Проектування програмного забезпечення

Лабораторна робота №5

Тема: Бази даних типу ключ-значення та атомарні операції
Мета: Набуття навичок в імплементації атомарних операцій за допомогою файлової системи та розуміння принципів роботи нереляційних баз даних

Завдання на дану роботу складається з 4-х частин:

1. Імплементація підтримки та злиття сегментів на базі коду, який розглядався під часлекційних занять.

2. Реалізація аспекту конкурентності у вашій базі даних.

3. Реалізація одного з додаткових аспектів відповідно до вашого варіанта.

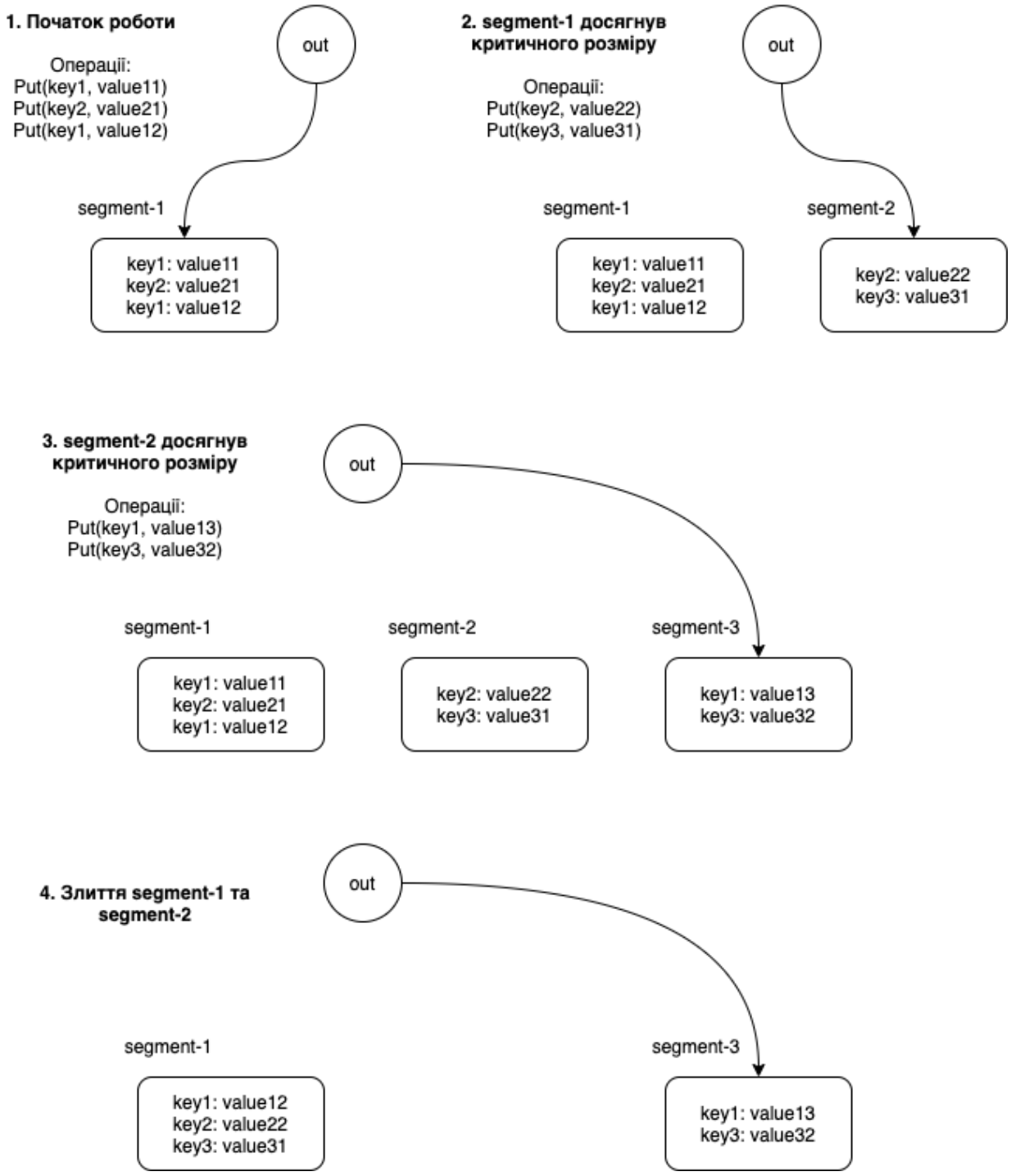
4. Інтеграція вашої бази даних із з іншими компонентами з попередньої лабораторної роботи.

Завдання 1

У репозиторії github.com/roman-mazur/design-db-practice ви знайдете реалізацію бази даних типу ключ-значення дуже близьку до того, що розглядається на лекції присвяченій реалізації БД. У рамках заняття ми залишаємо реалізацію в стані, коли файл, з яким ми виконуємо append операцію, нескінченно росте, навіть якщо ми змінюємо значення одного й того самого ключа.

Рішення даної проблеми полягає в реалізації файлів-сегментів: коли файл, з яким ми виконуємо append операцію досягає певного розміру, ми перестаємо писати в нього та створюємо новий файл для нових записів. Файли, які ми таким чином створюємо, не змінюються в результаті виконання Put операцій та можуть бути злиті у фоновому процесі (рутині). Даний процес проілюстровано на діаграмі нижче.

Метою першого завдання є реалізація даної ідеї та досягнення стану, коли ваша БД здатна видаляти неакутальні дані з диску.



Ваші кроки для виконання даного завдання можуть виглядати наступним чином.

- Скопіюйте datastore пакет з репозиторія github.com/roman-mazur/design-db-practice у ваш репозиторій, який ви використовували для попередньої лабораторної роботи (це важливо для останнього завдання).

- Додайте тест для перевірки роботи підтримки сегментів (аналогічно до того, як ми це робили на лекційному занятті) симулюючи сценарії створення нового сегмента та викликаючи рутину злиття сегментів.

- Імплементуйте підтримку сегментів, задовольняючи ваші тести.

Зауважте, що в основному коді та в тестах ви захочете встановити різні максимальні межі для out файла (щоб швидше виконувати тести та не займати багато дискового простору), тому потурбуйтеся про можливість зміни даної межі в тестовому коді.

Злиття сегментів має відбуватися як атомарна операція: якщо помилка/відмова виникає під час злиття, операція Get має продовжувати звертатися до попередньої версії файлів-сегментів.

Непоганим розміром для сегмента є 10M.

Завдання 2

Іншою проблемою поточної реалізації БД є відсутність можливості паралельного використання операцій Put та Get. Тому в контексті даного завдання вам необхідно реалізувати коректну підтримку паралельної роботи.

Писати в файл out вам необхідно завжди з однієї виділеної рутини, яка може, наприклад, отримувати дані для запису через канал, у який дані записуються в методі Put. Окремо вам потрібно забезпечити захист доступ до хеш-індексу ключів у пам'яті.

Операція Get відкриває файл сегмента в режимі читання та створює новий файловий дескриптор для кожного читання. Тому додавання синхронізації доступу до хеш-індексу має бути достатньо для її коректної роботи.

У цьому завданні достатньо змінити основний код та переконатися, що всі ваші тести продовжують успішно виконуватися.

Завдання 3

Залежно від вашого варіанту (/team-variant 5) реалізуйте один з наступних аспектів.

Варіант	Завдання
1	Підтримка операції видалення (Delete). Для видалення даних, вам потрібно вилучити ключ з хеш-індекса в пам'яті та записати спеціальний токен (маркер) в out файл для відповідного ключа. Під час злиття сегментів, дані для цього ключа мають видалитися з диска.
2	Збереження контрольної хеш-суми ваших даних та її перевірка при читанні. При збереженні даних на диск в Put вам потрібно додавати sha1 суму значення в запис. При виконанні операції Get перевіряйте, що sha1 прочитаних даних відповідає збереженій сумі та повертайте помилку, якщо ці суми не співпадають.
3	Захистіть операцію Get від відкриття занадто великої кількості файлів одночасно. Поточна реалізація відкриває новий файловий дескриптор при кожному виклику Get. При паралельному використанні з великою кількістю клієнтів це може призвести до відмов (через ліміти на рівні файлової системи). Ви можете організувати обмежений пул виконавців (workers), які виконують операцію читання. В методі Get комунікуйте з даними виконавцями через канали, відправляючи їм повідомлення з ключем та каналом, куди записати результат.
4	Додайте можливість роботи зі значеннями типу int64. Для цього додайте методи GetInt64 та PutInt64 до структури Db. Вам потрібно буде таож змінити реалізацію методів entry, додавши інформацію про тип збережених даних, а не тільки про їхній розмір. Якщо при читанні даних, тип значення не відповідає очікуваному типу (у сигнатурі одного з методів Get), повертайте помилку.

В усіх варіантах окрім 3-ого вам потрібно додати додаткові unit-тести для перевірки нової функціональності.

Завдання 4

У попередній роботі ви користувалися реалізацію cmd/server яка обробляла запит HTTP GET /api/v1/some-data, повертаючи статичні дані. У цьому завданні вам потрібно змінити реалізацію cmd/server таким чином, щоб вона виконувала комунікацію з вашою БД.

При запуску сервера, він має зберегти рядок з поточною датою (наприклад, “2021-04-25”) під ключем, який відповідає імені вашої команди у вашій базі даних. Обробник GET /api/v1/some-data повинен приймати параметр key в URL вашого запиту та читати дані під цим ключем. Якщо дані в БД відсутні, ваш обробник має повертати код 404 з пустим тілом.

Інтеграційний тест для вашого балансувальника тепер повинен відправляти запити
GET /api/v1/some-data?key=<ім’я команди>
та перевіряти, що він отримав не пусті дані.

Ваша БД повинна працювати в окремому процесі. Для цього виконайте наступні кроки.

- Додайте main пакет cmd/db.
- Реалізуйте в ньому функцію main, яка створює екземпляр datastore.Db та запускає HTTP сервер, який реалізовує наступний інтерфейс.
    - **GET /db/<key>** викликає Db.Get з визначеним ключем та повертає JSON відповідь з ключем та значенням:
      {“key”: “<key>”, “value”: <value>}
        - Якщо дані за ключем відсутні, повертайте статус код 404.
        - *Якщо ви маєте варіант 4*, додайте також параметр type у ваш запит, щоб визначити, який Get ви виконуєте. За замовчуванням, type=string.
          GET /db/<key>?type=int64
        - *Якщо ви маєте варіант 3*, у випадку помилки з хеш-сумами повертайте 404.
    - **POST /db/<key>** з JSON тілом {“value”: <value>} викликає Db.Put з визначеними ключем та значенням
        - *Якщо ви маєте варіант 4*, проаналізуйте тип value та викличте відповідний Put.
- Додайте новий сервіс db у ваш docker-compose,yml та зробіть HTTP інтерфейс вашої БД доступним на новому порту
- Обробляючи HTTP запит в cmd/server використайте http.DefaultClient для відправки GET /db/<key> сервісу db
- При ініціалізації cmd/server виконайте POST /db/<ім’я команди>
- Відкорегуйте інтеграційний тест вашого балансувальника. В результаті, балансувальник має відправляти запит на один з серверів, а ті, у свою чергу, сервісу db.

Оцінювання

Для оцінки роботи, відправте її через команду у Slack звичним чином:
`/submit 5 <github url>`

Максимальный бал за виконання роботи – 8 (по 2 бала за кожне завдання).