NoSQL база даних

Теорія

Реляційні (SQL) та нереляційні (NoSQL) бази даних

Більш широко ця теорія описана тут (relational db та non-relational db)

Простими словами: реляційна база представлена у вигляді таблиць з рядками та стовпичками в яких є записи (records). Нереляційна база представлена у вигляді неструктурованого набору даних, наприклад JSON об'єктів або пар ключ-значення.

JSON (JavaScript Object Notation) (wiki)

JSON — текстовий формат для зберігання та передачі об'єктів. Кожен об'єкт представлений його атрибутами: парами ключ-значення. Серед типів які підтримує JSON:

- Number. (`1`, `2.35`)
- String. (`"foo"`, `"bar"`)
- Boolean. (`true`, `false`)
- Array. (`[true, "foo", 3.1415]`)
- Object. (`{"first_name": "John", "last_name": "Doe"}`)
- Null. (`null`)

Приклад JSON об'єкта:

```
{
  "first_name": "John",
  "last_name": "Smith",
  "is_alive": true,
```

```
  "age": 27,
  "address": {
    "street_address": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postal_code": "10021-3100"
  },
  "phone_numbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [
    "Catherine",
    "Thomas",
    "Trevor"
  ],
  "spouse": null
}
```

Задача

Реалізувати NoSQL базу даних яка буде зберігати JSON об’єкти.

База має підтримувати ізольовані середовища (таблиці) в яких дані зберігаються незалежно.

База даних має зберігати всі дані фізично на диску.

База має підтримувати наступні операції:

- `Insert` — додає об’єкт в базу даних

- `Update` — шукає об’єкт за наданими фільтрами та якщо об’єкт знайдено оновлює його згідно з параметрами
- `Select` — шукає та повертає об’єкти за наданими фільтрами.
- `Delete` — видаляє об’єкти за наданими фільтрами.
- `Create index` — індексує об’єкти по заданому набору полів (див. Оптимізації).
- `Flush` — примусово перезаписує стан даних на диску після видалення (див. Оптимізації).

База має вводити обмеження на максимальний розмір одного файлу з даними на диску. За необхідності використання більшої кількості місця база має створювати і керувати додатковими файлами.

(Optional) База має підтримувати JSON об’єкти будь якої вкладеності.

(Optional) **Мова запитів + Компілятор**. Мова запитів підтримує високорівневі запити

(напр. `insert users object {"name": "Jane"}`, `select users where "name" = "Jane"`).

Компілятор перетворює високорівневий запит на набір низькорівневих команд до Ядра бази даних.

(Optional) **Веб-Сервер Бази**. Приймає http запит з запитом до бази даних. Компілює його, виконує низькорівневі команди та повертає результат у формі http відповіді.

(Optional) Синхронізація даних у випадках коли більше ніж один користувач одночасно надсилає запити до бази

Оптимізації

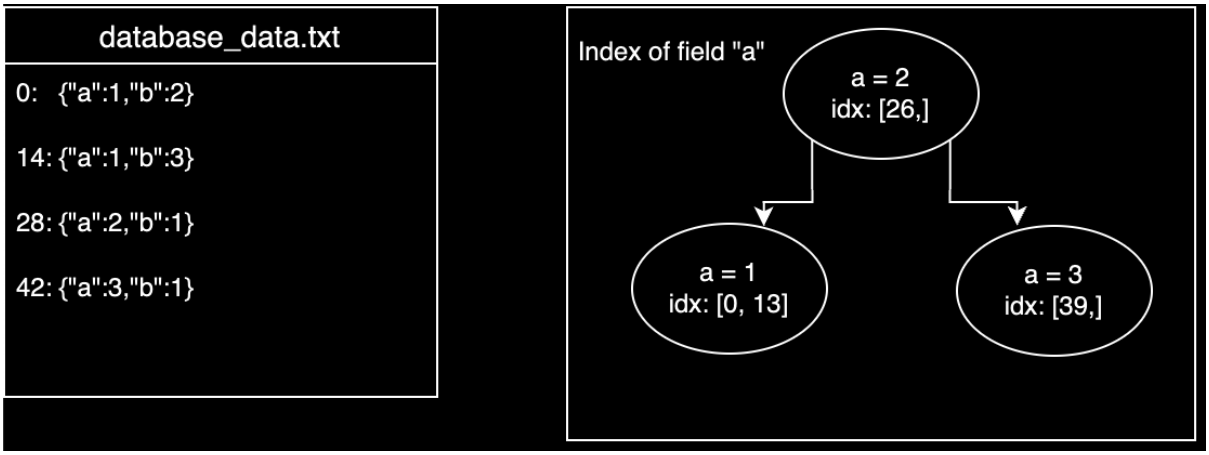
Ядро має використовувати наступні оптимізації:

Індексація пошуку (Indexing).

Уявимо, що у базу додано і записано у файл N об’єктів. Використовуючи наївний алгоритм для операції `select` базі необхідно буде перебрати O(N) об’єктів. Очевидна оптимізація — створення бінарного (або n-арного) дерева

пошуку. Імплементація бінарного (n-арного) дерева може бути власна або вже існуюча з посиланням на оригінал.

Нехай в базі існує M JSON об’єктів, які мають поле з назвою “a”. Створимо бінарне дерево, в якому ключем буде значення поля “a” в базі, а значенням — номер байта у файлі з якого починається об’єкт.



Приклад 1: Стан бази та індексу по одному з полів після додавання 4х об’єктів

Зауваження: нумерація байтів праворуч подана для наочності, файл фізично виглядає так:

```
{"a":1,"b":2}\n{"a":1,"b":3}\n{"a":2,"b":1}\n{"a":3,"b":1}\n
```

Тоді маючи таку структуру і запит вигляду `select where a=2` компілятор може **НАПРИКЛАД** згенерувати наступні низькорівневі команди для ядра:

```
get_index_by_key("a")
search_in_index_by_value(2)
get_from_file_by_address()
```

💡 Note: у запропонованому форматі збереження даних для того щоб вичитати об'єкт по його номеру першого байта достатньо почати читати з цього байта до наступного знака `\n` . Це можна зробити наступним набором команд:

```
file = open(…)
```

```
file.seek(start_byte)
```

```
file.readline()
```

Більш того, знаючи точний номер байта у файлі ми можемо за O(1) почати читати саме з цього місця за допомогою `file.seek(byte_no)` ця команда виставить вказівник у відповідний байт, подальші команди `file.read()` почнуть читання саме з цього місця.

`Create Index` має дозволяти створювати користувацький індекс по набору полів. Тобто у вершині дерева буде стояти не одне значення поля а декілька.

💡 TODO: Як зміниться розмір індексу (кількість вершин у дереві) в порівнянні зі стандартним індексом? Як буде виглядати алгоритм пошуку по двом полям з двома різними індексами, як він буде виглядати у варіанті з одним індексом по двом полям?
Як зміниться швидкість пошуку?

Автоіндексація

Автоматичне створення індексів. Як можна помітити лише даних від користувача достатньо щоб знати на які поля нам треба створювати індекси. Тому ядро має автоматично створювати індекси на всі поля які зустічаються в базі.

Повертаючись до попереднього прикладу має бути створено ДВА індекси: індекс по значенням поля “a” та окремий по значенням поля “b”

Відкладене видалення (Deferred deletion)

Уявімо що в нас записано N об’єктів у файл. Як нам видалити і-тий об’єкт? Насправді сучасні операційні системи таких операцій не підтримують з ряду причин. Список стандартних операцій над файлами можна знайти тут. Найближче що існує — функція `pwrite` яка перезаписує N байтів у файл починаючи з заданого байту, використовуючи її, наприклад, можна “затерти” об’єкт замість того щоб переписувати весь файл.

Зважаючи на це наївна імплементація виглядає наступним чином: на кожне видалення файл повністю перезаписується. Очевидно, що час виконання даної операції пропорційний розміру збережених даних та буде зростати лінійно.

Існує ряд оптимізацій як можна уникнути повного перезаписання на кожне видалення. В наведеній задачі пропонується наступний підхід:

- На видалення об’єкта збережемо номер байта з якого починається об’єкт у файлі. (помічаємо як видалений)
- Створимо окрему структуру даних, в якій будемо зберігати ці номера байтів, структура має підтримувати швидкий пошук.
- Роблячи пошук об’єкту в базі даних перевіряємо, чи не було знайдений об’єкт відмічено як видалений.
- Після кожних N операцій видалення виконуєм операцію `flush` :
    - Перезаписуємо файл з застосуванням усіх зроблених видалень
    - Очищаємо структуру з поміченими як видалені об’єктами

Таким чином замість того, щоб перезаписувати файл на кожен `delete` ми будемо його переписувати раз на N операцій `delete` заплативши за це часом пошуку об'єкту (тепер замість того щоб просто знайти об'єкт в індексі ми маємо ще і перевіряти кожен раз чи не було його видалено). В реальному світі в таких випадках будується статистичний експеримент на основі якого знаходиться значення N, таке після якого пошук стає недоцільно довгим в порівнянні з платою за повний перезапис.

(Optional) побудувати статистичний експеримент в якому заміряти тривалість перезапису файлу в залежності від його розміру і порівняти його зі збільшеням тривалості пошуку об'єкта. Зробити припущення що операцій `delete` в 4 рази менше ніж операцій `select` `update` (update теж має робити спочатку пошук, тому включаємо його сюди теж)

> 💡 Дозволяється обрати іншу оптимізацію для імплементації видалення

> 💡 TODO: зважаючи на підхід який ми використали для видалення, як імплементувати `update`?

> 💡 TODO: Інші оптимізації вітаються. Необхідно провести тестування і зробити висновки з приводу можливих інших оптимізацій.