie kopii bazy danych.
2. BTree - Klasa implementująca strukturę B-Drzewa wraz ze wszystkimi metodami zarządzającymi B-Drzewem
3. BTreeNode - Klasa implementująca węzeł B-Drzewa.
4. Client - Klasa służąca do wysyłania zapytań do serwera / repla.
5. Data - Klasa, w której trzymany jest klucz i wartość
6. DbFileHandler - Klasa obsługująca plik bazy danych
7. DbHandler - Klasa obsługująca samą bazę danych - operuje na `DBFileHandler` oraz `BTree`.
8. DbInterface - Interfejs implementujący metody dostępne w bazie.
9. Error - Zbiór klas reprezentujących zwracane błedy.
10. Evaluator - Klasa odpowiedzialna za przetwarzanie sparsowanych zapytań, czyli przetwarzanie `Request`.
11. Globals - Klasa w której trzymane są wartości globalne
12. NextDB - Klasa która udostępnia metodę `query`, dzięki której można korzystać z bazy danych z poziomu samego TypeScripta
13. Parser - Klasa prasująca nadchodzące zapytania, które są w postaci stringa lub obiektu.
14. REPL - Klasa tworząca repl'a.
15. Request - Klasa reprezentująca zapytanie do serwera.
16. Respons - Klasa reprezentująca odpowiedź serwera.


Response
+ status: String
+ found: String
Client
+ runWithRepl() : void
Repl
+ runRepl(): void
Evaluator
+ evaluate(Request): Response
Parser
+ parseString(string): Request
+ parseObject(Objet): Reqiest
Request
+ field: type
+ method(type): type
DbInterface
+ insert(Key, Value): void
+ delete(Key): void
+ update(Key, Value): void
+ search(Key): void
+ createNewDatabase(string): void
+ deleteDatabase(string): void
DbHandler
+ fhandler: DbFileHandler
+ insert(Key, Value, string): void
+ delete(Key, string): void
+ update(Key, Value, string): void
+ search(Key, string): Value
+ createNewDatabase(string): void
+ deleteDatabase(string): void
Backup
+ makeBackup(string): void
+ removeBackup(string): void
DbFileHandler
+ saveDbToFile(BTree, string): void
- prepareFile(string): string
+ readDbFromFile(string): BTree
+ createNewDbFile(BTree, string): void
+ deleteDbFile(string): void
Errors
+ UndefinedParam
+ DeleteFromEmpty
+ KeyNotFound
+ DbNotFound
+ DbAlreadyExists
+ EmptyString
BTree
+ fname: string
+ globals: Globals
+ root: BTreeNode
+ BTree(string) : BTree
+ search(Key): string
+ insert(Key, Value): void
+ delete(Key): void
+ splitChild(BTreeNode, Int, BTreeNode): void
+ insertNonFull(BTreeNode, Data): void
+ BTreeNode(Int, boolean): BTreeNode
+ delete(Key): void
- deleteFromLeaf(Int): void
- deleteFromNonLeaf(Int): void
- getSuccessor(Int): Data
- getPredecessor(Int): Data
- fill(Int): void
- borrowFromPrevious(Int): void
- borrwowFromNext(Int): void
- merge(Int): void
- findInTree(Key): Int
Data
+ Key: string
+ Value: string
+ Hash: Int
+ Data(string, string): Data
- generateHash(string): Int
BTreeNode
+ globals: Globals
+ n: Int
+ leaf: boolean
+ c: BTreeNode[]
+ data: Data[]
Globals
+ t: Int