Науковий вісник НЛТУ України
Scientific Bulletin of UNFU
ISSN 1994-7836 (print)
ISSN 2519-2477 (online)
https://doi.org/10.36930/40330608
Article received 21.11.2023
Article accepted 21.12.2023
UDC 004.4
Correspondence author: D. O. Horman dimahorman@gmail.com

Д. О. Горман, Т. О. Коротєєва
Національний університет "Львівська політехніка", м. Львів, Україна

АДАПТУВАННЯ КОРИСТУВАЦЬКИХ СЕРВІСІВ ДЛЯ ЗБЕРІГАННЯ ДАНИХ У ПРОЦЕСІ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Подано розробку та результати оцінювання нової бібліотечної архітектури для зберігання даних у форматі ключ-значення, яка використовує зовнішні кінцеві точки користувацьких API. Це дасть змогу полегшити та здешевити процес збереження даних і конфігурації сховища завдяки можливості безоплатного використання публічних API та мінімізації взаємодії з складною інфраструктурою хмарних провайдерів. Розроблена архітектура дає змогу зберегти модульність внутрішніх компонент, забезпечити безпеку збережених даних, оптимізувати процес зберігання даних і спростити налаштування для розробників. Проаналізовано попередні дослідження щодо спрощення взаємодії зі сховищами ключ-значення, з основним акцентом на підвищенні ефективності та зменшенні витрат. Під час аналізу акцентовано увагу на деталях щодо оптимізації конфігурації та ефективності використання систем ключ-значення. Окреслено ключові особливості цих досліджень та подано недоліки проаналізованих рішень для покращення користувацького досвіду використання сховищ. Наведено результати аналізу популярних рішень для зберігання ключів і значень на підставі хмарних платформ, зокрема Amazon Web Services (AWS) і Microsoft Azure. Встановлено особливості, переваги та недоліки використання цих хмарних рішень, а також особливості як безкоштовних, так і платних планів. Досліджено інтеграцію API GitHub, як системи зберігання ключів і значень, визначено основні функції та можливості цього методу зберігання, який ґрунтується на Git і функціях GitHub. Проаналізовано результати тестів інтеграції API, акцентуючи увагу на продуктивності та ефективності цього рішення. Проведено порівняння сховища "ключ-значення" на підставі інтеграції з GitHub API із традиційними хмарними рішеннями для зберігання. На підставі аналізу та порівняння сформульовано висновок щодо випадків використання та доцільності впровадження дослідженого рішення в розробленні програмного забезпечення.

Ключові слова: сховище; ключ-значення; хмарні платформи; користувацький API; транзакція; Git; бібліотека; модульна архітектура.

Вступ / Introduction

У сфері хмарних обчислень, що швидко розвиваються, використання систем зберігання "ключ-значення" стало повсюдним серед розробників для масштабованого керування даними. Однак природна складність налаштування і використання цих рішень для зберігання даних у хмарних платформах є актуальною проблемою, особливо для розробників швидких або невеликих проектів. Актуальність цієї проблеми підкреслюється поточним станом досліджень у цій галузі, де численні дослідження [3, 14, 18] визначили складну природу інтеграції сховищ ключ-значення як перешкоду для ефективної та економічно вигідної практики розроблення.

Отож, постало завдання розробити нову модель сховища "ключ-значення", яка використовує API популярних користувацьких веб-сервісів. Таке рішення потенційно спростить взаємодію зі сховищами та здешевить загальне розроблення програмного забезпечення за рахунок мінімізації взаємодії з хмарними провайдерами та використанню публічних кінцевих точок API.

Постановка завдання дослідження – розробити модель сховища ключ-значення на підставі адаптації API користувацьких веб-сервісів задля потенційного здешевлення та вдосконалення процесу розроблення програмного забезпечення.

Об'єкт дослідження – програмні реалізації сховищ ключ-значення.

Предмет дослідження – методика імплементації роботи сховища на підставі нової моделі з використанням стороннього API.

Мета роботи – адаптувати користувацькі веб-сервіси під задачі сховищ "ключ-значення" задля потенційного здешевлення та вдосконалення процесу розроблення програмного забезпечення.

Для досягнення зазначеної мети визначено такі основні *завдання дослідження*:

1) Дослідити наявні методи зберігання даних у форматі "ключ-значення".
2) Провести проектування програмної бібліотеки для нової моделі сховища з використанням API.
3) Провести тестування розробленої програмної бібліотеки та проаналізувати результати її роботи.

Інформація про авторів:
Горман Дмитро Олегович, магістр, кафедра програмного забезпечення. **Email:** dimahorman@gmail.com
Коротєєва Тетяна Олександрівна, канд. техн. наук, доцент, кафедра програмного забезпечення.
Email: tetyana.o.korotyeyeva@lpnu.ua; https://orcid.org/0000-0002-6287-5895

Цитування за ДСТУ: Горман Д. О., Коротєєва Т. О. Адаптування користувацьких сервісів для зберігання даних у процесі розроблення програмного забезпечення. Науковий вісник НЛТУ України. 2023, т. 33, № 6. С. 62–68.
Citation APA: Horman, D. O., & Korotyeyeva, T. O. (2023). Adapting user web services for data storing in software development. *Scientific Bulletin of UNFU*, 33(6), 62–68. https://doi.org/10.36930/40330608

Аналіз останніх досліджень та публікацій. Розглянуто публікації, які вивчають рішення для спрощення роботи зі сховищами ключів і значень, та зменшення витрат їхнього використання в межах розроблення ПЗ. Зокрема, в публікаціях [1, 7, 13, 18] окреслено дослідження, пов'язані з роботою новітніх технологій, такі як: Amazon DynamoDB Accelerator (DAX), CockroachDB, TiKV і Redis із RedisGears.

У Ceresnak [1] наведено роботу DAX, яка оптимізує DynamoDB, додаючи рівень кешування в пам'яті, зменшуючи затримку на читання та витрати. Ця розробка має на меті спростити конфігурацію, забезпечуючи повну інтеграцію з DynamoDB. Незважаючи на те, що DAX підвищує ефективність читання, його переваги найпомітніші в разі навантаження, пов'язаного з читанням. Програми, що потребують інтенсивного запису, можуть не мати такого ж рівня підвищення продуктивності.

DAX становить додатковий рівень, потенційно ускладнюючи архітектуру. Розробники повинні ретельно оцінити, чи виправдовує доданий рівень компроміс у простоті для певних випадків використання. Так, у роботі [18] наведено особливості роботи CockroachDB зі своєю моделлю розподіленого SQL та наголошено на масштабованості та узгодженості даних. Конфігурація спрощена завдяки автоматизованим завданням керування, що дає розробникам змогу зосередитися на логіці роботи програми. Модель надійної узгодженості CockroachDB, хоч і корисна, може спричинити затримку для розподілених транзакцій. Це може вплинути на продуктивність у сценаріях, де низька затримка є критичною. Розширені функції та розподілена природа CockroachDB можуть поставити крутішу криву навчання для розробників, які не знайомі зі системами розподілених баз даних.

У дослідженні Huang [7] наведено рішення у вигляді TiKV, розподіленого сховища ключ-значення з відкритим кодом, яке надає пріоритет горизонтальній масштабованості та високій продуктивності. Автоматизоване керування на підставі алгоритму консенсусу Raft спрощує конфігурацію, сприяючи простоті використання. Зосередження TiKV на горизонтальній масштабованості може призвести до підвищення операційної складності для менших розгортань. Переваги його архітектури стають помітнішими у великомасштабних, розподілених налаштуваннях. Хоча TiKV прагне до простоти, точне налаштування конфігурацій для оптимальної продуктивності може вимагати глибшого розуміння розподілених систем, що створює проблеми для менш досвідчених розробників.

У роботі [13] автори розглядають модульний сервіс RedisGears, який інтегрує безсерверне оброблення даних у базу даних Redis, зменшуючи потребу у зовнішніх ресурсах оброблення. Такий підхід спрощує конфігурацію та потенційно сприяє економічній ефективності. RedisGears покладається на Redis, яка є базою даних у пам'яті. Це може обмежити його придатність для випадків використання, які потребують зберігання великих наборів даних, які перевищують доступну оперативну пам'ять. Хоча безсерверне оброблення даних є сильною позицією, RedisGears може бути не найкращим рішенням для надскладних завдань оброблення даних, які виходять за межі його можливостей.

Отже, проаналізувавши останні дослідження можна констатувати, що низка рішень для зберігання ключів і їхніх значень істотні переваги, потенційні недоліки містять компроміси в продуктивності на підставі характеристик робочого навантаження, додаткову складність у певних сценаріях і міркування, пов'язані з кривими навчання та операційними тонкощами.

Результати дослідження та їх обговорення / Research results and their discussion

Сучасні засоби-аналоги програмної бібліотеки. Використання функціоналу сховищ ключ-значення традиційно відбувається в межах сервісів, які надаються потужностями популярних хмарних платформ. Найпоширенішими хмарними платформами є Google Cloud, Microsoft Azure та Amazon Web Services. Функціонал баз даних, який надається тим чи іншим хмарним провайдером, зіставний за особливостями користування, реалізації сервісів сховищ. Для дослідження хмарних рішень обрано Microsoft Azure [11, 17] та DynamoDB [8, 23, 24].

Azure надає кілька рішень для зберігання ключів і значень, кожне з яких має власний набір функцій, тарифні плани та випадки використання. Серед таких служб, які надає можливість використовувати Azure, є: Azure Cache для Redis, Azure Table Storage та Azure Cosmos DB.

Azure Cache для Redis – це сховище даних у пам'яті на підставі відкритої платформи Redis. Воно використовується для кешування та оброблення даних у реальному часі. Azure Cache для Redis пропонує базовий, стандартний і преміум-рівень. Серед особливостей можна виділити те, що це сховище має високошвидкісний доступ до даних завдяки зберіганню в пам'яті, підтримує реплікацію даних і високу доступність [11, 22]. Сховище можна використовувати для кешування, керування сеансами та аналітики в реальному часі. Серед потенційних недоліків можна відзначити, що Azure Cache для Redis не є безкоштовним сервісом та може бути відносно дорогим залежно від вибраного плану та використаного об'єму пам'яті.

Azure Table Storage – це сховище даних NoSQL, яке зберігає структуровані дані у форматі "ключ-значення", що робить його придатним для напівструктурованих і структурованих даних. Сервіс пропонує безкоштовні 5 ГБ пам'яті, 20 000 одиниць читання та 2000 одиниць запису на місяць [5, 21]. Додаткове використання оплачується на підставі ємності та одиниць транзакцій. Проаналізувавши особливості цього сервісу, можна виділити такі потенційні переваги його використання:

- відносно низька вартість, особливо для малих і середніх навантажень;
- добре підходить для зберігання структурованих даних із можливістю масштабування;
- легко інтегрується з іншими службами Azure.

До недоліків можна віднести обмежені можливості запитів порівняно зі сучасними базами даних і може не підходити для виконання транзакцій над великими кількостями даних.

Azure Cosmos DB – це глобально розподілена служба баз даних із кількома моделями, яка може зберігати дані у форматах "ключ-значення", "документ", "граф" або "стовпчик" [17]. Вона пропонує безкоштовний рівень із певними обмеженнями та платою залежно від пропускної здатності, зберігання та передачі даних. Azure Cosmos DB пропонує безкоштовний рівень із певними обмеженнями (наприклад, пропускна здатність

400 запитів/с і 5 ГБ пам'яті). Ціноутворення залежить від пропускної здатності, зберігання та передачі даних. Серед переваг цього сховища можна відзначити:

- глобальне розповсюдження з доступом із низькою затримкою по всьому світу;
- підтримка кількох моделей робить його універсальним для різних типів даних;
- гарантії високої доступності, масштабованості та угоди про рівень обслуговування (англ. *Service Level Agreement*, SLA) – угода між постачальником послуг і користувачем про рівень послуг [11].

Серед потенційних недоліків:

- дороге для масштабних застосувань, потенційно перевищуючи бюджет;
- складність у плані налаштування та управління.

Amazon DynamoDB – це керована служба бази даних NoSQL, яку надає AWS (англ. *Amazon Web Services*). Вона створена для високої доступності, масштабованості та продуктивності з низькою затримкою [23]. DynamoDB – це база даних "ключ-значення" та база даних документів, яка пропонує як безкоштовні, так і платні плани. DynamoDB надає безкоштовний рівень із такими можливостями використання на місяць [24]:

- 25 ГБ пам'яті;
- 25 одиниць ємності для читання (англ. *Read-Copy-Update*, RCU) і 25 одиниць ємності для запису (англ. *Write-Copy-Update*, WCU);
- 25 резервних одиниць ємності для запису;
- 2,5 мільйона одиниць ємності для читання на вимогу.

Платні плани DynamoDB пропонують дві основні моделі ціноутворення – фіксована ємність і ємність на вимогу. У фіксованій моделі користувачем вказується необхідна кількість RCU та WCU. Ціна базується на наданій ємності (за годину) і кількості збережених даних. Пропонує передбачувану продуктивність і вартість для робочих навантажень із стабільним трафіком. У моделі на вимогу не потрібно заздалегідь вказувати кількість RCU і WCU. Користувач платить за запит (читання та запис) і обсяг збережених даних. Добре підходить для робочих навантажень із змінними або непередбачуваними моделями трафіку. Серед потенційних переваг, які може надавати DynamoDB користувачам, є:

- автоматичне масштабування для оброблення високого трафіку без ручного втручання;
- забезпечує однозначну тривалість відповіді в мілісекундах, що робить його придатним для додатків у реальному часі;
- AWS виконує адміністративні завдання, як-от надання обладнання, встановлення виправлень і резервне копіювання даних;
- забезпечує шифрування в стані спокою;
- підтримує багаторегіональну та глобальну реплікацію [8].

Серед потенційних недоліків DynamoDB можна навести:

- значна ціна для додатків із високим трафіком, особливо в разі використання виділеної ємності;
- може бути не найкращим вибором для складних запитів, які потребують об'єднань або аналітики;
- розроблення схеми та запитів може бути складною справою, особливо для тих, хто звик до реляційних баз даних;
- кількість глобальних вторинних індексів обмежена наявною моделлю потужності;
- використання DynamoDB може призвести до блокування в екосистемі AWS, роблячи її менш переносимою для інших хмарних постачальників.

Модель сховища ключ-значення. Запропоновано нову модель сховища "ключ-значення", яка використовує API популярних веб-сервісів користувачів, таких як YouTube, Instagram або GitHub. Створені за такою моделлю сховища використовують наявну інфраструктуру і можливості зберігання даних цих служб і дадуть змогу розробникам залучати їх різні функціональні можливості (рис. 1).

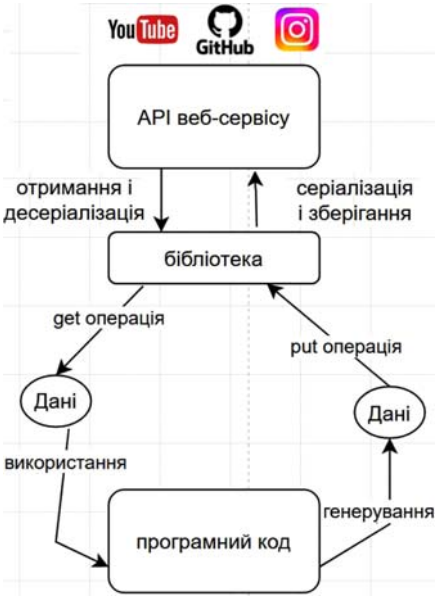

Рис. 1. Загальна схема роботи сховища на підставі користувацького API / General operation scheme of a storage based on a user API

Внаслідок цього, за рахунок мінімізації взаємодії з хмарними провайдерами, буде спрощено взаємодію зі сховищами та здешевлено загальне розроблення програмного забезпечення.

Проектування бібліотеки. Архітектура дотримується модульного та розширюваного шаблону проектування, який поширений у розробленні програмного забезпечення. Основні складники розробленої архітектури є такі:

1) Головна бібліотека слугує основою системи зберігання "ключ-значення" і визначає високорівневі абстракції та інтерфейси, які інкапсулюють основні функції. Вона містить компоненти для шифрування, стиснення, серіалізації та серверних модулів зберігання, кожен з яких абстрагований через інтерфейси або класи.
2) Компонент шифрування виконує шифрування та дешифрування даних. Він визначає інтерфейс або набір класів, які дають змогу розробникам під'єднати різні алгоритми або методи шифрування за потреби.
3) Компонент компресії відповідає за зменшення розміру збережених даних для оптимізації зберігання та ефективності передачі [15]. Як і компонент шифрування, він надає інтерфейси або класи розробникам для інтеграції різних алгоритмів або стратегій стиснення.
4) Компонент серіалізації має справу з перетворенням даних у формат, який можна легко зберігати, передавати та реконструювати. Він пропонує класи для вибору методів серіалізації, таких як JSON, XML або спеціальні формати.
5) Компонент інтеграції зі стороннім API абстрагує фактичний механізм зберігання, будь то традиційна база даних, розподілена файлова система чи хмарне сховище. Розробники можуть вибрати серверну систему зберігання та реалізувати необхідний інтерфейс для взаємодії з вибраним рішенням для зберігання.

Реалізації для кожного з основних компонент (шифрування, стиснення, серіалізація та зберігання) розроблені як окремі бібліотеки. Вони є незалежними, що дає

змогу розробникам вибирати конкретну реалізацію, яка найкраще відповідає потребам їх проекту.

Розробники залучають головну бібліотеку та необхідні бібліотеки компонент як залежності у файли конфігурації збірки свого проекту (наприклад, pom.xml для Maven або build.gradle для Gradle). Інструмент збірки (Maven або Gradle) отримує ці бібліотеки зі сховища (наприклад, Maven Central) і вносить їх у шлях до класів проекту.

Розробники використовуватимуть абстракції та інтерфейси, надані бібліотекою ядра у своїх проектах Java. Вони можуть налаштувати основну бібліотеку для використання конкретних реалізацій компонент, створюючи або вводячи їх за потреби.

Загалом розглянута архітектура (рис. 2) сприяє повторному використанню коду, гнучкості та зручності обслуговування. Розробники можуть легко замінювати різні реалізації компонент, не змінюючи основну логіку своєї системи зберігання ключів і значень, що робить її адаптованою до різних вимог і випадків використання. Окрім цього, використання керування залежностями спрощує інтеграцію цих бібліотек у проекти Java, покращуючи досвід розробника.

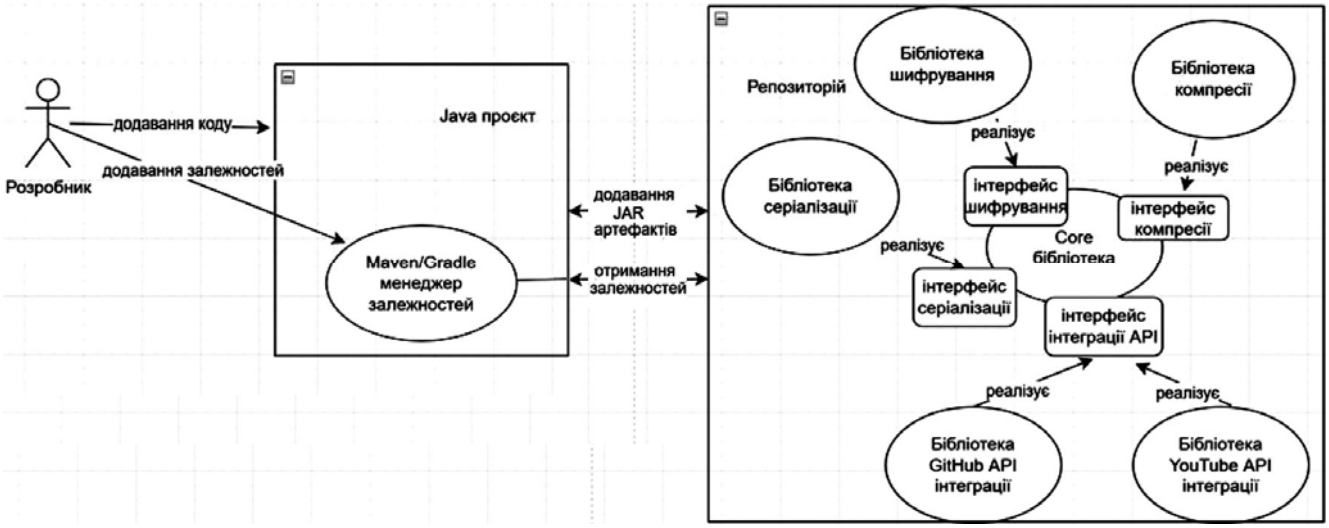

Рис. 2. Архітектура бібліотеки для використання сховища / Architecture of the library for storage utilization

Під час аналізу наявних технологій і рішень було обрано мову Kotlin в межах екосистеми Java. Кілька вагомих причин роблять Java прийнятним вибором для цієї спроби. Незалежність Java від платформи є однією з її особливостей. Це означає, що код, написаний на Java, може працювати на будь-якій платформі зі сумісною віртуальною машиною Java (JVM) [16]. Цей рівень переносимості є важливим під час створення бібліотеки, призначеної для використання розробниками з різноманітними середовищами.

Конфігурація бібліотеки в середовищі розробки. Для використання бібліотеки в Java-проектах потрібно додати її у проект через менеджер залежностей. Для цього вона має бути опублікована в публічному репозиторії. Це може бути локальний репозиторій, віддалений репозиторій або загальнодоступний репозиторій, наприклад Maven Central або JCenter.

У разі коли бібліотека є опублікована, її можна використати у проекті, додавши її через менеджер залежностей. Отже, це можна зробити, налаштувавши файли збірки Gradle (build.gradle) або Maven (pom.xml), щоб приєднати спеціальну бібліотеку як залежність [6, 20, 25].

Для того, щоб мати змогу використовувати сховище "ключів-значень", потрібно додати відповідну конфігурацію до коду проекту. Найпростіша конфігурація клієнту "бази даних" здійснюється за допомогою фрагмента коду, який утилізує шаблон "будівельник" (рис. 3). У цьому разі для клієнта бази даних було визначено дві конфігураційних змінні через змінні оточення. Перша змінна – API_KEY, яка відповідає за секретний ключ доступу до певного стороннього користувацького API. Друга змінна – CLIENT_ID, яка відповідає за ідентифікатор додатку, як клієнта API.

```
fun createApiClientFromEnv(): DatabaseApiClient {
    val apiKey = System.getenv( name: "API_KEY")
        ?: throw RuntimeException("API_KEY environment va
    val clientId = System.getenv( name: "CLIENT_ID")
        ?: throw RuntimeException("CLIENT_ID environment
    return DatabaseApiClient.Builder()
        .apiKey(apiKey)
        .clientId(clientId)
        .build()
}
```

Рис. 3. Конфігурація бібліотеки в коді через шаблон "будівельник" / Configuration of the library in code using the "builder" design pattern

Отже, для роботи бібліотеки потрібно виконати дві дії – додавання бібліотеки через менеджер залежностей та конфігурація в коді.

Робота компоненту інтеграції з стороннім API. Для інтеграції було обрано GitHub API. Створення системи зберігання ключ-значення на підставі загальнодоступного API GitHub має деякі переваги та недоліки. З переваг такого вибору можна виділити такі тези:

- *Масштабованість і гнучкість.* Використання GitHub API, як серверної частини для зберігання ключів і значень, може бути вигідним, оскільки GitHub – це масштабована та надійна платформа. Вона може обробляти велику кількість запитів і структуровано зберігати дані, що робить її придатною для зберігання "ключ-значення" [19].
- *Не потрібно налаштовувати та підтримувати власну базу даних або інфраструктуру зберігання*. Це може заощадити на операційних витратах і зменшити складність програми.
- *Контроль версій.* GitHub надає функції контролю версій, які можуть бути корисними, якщо потрібно відстежувати зміни даних з часом або гарантувати цілісність даних.

З можливих недоліків можна виділити такі:

- *Обмеження швидкості.* GitHub накладає обмеження на швидкість для свого API залежно від рівня автентифікації та типу запитів, які виконуються. Для неавтентифікованих запитів обмеження швидкості нижчі. Це може значно вплинути на продуктивність і доступність системи зберігання "ключ-значення". Щоб залишатися в межах цих обмежень, потрібно ретельно керувати своїми запитами та регулювати їх.
- *Затримка.* Доступ до даних через GitHub API може призвести до більшої затримки порівняно зі спеціальною базою даних або рішенням для зберігання. Це може вплинути на продуктивність.
- *Обмежені операції запису.* API GitHub призначений переважно для читання даних. Однак можливо створювати й оновлювати вміст за допомогою API. Це може бути не найкращим вибором, якщо програма потребує частих операцій запису.

Компонент інтеграції використовує API GitHub і Git (рис. 4), щоб забезпечити універсальне, ефективне рішення для зберігання даних із керуванням версіями.

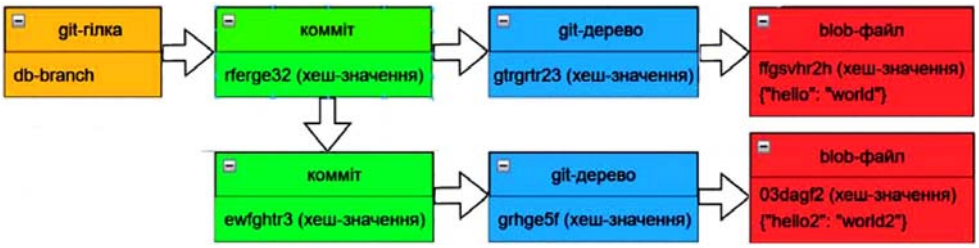

Рис. 4. Використання особливостей Git у сховищі ключів-значень / Utilization of Git features in the key-value store

Операції commit Git використовуються як механізм транзакцій. Вони є атомарними і їх можна легко скасувати. Коли дані потрібно змінити або оновити, можна розробити новий commit Git, забезпечуючи послідовність й атомарність даних [2].

Керування версією бази даних здійснюється через історію операцій commit. Кожен commit представляє версію бази даних. Це полегшує відкат від однієї версії commit до іншої, якщо базу даних потрібно повернути до старішої версії. Можливості контролю версій Git дають змогу відстежувати всю історію бази даних.

Сховище "ключ-значення" ефективно зберігає файли з однаковим вмістом. Можливості дедуплікації Git гарантують, що ідентичні файли зберігаються тільки один раз. Це допомагає оптимізувати простір для зберігання та зменшити надмірність.

Дані зберігаються у формі JSON, XML або будь-якому іншому відповідному форматі, що робить його легко адаптованим до різних структур і типів даних. Це забезпечує гнучкість для різних типів програм і випадків використання.

Гілки Git використовуються як ключі для зберігання даних про ключ-значення. Імена гілок можуть бути унікальними, а дані, пов'язані з певним ключем, зберігаються у файлі гілки. Це забезпечує організований і масштабований спосіб доступу до даних за ключем.

Реплікація даних здійснюється завдяки зберіганню даних у різних віддалених репозиторіях. Здатність Git керувати декількома віддаленими сховищами дає змогу резервувати дані та розподіляти їх між різними серверами чи місцями, що підвищує доступність даних, надійність і можливості аварійного відновлення.

Тестування GitHub API здійснювалося в межах HTTP запитів, ініційованих програмним кодом Kotlin з використанням бібліотеки JGit для взаємодії з Git репозиторієм та бібліотеки ktor для виконання специфічних HTTP запитів. Внаслідок тестування було отримано такі результати:

1) *Тривалість відповіді для операцій GET.* Зазвичай операції GET для даних малого та середнього розміру мали час відгуку в діапазоні від мілісекунд до кількох секунд. Для більших наборів даних або в періоди високого навантаження на сервер тривалість відповіді тривав до кількох секунд та більше. Часті запити GET, що близькі до обмежень швидкості або перевищують їх, призводили до збільшення затримки.
2) *Тривалість відповіді для операцій PUT.* Операції PUT зазвичай містять створення або оновлення вмісту, що може бути повільнішим, ніж операції GET. Тривалість відповіді для операцій PUT змінювалась від кількох до десятків секунд або більше залежно від таких факторів, як розмір даних і навантаження на сервер.
3) *Обмеження швидкості* (*Rate limiting*). GitHub накладає обмеження на швидкість для свого API. Для неавтентифікованих запитів ці обмеження є відносно низькими (наприклад, 60 запитів на годину для деяких кінцевих точок). Автентифіковані запити, особливо з вищими дозволами, мають вищі обмеження швидкості, але все ще підпадають під обмеження.
4) *Розмір даних.* Репозиторії GitHub можуть обробляти значний обсяг даних, але дуже великі набори даних можуть призвести до довшого часу відповіді. GitHub накладає обмеження на максимальний розмір файлу для окремих файлів у сховищі, що становить 100 МБ на файл. Файли, розмір яких перевищує це обмеження, не можна додавати до сховища GitHub. Існує м'яке обмеження на загальний розмір сховища, яке становить приблизно 2 ГБ для всього сховища, враховуючи всі файли та історію версій.
5) *Паралельні запити.* Кількість одночасних запитів, які може обробити система, залежить від обмежень швидкості, рівня автентифікації та навантаження на сервер. Щоб максимізувати паралельність, може знадобитися реалізувати інтелектуальну чергу запитів і обмеження швидкості. Якщо використовується автентифікація з дозволами низького рівня, можливо отримати дещо вищі обмеження швидкості, але вони все одно відносно низькі порівняно зі спеціальними базами даних. Паралельність може бути в діапазоні від десятків до сотень запитів на годину, залежно від кінцевої точки та дозволів. Кількість паралельних запитів може бути в діапазоні від сотень до тисяч запитів на годину, з дозволами високого рівня.
6) *Узгодженість і доступність даних.* GitHub забезпечує надійну узгодженість даних завдяки контролю версій. Кожна зміна даних відстежується як commit операція у сховищі Git, гарантуючи, що можна буде отримати доступ до попередніх версій даних. Commit GitHub є атомарною операцією, тобто зміна застосована повністю або не застосована взагалі. Це забезпечує узгодженість даних у кожній зафіксованій операції commit. Розподілена архітектура GitHub може призвести до певної затримки в реплікації даних на серверах, що водночас призводить до кінцевої узгодженості. Це означає, що може бути невелика затримка (від секунд до

хвилин), перш ніж зміни повністю поширяться на всі сервери GitHub і стануть доступними.

Архітектурний дизайн бібліотеки відіграв вирішальну роль у забезпеченні її гнучкості, ремонтопридатності та простоти використання. Бібліотека була структурована навколо кількох основних компонент, включаючи компоненти компресії, серіалізації, шифрування та інтеграції стороннього API. Зовнішнім API, вибраним для інтеграції з бібліотекою, був GitHub API, який був протестований, щоб перевірити його можливості зберігання "ключ-значення". Важливо зазначити, що ця реалізація не мала на меті замінити звичайні рішення для зберігання даних, оскільки вона мала низку обмежень, зокрема обмеження швидкості з боку GitHub, обмеження розміру даних і змінну тривалість відповіді серверу. Однак GitHub API запропонував переваги щодо достатньої загальної продуктивності, атомарності операцій і узгодженості даних.

Обговорення результатів дослідження. Обрана в цій роботі модульна архітектура бібліотеки з використанням зовнішніх кінцевих точок користувацьких API дасть змогу розробникам замінювати конкретні компоненти, не змінюючи основну логіку своєї системи зберігання ключів і значень, цим самим ефективно задовольняючи різноманітні вимоги та випадки використання.

Порівняно з аналогічним підходом використання модульної архітектури TiKV, що досліджена Huang в роботі [7], спроектована бібліотека має просту конфігурацію, доцільна для оброблення невеликої кількості запитів на годину та потребує менше виділеної пам'яті порівняно з відомими рішеннями.

Якщо порівнювати параметри продуктивності та кількості даних, які можна зберегти, використовуючи GitHub API, з типовими хмарними провайдерами та їхніми безкоштовними планами, то звичайно певна перевага в хмарних провайдерів є хмарні сховища "ключ-значення", що забезпечують високу масштабованість та низьку затримку при операціях GET та PUT, а також можуть обробляти велику кількість одночасних викликів [5]. Втім, якщо говорити про використання сховища в межах спроектованої бібліотеки, то ця взаємодія зі сховищем потенційно може бути спрощеною з точки зору конфігурацій та інтерфейсу доступу до даних. Тоді як робота з хмарними провайдерами вимагає певного рівня знань, навичок та досвіду, що водночас є істотним ускладненням [8, 26].

Отже, за результатами виконаної роботи можна сформулювати такі наукову новизну та практичну значущість результатів дослідження.

Наукова новизна отриманих результатів дослідження – вперше розроблено нову модель сховища "ключ-значення" з використанням стороннього API, що дає змогу потенційно здешевити та вдосконалити процес розроблення програмного забезпечення.

Практична значущість результатів дослідження – спроектовано бібліотеку, яку можна використати як сховище "ключ-значення" у програмних проектах, що дасть змогу здешевити та вдосконалити використання сховищ порівняно з популярними аналогами баз даних, які надаються разом з потужностями хмарних провайдерів, оскільки воно базується на використанні безкоштовного публічного API та його кінцевих точок.

Висновки / Conclusions

Спроектовано бібліотеку, що заснована на роботі стороннього API та можна використати в ролі сховища "ключ-значення" у програмних проектах. Досліджено роботу інтегрованого API в ролі сховища "ключ-значення". За результатами виконання роботи можна зробити такі висновки.

1. Проаналізовано особливості використання популярних сховищ "ключ-значення" в межах хмарних провайдерів. Показано можливості, обмеження безкоштовних і платних планів, переваги і недоліки використання хмарних сервісів. З основних недоліків виокремлено складність конфігурації і використання сервісів у межах інфраструктури хмарного провайдера та висока ціна використання за помірного навантаження. Тоді як з переваг – наявність безкоштовних планів, втім з обмеженнями розміру сховищ та кількості запитів на годину, безпека та шифрування даних, висока доступність за рахунок розподіленої структури деяких сховищ.

2. Наведено архітектурний дизайн бібліотеки, в основі якої лежить модульна структура, що забезпечує гнучкість та взаємозамінність компонент. Водночас, кожен модуль представляє окрему бібліотеку, яку можна підключити до власного проекту основних компонент. Загалом структура містить компоненти компресії, серіалізації, шифрування та інтеграції стороннього API.

3. Протестовано інтеграцію з GitHub API в ролі сховища "ключ-значення". Отримані результати свідчать про те, що дане рішення можна використовувати в розробленні програмного забезпечення, втім у нього є такі обмеження, як невелика загальна кількість запитів на годину та кількість виділеної пам'яті, яку можна використовувати в межах GitHub API й тривалий час відгуку операцій з даними. Перевагою створеного рішення можна назвати просту конфігурацію і використання його в ролі сховища "ключ-значення" в межах спроектованої бібліотеки, оскільки не вимагає спеціалізованих знань та має простий інтерфейс доступу до даних. Загалом використання такого сховища у великих корпоративних проектах, в яких є великі навантаження на систему, великі кількості запитів на секунду (сотні або тисячі) та величезні об'єми даних (терабайти), не має сенсу, оскільки описані обмеження продуктивності не дають змогу ефективно використати його у такий спосіб. Використання цього сховища має сенс у простих невеликих або тестових програмних проектах, які не вимагають високої продуктивності і великого об'єму пам'яті сховища як простого та швидкого рішення для зберігання даних.

References

1. Ceresnak, R., Kvet, M., & Matiasko, K. (2021). Improved method of selecting data in a nonrelational database. In *2021 International Conference on Information and Digital Technologies* (IDT), 59–64. https://doi.org/10.1109/IDT52577.2021.9497640
2. Chacon, S., & Straub, B. (2014). Pro Git 2nd Edition. Computer Science. URL: https://kupichitay.com.ua/product/pro-git-2nd-edition-scott-chacon-ben-straub/
3. Gill, S. S., Tuli, S., Xu, M., Singh, I., Singh, K. V., Lindsay, D., Garraghan, P., et al. (2019). Transformative effects of IoT, Blockchain and Artificial Intelligence on cloud computing: Evolution, vision, trends and open challenges. Internet of Things, 8. https://doi.org/10.1016/j.iot.2019.100118
4. Gupta, A., Tyagi, S., Panwar, N., Sachdeva, S., & Saxena, U. (2017). NoSQL databases: Critical analysis and comparison.

2017 International Conference on Computing and Communication Technologies for Smart Nation (IC3TSN). https://doi.org/10.1109/ic3tsn.2017.8284494
5. Gupta, B., Mittal, P., & Mufti, T. (2021). A Review on Amazon Web Service (AWS), Microsoft Azure & Google Cloud Platform (GCP) Services. ICIDSSD 2020. https://doi.org/10.4108/eai.27-2-2020.2303255
6. Hassan, F., Mostafa, S., Lam, E. S. L., & Wang, X. (2017). Automatic Building of Java Projects in Software Repositories: A Study on Feasibility and Challenges. 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). https://doi.org/10.1109/esem.2017.11
7. Huang, D., Liu, Q., Cui, Q., Fang, Z., Ma, X., Xu, F., Tang, X., et al. (2020). TiDB: a Raft-based HTAP database. Proceedings of the VLDB Endowment, 13(12), 3072–3084. https://doi.org/10.14778/3415478.3415535
8. Kalid, S., Syed, A., Mohammad, A., & Halgamuge, M. N. (2017). Big-data NoSQL databases: A comparison and analysis of "Big-Table", "DynamoDB", and "Cassandra." 2nd International Conference on Big Data Analysis. https://doi.org/10.1109/icbda.2017.8078782
9. Khalil, A., & Belaissaoui, M. (2020). Key-value data warehouse: Models and OLAP analysis. 2nd International Conference on Electronics, Control, Optimization and Computer Science (ICECOCS). https://doi.org/10.1109/icecocs50124.2020.9314447
10. Manchale, A. (2021). Tug Grall On Redis. Software, 38(4), 130–132. https://doi.org/10.1109/ms.2021.3073167
11. Mishra, A. (2023). Developing Cloud-Native Solutions with Microsoft Azure and NET: Build Highly Scalable Solutions for the Enterprise.
12. Mistrik, I., Galster, M., & Maxim, B. (2019). Software Engineering for Variability Intensive Systems: Foundations and Applications.
13. Patel, R. (2021). Data+ Education. Redis Is a Cache or More?. EasyChair Preprint, 88.
14. Pothukuchi, A. S., Kota, L. V., & Mallikarjunaradhya, V. (2021). A Critical Analysis of the Challenges and Opportunities to Optimize Storage Costs for Big Data in the Cloud.
15. Sayood, K. (2018). Introduction to Data Compression.
16. Schildt, H. (2018). Java: The Complete Reference, Eleventh Edition.
17. Stigler, M. (2018). Beginning Serverless Computing: Developing with Amazon Web Services, Microsoft Azure, and Google Cloud.
18. Taft, R., Sharif, I., Matei, A., VanBenschoten, N., Lewis, J., Grieger, T., Mattis, P., et al. (2020). Cockroachdb: The resilient geo-distributed sql database. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, 1493–1509. https://doi.org/10.1145/3318464.3386134
19. Tsitoara, M. (2019) Beginning Git and GitHub: A Comprehensive Guide to Version Control, Project Management, and Teamwork for the New Developer.
20. Varanasi, B. (2019). Introducing Maven: A Build Tool for Todays Java Developers.
21. Website. (2022). Azure Table Storage Overview. URL: https://learn.microsoft.com/en-us/azure/storage/tables/table-storage-overview.
22. Website. (2023). About Azure Cache for Redis. URL: https://learn.microsoft.com/en-us/azure/azure-cache-for-redis/cache-overview.
23. Website. (2023). Amazon DynamoDB features. URL: https://aws.amazon.com/dynamodb/features/.
24. Website. (2023). Amazon DynamoDB pricing. URL: https://aws.amazon.com/dynamodb/pricing/.
25. Website. (2023). Gradle User Guide. Part 1: Initializing the Project. URL: https://docs.gradle.org/current/userguide/part1_gradle_init.html.
26. Yang, C., Huang, Q., Li, Z., Liu, K., & Hu, F. (2017). Big Data and cloud computing: innovation opportunities and challenges. *International Journal of Digital Earth*, 10(1), 13–53. https://doi.org/10.1080/17538947.2016.1239771

D. O. Horman, T. O. Korotyeyeva

Lviv Polytechnic National University, Lviv, Ukraine

ADAPTING USER WEB SERVICES FOR DATA STORING IN SOFTWARE DEVELOPMENT

The article introduces a novel library architecture designed for key-value data storage utilizing external user API endpoints. This approach aims to simplify and economize data storage and configuration by leveraging public APIs, thus reducing dependence on complex cloud infrastructure. The developed architecture prioritizes internal component modularity, data safety, storage process optimization, and developer-friendly settings. The presented library architecture comprises the following key components: an encryption component ensuring secure data storage, a compressor component optimizing data storage in key-value format, an API integration component enabling key-value storage capabilities, and a serialization component for storing and retrieving serialized data. This article analyzes prior research on simplifying interactions with key-value storages, emphasizing enhanced efficiency and reduced costs. Examining various studies, it synthesizes insights aimed at streamlining configuration, usage, and cost-effectiveness of key-value storage systems. The delineation of key features of the research data is outlined, and the shortcomings of the analyzed solutions are presented to enhance the user experience in utilizing key-value storage. The paper includes the analysis of popular key-value storage solutions on cloud platforms such as Amazon Web Services (AWS) and Microsoft Azure, outlining their features, advantages, and drawbacks in both free and paid plans. Additionally, the integration of the GitHub API as a key-value storage system is explored, highlighting its features and capabilities based on Git and GitHub functionalities. The article concludes with an analysis of API integration tests, emphasizing the productivity and efficiency of the GitHub API integration compared to conventional cloud storage solutions. Based on the thorough analysis and comparison, conclusions have been derived regarding the applicability and viability of implementing the investigated solution in software development.

Keywords: storage; key-value; cloud platforms; user API; transaction; Git; library; modular architecture.