УДК 004.738 DOI: 10.30837/0135-1710.2022.178.050

Д.А. НЕФЬОДОВ, С.Г. УДОВЕНКО, Л.Е. ЧАЛА

МІКРОСЕРВІСНА АРХІТЕКТУРА СИСТЕМИ ПОТОКОВОЇ ОБРОБКИ ВЕЛИКИХ ДАНИХ

Розглянуто можливі шляхи використання мікросервісної архітектури у системах потокової обробки великих даних. Досліджено переваги і недоліки існуючих архітектур аналізу великих даних. Запропоновано варіант системи мікросервісної обробки великих даних з використанням розподіленої потокової платформи подій Kafka, платформи розробки та запуску програм Docker, об'єктно-реляційної системи управління базами даних Postgres, а також веб-платформи для створення додатків FastAPI. Розроблено архітектурні шаблони, які можуть спростити розробку застосунків для обробки великих даних з використанням мікросервісів, та концепти програм з використанням отриманих шаблонів. Наведено результати моделювання, які свідчать про те, що мікросервісна архітектура з розподіленим навантаженням на декілька реплік може забезпечити кращу масштабованість і швидкодію в порівнянні з монолітною архітектурою.

1. Вступ

Через постійно зростаючу кількість даних, які накопичуються у сучасному світі, за останні роки набула популярності концепція великих даних (ВД). Однак реалізація відповідних програм обробки ВД все ще залишається складним завданням. Застосування централізованих систем обробки та аналізу значно підвищує час розробки таких програм для реалізації інформаційних технологій. Одним з потенційних підходів до полегшення цієї задачі є використання нових децентралізованих технологій для створення аналітичних рішень для ВД. Зокрема, останнім часом отримали розвиток дослідження можливостей використання мікросервісів у системах обробки ВД. У зв'язку зі збільшенням вимог до аналізу ВД (АВД) з'явилися різноманітні технології та платформи для здійснення такого аналізу [1]. Відповідно до функціональних специфікацій, ці платформи АВД загалом поєднують логіку обробки даних із керуванням обчислювальними ресурсами під час обробки.

Актуальність використання мікросервісів для АВД пояснюється тим, що на сьогоднішній день існує великий попит на системи, здатні працювати з ВД, а мікросервісна архітектура дозволяє розробляти гнучкіші варіанти обробки потокових даних в реальному часі (зокрема, з використанням контейнерних технологій).

Традиційним підходом до побудови систем потокової обробки даних (СПОД) досі залишається використання монолітної архітектури [2]. Монолітна архітектура - це традиційний спосіб побудови програмних додатків СПОД, згідно з яким ці додатки будуються як неподільний блок, основними складовими якого є бази даних (БД), інтерфейс на стороні клієнта та сервер для обробки запитів. Компоненти такої архітектури тісно пов'язані один з одним і мають розроблятися та керуватися як одна сутність, оскільки їх виконання відбувається в рамках одного процесу операційної системи. При цьому навіть незначна зміна в одній частині будь-якого компонента вимагає нового релізу всього програмного забезпечення СПОД.

Функціонування складних організаційно-технічних систем зазвичай пов'язане з проблемами, обумовленими зростанням обсягу потоків даних, необхідністю інтеграції з новими інформаційними об'єктами, оновленням програмної платформи тощо. Зі зростанням таких проблем монолітні архітектури потроху втрачають свою актуальність через необхідність підтримувати їх та вносити зміни (наприклад, в разі розширення функцій систем обробки даних). У зв'язку з цим все більшої популярності набувають так звані мікросервісні архітектури (МСА). Головний принцип таких архітектур - розбиття монолітного програмного додатку деякої інформаційної системи на сукупність модулів (мікросервісів), що можуть бути в разі потреби вилучені або додані (за умови дотримання певних вимог) без порушення функціональності всієї системи. Такий підхід дає можливість розширювати інформаційну систему, масштабувати її та забезпечувати швидку та гнучку оркестрацію всіх модулів.

Можна визначити такі позитивні властивості МСА: модульність (кожний мікросервіс призначений для вирішення конкретної задачі); мінімальний об'єм програмного коду для

кожного з мікросервісів; багатоплатформеність (можливість використання різних засобів та технологій для модулів); відмовостійкість тощо. Цей перелік характеристик підкреслює виграшну позицію мікросервісних архітектур в порівнянні з монолітними. Однак МСА мають також і певні недоліки, зокрема: розробка складних програмних проєктів з використанням мікросервісів є довготривалим процесом; існують певні труднощі при перерозподілі функцій між мікросервісними компонентами системи; кожен мікросервіс потребує окремого обслуговування, тому необхідним є постійний автоматизований моніторинг та логування даних.

Слід зазначити, що на сьогодні немає універсальних рекомендацій щодо застосування мікросервісного підходу для деяких завдань обробки ВД в інформаційних системах різного функціонального призначення. Зокрема, потребує додаткових досліджень проблема децентралізації потокової обробки ВД, що може вирішуватися із застосуванням мікросервісного підходу. При цьому доцільним і актуальним є проведення аналізу МСА для дослідження їх переваг в порівнянні з використанням монолітних сервісів для обробки потоків ВД.

Розглянуті вище особливості побудови систем обробки ВД із застосуванням різних концептуальних підходів (моносервісного та мікросервісного) обумовили характер досліджень, результати яких обговорюються у даній роботі.

Метою цих досліджень є розробка варіанту МСА системи потокової обробки ВД, яка основана на використанні сучасних засобів розробки та запуску додатків в середовищах з підтримкою контейнеризації.

Відповідно до поставленої мети, необхідно вирішити наступні завдання:

- аналіз існуючих МСА потокової обробки ВД;
- розроблення варіанту системи мікросервісної обробки ВД з використанням розподіленої потокової платформи подій Kafka; платформи розробки, доставки та запуску програм Docker; об'єктно-реляційної системи управління БД Postgres; веб-платформи для створення додатків FastAPI;
- моделювання запропонованої МСА СПОД з метою доведення її переваг в порівнянні з монолітною архітектурою СПОД для обробки потоків ВД.

2. Сучасні архітектури потокової обробки великих даних з використанням мікросервісів

Для визначення сукупності компонентів, що є доцільним використати в подальшому у варіанті багатокомпонентної архітектури потокової обробки ВД, розглянемо властивості, переваги і недоліки існуючих схем та засобів такої обробки [3-5].

Відзначимо, що термін "великі дані" (Big Data) є досить розпливчастим. З еволюційної точки зору розмір "великих даних" постійно змінюється. Якщо, наприклад, використовувати поточну глобальну пропускну здатність Інтернет-трафіку як вимірну одиницю, то значення обсягу ВД будуть розташовані між терабайтним і зетабайтним діапазонами. Компанія IBM запропонувала концепт 4V для визначення ознак ВД: обсяг (Volume), що означає масштабність даних; щвидкість обробки (Velocity), що означає аналіз потокових даних; різноманіття (Variety), що вказує на різні форми ВД; правдивість (Veracity), що передбачає можливість невизначеності ВД. Деякі аналітики включають додаткові визначення ВД на основі літери V: варіативність (Variability), видимість (Visibility) та значимість (Value).

Здебільшого, ВД використовуються для АВД. Крім розвитку методів та алгоритмів АВД, необхідним і важливим є створення високопродуктивних систем для ефективної обробки ВД. На практиці використання таких систем має забезпечувати обчислювальні вимоги механізмів обробки даних для виконання завдань АВД під час виконання програм.

Більшість алгоритмів та систем АВД пристосовані для використання даних, що зберігаються в існуючих базах або сховищах даних. Це означає, що такі дані готові для безпосередньої роботи з ними. Проте існують ситуації, коли вхідні дані надходять потоками до систем їх аналізу та обробки з великою швидкістю оновлення, що не завжди робить можливим та доцільним їх зберігання в БД з метою подальшого опрацювання. Більшість алгоритмів потокової обробки використовують відсортування максимально небажаних/непотрібних даних на вході та подальшу роботу з вхідними даними. Існує також можливість подачі на вхід даних фіксованого розміру, що пришвидшить роботу першочергово, але разом з тим виникне ймовірність потрапляння есенційних даних до алгоритму, що призведе до втрати точності фінального аналізу. Різні алгоритми і підходи мають різні ідеї і можли-

вості для застосування, оскільки має місце питання пріоритетності (оптимальність/затратність).

Потокові системи достатьо часто використовуються в сучасних технологіях обробки даних, але існують суттєві обмеження щодо реалізаці потокової обробки. Перш за все, необхідно враховувати швидкість отримання потоків даних: якщо такі потоки надходять надто швидко, слід обрахувати надісланий пакет даних максимально швидко (при цьому треба брати до уваги ймовірність нерелевантності даних при надто великому очікуванні), тому слід підбирати алгоритм, що матиме змогу виконувати усі операції виключно в операційній системі, не використовуючи допоміжних пристроїв, адже це негативно позначається на швидкості самої системи. В той же час необхідно враховувати те, що даних може бути настільки багато, що не буде можливості отримувати ідеальний варіант, достатньо лише використовувати наближення, котрі задовольняють вимоги модуля: для цього слід використовувати наближені методи або методи, що працюють з хешами. Подібна практика використання є прийнятним компромісом між швидкістю і точністю.

Розглянемо основні операції, які зазвичай використовують для оптимізації роботи з даними. Базовий процес управління даними передбачає, насамперед, операцію вибірки даних, що у СПОД здійснюється з застосуванням хешування. Це дозволяє отримати максимально репрезентативну вибірку даних, а результат хешування буде застосовано для всього потоку.

Іншим типом операцій попередньої обробки потоків є фільтрація вхідних даних. Сама по собі фільтрація даних не є складним процесом, проте необхідно створити максимально швидкий фільтр який виконував би свою функцію максимально швидко [6]. Для цього в СПОД ефективним вважається використання фільтра Блума, що має такі компоненти: масив n біт, де кожне значення дорівнює 0; набір хеш-функцій $h1$, $h2$, ..., hk, що відображають значення ключів для n осередків; множину S, що містить m ключів. Головне призначення фільтра Блума - пропускати всі елементи потоку, ключі яких належать множині S, і відкидати більшість елементів з ключами, що не належать S. Після ініціалізізації бітового масиву встановимо в 1 біти, номери яких співпадають з hi(K) для деякої хеш-функції hi і деякого ключа K з множини S. При обробці ключа K, що надійшов з потоку, перевіряємо, чи значення всіх бітів з номерамі $h1(K)$, $h2(K)$, ..., hk(K) дорівнюють одиниці. Це означає, що при присутності усіх значень хеш-функцій у множині пропускаємо черговий елемент. Якщо хоча б один біт дорівнює 0, то ключ K не може належати S, тому цей елемент відкидаємо. Однак такий елемент може пройти і тоді, коли його ключ відсутній в S. Це може відбутися при недостатній кількості хеш-функцій або недостатній потужності множини S.

Ще однією важливою операцією потокової обробки є знаходження елементів в потоці. Найочевиднішим підходом для вирішення подібної задачі буде збереження в оперативній пам'яті тих елементів, які вже були попередньо зафіксовані. Більш того, можливо зберігати їх в структурі даних, що забезпечує ефективний пошук (наприклад, у хеш-таблиці або дереві пошуку, які дозволяють легко додавати нові елементи і перевіряти їх відповідність черговому елементу з потоку).

При відносно невеликій кількості елементів в потоках даних розглянуті операції можуть бути реалізовані в СПОД без особливих зусиль. Проте, при великій кількості елементів в потоці (або при опрацюванні занадто великої кількості потоків) лишається малоймовірною можливість збереження усіх потрібних даних в оперативній пам'яті СПОД з монолітною архітектурою. Існують різні рішення для подолання цієї проблеми. Можна, наприклад, використовувати декілька машин, кожна з яких буде відповідати за обробку одного або декількох потоків. Крім того, можна зберігати значну кількість даних у зовнішній пам'яті та збирати елементи потоку до пакету, виконуючи багато перевірок та оновлень даних в цьому пакеті та переміщаючи його з диска до оперативної пам'яті. Це деякою мірою відповідає концепції використання МСА для потокової обробки ВД, що досліджується в даній роботі.

Розглянемо особливості уніфікованої архітектури ВД для обробки ВД, що дозволяє приймати, обробляти та аналізувати дані, які є надто об'ємними або надто складними для традиційних систем БД [7].

Уніфікована архітектура ВД (Unified Big Data Architecture) - це архітектура ВД, що включає інфраструктуру, інструменти та технології, які створюють, керують і підтримують збір даних, їх обробку, аналітику та елементи машинного навчання. Уніфікована архітектура передбачає централізовану інфраструктуру даних, щоб уникнути дублювання даних і додаткових програмних зусиль. Рішення для обробки ВД зазвичай призначені для одного або кількох з таких типів

робочого навантаження: пакетне оброблення джерел неактивних ВД; обробка ВД в реальному часі; інтерактивне вивчення ВД; прогнозна аналітика та машинне навчання.

Архітектура потокової обробки ВД (Streaming Big Data Processing Architecture) - це структура програмних компонентів, створена для прийому та обробки великих обсягів потокових даних із багатьох джерел [8]. У той час, як традиційні рішення для обробки даних зосереджені на записі та зчитуванні даних пакетами, архітектура потокових даних споживає дані відразу після їх створення, зберігає їх у сховищі та може включати різноманітні додаткові компоненти для кожного випадку використання, такі як інструменти для обробки в реальному часі, дані маніпуляції та аналітика (рис. 1).

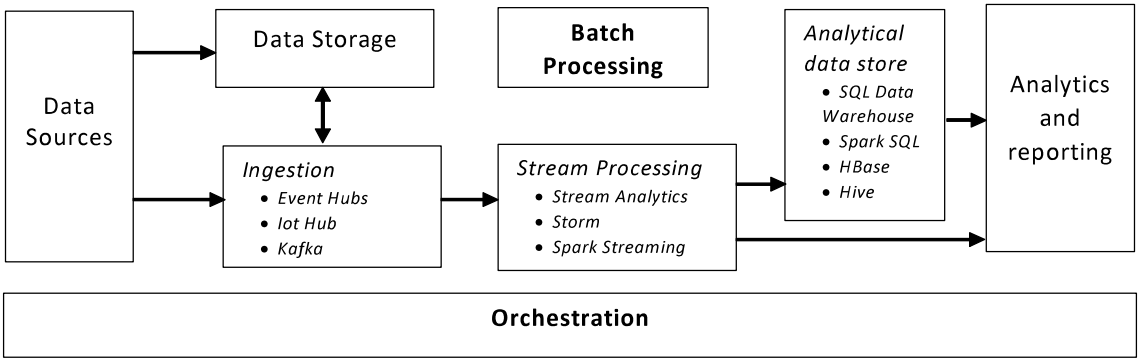

Рис. 1. Базова архітектура потокової обробки великих даних

Варіант базової архітектури потокової обробки ВД, наведений на рис. 1, включає практично всі компоненти уніфікованої архітектури ВД. Основним компонентом є джерела даних (Data Sources). Всі рішення для обробки ВД починаються з одного або кількох джерел даних (це можуть бути: сховища даних додатків, наприклад, реляційні БД; статичні файли, які створюються програмами, наприклад, файли журналу вебсервера; джерела даних з передачею в режимі реального часу, наприклад, пристрої Інтернету речей (IoT Hub). Для систем обробки ВД характерна наявність сховища даних (Data Storage). Дані пакетної обробки зазвичай зберігаються в розподіленому сховищі файлів, де можуть міститися значні обсяги великих файлів у різних форматах. Цей тип сховищ часто називають озером даних (Data Lake).

Слід також розглянути інші компоненти, наявні у більшості архітектур ВД. До таких компонент можна віднести засоби пакетного оброблення (Batch Processing). Оскільки набори даних зазвичай великі, то часто у СПОД обробляються пакетні завдання. Над даними виконуються різні операції, такі як фільтрація, статистична обробка та інші процеси підготовки даних до аналізу. Зазвичай у ці завдання входить також читання вихідних файлів, їх обробка і запис вихідних даних у нові файли.

Ще одним важливим компонентом є отримання повідомлень у режимі реального часу. Якщо рішення містить джерела, що працюють у реальному часі, в архітектурі має бути передбачений спосіб збирання та збереження повідомлень у режимі реального часу для потокової обробки (Stream Processing). Це може бути просте сховище даних з папкою, в яку вхідні повідомлення розміщуються для обробки. Але для прийому повідомлень багатьом рішенням потрібне сховище, яке можна використовувати як буфер. Таке сховище має підтримувати обробку з горизонтальним масштабуванням, надійну доставку та іншу семантику черги повідомлень.

Важливим компонентом систем ВД є також сховища аналітичних даних (Analytical data store). У багатьох рішеннях обробки ВД дані готуються до аналізу. Потім оброблені дані структуруються відповідно до формату запитів для засобів аналітики. Сховище аналітичних даних, яке використовується для обробки таких запитів, може бути і реляційною БД, наприклад, Kimball, що можна побачити в більшості традиційних рішень бізнес-аналітики.. Крім того, дані можна представити за допомогою технологій NoSQL (SQL Data Warehouse, Spark SQL), що мають низьку затримку на читання і запис, а також технології HBase або інтерактивної БД Hive, яка надає абстракцію метаданих для файлів даних у розподіленому сховищі.

Слід відзначити також аналіз та створення звітів (Analytics and reporting). Більшість рішень для обробки ВД дозволяють отримати уявлення про дані за допомогою аналізу та звітів. Для того, щоб розширити можливості аналізу даних, можна включити в архітектуру шар моделювання, наприклад, модель багатовимірного куба OLAP (Online analytical

processing), що сприяє організації великх БД і підтримує їх комплексний аналіз [9]. Їх можна використовувати для виконання складних аналітичних запитів без негативного впливу на транзакційні системи. Останнім компонентом розглянутої архітектури є оркестрація (Orchestration). Більшість рішень для обробки ВД складаються з повторюваних робочих процесів, під час яких перетворюються вихідні дані, дані переміщаються між декількома джерелами і приймачами, оброблені дані завантажуються в сховища аналітичних даних або результати передаються безпосередньо у звіт або на панель моніторингу.

Microsoft рекомендує використовувати подібну архітектуру для таких сценаріїв: зберігання та обробка даних в обсягах, надто великих для традиційної БД; перетворення неструктурованих даних для аналізу та створення звітів; запис, обробка та аналіз асінхронних потоків даних у режимі реального часу або з низькою затримкою;

До переваг такої архітектури можна віднести: вибір технологій та їх комбінацій (наприклад, можна комбінувати та зіставляти керовані сервіси Azure та технології Apache у кластерах HDInsight, щоб з максимальною вигодою застосовувати існуючі навички); підвищення продуктивності за допомогою паралелізму (у рішеннях обробки ВД використовується перевага паралелізму, що дозволяє застосовувати високопродуктивні рішення, які можуть масштабуватися до роботи з великими обсягами даних); еластичне масштабування (всі компоненти архітектури для обробки ВД підтримують горизонтальне масштабування для адаптування рішень для малих і великих робочих навантажень і плати лише за реально використовувані ресурси мережі); взаємодію з поточними рішеннями (компоненти архітектури для обробки ВД також використовуються у відповідних рішеннях Інтернету речей та корпоративних рішеннях бізнес-аналітики, що дозволяє створювати інтегровані засоби для робочих навантажень обробки даних).

Втім розглянута базова архітектура несе додаткову складність, адже практична реалізація обробки ВД може потребувати значної кількості компонентів для прийому даних з декількох джерел. Створення, тестування та усунення неполадок процесів обробки ВД також може стати непростим завданням через велику кількість параметрів конфігурації для оптимізації продуктивності.

Слід також врахувати набір навичок, необхідних для розробників (велика кількість технологій для обробки ВД є вузькоспеціалізованими та використовують платформи і мови, які є стандартними для більш загальних архітектур додатків). З іншого боку, технології обробки ВД сприяють розвитку нових інтерфейсів API з урахуванням більш традиційних мов.

Суттєвою проблемою може стати зрілість технологій, адже багато технологій, що використовуються для обробки ВД, ще знаходяться в розвитку (якщо основні технології Hadoop, Hive та Pig вже сформувалися та широко використовуються, то такі нові технології, як Spark, з кожним релізом зазнають значних змін і вдосконалень).

Важливою характеристикою архітектури ВД є також безпека. У рішеннях з обробки ВД всі статичні дані зазвичай зберігаються у централізованому озері даних Data Lake. Захист доступу до цих даних є нетривіальним завданням, якщо дані повинні прийматися та використовуватися кількома програмами та платформами. У таких системах рекомендується використання паралелізму. У більшості технологій обробки ВД робоче навантаження розподіляється між кількома одиницями обробки, тому статичні файли даних створюються та зберігаються у доступному для розбивки форматі. Розподілені файлові системи типу HDFS мають забезпечити оптимізацію продуктивності читання та запису (зазвичай фактична обробка виконується тут паралельно в кількох вузлах кластера, що скорочує загальний час виконання завдань).

У таких системах пакетна обробка (Batch processing) виконується за регулярним розкладом (наприклад, щотижня або щомісяця). Секціоновані файли та табличні структури даних ґрунтуються на темпоральних періодах, що відповідають розкладу обробки. Це спрощує прийом даних, планування завдань та усунення помилок. Крім того, таблиці розділів, які використовуються у запитах Hive, U-SQL або SQL, можуть значно підвищити їхню продуктивність.

У традиційних рішеннях бізнес-аналітики для переміщення даних до сховища використовується процес екстракції, перетворення та завантаження ETL (Extract-Transform-Load) (рис. 2).

Першим кроком цього процесу є екстракція (Extract) даних із цільових джерел, які зазвичай неоднорідні (наприклад, бізнес-системи, API, дані датчиків, маркетингові інструменти та БД транзакцій тощо). Деякі з цих типів даних є структурованими виходами широко використовуваних систем, тоді як інші є напівструктурованими серверними журналами JSON.

Другий крок полягає в перетворенні (Transform) необроблених даних, отриманих із джерел, у формат, який можна використовувати різними програмами. На цьому етапі дані очищуються, відображаються та перетворюються, щоб відповідати оперативним потребам. Цей процес передбачає кілька типів перетворень, які забезпечують якість і цілісність даних. Дані зазвичай не завантажуються безпосередньо в цільове джерело даних, натомість їх завантажують у проміжну БД. Цей крок забезпечує швидкий відкат, якщо щось піде не за планом.

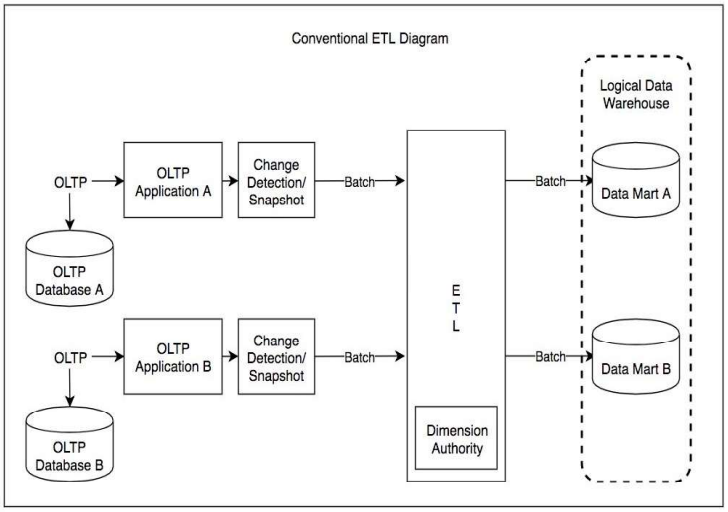

Рис. 2. Уніфікована ETL-діаграма

Останньою функцією ETL є завантаження (Load) - це процес запису перетворених даних із проміжної області в цільову БД. Залежно від вимог програми, цей процес може бути простим або складним. Кожен із цих кроків можна виконати за допомогою інструментів ETL або спеціального коду.

Для великих обсягів даних і різноманітних форматів у рішеннях обробки ВД зазвичай використовуються різні варіації ETL, наприклад, схема перетворення, екстракції і завантаження TEL (Transform-Extract-Load). В цьому випадку дані обробляються в розподіленому сховищі даних та перетворюються на необхідну структуру перед переміщенням до сховища аналітичних даних.

За даної архітектури слід також регулювати витрати при тарифікації на основі обсягу та часу використання. Для завдань пакетної обробки дуже важливо враховувати два фактори: витрати на одиницю обчислювальних вузлів та щохвилинну вартість використання цих вузлів для виконання завдання. Наприклад, виконання пакетного завдання може тривати вісім годин під час використання чотирьох вузлів кластера. При цьому може виявитись, що всі чотири вузли використовуються для завдання лише протягом перших двох годин, а після цього достатньо двох вузлів. У такому разі виконання всього завдання на двох вузлах збільшить загальний час, але не подвоїть його, що зменшить сукупну вартість виконання. У деяких бізнес-сценаріях триваліший час обробки може бути кращим, ніж вища вартість використання ресурсів кластера, що не використовуються.

Таким чином, уніфікована архітектура ВД щільно пов'язана з ETL-процесами, які забезпечують потрапляння даних в реляційне сховище. Перевагою такої моделі є відносна простота і легкість реалізації, а її відмінність від традиційних систем обумовлюється лише вибором технології реалізації, де компоненти бізнес-системи замінені на інструменти Big Data.

Однак, розглянуті технології вимагають багато часу та зусиль для налаштування аналітичних звітів. Сама уніфікована архітектура призначена для пакетної обробки даних і не завжди підтримує потокову передачу подій у реальному часі. Усунути цей недолік можна за допомогою удосконалення архітектурних моделей.

Потокові архітектури мають враховувати унікальні характеристики потоків ВД, які генерують величезні обсяги даних (від терабайтів до петабайтів). Ці дані в кращому випадку є напівструктурованими та потребують значної попередньої обробки та застосування схеми ETL для того, щоб стати інформаційно корисними.

Завдання практичної реалізації потокової обробки ВД рідко вирішується за допомогою однієї БД або ETL-інструментів, тому доцільно спроєктувати рішення, що складається з кількох будівельних блоків. Наприклад, ідея Upsolver SQLake полягає в тому, щоб замінити точкові продукти інтегрованою платформою, яка забезпечує самоорганізовані конвеєри декларативних даних. Далі буде продемонстровано, як цей підхід проявляється в кожній частині ланцюга постачання потокових даних.

Раніше потокова обробка була нішевою технологією, яка використовувалася лише невеликою групою компаній. Однак з розвитком концепції програмного забезпечення як послуги SaaS (Software as a Service), а також IoT і машинного навчання в різних галузях все частіше

використовують потокову аналітику. Оскільки трафік до цифрових активів (додатків або вебсайтів) сучасних компаній зростає, застосування сучасної інфраструктури даних та комплексної аналітики в реальному часі стає необхідним.

У порівнянні з традиційними пакетними архітектурами, що не завжди можуть бути цілком достатніми, системи потокової обробки забезпечують кілька важливих переваг.

Головною з таких переваг є здатність працювати з нескінченними потоками подій. Традиційні інструменти пакетної обробки вимагають зупинки потоку подій, збору пакетів даних і об'єднання пакетів для отримання загальних висновків. Потокова обробка, незважаючи на складність об'єднання та захоплення даних із кількох потоків, дозволяє отримати оброблену інформацію з великих обсягів потокових даних. Іншою перевагою потокових систем є можливість обробки в режимі реального часу (або майже в режимі реального часу). Потокова обробка також спрощує виявлення закономірностей у даних часових рядів (виявлення закономірностей у часі, наприклад, пошук тенденцій у даних трафіку веб-сайту, потребує постійної обробки та аналізу даних). Останньою перевагою є легкість масштабування даних (це об'єднує потокову обробку з МСА), адже зростання обсягів даних може порушити роботу системи пакетної обробки, вимагаючи надання додаткових ресурсів або зміни архітектури. Сучасна інфраструктура потокової обробки є гіпермасштабованою, тобто здатною працювати з гігабайтами даних на секунду за допомогою одного потокового процесора. Це дає змогу обробляти зростаючі обсяги даних без змін інфраструктури.

Більшість потокових систем все ще будуються на конвеєрі з відкритим вихідним кодом та на пріоритетних рішеннях для проблем прийому і зберігання даних, обробки потоку та оркестровки завдань. Будь-яка потокова архітектура має включати чотири ключові архітектурні блоки.

Першим з них є брокер повідомлень (The Message Broker). Це елемент, що отримує дані з джерела, яке називається виробником (Producer), перетворює їх у стандартизований формат повідомлення та постійно передає їх потоком, після чого інші компоненти можуть використовувати повідомлення, передані брокером.

Перше покоління брокерів повідомлень, таких як RabbitMQ і Apache ActiveMQ, покладалося на парадигму підпрограмного забезпечення, орієнтованого на обробку повідомлень MOM (Message Oriented Middleware). Пізніше з'явилися надпродуктивні платформи обміну повідомленнями, або потокові процесори (Stream Processor), які більше підходять для потокової парадигми. Найбільш популярними інструментами обробки потоків є Apache Kafka та Amazon Kinesis Data Streams.

На відміну від MOM-брокерів, потокові брокери підтримують дуже високу продуктивність, зберігаючи цілісність даних, мають величезну пропускну здатність трафіку повідомлень (гігабайт за секунду або навіть більше) і зосереджені на потоковій передачі з незначною підтримкою перетворення даних або планування завдань.

Другим блоком є платформи ETL. Потоки даних від одного чи кількох брокерів повідомлень необхідно агрегувати, трансформувати та структурувати, перш ніж дані можна буде проаналізувати за допомогою інструментів аналітики на основі SQL. Це можна зробити за допомогою платформи ETL, яка отримує запити від користувачів, витягує події з черг повідомлень, а потім застосовує запит для генерації результату (у цьому процесі часто виконуються додаткові об'єднання, перетворення або агрегації даних). Результатом може бути виклик API, дія, візуалізація, сповіщення або новий потік даних. Прикладами інструментів ETL з відкритим кодом для потокової передачі даних є Apache Storm, Spark Streaming і WSO2 Stream Processor. Хоча ці структури працюють по-різному, усі вони здатні прослуховувати потоки повідомлень, обробляти дані та зберігати їх у пам'яті. Деякі потокові процесори, включаючи Spark і WSO2, забезпечують синтаксис SQL для запитів і обробки даних.

Після того, як потокові дані підготовлені для використання потоковим процесором, їх необхідно проаналізувати. Існує багато різних інструментів аналізу потокових даних, найпоширенішими з яких є Amazon Athena, Amazon Redshift, Elasticsearch та Cassandra.

З появою відносно недорогих технологій зберігання ВД більшість організацій сьогодні зберігають дані своїх потокових подій. Ці дані можна зберігати або у базах та сховищах даних (наприклад, Postgre SQL або Amazon Redshift), або у брокерах повідомлень (наприклад, за допомогою постійного сховища Kafka) або у озерах даних (наприклад, Amazon S3). Озеро даних є найбільш гнучким і недорогим варіантом для зберігання даних про події, але його створення та обслуговування часто є дуже технічно складним завданням.

Розглянемо приклад організації системи потокової обробки ВД за допомогою мікросервісів на прикладі геоплатформи, запропонованої компанією MTS IT (рис. 3) [4].

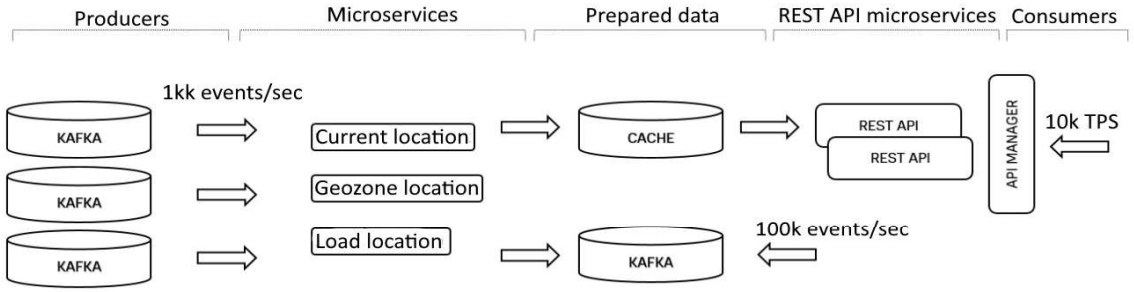

Рис. 3. Схематична взаємодія сервісів у геоплатформі MTS IT

В даній платформі джерелом даних є кластер Kafka, а далі йде набір сервісів, кожен з яких реалізує окремий бізнес-процес (наприклад, сервіс визначення поточного місцезнаходження клієнта, сервіс моніторингу входу клієнтів в дану геозону або визначення завантаженості станцій в реальному часі). Після чого ці дані можна запросити на вимогу через API, які також поділяються на сервіси за бізнес-кейсами, або зчитати з черги повідомлень кластеру Kafka.

Серед переваг такої архітектури можна підкреслити самостійне розгортання та оновлення служб [10]. Так система дозволяє швидко запустити в комерційну експлуатацію нову бізнес-функцію або цілий сервіс, не боячись порушити роботу інших сервісів. Це особливо стосується обробки ВД, адже великий обсяг вхідних даних і їх різноманітність призводять до формування великої кількості нових бізнес-кейсів і частої зміни вже існуючих.

Слід також відзначити і ефективність горизонтального масштабування. Індивідуальне масштабуваня послуг як окремий сервіс є ефективнішим, ніж масштабування послуг як частини моноліту.

Вагомою перевагою є також те, що сервіси можуть бути реалізовані за допомогою різних мов програмування та фреймворків. Це дуже важливо в Big Data, де через велику кількість бізнес-процесів багато різних команд одночасно працюють на одній платформі [11]. З точки зору розробки слід відзначити низький поріг входу для нових розробників, адже маленьку кодову базу легко зрозуміти. Зазвичай, сервіси можна переписати з нуля за кілька SCRUM-спринтів.

Нарешті, останньою вагомою перевагою з точки зору управління проєктами є масштабування команд розробників. Якщо відразу зрозуміло, як розділити систему на мікросервіси, то за наявності ресурсів можна розпаралелити розробку та значно скоротити загальний час впровадження платформи. Але, як і будь-яка інша архітектура, МСА має і свої недоліки, головним з яких є розподілення системи. Систему, що складається з багатьох сервісів, стає важко спроєктувати та розробити (для таких комплексних задач потрібні архітектори та забудовники високої кваліфікації). Слід відзначити ненадійність мережних взаємодій в МСА (сервіси спілкуються один з одним через мережу, а мережеві з'єднання є схильними до збоїв). Останнім суттєвим недоліком використання МСА є складність оперування, адже розподілену систему важче підтримувати.

Порівняємо МСА (рис. 3) з рішенням, реалізованим згідно з концепцією монолітної архітектури (рис. 4). Серед переваг монолітної архітектури над МСА можна зазначити лише швидкість

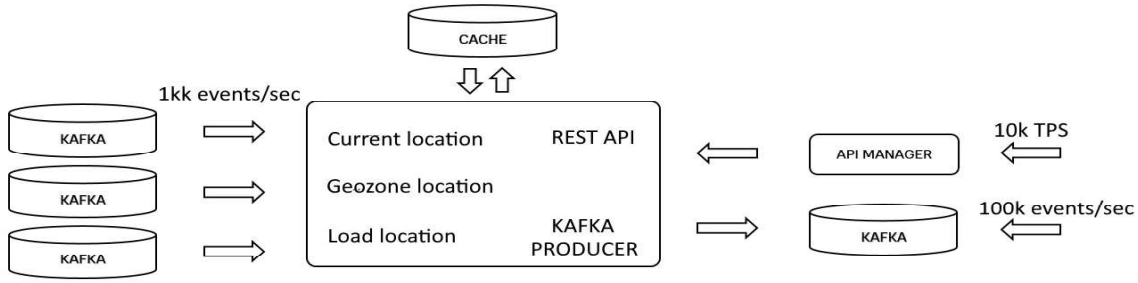

Рис. 4. Схематична взаємодія монолітної геоплатформи MTS IT

розгортання програми та відсутність мережевої взаємодії, що наявна у рішенні з використанням МСА [48], однак кількість недоліків монолітної платформи значно переважає ці два пункти.

Головним з таких недоліків слід вважати знижену загальну надійність (будь-яке доопрацювання одного з модулів може порушити дієздатність цілої низки інших сервісів).

Застосування монолітної архітектури передбачає також сильний зв'язок модулів (часто деякі модулі використовують методи інших модулів у своїй реалізації, тому для модифікації одного модуля потрібно модифікувати і кілька інших, щоб не зламати зворотню сумісність).

Ще одним важливим недоліком використання монолітної архітектури є великий вхідний поріг для нових розробників (розробнику потрібний значний час, щоб зрозуміти чималу кодову базу), що значно ускладнює впровадження нових технологій.

Суттєво різняться і вимоги до інфраструктури МСА та монолітної архітектури для систем потокової обробки ВД. В таблиці 1 наведені оцінки вимог до апаратних ресурсів інфраструктури систем обробки ВД з монолітною та мікросервісною архітектурами. Тут використані позначення: S - small (низькі вимоги), M - middle (середні вимоги), L - large (значні вимоги) для таких типів ресурсів як RAM (об'єм оперативної пам'яті), CPU (ресурси центрального процесора), HDD / SSD (потрібне місце на дисках) та LAN (мережа комунікації).

Таблиця 1

Характеристика вимог до апаратних ресурсів інфраструктури систем обробки великих даних з монолітною та мікросервісною архітектурами

Архітектура	RAM	CPU	HDD/SSD	LAN
Монолітна	M	M	M	S
Мікросервісна	L	L	S	L

3. Пропонований варіант архітектури потокової обробки великих даних з використанням мікросервісів

Для визначення сукупності компонентів, що є доцільним використати в подальшому у варіанті МСА системи потокової обробки ВД, були розглянуті та обрані такі мови програмування і технології: інтерпретована об'єктно-орієнтована мова програмування Python; програмне забезпечення для автоматизації розгортання та керування програмами в середовищах з підтримкою контейнеризації; контейнеризатор додатків Docker; розподілена потокова платформа подій Apache Kafka, об'єктно-реляційна СУБД PostgreSQL; веб-платформа для створення API FastAPI [12].

Наведемо короткий опис цих елементів.

Python є інтерпретованою високорівневою об'єктно-орієнтованою кросплатформеною мовою програмування. На сьогодні ця мова є провідною у сферах штучного інтелекту, машинного навчання та нейромережевого моделювання.

Docker є відкритою платформою для розробки, доставки та запуску програм. Скориставшись методологіями Docker для швидкої доставки, тестування та розгортання коду в середовищі контейнерів, можна значно зменшити затримку між написанням коду та його використанням в МСА.

Apache Kafka є розподіленою потоковою платформою подій із відкритим кодом, що використовується для високопродуктивних конвеєрів даних, потокової аналітики, інтеграції даних і критично важливих програм, та складається з серверів і клієнтів, які спілкуються через високопродуктивний мережевий протокол TCP. Kafka працює як кластер з одного або кількох серверів, які можуть охоплювати кілька центрів обробки даних або хмарних регіонів. Kafka дозволяє створювати розподілені програми та мікросервіси, які зчитують, записують і обробляють потоки подій паралельно у відмовостійкий спосіб.

Postgres (повна назва PostgreSQL) є об'єктно-реляційною СУБД, що характеризується масштабованістю і відповідає стандартам SQL. Його основною функцією є безпечне зберігання даних та подальше їх отримання за запитом інших програм (тих, що знаходяться на тому самому комп'ютері або тих, що працюють на інших комп'ютерах мережі. Postgres може обробляти робочі навантаження від невеликих однопоточних додатків до великих Інтернет-додатків з багатьма одночасними користувачами.

FastAPI є сучасною високопродуктивною веб-платформою для створення API на основі стандартного синтаксису мови Python. Ключовими особливостями цієї платформи є висока продуктивність (це один із найшвидших фреймворків Python); висока швидкість програмування; інтуїтивно зрозумілий синтаксис; мінімалістичність (дублювання коду зведено тут до мінімуму).

Отже, з огляду на всі переваги розглянутих засобів реалізацій, такий технологічний стек дозволяє швидко розробити систему потокової обробки ВД (з урахуванням перспектив подальшого розвитку системи). Усі обрані технології є загально визнаними та широко застосовуваними.

Аналіз різних архітектур ВД, що організовані за допомогою мікросервісів, дозволив розробити прототип системи ВД з використанням розглянутих вище методів і технологій.

Схему взаємодії мікросервісів у запропонованому рішенні наведено на рис. 5.

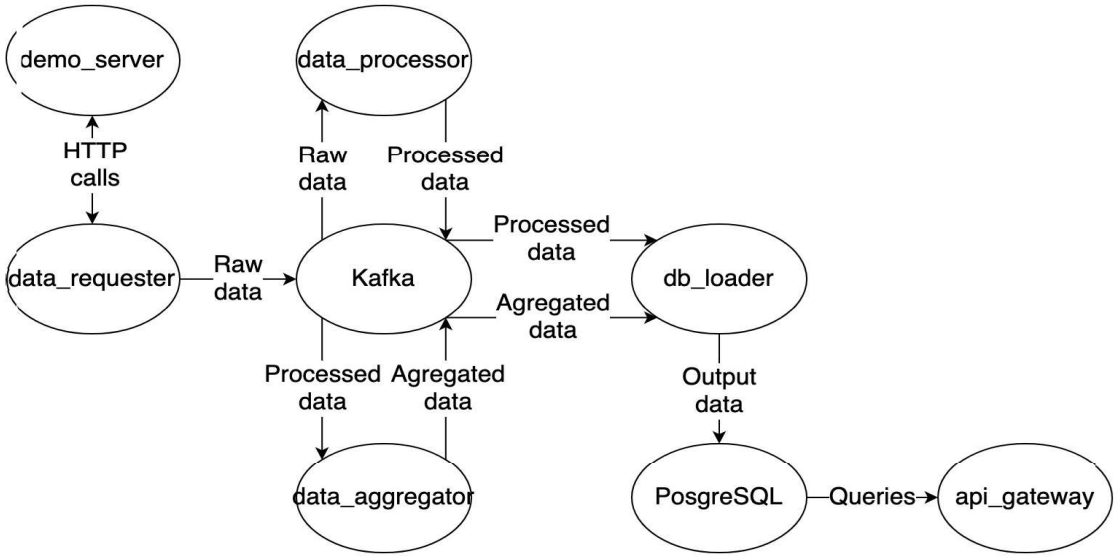

Рис. 5. Схема взаємодії мікросервісів у запропонованому рішенні

Розглянемо працездатність цієї схеми на прикладі її використання для створення системи бізнес-аналізу, яка приймає в потоковому режимі велику кількість записів, що являють собою пару курсів валют, та обчислює середні значення цих пар для деякої кількості запитів. Даний прототип системи розглянуто для перевірки концепції (Proof of Concept) застосування МСА у системах потокової обробки ВД.

Іншим напрямом проведення експериментальних досліджень було порівняння характеристик систем потокової обробки ВД для монолітного та мікросервісного варіантів їх реалізації.

4. Результати розробки та експериментального дослідження пропонованого варіанту МСА для систем потокової обробки великих даних

Для прототипизації мікросервісного застосунку, що працює з ВД згідно з запропонованою схемою взаємодії мікросервісів, було обрано потокову архітектуру.

За допомогою інструментарію Docker було створено систему Docker-контейнерів, кожен з яких відповідає за поставлене йому завдання:

- контейнер infrastructure, що містить базову та запропоновану інфраструктури, а також усі інфраструктурні послуги (такі як Kafka та Postgres);
- контейнер demo_server, що містить демонстраційну серверну програму, що створює випадкові тестові дані, необхідні для демонстрації;
- контейнер data_requester, що виконує мікросервісну програму запиту даних та запитує дані з демонстраційного сервера та надсилає їх до Kafka;
- контейнер data_processor, що відповідає за читання повідомлення, отриманого від сервісу data_requester через Kafka, після чого розділяє дані та завантажує оброблені дані назад у Kafka;
- контейнер data_aggregator, що містить програму агрегатора даних; служба агрегатора даних читає повідомлення, отримані від процесора даних через Kafka, і обчислює середнє значення для останніх 10 відповідей;

- контейнер db_loader, що завантажує БД та службу завантажувача БД, читає повідомлення від сервісів data_processor і data_aggregator і зберігає ці дані в БД Postgres;

- контейнер api_gateway, що надає HTTP-ендпоінти для запиту даних із бази даних Postgres.

Для реалізації процедур створення та запуску зазначених Docker-контейнерів були написані відповідні скрипти та виконавчі файли.

Тестові дані формуються у мікросервісі demo_server, з якого щосекундно дані запитуються мікросервісом data_requester. Сервіс data_requester є першим мікросервісом системи. Після запиту цей сервіс зберігає відповідь у Kafka.

Далі це повідомлення вичитує data_processor, обробляє його та теж зберігає у Kafka. Потім його вичитують сервіси data_aggregator та db_loader. Перший сервіс вираховує середні значення, а другий переміщає до бази всі повідомлення, згенеровані сервісом data_aggregator. Сервіс api_gateway формує запити до БД за вимогою користувача. Весь проєкт та всі додаткові сервіси запускатимуться у сервісі docker-compose.

Для тестування та демонстрації роботи всієї системи розроблено сервіс, який формує значення валютних пар у JSON-форматі з залученням бібліотеки FastAPI (лістинг 1, див. рис. 6).

Дані з сервера запитує мікросервіс api_requester. Під час створення було визначено ім'я програми, що є обов'язковим аргументом. Якщо ми запустимо кілька екземплярів сервісу з однаковим ім'ям, Kafka розподілить партиції між ними, що дозволить масштабувати у систему горизонтально.

```
@router.get("/pairs", tags=["pairs"])
async def get_pairs() -> Dict:
  metrics.GET_PAIRS_COUNT.inc()
  return {
    "USDUAH": round(random.random() * 100, 2),
    "EURUAH": round(random.random() * 100, 2)
  }
```

Рис. 6. Лістинг 1. Ендпоінт для генерації пари двох курсів валют

Сервіс data_requester не читає повідомлень з тем Kafka, але кожну секунду відправляє запит до емулятора даних і обробляє відповідь. Для опису функції, що викликається за таймером із заданим інтервалом, розроблено відповідний сервіс (лістинг 2, див. рис. 7).

```
@app.timer(interval=1.0)
async def request_data() -> None:
  provider = data_provider.DataProvider(
    base_url=config.get(config_loader.BASE_URL))
  pairs = await provider.get_pairs()
  metrics.REQUEShbc/T_CNT.inc()
  logger.info(f"Received new pairs: {pairs}")
  if pairs:
    await src_data_topic.send(key=uuid.uuid1().bytes,
      value=json.dumps(pairs).encode())
```

Рис. 7. Лістинг 2. Циклічна функція, що запрошує дані з data_provider

Періодичність виконання функції контролює декоратор @app.timer. Функція створює екземпляр класу DataProvider, який відповідає за запит даних.

Наступним мікросервісом є data_processor, що займається обробкою пар, отриманих від мікросервісу api_requester. Цей сервіс отримує повідомлення з топіка та обробляє їх.

Для цієї функції використано декоратор @app.agent(src_data_topic), який контролює те, що функція реагує на повідомлення в темі src_data_topic. Повідомлення вичитуються в циклі async for msg_key, msg_value in stream.items(). Далі здійснюються серіалізація отриманого повідомлення та формування валютних пар та їх значень, причому кожна пара записується в наступний топік окремо (лістинг 3, див. рис. 8).

Два попередні сервіси читають та пишуть дані в Kafka потоками. Кожне нове повідомлення вони опрацьовують незалежно від попередніх, але періодично виникає необхідність обробки повідомлення разом із попередніми. Система має порахувати середнє значення для останніх 10 значень пар, що мають бути збережені.

Якщо запустити кілька примірників сервісу агрегацій, то кожен з них зберігатиме локально лише свої значення, що призведе до формування некоректних середніх значень. Тому необхідно реалізувати функцію збереження історії та вирахування середнього значення (лістинг 4, див. рис. 9).

Мікросервіс db_loader прослуховує одразу дві теми - processed_data і data-aggregator-average-changelog. До першої теми пише повідомлення data_processor, а до другої -

```
@app.agent(src_data_topic)
async def on_event(stream) -> None:
  async for msg_key, msg_value in stream.items():
    metrics.SRC_DATA_RECEIVED_CNT.inc()
    logger.info(f"Received new pair message {msg_value}")
    serialized_message = json.loads(msg_value)
    for pair_name, pair_value in serialized_message.items():
      logger.info(f"Extracted pair: {pair_name}: {pair_value}")
      metrics.PROCESSED_PAIRS_CNT.inc()
      await processed_data_topic.send(key=msg_key,
        value=json.dumps({pair_name: pair_value}).encode())
      metrics.PROCESSED_DATA_SENT_CNT.inc()
      yield msg_value
```

Рис. 8. Лістинг 3. Код отримання пар даних

```
@app.agent(processed_data_topic)
async def on_event(stream) -> None:
  async for msg_key, msg_value in stream.items():
    metrics.PROCESSED_DATA_RECEIVED_CNT.inc()
    logger.info(f'Received new processed data message {msg_value}')
    serialized_message = json.loads(msg_value)
    for pair_name, pair_value in serialized_message.items():
      average_value = average_table.get(pair_name, {})
      if average_value:
        average_value['history'].append(pair_value)
        average_value['history'] = average_value['history'][-10:]
        average_value['average'] = round(sum(average_value['history']) / \
          len(average_value['history']), 2)
      else:
        average_value['history'] = [pair_value]
        average_value['average'] = pair_value
      logger.info(f"Aggregated value: {average_value}")
      average_table[pair_name] = average_value
      metrics.PAIRS_AVERAGE_AGGREGATED_CNT.inc()
```

Рис. 9. Лістинг 4. Функція підрахунку середнього значення двох пар

data_aggregator, що викликає необхідність опису двох функцій обробки повідомлень. За аналогією з іншими сервісами, ці повідомлення читаються з Kafka та передаються до БД. Для роботи з базою скористаємося інструментом об'єктно-реляційного відображення ORM (Object-relational mapping) SQLAlchemy.

Для запиту результатів було написано сервіс на FastAPI, який вичитує дані з БД PosgreSQL та повертає їх у JSON форматі (рис. 10).

Для роботи з БД при цьому використовується ORM, як і на попередньому кроці (лістинг 5, див. рис. 11).

Для тестування можливостей запропонованої МСА було здійснено експериментальне порівняння характеристик систем потокової обробки ВД для монолітного та мікросервісного варіантів їх реалізації (на прикладі бізнес-системи, що здійснює управління поставкою продуктів, а також обробку відповідних замовлень і платежів).

Порівняння здійснювалося для двох вебзастосунків з різними архітектурами: перший з монолітною архітектурою (рис. 12), а другий - з МСА (запропонований варіант) системи (рис. 13), яка складається з трьох компонентів: auth (автентифікація), products (продукти) і checkout (оформлення замовлення).

Монолітна архітектура вебзастосунку передбачає, що весь код і функціональність знаходяться в одній програмі або модулі. Всі компоненти та функції, такі як обробка запитів, доступ

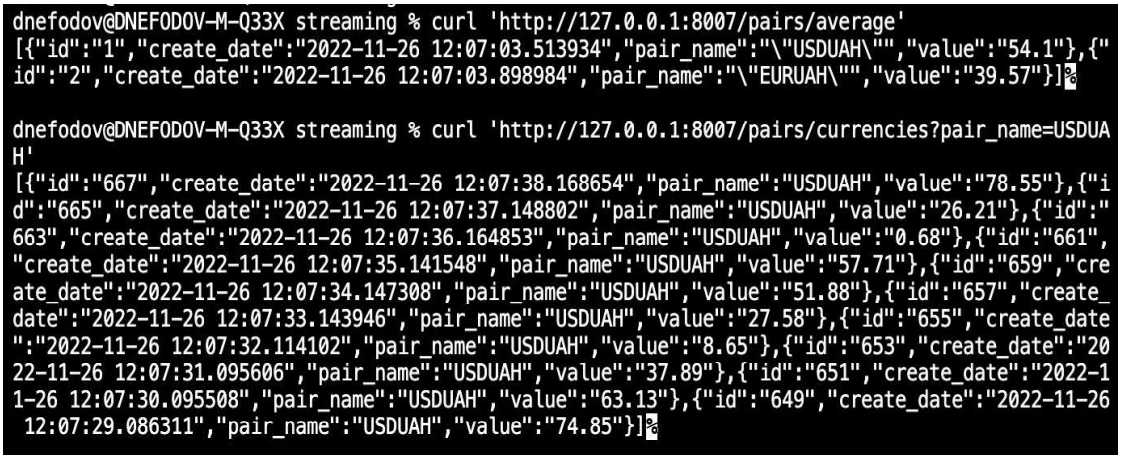

Рис. 10. Фрагмент результатів роботи системи з мікросервісною архітектурою

```
@router.get("/pairs/currencies", tags=["pairs"])
async def get_currencies(pair_name: str, limit: int = 10) -> List[Dict]:
metrics.GET_CURRENCIES_CNT.inc()
  return await db.get_currencies(pair_name, limit)
@router.get("/pairs/average", tags=["pairs"])
async def get_average() -> List[Dict]:
 metrics.GET_AVERAGE_CNT.inc()
 return await db.get_averages()
```

Рис. 11. Лістинг 5. Ендпоінти отримання курсів валют та середнього значення останніх 10 відповідей

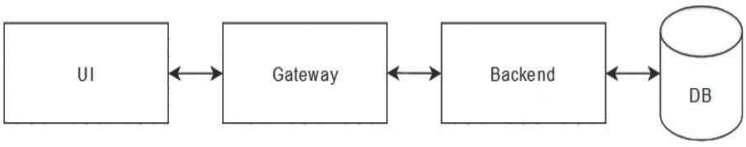

Рис. 12. Структурна схема застосунку з монолітною архітектурою

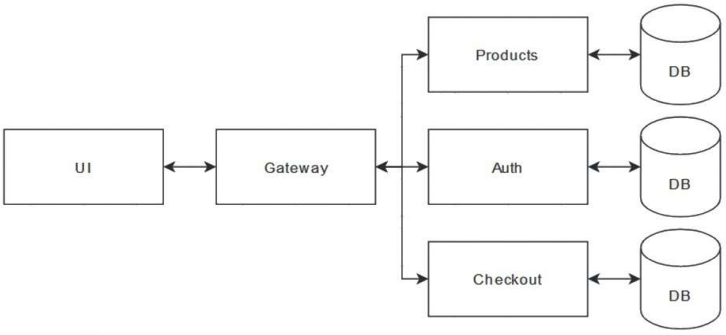

Рис. 13. Структурна схема застосунку з мікросервісною архітектурою

до БД і бізнес-логіка, розташовані в одному монолітному середовищі. Це означає, що всі запити на сервер обробляються одним програмним кодом.

Вебзастосунок з розділеною МСА розбитий на три компоненти: auth, products і checkout. Кожен компонент виконує конкретні функції, пов'язані зі своєю областю відповідальності. Компонент auth відповідає за автентифікацію і авторизацію користувачів, products - за управління продуктами та інформацією про них, а checkout - за обробку замовлень і платежів.

Під час тестування з великою кількістю асинхронних запитів (50 000 та 500 000) до трьох різних варіантів систем (монолітного, мікросервісного з однією реплікою та мікросервісного з двома репліками) були отримані наступні результати щодо середнього часу відповіді.

Для монолітного застосунку:

- при 50 000 запитів середній час відповіді склав 13 мс;
- при 500 000 запитів середній час відповіді збільшився до 110 мс.

Для мікросервісного застосунку з однією реплікою:
- при 50 000 запитів середній час відповіді становив 14 мс;
- при 500 000 запитів середній час відповіді скоротився до 99 мс.
Для мікросервісного застосунку з двома репліками:
- при 50 000 запитів середній час відповіді склав 16 мс;
- при 500 000 запитів середній час відповіді скоротився до 87 мс.
На основі цих результатів можна зробити декілька спостережень.

Монолітний застосунок мав найшвидшу середню відповідь при 50 000 запитів, але при збільшенні кількості запитів до 500 000 його час відповіді значно зріс.

Мікросервісний застосунок з однією реплікою показав хороші показники швидкості відповіді як при 50 000, так і при 500 000 запитів.

Мікросервісний застосунок з двома репліками демонстрував найкращу продуктивність, зменшуючи час відповіді зі зростанням кількості запитів (він дозволяє відстежити середнє значення курсів валют за останні 400 тисяч випадково згенерованих записів в середньому за 30 секунд).

Це свідчить про те, що МСА з розподіленим навантаженням на декілька реплік може забезпечити кращу масштабованість і швидкодію в порівнянні з монолітною архітектурою. Застосування реплік дозволяє розділити навантаження між декількома екземплярами сервісу, що призводить до поліпшення продуктивності та зниження часу відповіді. Слід зазначити, що у варіанті тестування застосунка з однією реплікою використання мікросервісної арзітектури характеризується певними затратами через застосування проксі-серверу для розподілу запитів по мікросервісах.

5. Висновки та перспективи подальших досліджень

На сьогодні не існує універсальних рекомендацій щодо застосування мікросервісного підходу для деяких завдань обробки ВД в інформаційних системах різного функціонального призначення. Зокрема, потребує додаткових досліджень проблема децентралізації потокової обробки ВД, що може вирішуватися із застосуванням мікросервісного підходу. Актуальність використання мікросервісів у ВД пояснюється тим, що на сьогоднішній день існує великий попит на системи, здатні працювати з ВД, а МСА дозволяє отримати більш гнучкі для підтримки довготривалі рішення та спростити управління розробкою.

Аналіз різних архітектур ВД, організованих за допомогою мікросервісів, дозволив запропонувати варіант системи мікросервісної обробки ВД з використанням розподіленої потокової платформи подій Kafka, платформи розробки та запуску програм Docker; об'єктно-реляційної системи управління БД Postgres, а також веб-платформи для створення додатків FastAPI. З огляду на всі можливості розглянутих засобів, такий технологічний стек дозволяє прискорити розробку працездатної та ефективної схеми взаємодії мікросервісів для обробки потоків ВД (з урахуванням перспектив подальшого розвитку системи). Розроблені архітектурні шаблони можуть спростити розробку систем обробки ВД з використанням мікросервісів. Результати моделювання запропонованої МСА доводять її переваги в порівнянні з монолітною архітектурою в системах аналізу обробки потоків ВД. МСА з розподіленим навантаженням на декілька реплік може забезпечити кращу масштабованість і швидкодію в порівнянні з монолітною архітектурою.

Перспективою продовження досліджень, повязаних з розробкою МСА обробки ВД, може бути удосконалення схеми взаємодії компонентів запропонованого варіанту архітектури, поліпшення його експлуатаційних характеристик та розширення функціональних можливостей.

Список літератури: 1. *Efremov A.* Application of microservice architecture in Big Data streaming https://prog.world/application-of-microservice-architecture-in-big-data-streaming (Last accessed: 26.11.2022). 2. *Newman S.* Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly Media, Incorporated, 2019. 272 p. 3. *Zikopoulos P.* Harness the power of big data, the IBM Big data platform. US: McGraw-Hill. 2013. P. 4. *Demchenko Y.* Defining architecture components of the big data ecosystem. In: IEEE Collaboration Technologies and System (CTS). 2014. 104 p. 5. *Стиль* архитектуры для обработки больших данных. 2022. URL: https://learn.microsoft.com/ru-ru/azure/architecture/guide/architecture-styles/big-data (Last accessed: 26.11.2022). 6. *Miliauskas E.* A guide to data warehousing clickstream data [Електронний ресурс] / Evaldas Miliauskas. Stacktome. 2019. Режим доступу до ресурсу: https://stacktome.com/blog/a-guide-to-data-warehousing-clickstream-data. 7. *Miao K., Li J., Hong W.,Chen M. A.* Microservice-

Based Big Data Analysis Platform for Online Educational Applications. 2020. URL: https://www.hindawi.com/journals/sp/2020/6929750 (Last accessed: 26.11.2022). 8. *Levy E.* Streaming Data Architecture in 2022: Components and Examples. 2022. URL: https://www.upsolver.com/blog/streaming-data-architecture-key-components (Last accessed: 26.11.2022). 9. *Tejada Z.* Online analytical processing (OLAP). 2022. URL: https://learn.microsoft.com/en-us/azure/architecture/data-guide/relational-data/online-analytical-processing (Last accessed: 26.11.2022). 10. *Dean Z.* How to Overcome Data Order Issues in Apache Kafka [Електронний ресурс] / Zeke Dean. 2020. Режим доступу до ресурсу: https://www.dataversity.net/how-to-overcome-data-order-issues-in-apache-kafka/. 11. *Soylemez M., Tekinerdogan B., Tarhan A.* Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review. URL: https://doi.org/10.3390/app12115507. (Last accessed: 26.11.2022). 12. *Richards M., Ford N.* Fundamentals of Software Architecture: An Engineering Approach. O'Reilly Media. 2020. 432 p.

Надійшла до редколегії 14.11.2022

Нефьодов Даніїл Андрійович, аспірант кафедри штучного інтелекту ХНУРЕ. Наукові інтереси: розробка архітектури розподілених систем, методи стримінгової класифікації. Адреса: Україна, 61000, Харків, проспект Науки, 14, тел. +38(095)3643243, e-mail: danyil.nefodov@nure.ua, ORCID: 0009-0008-3171-1397

Удовенко Сергій Григорович, доктор технічних наук, професор, завідувач кафедри інформатики та комп'ютерної техніки ХНЕУ ім. С. Кузнеця. Наукові інтереси: інтелектуальні системи керування та обробки інформації. Адреса: Україна, 61000, Харків, проспект Науки, 9-А, тел. +38(067)9098331, e-mail: udovenkosg@gmail.com, ORCID: 0000-0001-5945-8647

Чала Лариса Ернестівна, кандидат технічних наук, доцент, доцент кафедри штучного інтелекту ХНУРЕ. Наукові інтереси: проєктування інформаційних систем, обробка природномовної інформації. Адреса: Україна, 61099, Харків, проспект Науки, 14, тел. +38(068)3511405, e-mail: larysa.chala@nure.ua, ORCID: 0000-0002-9890-4790