

Міністерство освіти і науки України
Державний заклад
«Луганський національний університет імені Тараса Шевченка»

Навчально-науковий інститут математики та інформаційних технологій

Кафедра інформаційних технологій та систем

Чепрасов Віталій Вікторович

**РОЗРОБКА МЕТОДУ АСИНХРОННОЇ ОБРОБКИ ВЕЛИКИХ
ПОТОКІВ ДАНИХ ДЛЯ ПРОЕКТІВ НА SCALA**

кваліфікаційна робота

здобувача вищої освіти другого (магістерського) рівня

освітньої програми «Мультимедійні системи»

за спеціальністю 121 «Інженерія програмного забезпечення»

Особистий підпис _____ Віталій ЧЕПРАСОВ

Науковий керівник _____, Микола СЕМЕНОВ,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

Завідувач кафедри _____ Микола СЕМЕНОВ,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

Полтава – 2025

АНОТАЦІЯ

Тема: Розробка методу асинхронної обробки великих потоків даних для проєктів на Scala.

Спеціальність: 121 «Інженерія програмного забезпечення».

Установа: ЛНУ імені Тараса Шевченка, 2025 р.

Магістерська робота містить: 75 с., 18 рис., 8 табл., 40 джерел.

Об'єкт дослідження – методи асинхронної обробки великих потоків даних.

Предмет дослідження – розробка методів асинхронної обробки великих потоків даних у Scala.

Мета дослідження – розробити та впровадити метод, який покращує продуктивність та масштабованість обробки великих потоків даних у проєктах на базі Scala.

Результати дослідження – було розроблено та впроваджено метод асинхронної обробки потоків великих даних, який поєднує сегментоване розбиття, інкрементальні обчислення та адаптивні моделі виконання. Рішення розв'язує проблеми традиційної обробки, вводячи новий підхід до паралельної обробки потоків даних, покращуючи обчислювальну ефективність та знижуючи затримки.

Ключові слова: АСИНХРОННА ОБРОБКА, ВЕЛИКІ ДАНІ, ОПТИМІЗАЦІЯ, ПОТОК, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ, ІНФРАСТРУКТУРА ДАНИХ, SCALA, BIG DATA.

ANNOTATION

Topic: Development of an Asynchronous Processing Method for Big Data Streams in Scala Projects.

Speciality: 121 "Software engineering".

Institution: Luhansk Taras Shevchenko National University (LTSNU), 2025.

Master's thesis consists of: 75 p., 18 im., 8 tables, 40 sources

Object of the study – methods of asynchronous processing for Big Data streams.

Subject of the study – implementation of asynchronous processing methods for Big Data streams.

Objective of the study – to develop and implement an innovative asynchronous processing method that enhances performance and scalability of Big Data stream processing in Scala-based applications.

Results of the study – an advanced asynchronous processing method for big data streams was developed and implemented. The developed method combines segmented partitioning, incremental computation, and adaptive execution models. The solution addresses challenges of traditional processing by introducing a novel approach to parallel data stream handling, improving computational efficiency and reducing latency.

Keywords: SCALA, ASYNCHRONOUS PROCESSING, BIG DATA, OPTIMIZATION, STREAM, PARALLEL COMPUTING, DATA INFRASTRUCTURE.

ЗМІСТ

ВСТУП	5
РОЗДІЛ 1. ОБРОБКА ВЕЛИКИХ ПОТОКІВ ДАНИХ У SCALA	8
1.1. Великі потоки даних	8
1.2. Основи асинхронної обробки даних	9
1.3. Екосистема Scala в обробці потоків	12
1.4. Теоретичні моделі асинхронної обробки великих потоків даних	17
Висновки до розділу 1	22
РОЗДІЛ 2. ПРАКТИЧНА РЕАЛІЗАЦІЯ МЕТОДУ АСИНХРОННОЇ ОБРОБКИ ВЕЛИКИХ ПОТОКІВ ДАНИХ	23
2.1. Методологія розробки	23
2.2. Імплементация методу	28
2.3. Методи оптимізації продуктивності	52
Висновки до розділу 2	55
РОЗДІЛ 3. ЕКСПЕРИМЕНТАЛЬНА ПЕРЕВІРКА МЕТОДУ	56
3.1. Конфігурація тестового середовища	56
3.2. Показники продуктивності	57
3.3. Компаративний аналіз	61
3.4. Аналіз результатів і перспективи вдосконалення	64
Висновки до розділу 3	67
ВИСНОВКИ	68
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	70

ВСТУП

Сучасний стан галузі обчислювальних технологій вимагає від розробників програмного забезпечення та науковців шукати ефективні способи обробки великих потоків даних (Big Data Streams). Експоненційне зростання обсягів даних, що генеруються інформаційними системами, пристроями Інтернету речей (IoT) та іншими джерелами, вимагає нових підходів до їх обробки. Традиційні моделі обчислень стикаються з обмеженнями масштабованості, продуктивності та ефективності використання ресурсів, що ускладнює їх застосування в умовах реального часу, де швидкість реакції на події є критично важливою [23].

Одним із ключових викликів є необхідність забезпечення високої пропускну здатності систем обробки даних за умови їхньої стійкості до раптових змін інтенсивності потоків. Дані сучасних систем надходять нерівномірно, містять структурні неоднорідності, а також можуть бути географічно розподіленими. Ці фактори породжують додаткові технічні труднощі, пов'язані з синхронізацією, розподілом навантаження та мінімізацією затримок.

Виклики ускладнюються тим, що синхронні методи обробки даних, хоч і залишаються основою для багатьох рішень, мають суттєві обмеження. Вони часто стають слабким місцем у реалізації сценаріїв реального часу, таких як аналіз поведінки користувачів у соціальних мережах або моніторинг роботи критичних інфраструктур, де навіть незначна затримка може призвести до втрати релевантності результатів.

У цьому контексті асинхронні підходи до обробки великих потоків даних постають як альтернатива, яка дозволяє уникнути деяких фундаментальних обмежень синхронної обробки. Використовуючи моделі обчислень, що базуються на концепціях нелінійного виконання задач, паралелізму та реактивного програмування, асинхронні методи демонструють вищу адаптивність до змін інтенсивності потоків даних та оптимальне використання обчислювальних ресурсів [14]. Мова програмування Scala, з її

потужними засобами функціонального програмування та підтримкою бібліотек для реактивного програмування, таких як Akka Streams та FS2, є одним із інструментів для створення ефективних рішень в цій сфері.

Об’єкт дослідження – методи асинхронної обробки великих потоків даних у проєктах на Scala.

Предмет дослідження – розробка методів асинхронної обробки великих потоків даних у Scala.

Мета дослідження – розробити та впровадити метод, який покращує продуктивність та масштабованість обробки великих потоків даних у проєктах на базі Scala.

Досягнення поставленої мети обумовлюється виконанням таких завдань:

- 1) проаналізувати сучасні підходи до обробки потоків великих даних, включаючи традиційні синхронні та асинхронні методи;
- 2) розробити концептуальну модель методу асинхронної обробки великих потоків даних для Scala;
- 3) реалізувати прототип методу для обробки великих потоків даних на основі Scala з використанням асинхронних механізмів, таких як неблокуюче I/O та акторна модель;
- 4) провести тестування методу SASP на реальних сценаріях, включаючи обробку даних IoT, оновлення моделей машинного навчання та обробку даних на основі подій;
- 5) виконати порівняльний аналіз продуктивності SASP з існуючими синхронними методами та альтернативними підходами, оцінюючи затримки, пропускну здатність та ефективність використання ресурсів.

Наукова новизна дослідження полягає в розробці нового методу, який забезпечує асинхронну обробку потоків великих даних шляхом сегментованого поділу даних, інкрементального обчислення та адаптивного управління ресурсами. Вперше реалізовано механізм динамічного розподілу

сегментів із контекстуально залежним виконанням, що мінімізує затримки, оптимізує використання ресурсів і забезпечує стійкість системи до збоїв. Експериментально доведено переваги методу SASP над традиційними підходами на прикладі реальних сценаріїв обробки даних.

Методи дослідження: аналіз – для вивчення теоретичних засад асинхронного програмування, синтез – для розробки архітектури методу; вимірювання – для визначення продуктивності методу за допомогою таких метрик, як затримка, пропускна здатність та використання ресурсів; експеримент – для тестування методу на реальних даних і підтвердження його ефективності; порівняння – для оцінки ефективності запропонованого методу порівняно з традиційними підходами.

У першому розділі здійснено огляд сучасних підходів до обробки великих потоків даних. Розглянуто теоретичні основи асинхронного програмування, такі як акторна модель, неблокуюче I/O, механізми зворотного тиску та реактивне програмування. Проведено аналіз існуючих платформ і фреймворків для обробки потоків у Scala, зокрема Akka Streams і Spark Streaming, їхніх можливостей та обмежень.

У другому розділі описано розробку архітектури методу «Segmented Asynchronous Shard Processing (SASP)» та описано його основні компоненти, зокрема динамічне управління сегментами, інкрементальні обчислення, адаптивне виконання та контекстуально залежні сегментів. Розроблено алгоритми для реалізації кожного етапу обробки, включаючи динамічну алокацію ресурсів, моніторинг продуктивності та управління станом.

У третьому розділі проведено тестування розробленого методу. Представлено методику експерименту, описано середовище тестування, характеристики даних і використані метрики оцінки. Виконано порівняльний аналіз продуктивності SASP з існуючими методами, продемонстровано переваги запропонованого підходу та проаналізовано обмеження.

РОЗДІЛ 1

ОБРОБКА ВЕЛИКИХ ПОТОКІВ ДАНИХ У SCALA

1.1. Великі потоки даних

Великі потоки даних (англ. Big Data Streams) – це безперервні потоки інформації, що надходять із різних джерел у режимі реального часу або близькому до нього [30]. Основною особливістю таких потоків є їх динамічність: дані генеруються з високою швидкістю, мають різноманітні структури та зазвичай вимагають обробки в міру надходження. Ця характеристика, яка відома під назвою *velocity* в моделі "3V" (*volume, velocity, variety*), відрізняє потоки даних від статичних наборів даних.

Потоки даних можуть бути структурованими (наприклад, лог-файли з вебсерверів), напівструктурованими (JSON, XML) або неструктурованими (відео, аудіо, текстові повідомлення). Типові джерела великих потоків даних включають Інтернет речей (IoT), соціальні мережі, фінансові системи, мобільні пристрої та телекомунікаційні мережі. Однією з ключових властивостей потоків даних є їх непередбачуваність: інтенсивність та обсяги потоку можуть різко змінюватися, що створює додаткові виклики для систем їхньої обробки [26].

Одним із головних викликів обробки великих потоків даних є необхідність забезпечення низької затримки (*latency*). У багатьох випадках навіть незначне сповільнення обробки може призвести до втрати актуальності отриманих результатів, що є критичним у таких областях, як системи прогнозування погоди, аналіз біржових операцій чи моніторинг кібербезпеки. Другою важливою проблемою є масштабованість (*scalability*): обробка потоків вимагає від систем здатності обробляти зростаючі обсяги даних без втрати продуктивності. Третій виклик — гетерогенність даних. Дані з різних джерел можуть мати різний формат, структуру та вимоги до обробки. Це ускладнює їх інтеграцію в єдину систему. Четвертою проблемою є стійкість до збоїв (*fault tolerance*). Через високі обсяги та швидкість даних будь-які збої в системі можуть призвести до втрати значних обсягів інформації.

Складність управління обчислювальними ресурсами для оптимізації використання CPU, пам'яті та дискової підсистеми також створює додаткові труднощі. Розподілені системи обробки потоків, що працюють у кластерному середовищі, стикаються з проблемами синхронізації між вузлами та мінімізації накладних витрат на передачу даних [10, 29].

Сучасні підходи до обробки великих потоків даних базуються на двох основних парадигмах: синхронна обробка та асинхронна обробка.

Синхронна обробка, яка реалізована в традиційних обчислювальних моделях, передбачає послідовну обробку даних із обов'язковим завершенням одного етапу перед початком іншого. Основним її недоліком є висока затримка та низька адаптивність до змін у потоках. Наприклад, MapReduce – класичний приклад синхронної обробки – є доцільним для пакетної обробки, але не для динамічних потоків даних.

Асинхронна обробка пропонує більш гнучкий підхід, використовуючи реактивне програмування та паралелізм. Реактивна парадигма (Reactive Streams) дозволяє системам адаптуватися до змін інтенсивності потоку та мінімізувати затримки завдяки управлінню «зворотним тиском» (backpressure). Її реалізації, такі як Akka Streams та Apache Flink, є прикладами успішного використання асинхронної обробки для забезпечення високої продуктивності, масштабованості та стійкості до збоїв [17, 18, 22].

Інша важлива концепція – обробка на основі подій (event-driven processing), де система реагує на події у міру їхнього надходження, замість обробки даних у фіксованих пакетах. Цей підхід активно використовується у фінансових технологіях, ігровій індустрії та системах управління IoT [11, 35].

1.2. Основи асинхронної обробки даних

Асинхронні обчислення є фундаментальною концепцією сучасних розподілених систем та обробки потоків даних, які мають справу зі значними обсягами інформації. Асинхронність у цьому контексті визначається як здатність системи виконувати операції незалежно від завершення інших, що

дозволяє уникати блокувань, підвищувати продуктивність і ефективніше використовувати обчислювальні ресурси.

Основна ідея асинхронних обчислень полягає у відокремленні запитів та обробки, що дозволяє системі працювати над іншими задачами під час очікування результатів. У термінах теорії обчислень, асинхронні процеси моделюються за допомогою недетермінованих автоматів, що забезпечує можливість реалізації конкурентного виконання та високої масштабованості.

Асинхронні обчислення є основою для створення реактивних систем, які характеризуються високою гнучкістю, стійкістю до збоїв, адаптивністю до змін навантаження та здатністю обробляти дані у режимі реального часу. Ці системи широко застосовуються в таких областях, як фінансові технології, стрімінгові платформи, управління транспортними мережами та Інтернет речей (IoT).

Модель акторів (Actor Model), запропонована Карлом Г'юїтом у 1973 році, є однією з ключових концепцій асинхронних обчислень. Вона базується на ідеї незалежних «акторів», які комунікують між собою шляхом передачі повідомлень. Кожен актор має свій стан і обробляє повідомлення послідовно, уникаючи необхідності синхронізації між потоками. Ця модель забезпечує природну підтримку паралелізму та стійкості до збоїв, оскільки відмова одного актора не впливає на роботу інших.

Scala, завдяки бібліотеці Akka, є однією з найпоширеніших мов програмування для реалізації акторної моделі. Akka дозволяє створювати системи, які можуть масштабуватися горизонтально, працюючи на тисячах вузлів у кластері. Крім того, вона підтримує механізми відновлення після збоїв, гарантуючи стабільність навіть у складних середовищах.

Реактивне програмування, тісно пов'язане з моделлю акторів, зосереджується на управлінні потоками даних і подій у динамічних системах. Його основна концепція полягає в тому, що обчислення є реакцією на події, які виникають у потоці. Реактивні бібліотеки, такі як Akka Streams та FS2 у

Scala, використовують механізм зворотного тиску (backpressure), що дозволяє балансувати навантаження між постачальниками та споживачами даних.

Важливими аспектами асинхронних обчислень є **паралелізм** і **конкурентність**, але ці поняття мають суттєві відмінності (див. рис. 1.1). Паралелізм передбачає виконання кількох завдань одночасно на різних ядрах або вузлах, тоді як конкурентність зосереджується на ефективному управлінні декількома задачами, які можуть взаємодіяти між собою [14].

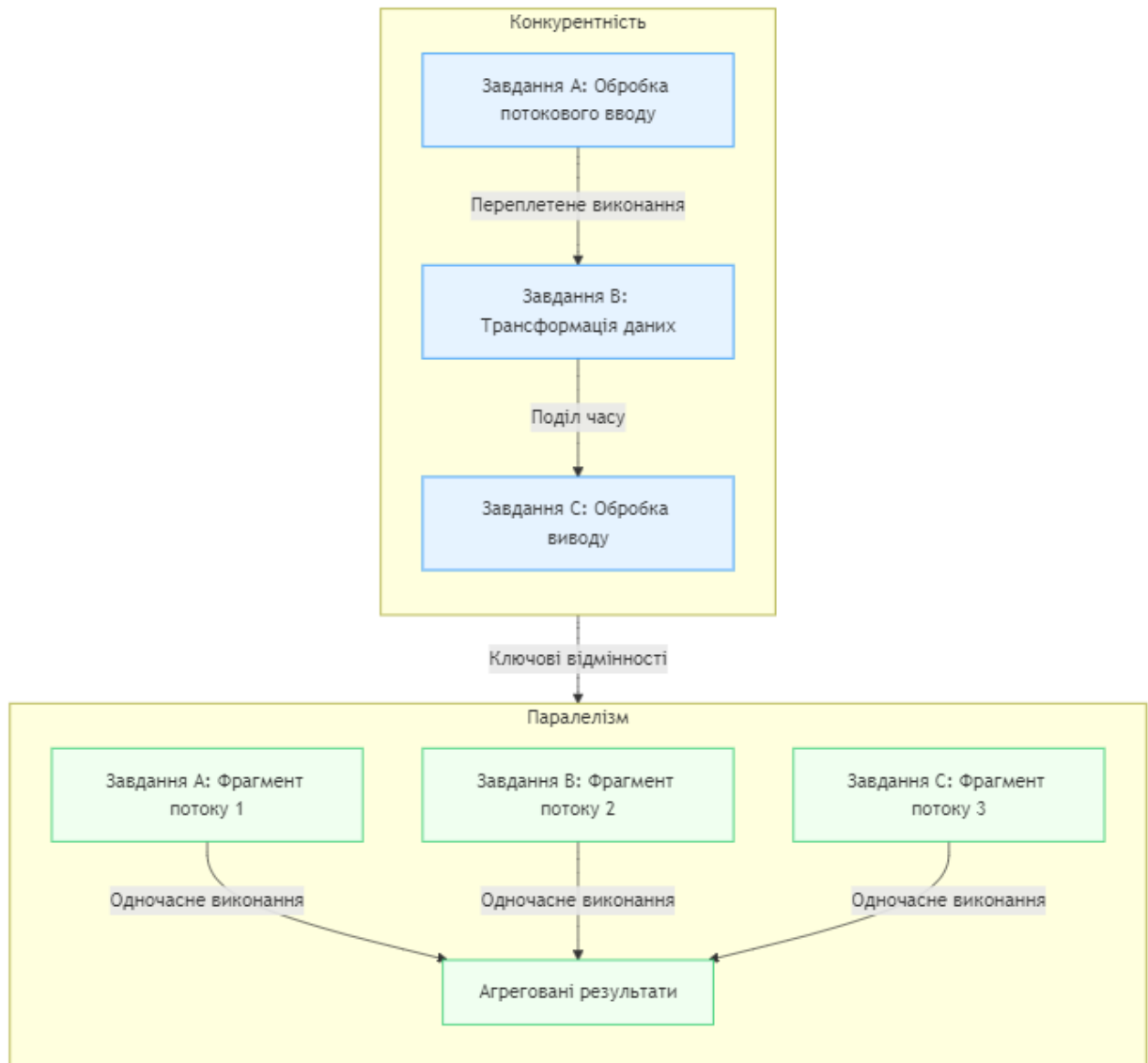


Рис. 1.1 – Різниця між конкурентністю та паралелізмом у контексті потокової обробки

Асинхронні обчислення у мові Scala підтримують ці обидва аспекти завдяки використанню механізмів Future, Promise та Execution Context. Ці інструменти дозволяють розробникам визначати задачі, які виконуються

незалежно, та управляти їхнім станом у реальному часі. Наприклад, бібліотека Cats Effect забезпечує високий рівень абстракції для управління асинхронними процесами у функціональному стилі.

Паралельна обробка даних, що інтегрує ці концепції, дозволяє ефективно працювати із великими обсягами інформації. Це досягається шляхом розподілу завдань між різними вузлами системи та використання потокового процесингу, який дозволяє аналізувати дані у міру їх надходження.

1.3. Екосистема Scala в обробці потоків

Scala як мова програмування, що об'єднує об'єктно-орієнтовані та функціональні підходи, є однією з провідних платформ для обробки великих потоків даних. Її елегантна синтаксична структура, потужна стандартна бібліотека та підтримка паралельних і асинхронних обчислень забезпечують високу ефективність і гнучкість у реалізації складних обчислювальних задач. Крім того, екосистема Scala пропонує широкий набір інструментів і бібліотек для реалізації потокової обробки, таких як Akka Streams, Apache Flink і FS2.

Функціональне програмування є ключовою особливістю Scala, яка відкриває нові горизонти у створенні ефективних і надійних систем потокової обробки. Його принципи базуються на використанні чистих функцій, імутабельності даних і відсутності побічних ефектів, що сприяє більш передбачуваній поведінці програм і полегшує паралелізацію та масштабування.

- 1. Чисті функції та композиційність.** У Scala функції є так званими «громадянами першого класу» (*first-class citizens*), що дозволяє використовувати їх як аргументи, повертати з інших функцій або зберігати у структурах даних. Це забезпечує високу композиційність: складні обчислення можна розбивати на дрібніші модулі, які легко поєднувати. У потоковій обробці це дозволяє створювати конвеєри обробки даних, де кожен етап є окремою функцією.

2. **Імутабельність даних.** Функціональне програмування в Scala активно пропагує використання незмінних (імутабельних) структур даних. У контексті потокової обробки це зменшує ризик виникнення помилок через зміну стану і полегшує реалізацію конкурентних і паралельних обчислень.
3. **«Лінивість» (латентність).** Scala підтримує механізми так званої лінивої, або латентної обробки даних через структури, такі як *Stream* або функції *lazy*. Це дозволяє обробляти лише ті елементи потоку, які потрібні в конкретний момент, оптимізуючи споживання пам'яті та зменшуючи час обчислень. Лінівні колекції дозволяють реалізувати великі конвеєри обробки даних, які можуть виконуватись по мірі надходження нових даних.
4. **Паралелізм та асинхронність.** Scala надає потужні інструменти для управління асинхронними обчисленнями, такі як *Future* та *Promise*, які інтегруються з функціональними бібліотеками. Це дозволяє виконувати обробку даних паралельно, що є критично важливим для масштабованих поточкових систем.
5. **Типова система та монади.** Scala має потужну типову систему, яка дозволяє розробникам визначати й забезпечувати коректність обробки на етапі компіляції. Монади, такі як *Option*, *Either* та *IO*, використовуються для обробки помилок і роботи з побічними ефектами, що спрощує реалізацію складних поточкових сценаріїв.

Функціональне програмування у Scala також тісно інтегроване з реактивними бібліотеками, такими як FS2 (Functional Streams for Scala), яка дозволяє створювати поточкові обчислення, використовуючи функціональні абстракції.

Широкий вибір фреймворків в екосистемі Scala які дозволяє досягти високої продуктивності та гнучкості при роботі з великими потоками даних. Кожен із них має власні переваги, орієнтовані на різні аспекти обробки потоків, такі як паралелізм, асинхронність, обробка в режимі реального часу

чи інтеграція з розподіленими системами. Нижче наведено огляд ключових фреймворків, що використовуються для реалізації потокової обробки на Scala.

Akka Streams є частиною екосистеми Akka і реалізує реактивну програмну модель, сумісну з принципами Reactive Streams. Цей фреймворк надає потужні засоби для створення потоків, керування зворотним тиском (backpressure) та адаптації до змін інтенсивності потоку. Його ключові можливості:

- підтримка декларативного стилю програмування, де потоки описуються у вигляді графів;
- інтеграція з Akka Actors, що дозволяє легко масштабувати систему та забезпечує високу стійкість до збоїв;
- підтримка паралельної обробки через розподіл потоків на декілька вузлів.

Реактивне програмування, яке лежить в основі Akka Streams, дозволяє розробникам описувати потоки даних як графічні структури. Це декларативне програмування спрощує процес створення складних поточкових застосунків. Розробники можуть визначити джерела даних, перетворення та операції збору результатів, не занурюючись у складності низькорівневого керування потоками. Використання Akka Streams є доцільним у сценаріях, що потребують стійкості до збоїв і роботи в розподіленому середовищі, наприклад, у фінансових системах чи IoT-платформах.

FS2 (Functional Streams for Scala) – це функціональний фреймворк для роботи з потоками, побудований на базі Cats Effect, що реалізує концепції чистого функціонального програмування. Перевагами FS2 є [25, 28]:

- повна типобезпека: кожен потік визначається як функція, що приймає і повертає строго типізовані дані;
- «лінива» обробка даних, яка дозволяє оптимально використовувати ресурси;
- зручність інтеграції з іншими бібліотеками в екосистемі Cats, що забезпечує високу модульність і простоту тестування.

FS2 показує найвищу ефективність у проєктах, де потрібна висока передбачуваність обробки та елегантний код [15]. Крім того, FS2 можна використовувати для створення мікросервісів, які необхідно інтегрувати з іншими системами, а також для розробки розподілених систем обробки даних.

Apache Flink – це фреймворк для обробки потоків даних у розподілених системах, який підтримує Scala як одну з основних мов програмування [27]. Хоча Flink часто асоціюється з Java, його інтеграція зі Scala дозволяє використовувати функціональні парадигми для створення масштабованих систем обробки в реальному часі. Його ключові особливості:

- підтримка як потокової, так і пакетної обробки;
- можливість виконання обчислень у режимі реального часу із мінімальною затримкою;
- висока стійкість до збоїв через механізми відновлення з контрольних точок (*checkpoints*).

Flink є доцільним для роботи у великих кластерах і використовується в таких галузях, як телекомунікації, обробка клієнтських даних та моніторинг систем.

Apache Kafka Streams є бібліотекою, створеною для обробки даних безпосередньо в межах екосистеми Kafka. Для Scala розробники можуть використовувати Kafka Streams API, яка забезпечує простий спосіб обробки повідомлень із черг Kafka. Основні можливості включають [6]:

- легкість інтеграції з існуючими потоками Kafka;
- підтримка збереження стану для складних обчислень;
- можливість паралельної обробки у реальному часі.

Kafka Streams часто використовується у проєктах, де Kafka виконує роль основного засобу транспортування даних.

Monix – це високооптимізований фреймворк для реактивного програмування у Scala, який надає розробникам інструментарій для створення високопродуктивних та масштабованих асинхронних додатків. Спираючись на фундаментальні концепції *observable* (спостережувані об'єкти) та *task*

(завдання), Monix дозволяє ефективно працювати з асинхронними потоками даних, забезпечуючи при цьому високу пропускну здатність та реактивність. Крім того, фреймворк пропонує низку операторів для трансформації та об'єднання потоків, що дозволяє будувати складні асинхронні конвеєри. Його особливості [32]:

- потужні інструменти для асинхронної обробки потоків;
- механізм зворотного тиску для управління навантаженням у потоках;
- інтеграція з іншими бібліотеками в екосистемі Scala, такими як Cats та Akka.

Функціональні підходи, що застосовуються в Scala, пропонують можливості для створення високопродуктивних і стійких поточкових систем. Зокрема, акцент на чистих функціях та імутабельності даних дозволяє значно знизити ризики помилок, пов'язаних з маніпуляцією станом, і полегшує інтеграцію паралельних обчислень у середовищах з великими потоками даних. Враховуючи це, важливим аспектом є також використання моделей асинхронного програмування.

Вибір інструментів для потокової обробки даних у Scala визначається специфікою завдань. Важливою особливістю є можливість комбінування різних бібліотек та фреймворків з метою адаптування до конкретних вимог. При розробці додатків для обробки даних в реальному часі, таких як фінансові платформи чи системи Інтернету Речей (IoT), може використовуватися підхід на основі реактивних програмних моделей, переваги якого полягають у здатності адаптуватися до змінної інтенсивності потоків.

Незважаючи на потенціал Scala для потокової обробки, важливим фактором залишається забезпечення узгодженості даних та синхронізації між різними компонентами системи. Одним з найбільших викликів є правильне управління зворотним тиском у розподілених середовищах, де потоки можуть змінювати свою швидкість [34]. Крім того, для досягнення високої продуктивності та ефективності при обробці великих даних необхідно

враховувати фактори масштабованості та оптимізації витрат ресурсів. Для реального впровадження таких систем необхідна глибока інтеграція з інфраструктурними та операційними платформами, що забезпечує коректну синхронізацію і обробку даних у розподілених середовищах.

1.4. Теоретичні моделі асинхронної обробки великих потоків даних

Асинхронна обробка великих потоків даних базується на низці теоретичних моделей. Однією з ключових концепцій, що лежать в основі цих моделей, є подійно-орієнтовані архітектури.

Подійно-орієнтовані архітектури (Event-Driven Architectures, EDA) — це модель обчислень, в якій основною одиницею роботи є подія. Подія визначається як будь-яка зміна стану системи або значуща інформація, яка надходить у систему ззовні або генерується в її межах. Основний принцип EDA полягає в тому, що система реагує на події у реальному часі, обробляючи їх у міру надходження [31].

Схема на рис. 1.2 ілюструє потік подій у типовій подійно-орієнтованій архітектурі.

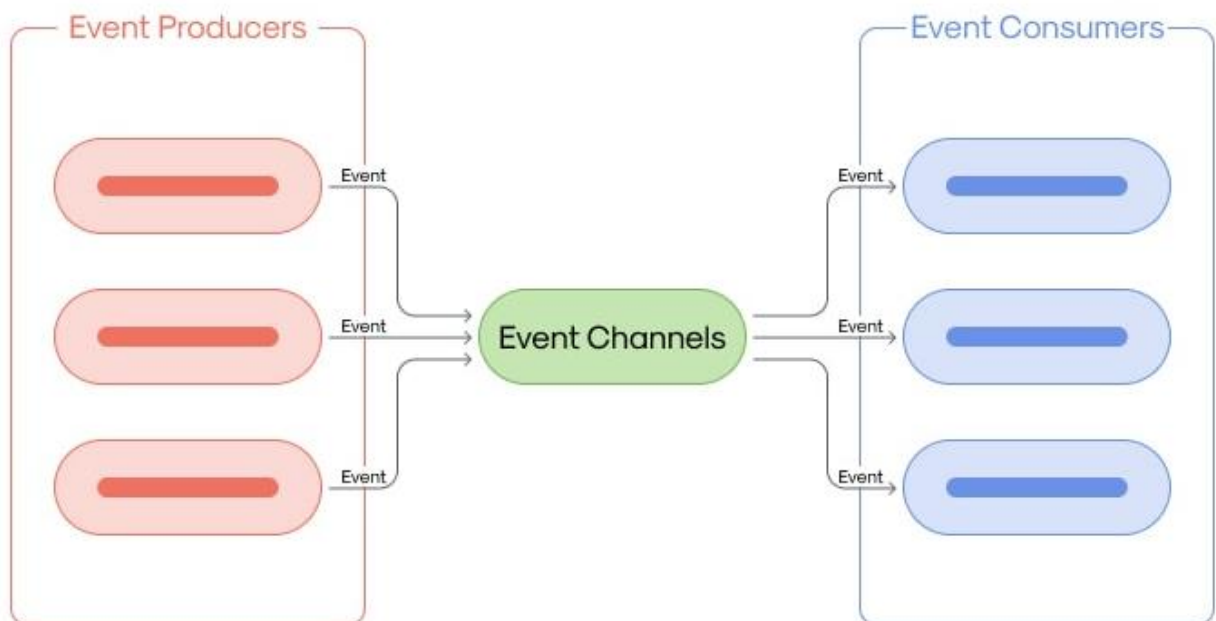


Рис. 1.2 – Подійно-орієнтована архітектура [24]

Подійно-орієнтована архітектура працює за принципом публікації-підписки (publish-subscribe), у якій продуценти подій публікують дані, а споживачі підписуються на певні типи подій і реагують на них у реальному часі. Цей підхід забезпечує низьку затримку та високу продуктивність навіть за умов великої кількості одночасно оброблюваних подій [19]. Реалізація EDA часто базується на чергах повідомлень, таких як Apache Kafka, RabbitMQ чи Pulsar, які забезпечують транспортний шар для подій, дозволяючи масштабувати систему горизонтально.

Ключовою перевагою подійно-орієнтованих архітектур є їхня здатність забезпечувати децентралізованість обчислень. У EDA відсутня необхідність у централізованому координаторі, оскільки кожен компонент системи працює незалежно, реагуючи лише на ті події, які для нього релевантні. Це дозволяє досягати високої стійкості до збоїв, адже вихід з ладу одного компонента не впливає на роботу інших [39].

Системи, побудовані на основі цієї моделі, можуть масштабуватися як горизонтально, шляхом додавання нових вузлів, так і вертикально, шляхом збільшення ресурсів існуючих компонентів. Подійно-орієнтовані архітектури також інтегруються з іншими теоретичними моделями, такими як реактивне програмування та модель акторів, що дозволяє створювати більш складні системи.

У середовищі Scala фреймворки Akka Streams та Kafka Streams надають інструменти для реалізації подійно-орієнтованих архітектур із використанням типобезпечного функціонального підходу. Це забезпечує більш високу продуктивність обчислень і мінімізацію часу розробки завдяки ефективним абстракціям.

Одним із принципів асинхронної обробки є також неблокуючий ввід-вивід (Non-blocking I/O), який дозволяє створювати високопродуктивні системи, здатні працювати з великими потоками даних без затримок, пов'язаних із очікуванням завершення операцій вводу або виводу [40]. Цей підхід протиставляється традиційному блокуючому вводу/виводу, де потік або

процес «зупиняється» до завершення операції, що призводить до втрати часу та ресурсів.

Неблокуючий I/O базується на концепції обробки операцій у фоновому режимі, без блокування основного потоку виконання. У цьому підході виклик функції вводу/виводу негайно повертає керування програмі, а результат операції стає доступним через механізми зворотного виклику (*callback*), майбутніх результатів (*Future/Promise*) або подій. Такий підхід дозволяє ефективно використовувати обчислювальні ресурси, оскільки основний потік виконує інші задачі, поки операція вводу/виводу не завершиться.

Основні переваги неблокуючого I/O включають збільшення пропускної здатності та зменшення затримок у системах, які обробляють великі обсяги одночасних з'єднань або потоків даних. У розподілених системах це стає критичним для забезпечення швидкого обміну повідомленнями між вузлами, обробки запитів до баз даних чи отримання даних від зовнішніх API.

Технічна реалізація неблокуючого I/O, як правило, базується на системних викликах операційної системи, таких як *epoll* у Linux, *kqueue* у macOS і BSD, або *IOCP* у Windows. Ці механізми дозволяють одній ниті обробляти тисячі одночасних з'єднань через ефективну модель спостереження за станом файлів і сокетів.

У мові Scala неблокуючий I/O підтримується через бібліотеки та фреймворки, які використовують цей принцип для асинхронної обробки:

- **Akka HTTP** – забезпечує неблокуючий ввід/вивід для створення високопродуктивних вебсерверів і клієнтів, інтегруючись із акторною моделлю Akka;
- **Cats Effect з FS2** надає функціональні абстракції для управління неблокуючими операціями, що доцільно для побудови складних потокових обчислень у чистому функціональному стилі [20];
- **Netty** – низькорівневий фреймворк для роботи з мережевими з'єднаннями, який використовується у багатьох високопродуктивних бібліотеках для Scala, таких як Play Framework [33].

Механізм неблокуючого I/O дозволяє реалізувати ефективне управління **зворотним тиском** (backpressure), який є критично важливим механізмом у теоретичних моделях асинхронної обробки великих потоків даних. Він забезпечує стабільність і контроль у системах, де дані надходять з високою швидкістю та обробляються асинхронно.

У контексті потокової обробки даних, постачальник (producer) генерує або надсилає дані, тоді як споживач (consumer) їх обробляє. Коли швидкість надходження даних перевищує швидкість їх обробки, споживач стає вузьким місцем у системі. Без належного управління це призводить до переповнення буферів, зростання затримок або аварійного завершення роботи системи через нестачу ресурсів.

Зворотний тиск вирішує цю проблему, дозволяючи споживачу контролювати обсяг даних, які він отримує від постачальника. Якщо споживач не встигає обробляти дані, він може надіслати сигнал постачальнику про необхідність зменшити швидкість передавання або тимчасово призупинити її.

Зворотний тиск є фундаментальною частиною реактивних потоків (*Reactive Streams*), що визначають стандарт для асинхронної обробки потоків даних із зворотним тиском [13]. Реактивні потоки описують взаємодію між постачальником і споживачем через чотири основні ролі:

1. **Постачальник (Publisher)** — генерує дані та передає їх споживачу.
2. **Споживач (Subscriber)** — приймає дані та обробляє їх.
3. **Підписка (Subscription)** — керує взаємодією між постачальником і споживачем, дозволяючи споживачу запитувати певну кількість даних.
4. **Processor** — проміжний компонент, який діє як і постачальник, і споживач, здійснюючи трансформацію даних.

У реактивному програмуванні зворотний тиск реалізується через механізм, коли споживач чітко запитує у постачальника певну кількість елементів для обробки. Постачальник не надсилає більше даних, ніж запитав споживач, що дозволяє уникнути перевантаження.

У мові Scala зворотний тиск підтримується у кількох популярних фреймворках для потокової обробки. Akka Streams реалізує зворотний тиск автоматично в рамках реактивних потоків. Кожен етап обробки у потоці може сигналізувати попередньому етапу про те, скільки даних він готовий прийняти, забезпечуючи контроль над потоком даних і балансування навантаження. FS2 натомість використовує чисто функціональний підхід до обробки потоків із вбудованою підтримкою зворотного тиску. Потоки у FS2 є «лінівими», що дозволяє споживачу обробляти дані у своєму темпі, не перевантажуючи ресурси. Kafka Streams підтримує зворотний тиск завдяки можливостям Apache Kafka. Споживачі Kafka можуть контролювати швидкість отримання повідомлень через механізм підтверджень (commit).

Застосування механізмів зворотного тиску дозволяє системам потокової обробки підтримувати стабільну продуктивність навіть у разі різких змін інтенсивності потоку даних. Тому вони є невід'ємною частиною асинхронної обробки великих потоків даних.

Висновки до розділу 1

У першому розділі було проведено системний аналіз теоретичних основ асинхронної обробки великих потоків даних на мові програмування Scala та в контексті її екосистеми. Було розглянуто виклики, що виникають під час обробки потоків даних у сучасних обчислювальних системах, та обмеження синхронних методів, які стають вузьким місцем для масштабованих і продуктивних рішень. Асинхронні моделі обробки даних були визначені як ефективна альтернатива, що дозволяє мінімізувати затримки та оптимізувати використання ресурсів.

Детально розглянуто теоретичні моделі асинхронних обчислень, включаючи подійно-орієнтовані архітектури, неблокуючий I/O та механізми зворотного тиску. За підсумками аналізу наукових джерел виявлено, що подійно-орієнтовані архітектури забезпечують високу гнучкість і стійкість до збоїв завдяки реактивному підходу до обробки подій. Неблокуючий I/O дозволяє уникати простоїв у системах, що працюють із великим числом одночасних з'єднань, а механізми зворотного тиску забезпечують динамічне балансування навантаження між постачальниками та споживачами даних, підтримуючи стабільність потокової обробки.

Окрім теоретичних моделей, було проаналізовано екосистему Scala, яка включає фреймворки та бібліотеки для асинхронної та потокової обробки, такі як Akka Streams, FS2, Apache Flink та Kafka Streams. Ці інструменти дозволяють реалізувати складні сценарії обробки даних, поєднуючи функціональне програмування, паралелізм та реактивні підходи. Таким чином, перший розділ закладає основу для розробки методу асинхронної обробки великих потоків даних, реалізація та дослідження якого описується в наступних розділах роботи.

РОЗДІЛ 2

ПРАКТИЧНА РЕАЛІЗАЦІЯ МЕТОДУ АСИНХРОННОЇ ОБРОБКИ ВЕЛИКИХ ПОТОКІВ ДАНИХ

2.1. Методологія розробки

Сучасні вимоги до обробки великих потоків даних у реальному часі зумовлюють необхідність розробки методів, що здатні динамічно адаптуватися до мінливих умов та ефективно використовувати доступні ресурси. Статичні підходи, які передбачають фіксований розподіл завдань і синхронну обробку, часто виявляються недостатньо гнучкими для роботи зі стрімко зростаючими обсягами даних. У цій роботі досліджується авторський метод **Segmented Asynchronous Shard Processing (SASP)**, який пропонує новий підхід до організації асинхронного виконання і базується на сегментуванні даних та динамічному управлінні потоками.

Архітектура **Segmented Asynchronous Shard Processing (SASP)** складається з чотирьох основних компонентів: менеджера сегментів, вузлів обробки, маршрутизатора даних та адаптивного монітора. Наочне представлення архітектури зображено на рис. 2.1.

1. Менеджер сегментів (Shard Manager) – відповідає за створення, розподіл, об'єднання та поділ сегментів даних (або шарів). Цей компонент підтримує глобальний реєстр, що зберігає метадані про всі активні сегменти, їх поточний стан та залежності між ними. Під час надходження нового потоку даних менеджер аналізує сегментаційні правила і визначає, до якого шару віднести нові дані. Динамічний розподіл сегментів відбувається з урахуванням доступних ресурсів та поточного завантаження вузлів обробки.

2. Вузли обробки (Processing Nodes) – незалежні робочі процеси, що виконують обробку сегментів у паралельному та асинхронному режимах. Кожен вузол працює з одним або кількома сегментами, забезпечуючи їх обробку без блокування інших задач. Вузли реалізують логіку обробки, збереження проміжних результатів та оновлення стану сегментів. Використання асинхронних обчислень дозволяє вузлам виконувати операції

введення/виведення без блокування, підвищуючи ефективність обробки потоків даних.

3. Маршрутизатор даних (Data Router) – відповідає за направлення вхідних потоків до відповідних сегментів. Рішення про маршрутизацію приймаються на основі правил сегментації, які можуть включати часові діапазони, ієрархічну структуру або контекстуальні атрибути. Маршрутизатор підтримує як статичне, так і динамічне маршрутизаційне рішення, що дозволяє оперативно перенаправляти потоки у разі зміни умов або появи нових сегментів.

4. Адаптивний монітор (Adaptive Monitor) – здійснює постійний контроль за навантаженням на вузли, розподілом сегментів і загальною продуктивністю системи. Монітор відстежує стан ресурсів, здоров'я вузлів та ефективність обробки. На основі зібраних даних монітор ініціює адаптивні дії, такі як перенесення сегментів між вузлами, об'єднання дрібних сегментів або їх поділ у разі перевантаження.

Архітектура SASP використовує як подійно-орієнтовану модель для обробки нових даних у режимі реального часу, так і періодичну консолідацію для агрегації результатів.

Таким чином, у методі реалізовано динамічний підхід до сегментації та обробки даних на розподілених вузлах. Алгоритм базується на комбінації сегментованого поділу, інкрементальних обчислень та адаптивного виконання. Опис алгоритму можна представити у кілька етапів, які відображають взаємодію між основними компонентами системи.

Етап 1. Сегментація вхідних даних.

1. Надходження даних. Система отримує потік вхідних даних із різних джерел, таких як сенсори IoT, лог-файли або транзакційні записи.

2. Застосування правил сегментації. Менеджер сегментів аналізує вхідні дані та застосовує правила сегментації для поділу даних на логічні сегменти – шари (shards). Правила сегментації можуть базуватися на часових мітках, категоріях або інших атрибутах.

3. Створення або оновлення сегментів. Якщо для певного сегмента ще не створено відповідний шар, менеджер створює новий шар із унікальним ідентифікатором. У разі існування відповідного сегмента, дані додаються до наявного шару.

Етап 2. Динамічний розподіл завдань.

1. Визначення доступних ресурсів. Менеджер сегментів оцінює завантаженість вузлів обробки та визначає, які з них готові приймати нові задачі.

2. Розподіл сегментів. Менеджер передає сегменти вузлам обробки відповідно до їх поточного стану та пріоритетів обробки. Розподіл відбувається динамічно, залежно від поточної доступності ресурсів.

3. Виконання асинхронних завдань. Вузли обробки отримують сегменти та запускають асинхронні задачі для їх обробки. Використовуються механізми Future або Promise для виконання завдань без блокування основного потоку.

Етап 3. Інкрементальна обробка.

1. Обробка змін у сегментах. Вузли обробки аналізують дані у сегментах та виконують інкрементальні обчислення, обробляючи лише нові або змінені дані. Це дозволяє уникнути повторної обробки всієї вибірки.

2. Оновлення стану сегмента. Після обробки кожного сегмента вузол оновлює його стан, зберігаючи проміжні результати, журнали помилок та іншу необхідну інформацію для подальшого використання.

Етап 4. Адаптивне виконання.

1. Моніторинг стану системи. Адаптивний монітор відстежує завантаження вузлів, продуктивність обробки та цілісність сегментів.

2. Динамічна оптимізація. У разі зміни умов, таких як перевантаження вузла або нерівномірний розподіл даних, монітор ініціює дії з оптимізації: об'єднання дрібних сегментів, поділ великих сегментів або їх перенесення на інші вузли.

3. Відновлення після збоїв. У разі збою вузла сегмент із його станом може бути перенесений на інший вузол для відновлення обробки.

Етап 5. Консолідація результатів.

1. Агрегація даних. Після завершення обробки сегментів система періодично виконує консолідацію результатів для створення узагальнених висновків або збереження даних у сховищах.

2. Експорт результатів. Консолідовані результати передаються на зовнішні системи, бази даних або інтерфейси для подальшого аналізу чи візуалізації.

Блок-схема на рис. 2.1 ілюструє послідовність основних етапів обробки великих потоків даних у розподіленій асинхронній системі відповідно до алгоритму SASP. Ключовим принципом розробленого методу є логічне поділення даних на самостійні сегменти, які обробляються незалежно один від одного. Метою такого підходу є мінімізація затримок та підвищення продуктивності системи, оскільки кожен сегмент може виконуватися паралельно на різних вузлах обробки. Кожен шар містить не лише дані, а й власний стан, що включає метадані, історію обробки та проміжні результати. Завдяки цьому досягається ізолюваність обробки, що полегшує управління помилками та забезпечує можливість відновлення після збоїв.

Динамічний розподіл сегментів між вузлами обробки є ключовим етапом алгоритму SASP. Цей процес базується на моніторингу поточного стану ресурсів і застосуванні адаптивних рішень для балансування навантаження. Коли один вузол завершує обробку сегмента, він може одразу взяти в роботу новий сегмент, забезпечуючи безперервність обчислювального процесу.

Важливість інкрементальних обчислень аргументується тим, що, на відміну від традиційних підходів, де весь потік даних обробляється цілком, SASP дозволяє оновлювати лише ті сегменти, які зазнали змін. Оскільки система безперервно аналізує продуктивність вузлів, то у разі виявлення дисбалансу виконуються коригувальні дії. Це може бути об'єднання дрібних сегментів для зниження накладних витрат або поділ великих сегментів для рівномірного розподілу навантаження між вузлами.

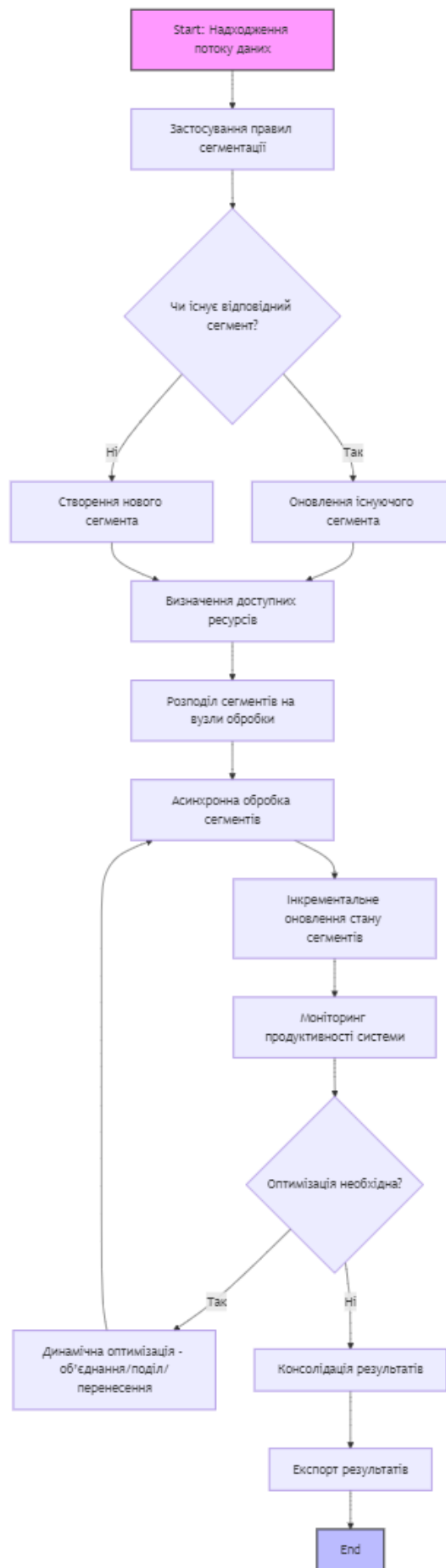


Рис. 2.1 – Алгоритм методу обробки великих потоків даних SASP

2.2. Імплементация методу

Ефективна реалізація методу SASP потребує технологічного стеку, компоненти якого вибрані відповідно до їх здатності підтримувати асинхронні обчислення, динамічний розподіл завдань, інкрементальну обробку та адаптивне виконання. Використання інструментів і бібліотек, сумісних із парадигмами функціонального програмування та асинхронної обробки, є ключовим фактором для забезпечення продуктивності системи [36]. Додатково вибір технологій враховує можливість інтеграції з існуючими системами та здатність до масштабування у розподілених середовищах.

Далі наведено ключові елементи технологічного стеку разом із їхніми відповідними ролями в системі:

1. **Мова програмування Scala** – відповідає обраному предмету дослідження та, завдяки своїй типобезпечності, підтримці «лінивих» обчислень і дієвому механізму обробки потоків даних забезпечує природну підтримку функціонального програмування й масштабованих асинхронних обчислень [37]. Scala дозволяє інтегрувати інструменти для реактивного програмування, такі як Akka Streams та Cats Effect, що є необхідними для побудови архітектури SASP.

2. **Akka** використовується для реалізації моделі акторів та асинхронного виконання завдань. Akka забезпечує створення незалежних акторів, що можуть виконувати обробку сегментів паралельно та обмінюватися повідомленнями. Це дозволяє легко масштабувати систему та гарантує стійкість до збоїв завдяки механізмам відновлення.

3. **Akka Streams** – реалізує потокову обробку із зворотним тиском. Використання Akka Streams дозволяє будувати конвеєри обробки даних, які автоматично адаптуються до змін інтенсивності потоку. Це критично важливо для динамічної обробки сегментів у режимі реального часу, де інтенсивність даних може змінюватися непередбачувано.

4. **Cats Effect** – забезпечує функціональні абстракції для асинхронних обчислень та управління побічними ефектами. Завдяки Cats Effect можна

реалізовувати чисті функціональні операції для обробки даних, що підвищує передбачуваність та тестованість коду. Комбінуючи Cats Effect із бібліотекою FS2, можна створювати потужні функціональні конвеєри для обробки потоків даних.

5. Apache Kafka – забезпечує зберігання та маршрутизацію даних. Kafka виконує роль буфера для вхідних потоків і забезпечує механізм передачі даних між компонентами системи. Кожен сегмент даних може бути представлений як окрема топіка Kafka, що полегшує динамічну маршрутизацію та балансування навантаження.

6. Apache Spark – використовується для аналітичної обробки та консолідації результатів. Spark інтегрується з Kafka для виконання обчислень на поточкових даних у реальному часі та підтримує інкрементальне оновлення результатів [7].

7. Prometheus і Grafana – здійснюють моніторинг та адаптивне управління ресурсами. Prometheus збирає метрики продуктивності вузлів обробки та адаптивного монітора, а Grafana забезпечує візуалізацію цих даних для оперативного прийняття рішень [21, 38].

Реалізація методу SASP зумовлює необхідність формування структурованої архітектури коду та використання релевантних патернів проєктування з метою забезпечення ефективної асинхронної обробки, масштабованості та супроводу системи. Архітектура коду, що реалізує SASP, ґрунтується на принципах модульності, розподілу відповідальностей та застосуванні парадигми реактивного програмування.

Структура коду в SASP є компонентно-орієнтованою, тобто складається з чітко визначених модулів, кожен з яких виконує окремі функції в процесі асинхронної обробки великих потоків даних. Реалізація базується на окремих класах і об'єктах, що взаємодіють між собою через уніфіковані інтерфейси та обмін повідомленнями. Основні компоненти включають **ShardManager**, **ShardExecutor**, **DataRouter** та **AdaptiveMonitor**.

Компонент **ShardManager** відповідає за створення, сегментацію та динамічний розподіл завдань. Його архітектура реалізована у вигляді керівного компонента, який використовує шаблон **Singleton** для збереження глобального реєстру сегментів. Наведений на рис. 2.2 фрагмент коду демонструє реалізацію класу **ShardManager**. Клас **Shard** представляє собою контейнер для даних (**data: List[DataSegment]**) та їхнього стану (**state: ShardState**), визначений унікальним ідентифікатором (**id: String**).

```
case class Shard(id: String, data: List[DataSegment], state: ShardState)

class ShardManager {
  private var shards: Map[String, Shard] = Map()

  // Створення або оновлення сегмента
  def allocateShard(data: List[DataSegment]): Shard = {
    val shardId = generateShardId(data)
    val shard = shards.getOrElseUpdate(shardId, Shard(shardId, List(), ShardState()))
    shard.copy(data = shard.data ++ data)
  }

  // Отримання сегмента за ідентифікатором
  def getShard(id: String): Option[Shard] = shards.get(id)

  // Видалення сегмента після завершення обробки
  def removeShard(id: String): Unit = {
    shards -= id
  }

  // Генерація унікального ідентифікатора для сегмента
  private def generateShardId(data: List[DataSegment]): String = {
    s"${data.head.category}-${data.head.timestamp / 10000}"
  }
}
```

Рис. 2.2 – Управління сегментами даних за допомогою **ShardManager**

ShardManager забезпечує такі операції з сегментами:

1. allocateShard(data: List[DataSegment]). Відповідає за створення або оновлення фрагмента. Він приймає список сегментів даних (**List[DataSegment]**) як аргумент. Спочатку генерується унікальний ідентифікатор фрагмента за допомогою методу **generateShardId()**. Якщо фрагмент з таким ідентифікатором вже існує в кеші (**shards: Map[String, Shard]**), він оновлюється шляхом додавання нових даних до існуючих. У разі відсутності фрагмента створюється новий екземпляр **Shard** з отриманим

ідентифікатором, порожнім списком даних та початковим станом. Для створення нового об'єкта `Shard` з оновленими даними використовується `shard.copy`, що зберігає імутабельність.

2. **`getShard(id: String)`**. Дозволяє отримати сегмент за його ідентифікатором. Він повертає `Option[Shard]`, що дозволяє коректно обробляти ситуацію, коли фрагмент з заданим ідентифікатором не знайдено (в такому випадку повертається `None`).

3. **`removeShard(id: String)`**. Видаляє сегмент з кешу за його ідентифікатором. Він використовується після завершення обробки фрагмента для звільнення пам'яті.

4. **`generateShardId(data: List[DataSegment])`**. Приватний метод, що генерує унікальний ідентифікатор для фрагмента на основі категорії та часу першого елемента даних у списку. Він використовує значення `category` та `timestamp`, розділений на 10000, з першого елемента списку `data`.

Компонент **`ShardExecutor`** забезпечує асинхронну обробку фрагментів даних. Він запускає обробку кожного сегмента даних у фрагменті окремо та оновлює стан фрагмента після обробки кожного сегмента. На рис. 2.3 показана реалізація об'єкту **`ShardExecutor`**, відповідальний за асинхронну обробку сегментів даних (`Shard`). Він містить метод `processShard(shard: Shard): Future[Unit]`, який приймає фрагмент як аргумент і повертає `Future[Unit]`. Це означає, що обробка сегмента відбувається асинхронно, а `Future` дозволяє отримати результат обробки або обробити його після завершення.

```
import scala.concurrent.{Future, ExecutionContext}

object ShardExecutor {
  implicit val ec: ExecutionContext = ExecutionContext.global

  def processShard(shard: Shard): Future[Unit] = Future {
    shard.data.foreach { segment =>
      // Логіка обробки сегмента
      shard.state.updateWith(segment)
    }
    println(s"Shard ${shard.id} processed.")
  }
}
```

Рис. 2.3 – Виконання обробки фрагмента за допомогою **`ShardExecutor`**

Ключові аспекти реалізації:

1. Рядок **«implicit val ec: ExecutionContext = ExecutionContext.global»**. Оголошує імпліцитний параметр **ec** типу **ExecutionContext**. Він необхідний для виконання асинхронних задач за допомогою **Future**. **ExecutionContext.global** створює глобальний пул потоків для виконання асинхронних операцій.
2. **def processShard(shard: Shard)**. Цей метод є основним для **ShardExecutor**. Він приймає фрагмент **shard** для обробки та повертає **Future[Unit]**. Всередині методу використовується **Future { ... }** для запуску асинхронної операції.
3. **shard.data.foreach { segment => ... }**. Ітерація по кожному сегменту даних (**segment**) всередині фрагмента (**shard**). В тілі циклу розміщується логіка обробки окремого сегмента даних.
4. **shard.state.updateWith(segment)**. Оновлює стан фрагмента (**shard.state**) на основі обробки поточного сегмента даних (**segment**).

Компонент **DataRouter** відповідає за маршрутизацію вхідних даних до відповідних сегментів. Він застосовує правила сегментації та викликає метод **allocateShard()** у **ShardManager**. Як показано на рис. 2.4, конструктор класу **DataRouter** приймає екземпляр **ShardManager** як аргумент. Це дозволяє **DataRouter** взаємодіяти з **ShardManager** для створення та отримання фрагментів.

```
class DataRouter(manager: ShardManager) {
  def routeData(data: List[DataSegment]): Map[(String,String), Shard] = {
    data.groupBy(segment => {
      val instant = Instant.ofEpochMilli(segment.timestamp)
      val localDateTime = LocalDateTime.ofInstant(instant, ZoneId.of("UTC"))
      (segment.category, s"${localDateTime.getYear}-${localDateTime.getMonthValue()}-${localDateTime.getDayOfMonth()}")
    }).map { case ((category, date), segments) =>
      ((category, date), manager.allocateShard(segments))
    }
  }
}

object ShardManager {
  private def generateShardId(data: List[DataSegment]): String = {
    val categories = data.map(_.category).distinct.sorted.mkString("-")
    s"${categories}-${data.head.timestamp / (1000 * 60 * 60 * 24)}"
  }
}
```

Рис. 2.4 – Маршрутизація даних до фрагментів за допомогою **DataRouter**

Основним методом для `DataRouter` є `routeData(data: List[DataSegment])`, який приймає список сегментів даних як аргумент та повертає фрагмент (`Shard`), до якого було маршрутизовано ці дані. Вся логіка маршрутизації делегується методу **`allocateShard(data)`** класу `ShardManager`. У такий спосіб `DataRouter` передає отримані дані до `ShardManager`, який вже відповідає за створення або оновлення відповідного фрагмента на основі логіки, що міститься в ньому (генерація `shardId` та додавання даних до існуючого або створення нового `Shard`). У представленому фрагменті логіка маршрутизації є комбінованою: дані спочатку групуються за категорією, а потім за часовим діапазоном.

Таким чином, `DataRouter` виконує роль посередника між вхідними даними та `ShardManager`. Цей підхід дозволяє розділити відповідальності: `DataRouter` відповідає за отримання та передачу даних, а `ShardManager` – за управління фрагментами.

Компонент **`AdaptiveMonitor`** відповідає за моніторинг стану системи та ініціює оптимізацію розподілу сегментів у разі зміни умов навантаження. Він аналізує всі сегменти в системі та визначає, чи потребують вони оптимізації.

```
class AdaptiveMonitor(manager: ShardManager) {
  def monitorAndOptimize(): Unit = {
    // Логіка моніторингу стану сегментів
    manager.getAllShards.foreach { shard =>
      if (shardNeedsOptimization(shard)) {
        optimizeShard(shard)
      }
    }
  }

  private def shardNeedsOptimization(shard: Shard): Boolean = {
    // Умови для оптимізації сегмента
    shard.data.size > 1000
  }

  private def optimizeShard(shard: Shard): Unit = {
    // Логіка оптимізації сегмента (поділ або об'єднання)
    println(s"Shard ${shard.id} optimized.")
  }
}
```

Рис. 2.5 – Адаптивний моніторинг та оптимізація фрагментів

У разі перевищення заданих умов запускається процес оптимізації сегментів. Таким чином, клас **AdaptiveMonitor** забезпечує динамічну адаптацію системи обробки даних до змін у навантаженні або структурі даних шляхом оптимізації фрагментів. Ключові аспекти реалізації класу є такими:

1. monitorAndOptimize(). Цей метод є основним для AdaptiveMonitor. Він періодично викликається для моніторингу стану всіх фрагментів, отриманих за допомогою `manager.getAllShards`, та застосування оптимізації до тих, які цього потребують.

2. shardNeedsOptimization(shard: Shard). Цей приватний метод визначає, чи потребує даний фрагмент оптимізації. Наразі умова оптимізації проста: якщо кількість елементів даних у фрагменті (`shard.data.size`) перевищує 1000, фрагмент вважається таким, що потребує оптимізації. Цю логіку можна змінювати та ускладнювати, додаючи інші критерії, наприклад, розмір даних, частоту доступу до фрагмента, час останньої оптимізації тощо.

3. optimizeShard(shard: Shard). Цей приватний метод виконує оптимізацію заданого сегмента. Наразі реалізація містить лише виведення повідомлення про оптимізацію. В реальній системі буде реалізована конкретна логіка оптимізації, а саме:

3.1. Поділ сегмента. Якщо сегмент занадто великий, він ділиться на кілька менших, що дозволить паралелізувати обробку даних та зменшити час обробки кожного окремого сегмента.

3.2. Об'єднання сегментів. Якщо є багато маленьких сегментів, вони об'єднуються в більші, що може зменшити накладні витрати на управління великою кількістю сегментів.

3.3. Перерозподіл даних. Дані у сегменті можуть бути перерозподілені для покращення локальності даних або балансування навантаження між вузлами кластера.

Таким чином, AdaptiveMonitor забезпечує автоматичне реагування системи на зміни в даних та навантаженні. Важливою частиною є реалізація

методів `shardNeedsOptimization()` та `optimizeShard()`, які повинні враховувати специфіку системи та вимоги до оптимізації.

Діаграма на рис. 2.6 наочно ілюструє послідовність викликів та обміну повідомленнями між основними компонентами системи SASP.

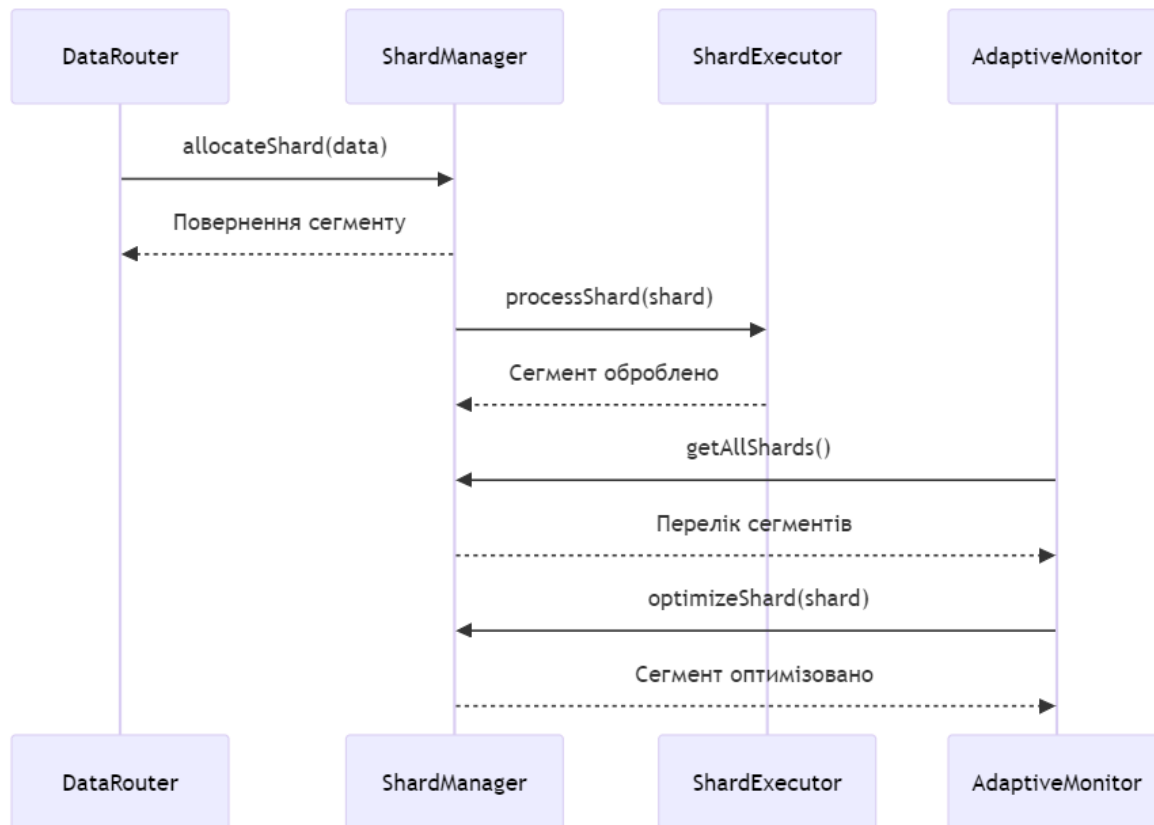


Рис. 2.6 – Взаємодія між компонентами SASP

Ключовим патерном для забезпечення асинхронного та конкурентного виконання в рамках методу SASP є **Actor Model**, який дозволяє розробляти системи, що обробляють великі потоки даних у паралельному та ізольованому середовищі. Реалізація Actor Model у цьому проєкті базується на використанні фреймворку Akka. Основні актори в системі SASP – це **ShardManagerActor**, **ShardExecutorActor** та **AdaptiveMonitorActor**. Кожен актор у системі має свій власний стан і обробляє повідомлення незалежно від інших акторів. По-перше, це забезпечує: ізоляцію стану – зміни у стані одного актора не впливають на інших акторів; конкурентне виконання – декілька акторів можуть одночасно обробляти різні сегменти даних; стійкість до збоїв – у разі

помилки в одному акторі система може автоматично перезапустити його, не впливаючи на роботу інших компонентів.

На рис. 2.7 показано реалізацію актора **ShardManagerActor**, який відповідає за керування фрагментами даних (Shard) та взаємодію з іншими акторами системи за допомогою повідомлень.

```
import akka.actor.{Actor, ActorRef, Props}

case class AllocateShard(data: List[DataSegment])
case class ShardAllocated(shardId: String, shard: Shard)

class ShardManagerActor extends Actor {
  private var shards: Map[String, Shard] = Map()

  def receive: Receive = {
    case AllocateShard(data) =>
      val shardId = generateShardId(data)
      val shard = shards.getOrElseUpdate(shardId, Shard(shardId, List(), ShardState()))
      shards = shards.updated(shardId, shard.copy(data = shard.data ++ data))
      sender() ! ShardAllocated(shardId, shard)
  }

  private def generateShardId(data: List[DataSegment]): String = {
    s"${data.head.category}-${data.head.timestamp / 10000}"
  }
}
```

Рис. 2.7 – Актор керування фрагментами даних

Як видно на рисунку, клас **ShardManagerActor** успадковується від класу **Actor** бібліотеки Akka, що робить його актором у системі. Актор зберігає внутрішній стан у вигляді кешу фрагмента (**shards**), який є мапою, де ключ – це унікальний ідентифікатор фрагмента (**shardId**), а значення – сам об'єкт **Shard**. Метод **receive: Receive = { ... }** визначає поведінку актора, реагуючи на різні типи повідомлень. В даному випадку актор очікує лише повідомлення типу **AllocateShard(data)**, містить список сегментів даних. Коли актор отримує це повідомлення, він виконує такі кроки:

1. Генерує унікальний ідентифікатор фрагмента (**shardId**) за допомогою методу **generateShardId**.
2. Використовує **shards.getOrElseUpdate** для отримання існуючого фрагмента за **shardId**. Якщо фрагмент не знайдено, створюється новий.

3. Оновлює дані фрагмента, додаючи нові сегменти до вже існуючих за допомогою `shard.copy(data = shard.data ++ data)`.
4. Оновлює кеш фрагментів (`shards`) з новим або оновленим фрагментом.
5. Відправляє повідомлення `ShardAllocated(shardId, shard)` відправнику (тому актору, який надіслав запит на виділення фрагмента). Це повідомлення містить ідентифікатор і сам об'єкт фрагмента.

За генерацію унікального ідентифікатора фрагмента відповідає приватний метод **`generateShardId(data: List[DataSegment])`**, на основі категорії та часу першого елемента даних у списку.

Прерогативою іншого актора – `ShardExecutorActor` – є асинхронна обробка сегментів. Кожен актор обробляє окремий сегмент, що забезпечує ізолюваність виконання та можливість паралельної обробки декількох сегментів. На рис. 2.8 продемонстровано реалізацію актора для асинхронної обробки фрагментів даних. Він отримує завдання на обробку через повідомлення, послідовно обробляє кожен сегмент даних фрагмента та оновлює його стан.

```
import akka.actor.{Actor, Props}

case class ProcessShard(shard: Shard)

class ShardExecutorActor extends Actor {
  def receive: Receive = {
    case ProcessShard(shard) =>
      shard.data.foreach(segment => shard.state.updateWith(segment))
      println(s"Shard ${shard.id} processed by ${self.path.name}")
  }
}
```

Рис. 2.8 – Актор обробки сегмента даних

Згідно з наданим фрагментом коду, клас **`ShardExecutorActor`** успадковується від класу **`Actor`** бібліотеки Akka, що робить його актором у системі. Метод **`receive: Receive = { ... }`** визначає поведінку актора і в даному випадку визначає очікування повідомлення типу `ProcessShard(shard)`. Коли актор отримує повідомлення цього типу (воно містить об'єкт фрагмента даних `shard`), то виконує дії у такій послідовності:

1. Ітерація по кожному сегменту даних (segment) всередині фрагмента (shard).
2. Для кожного сегмента викликається метод `shard.state.updateWith(segment)` для оновлення стану фрагмента (`shard.state`). Цей метод реалізований окремо та забезпечує оновлення стану сегмента на основі конкретної логіки обробки даних. Наприклад, якщо сегмент містить дані сенсорів, метод може агрегувати значення, зберігати статистику або логувати помилки.
3. Виводиться повідомлення про завершення обробки фрагмента, вказуючи його ідентифікатор (`shard.id`) та власне ім'я актора (`self.path.name`).

Поточний підхід (виведення результату в консоль) є доцільним, якщо немає необхідності передавати результат обробки назад до **ShardManagerActor** або іншого компонента. Проте, якщо система потребує підтвердження обробки або подальшого аналізу результатів, варто сповістити компонент про завершення обробки, наприклад, за допомогою рядка:

```
manager ! ShardProcessed(shard.id)
```

Для реалізації механізму моніторингу в контексті методу SASP використовується патерн «**Observer**» (укр. «спостерігач»), який дозволяє здійснювати постійне спостереження за станом компонентів системи та автоматично реагувати на зміни завантаження або інші критичні події.

Патерн «Observer» у системі SASP складається з двох основних компонентів: суб'єкт (subject) — компонент, який генерує події та сповіщає про зміни свого стану (у SASP це може бути **ShardManager** або окремі вузли обробки **ShardExecutor**); спостерігач (observer) — компонент, який підписується на оновлення від суб'єкта та виконує певні дії при отриманні сповіщень (у цьому випадку це **AdaptiveMonitor**).

ShardManager виступає суб'єктом і відповідає за відстеження стану сегментів і ресурсів обробки. Він зберігає список підписників (спостерігачів) і повідомляє їх про зміни стану.

На рис. 2.9 наведено код `ShardManager`, який забезпечує механізм для незалежних компонентів (спостерігачів) отримувати інформацію про зміни стану фрагментів, не маючи прямої залежності від внутрішньої реалізації керування фрагментами, що сприяє декомпозиції, повторному використанню коду та більшій гнучкості системи.

```
import akka.actor.{Actor, ActorRef}

case class ShardStateUpdated(shardId: String, newState: ShardState)
case class Subscribe(observer: ActorRef)

class ShardManager extends Actor {
  private var observers: List[ActorRef] = List()

  def receive: Receive = {
    case Subscribe(observer) =>
      | observers = observer :: observers

    case ShardStateUpdated(shardId, newState) =>
      | notifyObservers(shardId, newState)
  }

  // Сповіщення всіх підписаних спостерігачів про зміну стану
  def notifyObservers(shardId: String, newState: ShardState): Unit = {
    | observers.foreach(_ ! ShardStateUpdated(shardId, newState))
  }
}
```

Рис. 2.9 – Реалізація суб'єкту керування фрагментами даних

У наведеному коді клас **`ShardManager`** є актором Akka, що дозволяє йому обробляти повідомлення асинхронно. `ShardManager` зберігає список акторів-спостерігачів (`observers`), які підписалися на отримання сповіщень про зміни стану фрагментів. **`ActorRef`** – це посилання на актора, що дозволяє `ShardManager` надсилати йому повідомлення.

Блок **`case Subscribe(observer) => observers = observer :: observers`** обробляє повідомлення `Subscribe(observer)`, яке надсилається актором-спостерігачем для підписки на оновлення. При отриманні цього повідомлення `ShardManager` додає посилання на актора-спостерігача до списку `observers`. Використання «`::`» забезпечує додавання елемента на початок списку для кращої продуктивності.

У блоці **ShardStateUpdated(shardId, newState) => notifyObservers(shardId, newState)** обробляється повідомлення **ShardStateUpdated(shardId, newState)**, яке сигналізує про зміну стану фрагмента з ідентифікатором **shardId** на **newState**. Після отримання цього повідомлення, **ShardManager** викликає метод **notifyObservers**, щоб сповістити всіх підписаних спостерігачів. Останній виконує розсилання повідомлень **ShardStateUpdated(shardId, newState)** всім акторам-спостерігачам, що знаходяться в списку **observers**. Він ітерується по списку та за допомогою оператора «!» надсилає кожному спостерігачу відповідне повідомлення.

AdaptiveMonitor – виконує роль спостерігача у патерні **Observer**, взаємодіючи з **ShardManager** (суб'єктом). **AdaptiveMonitor** відповідає за моніторинг стану фрагментів даних (**Shard**) та ініціювання адаптивних дій, таких як оптимізація, у разі необхідності. На рис. 2.8 представлений код **AdaptiveMonitor**, який забезпечує зв'язаність між цими компонентами.

```
import akka.actor.{Actor, ActorRef}

class AdaptiveMonitor(shardManager: ActorRef) extends Actor {
  // Підписка на оновлення стану
  override def preStart(): Unit = {
    shardManager ! Subscribe(self)
  }

  def receive: Receive = {
    case ShardStateUpdated(shardId, newState) =>
      monitorShardState(shardId, newState)
  }

  // Логіка моніторингу та адаптивних дій
  def monitorShardState(shardId: String, newState: ShardState): Unit = {
    if (newState.needsOptimization) {
      println(s"Shard $shardId needs optimization. Initiating adaptive action.")
      shardManager ! OptimizeShard(shardId)
    }
  }
}
```

Рис. 2.8 – Реалізація актора **AdaptiveMonitor**

У коді показано, що конструктор класу **AdaptiveMonitor** приймає посилання на актора **ShardManager** (**shardManager: ActorRef**). Це дозволяє

AdaptiveMonitor взаємодіяти з ShardManager для підписки на оновлення та надсилання запитів на оптимізацію. Метод **preStart()** викликається перед запуском актора. В цьому методі AdaptiveMonitor надсилає повідомлення **Subscribe(self)** актору ShardManager, реєструючись як спостерігач. Параметр **self** – це посилання на самого себе (актора AdaptiveMonitor).

AdaptiveMonitor очікує повідомлення **ShardStateUpdated(shardId, newState)**, яке надходить від ShardManager після зміни стану фрагмента. При отриманні цього повідомлення викликається метод **monitorShardState()**. В останньому виконується перевірка, чи потребує фрагмент оптимізації, шляхом виклику методу **newState.needsOptimization()**. Якщо умова виконується, AdaptiveMonitor виводить повідомлення в консоль та надсилає повідомлення **OptimizeShard(shardId)** актору ShardManager для запуску процесу оптимізації.

У прикладі, наведеному на рис. 2.9, показано, як ShardManager сповіщає AdaptiveMonitor про зміну стану сегмента shard-123. AdaptiveMonitor перевіряє стан і ініціює оптимізацію, якщо сегмент перевищує певний поріг оброблених даних.

```
import akka.actor.{ActorSystem, Props}

object SASPSystem extends App {
  val system = ActorSystem("SASPSystem")

  val shardManager = system.actorOf(Props[ShardManager], "shardManager")
  val adaptiveMonitor = system.actorOf(Props(new AdaptiveMonitor(shardManager)), "adaptiveMonitor")

  // Імітація оновлення стану сегмента
  shardManager ! ShardStateUpdated("shard-123", ShardState(processedCount = 1001))
}
```

Рис. 2.9 – Застосування патерну Observer

Особливості використання патерну Observer є наступними:

1. **Слабка зв'язаність.** AdaptiveMonitor не залежить від конкретної реалізації ShardManager. Він лише знає про інтерфейс сповіщень.

2. **Розширюваність.** Можна додавати нових спостерігачів, не змінюючи код `ShardManager`.
3. **Повторне використання.** `AdaptiveMonitor` можна використовувати для моніторингу різних аспектів стану фрагментів, змінюючи лише логіку в методі `monitorShardState`.

Таким чином, застосування патерну `Observer` у проєкті `SASP` дозволяє відстежувати зміни в системі та автоматично ініціювати адаптивні дії. Це забезпечує динамічне управління ресурсами, балансування навантаження та підтримання стабільної продуктивності системи в умовах непередбачуваних змін у потоках даних.

Одним із ключових принципів, який спрощує паралельну обробку даних та забезпечує передбачуваність результатів у багатопотокових або асинхронних системах, таких як `SASP`, є підхід **Immutable Data** (укр. «незмінні дані»). Незмінні дані визначаються як структури, які після створення не можуть бути змінені. Усі операції, що змінюють такі дані, створюють нову копію з урахуванням змін, залишаючи оригінал незмінним. Наприклад, якщо потрібно оновити значення в колекції, `Immutable Data` забезпечує створення нової колекції з новим значенням, не модифікуючи початкову.

`Scala` має підтримку `Immutable Data` через такі структури, як `List`, `Map`, `Set` та `Vector`, які доступні у пакеті `scala.collection.immutable`. Незмінність гарантується на рівні компіляції, що сприяє більшій безпеці коду. Стан кожного сегмента в `SASP` представлений класом `ShardState`, який реалізовано як незмінний. Це ключовий аспект, що забезпечує коректну роботу системи в умовах конкурентного доступу з різних потоків або акторних вузлів. Завдяки незмінності стану, усувається необхідність використання складних механізмів синхронізації, таких як блокування або семафори, що значно зменшує ризик виникнення помилок, пов'язаних з конкурентним доступом, таких як **race conditions** або **deadlocks**.

На рис. 2.10 показано, як метод **updateWith** повертає новий об'єкт **ShardState**, створений за допомогою методу **copy**. Оригінальний стан залишається незмінним.

```
case class ShardState(processedCount: Int = 0, errors: List[String] = List()) {
  def updateWith(segment: DataSegment): ShardState = {
    try {
      // Логіка обробки сегмента (приклад)
      val newValue = segment.value * 2 // Припустимо, що обробка множить значення на 2
      this.copy(processedCount = this.processedCount + 1)
    } catch {
      case e: Exception =>
        this.copy(errors = this.errors ++ e.getMessage)
    }
  }

  def merge(other: ShardState): ShardState = {
    this.copy(processedCount = this.processedCount + other.processedCount, errors = this.errors ++ other.errors)
  }

  override def toString: String = s"processedCount: $processedCount, errors: $errors"
}
```

Рис. 2.10 – Реалізація незмінності стану сегментів за допомогою **ShardState**

Використання **case class** в Scala автоматично генерує метод **copy**, що є ключовим для реалізації незмінності. Також додано значення за замовчуванням для полів. Метод **updateWith** приймає об'єкт **DataSegment** та повертає новий об'єкт **ShardState** з оновленим станом. Оригінальний об'єкт **ShardState** залишається незмінним. У прикладі додано просту логіку обробки сегмента – множення значення на 2 (замість простого збільшення лічильника). Це показує, як можна виконувати обчислення та зберігати результати в новому стані.

Блок **try-catch** обробляє можливі винятки під час обробки сегмента. У разі виникнення помилки, новий об'єкт **ShardState** створюється з доданим повідомленням про помилку до списку **errors**.

Для злиття станів двох фрагментів додано метод **merge(other: ShardState): ShardState**. Він також повертає новий об'єкт **ShardState**, що є результатом об'єднання.

Другим аспектом реалізації **Immutable Data** у **SASP** є обробка даних у паралельному середовищі. Згідно з нею, окрім незмінного стану сегментів (**ShardState**), у **SASP** також дотримується принцип незмінності самих сегментів (**Shard**). Це означає, що після створення об'єкта **Shard** його дані не

можуть бути змінені безпосередньо. Будь-яка операція, що потребує зміни даних або стану сегмента, призводить до створення нового екземпляра **Shard** з відповідними модифікаціями. На рис. 2.11 показано реалізацію **Immutable Shard** (забезпечення незмінності стану сегмента шляхом обробки в паралельному середовищі).

```
case class Shard(id: String, data: List[DataSegment], state: ShardState) {  
  def withUpdatedState(newState: ShardState): Shard = {  
    this.copy(state = newState)  
  }  
  
  def withAddedData(newData: List[DataSegment]): Shard = {  
    this.copy(data = this.data ++ newData)  
  }  
  
  override def toString: String = s"id: $id, data size: ${data.size}, state: $state"  
}
```

Рис. 2.11 – Обробка даних у паралельному середовищі

Так, метод **withUpdatedState(newState: ShardState)** створює новий об'єкт **Shard** з оновленим станом (**newState**), залишаючи оригінальний **Shard** незмінним. Далі створюється новий об'єкт **Shard** в методі **withAddedData(newData: List[DataSegment])** з доданими даними (**newData**), залишаючи оригінальний **Shard** незмінним. Це необхідно, наприклад, при первинному розподілі даних по сегментах або при отриманні додаткових даних для існуючого сегмента.

Прикладом доцільного застосування цього принципу є ситуація, коли актор **ShardManager** розподіляє дані між акторами **ShardExecutor**. Замість того, щоб передавати посилання на змінюваний об'єкт **Shard**, він створює його копію за допомогою **withAddedData** та передає цю копію кожному **ShardExecutor**. Таким чином, кожен **ShardExecutor** працює з власною незалежною копією даних, що повністю виключає можливість конфліктів. Після обробки, **ShardExecutor** може створити новий **Shard** з оновленим станом за допомогою **withUpdatedState** та надіслати його назад до **ShardManager**, який, в свою чергу, збереже цей новий стан.

Ще одним важливим аспектом реалізації Immutable Data є використання незмінних колекцій. У системі SASP для зберігання даних сегментів (**DataSegment**) та станів (**ShardState**) використовуються незмінні колекції, такі як List та Map (або інші незмінні колекції з бібліотек, таких як `scala.collection.immutable`). Це ключове рішення, що забезпечує коректну та передбачувану поведінку системи в умовах конкурентності.

Незмінні колекції мають дистинктивну властивість: будь-яка операція, що змінює колекцію (наприклад, додавання, видалення або оновлення елементів), не модифікує оригінальну колекцію, а створює нову колекцію з внесеними змінами. Оригінальна колекція залишається незмінною протягом усього свого життєвого циклу. У фрагменті коду на рис. 2.12 демонструється використання незмінних колекцій List та Map у Scala, що підкреслює їхню властивість створювати нові колекції при зміні, не модифікуючи оригінальні.

```
case class DataSegment(sensorId: String, timestamp: Long, value: Double)

val initialData: List[DataSegment] = List(
  DataSegment("sensor1", 123456, 10.0),
  DataSegment("sensor2", 123457, 20.0)
)

val updatedData = initialData :+ DataSegment("sensor3", 123458, 30.0) // Створюється новий список

println(s"Initial Data: $initialData") // Залишається незмінним
println(s"Updated Data: $updatedData") // Містить новий елемент

val initialMap: Map[String, Int] = Map("a" -> 1, "b" -> 2)
val updatedMap = initialMap + ("c" -> 3)
println(s"Initial Map: $initialMap") // Залишається незмінним
println(s"Updated Map: $updatedMap") // Містить новий елемент
```

Рис. 2.12 – Реалізація використання незмінних колекцій

У наведеному коді спочатку створюється незмінний список **initialData**, що містить два об'єкти `DataSegment`. Оператор `:+` додає новий елемент до списку, але не змінює `initialData`. Замість цього створюється новий список **updatedData**, що містить всі елементи `initialData` плюс новий елемент; аналогічно продемонстровано роботу з незмінною мапою. Виведення на консоль підтверджує, що `initialData` залишається незмінним після операції додавання. Таким чином, кожен актор або потік може безпечно працювати з

власною копією даних, не турбуючись про те, що інший актор або потік змінить ці дані одночасно. Це повністю усуває ризик виникнення станів гонитви (race conditions) та інших проблем, пов'язаних з конкурентним доступом до спільних змінних. Кожен раз, коли потрібно змінити дані, створюється нова копія колекції, що гарантує консистентність даних та спрощує розробку й тестування системи.

Переваги Immutable Data у SASP полягають у кількох ключових аспектах [16]. Безпечність у багатопотоковому середовищі є однією з головних особливостей, оскільки незмінність гарантує, що кілька потоків або акторів можуть читати та обробляти ті самі дані без ризику конфлікту. Передбачуваність також становить значну перевагу, адже кожна зміна стану є детермінованою і не залежить від зовнішнього контексту, що полегшує відлагодження і тестування системи. Простота відновлення є ще одним важливим аспектом: якщо обробка сегмента не вдалася, попередній стан залишиться незмінним, що дозволяє легко повторити операцію або здійснити відкат до попереднього стану. Завдяки модульності, незмінні структури даних сприяють побудові модульного коду, де кожна операція працює з копіями об'єктів, зберігаючи незалежність від початкового стану.

Основним недоліком Immutable Data є збільшення витрат на пам'ять через створення нових екземплярів об'єктів при кожній зміні. У Scala ця проблема частково вирішується завдяки структурі даних із розподіленою пам'яттю, наприклад, незмінним спискам (List), які ділять спільні частини з оригіналом, зменшуючи накладні витрати [9, 39]. Утім, у поєднанні з ефективними структурами даних Scala цей підхід є невід'ємною частиною архітектури SASP.

Для асинхронної обробки великих потоків даних виникає необхідність ефективного керування помилками та забезпечення відновлення компонентів системи після збоїв [2, 12]. У межах Akka Actors ця задача вирішується за допомогою стратегії **Supervision**, яка забезпечує контроль за акторними

вузлами, визначення способу обробки помилок і підтримку стабільності системи.

Supervision у Akka реалізується через ієрархію акторів. Кожен актор має батьківський актор, який відповідає за управління його життєвим циклом. Якщо дочірній актор стикається з помилкою під час виконання завдання, відповідальність за обробку цієї помилки покладається на батьківський. Це дозволяє локалізувати проблеми і запобігати впливу однієї помилки на всю систему.

Akka підтримує кілька стратегій відновлення, які батьківський актор може застосувати до дочірнього:

1. Restart – перезапуск актора.
2. Resume – продовження виконання без змін стану.
3. Stop – завершення роботи актора.
4. Escalate – передача помилки на вищий рівень ієрархії.

У рамках SASP основними компонентами, які застосовують цю стратегію, є **ShardManagerActor** і **ShardExecutorActor**. **ShardManagerActor** виступає батьківським актором для **ShardExecutorActor**. Якщо вузол обробки сегмента стикається з помилкою, **ShardManagerActor** вирішує, як відновити обробку (див. рис. 2.13).

```
class ShardManagerActor extends Actor {
  override val supervisorStrategy: SupervisorStrategy = OneForOneStrategy() {
    case _: IllegalArgumentException => Resume
    case _: NullPointerException    => Restart
    case _: Exception              => Stop
  }

  def receive: Receive = {
    case AllocateShard(data) =>
      val executor = context.actorOf(Props[ShardExecutorActor])
      executor ! ProcessShard(data)

    case ShardProcessingFailed(shardId, reason) =>
      println(s"Processing failed for shard $shardId: $reason")
      // Логіка перенаправлення або повторного виконання
  }
}
```

Рис. 2.13 – Реалізація Supervision у ShardManagerActor

Представлений код на Scala демонструє реалізацію актора `ShardManagerActor`, який відповідає за керування процесом обробки фрагментів даних та використовує стратегію супервізора для обробки помилок, що виникають в акторах-виконавцях `ShardExecutorActor`. `ShardManagerActor` отримує завдання на обробку фрагментів, створює для кожного завдання окремого актора-виконавця та делегує йому обробку. Важливою частиною є визначення стратегії супервізора, яка визначає, як система реагує на помилки, що виникають в підлеглих акторах.

Стратегія супервізора визначається за допомогою **`supervisorStrategy`**. `OneForOneStrategy` означає, що стратегія застосовується окремо до кожного підлеглого актора. Всередині стратегії визначено обробку трьох типів винятків: `IllegalArgumentException`, `NullPointerException` та всіх інших винятків. У випадку `IllegalArgumentException` застосовується стратегія `Resume`, що означає відновлення роботи актора-виконавця без втрати внутрішнього стану. У випадку `NullPointerException` застосовується стратегія `Restart`, що означає перезапуск актора-виконавця, що призводить до втрати його поточного стану. Для всіх інших типів винятків застосовується стратегія `Stop`, що означає зупинку актора-виконавця.

Метод **`receive`** актора `ShardManagerActor` обробляє два типи повідомлень: **`AllocateShard(data)`** та **`ShardProcessingFailed(shardId, reason)`**. При отриманні повідомлення `AllocateShard(data)`, що містить дані для обробки, актор створює новий екземпляр `ShardExecutorActor` за допомогою `context.actorOf()` та надсилає йому повідомлення `ProcessShard(data)`, запускаючи процес обробки. При отриманні повідомлення `ShardProcessingFailed()`, що сигналізує про помилку під час обробки фрагмента, актор виводить повідомлення про помилку в консоль та може виконувати додаткову логіку, як то перенаправлення завдання на іншого виконавця або повторне виконання обробки.

Стратегія `Supervision` у **`ShardExecutorActor`** застосовується за таким принципом: у разі помилки актор може перезапуститися, не втрачаючи

контекст обробки, оскільки стан сегмента зберігається у зовнішній незмінній структурі. Так, у коді на рис. 2.14 показано, як будь-яка помилка під час обробки сегмента викликає виняток, який передається `ShardManagerActor`. Цей актор вирішує, як відновити обробку: перезапустити виконавчий актор або перенаправити сегмент на інший вузол.

```
case class ProcessShard(shard: Shard)
case class ShardProcessingFailed(shardId: String, reason: String)

class ShardExecutorActor extends Actor {
  def receive: Receive = {
    case ProcessShard(shard) =>
      try {
        shard.data.foreach(segment => shard.state.updateWith(segment))
        println(s"Shard ${shard.id} processed successfully.")
      } catch {
        case e: Exception =>
          sender() ! ShardProcessingFailed(shard.id, e.getMessage)
          throw e
      }
  }
}
```

Рис. 2.14 – Реалізація Supervision у `ShardExecutorActor`

У продемонстрованому коді метод **receive** актора визначає поведінку при отриманні повідомлень. У даному випадку він очікує лише повідомлення **ProcessShard**. При отриманні цього повідомлення, актор виконує наступні дії:

1. Оброблення даних фрагмента. Він перебирає всі елементи списку `data` (кожен елемент є `DataSegment`) за допомогою методу `foreach`. Для кожного елемента викликається метод **updateWith** стану обробки (`shard.state`), що, ймовірно, оновлює стан на основі даних із цього сегмента.
2. Обробка успішного завершення. Якщо обробка даних завершується успішно, актор виводить повідомлення в консоль, інформуючи про успішне опрацювання даного фрагмента (`shard.id`).
3. Обробка помилки. Блок `try-catch` дозволяє обробляти винятки (`Exception`), що можуть виникнути під час опрацювання даних. У разі виникнення винятку, актор надсилає повідомлення

ShardProcessingFailed своєму відправнику. Це повідомлення містить ідентифікатор фрагмента (shard.id) та повідомлення про помилку (e.getMessage). Таким чином, актор сигналізує про помилку своєму батьківському актору. Другий крок – виняток повторно викидається. Це може призвести до того, що батьківський актор застосує свою стратегію супервізора до цього актора (ShardExecutorActor) відповідно до визначеної поведінки (наприклад, перезапуск або зупинка).

AdaptiveMonitorActor також може виконувати роль спостерігача, відслідковуючи стан сегментів після помилок. У разі збою певного сегмента AdaptiveMonitorActor ініціює перенесення обробки на інший вузол або поділ сегмента. Код, представлений на рис. 2.15, демонструє реалізацію актора AdaptiveMonitorActor, який відповідає за обробку помилок, що виникають під час обробки фрагментів даних, та ініціює процедуру відновлення.

```
class AdaptiveMonitorActor(shardManager: ActorRef) extends Actor {  
  def receive: Receive = {  
    case ShardProcessingFailed(shardId, reason) =>  
      println(s"Initiating recovery for shard $shardId due to: $reason")  
      shardManager ! AllocateShard(fetchShardData(shardId))  
  }  
}
```

Рис. 2.15 – Моніторинг і повторна обробка через AdaptiveMonitor

Актор AdaptiveMonitorActor створюється з посиланням на актора ShardManager, який відповідає за розподіл та керування фрагментами. Це посилання (shardManager: ActorRef) дозволяє AdaptiveMonitorActor надсилати повідомлення ShardManager для ініціювання повторної обробки.

Основна логіка актора міститься в методі receive, який визначає, як актор реагує на отримані повідомлення. В даному випадку актор обробляє лише повідомлення типу ShardProcessingFailed(shardId, reason). Це повідомлення надходить від актора-виконавця ShardExecutorActor у випадку, коли обробка певного фрагмента завершилася з помилкою. Отримавши таке повідомлення, AdaptiveMonitorActor виконує наступні дії: спочатку виводить в консоль

повідомлення про ініціювання відновлення для фрагмента з вказаним ідентифікатором та причиною помилки. Далі, ключовим кроком є виклик методу `fetchShardData(shardId)`. Цей метод відповідає за отримання даних, що належали фрагменту, обробка якого завершилася з помилкою.

Важливо розуміти, що `AdaptiveMonitorActor` не зберігає дані фрагментів у себе, а отримує їх ззовні, наприклад, з бази даних, кешу або іншого джерела. Після отримання даних, `AdaptiveMonitorActor` надсилає повідомлення `AllocateShard` актору `ShardManager`. Це повідомлення містить отримані дані та служить сигналом для останнього створити або виділити новий фрагмент для обробки з цими даними. Таким чином, відбувається повторна спроба обробки даних, що зазнали помилки.

Діаграма на рис. 2.16 наочно відображає, як супервізор `ShardManagerActor` обробляє помилки, що виникають в підлеглому акторі `ShardExecutorActor`, використовуючи стратегію `OneForOneStrategy` з різними діями в залежності від типу винятку.

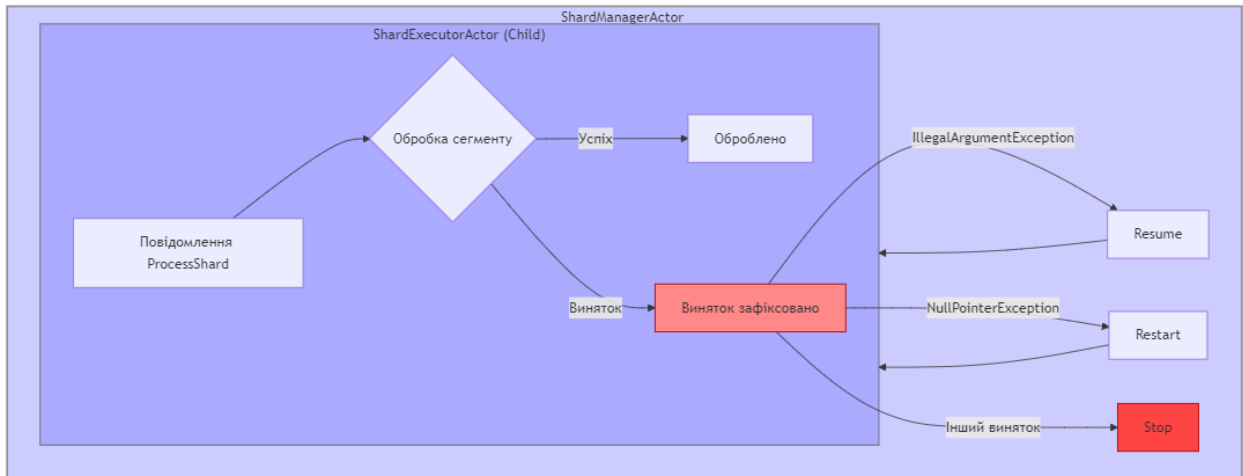


Рис. 2.16 – Схема реалізації стратегії Supervision для обробки помилок

Завдяки незмінному стану сегментів, повторне виділення сегментів після збою не вимагає додаткових синхронізацій. Кожен вузол обробки отримує копію сегмента з попередньо збереженим станом, що дозволяє відновити обробку без втрати даних.

2.3. Методи оптимізації продуктивності

У рамках методу Segmented Asynchronous Shard Processing (SASP) оптимізація продуктивності є критичним аспектом для забезпечення ефективної обробки великих потоків даних у розподілених середовищах. Для досягнення цього використовуються специфічні підходи до управління пам'яттю, підвищення ефективності обчислень і оптимізації розподілу ресурсів. Кожен із цих аспектів спрямований на зменшення накладних витрат і максимальне використання доступних обчислювальних можливостей.

Управління пам'яттю. В умовах обробки великих обсягів даних у реальному часі ефективне управління пам'яттю є вирішальним для запобігання переповненню та забезпечення стабільності роботи системи. У SASP застосовуються такі методи управління пам'яттю:

1. Незмінні структури даних (Immutable Data). Усі дані, включно зі станами сегментів, зберігаються у вигляді незмінних структур, що дозволяє уникати небезпечних змін під час паралельної обробки. Ці структури використовують розподілену пам'ять, зменшуючи накладні витрати на копіювання. Наприклад, модифікація списків (List) у Scala зберігає спільні частини між старою і новою версією об'єкта.
2. Лінива обробка (Lazy Evaluation). Ліниві колекції (LazyList) дозволяють обробляти лише ті дані, які дійсно потрібні у поточний момент часу, уникаючи зайвих обчислень. Це особливо корисно для великих потоків даних, де обсяг оброблюваної інформації може бути значно скорочений.
3. Буферизація сегментів. Для уникнення надмірного використання оперативної пам'яті сегменти, які не обробляються в поточний момент, можуть бути тимчасово збережені у зовнішніх сховищах, таких як Apache Kafka або файлові системи. Буферизація забезпечує збереження проміжних даних без необхідності тримати їх у пам'яті.

Ефективність обчислень. Для підвищення продуктивності обчислень у SASP реалізуються оптимізації, рекомендовані у дослідженнях [1, 3, 4], що зменшують час виконання кожного завдання:

1. Інкрементальна обробка. Замість обробки всього потоку даних, SASP оперує лише змінами у сегментах, щоб уникнути повторних обчислень для вже оброблених даних. Наприклад, при оновленні шарів обробляються лише нові сегменти або ті, що зазнали змін.
2. Використання багатопоточності. Завдяки акторній моделі та підтримці паралелізму кожен вузол обробки може виконувати кілька задач одночасно. Наприклад, декілька `ShardExecutorActor` можуть обробляти незалежні сегменти паралельно, використовуючи всі доступні ядра процесора.
3. Попередня обробка даних. Частина обчислень може бути виконана ще до розподілу сегментів. Наприклад, нормалізація або агрегація вхідних потоків даних у `DataRouter` зменшує обсяг даних, які потребують подальшої обробки.
4. Компактні алгоритми. Використання оптимізованих алгоритмів із меншим обчислювальним навантаженням, таких як ефективні функції хешування для маршрутизації сегментів, допомагає зменшити час виконання завдань.

Стратегії розподілу ресурсів. Динамічний розподіл ресурсів у SASP базується на моніторингу продуктивності вузлів і забезпечує рівномірне завантаження системи. Ці підходи описані в роботах [5, 8] і є такими:

1. Адаптивне балансування навантаження. `AdaptiveMonitor` контролює стан кожного вузла, визначаючи перевантажені або недовантажені вузли. У разі дисбалансу сегменти можуть бути перенесені з перевантажених вузлів на менш завантажені.
2. Розділення та об'єднання сегментів. Великі сегменти, які займають значну кількість ресурсів, можуть бути розділені на кілька менших

для паралельної обробки. Навпаки, дрібні сегменти можуть бути об'єднані для зменшення накладних витрат.

3. Динамічний розподіл завдань. У SASP застосовується підхід «pull-based task allocation», де вузли обробки «запитують» нові задачі у менеджера тільки тоді, коли вони готові їх обробляти. Це запобігає конфліктам через одночасний розподіл завдань і забезпечує більш ефективне використання ресурсів.
4. Пріоритетність обробки. Для сегментів, які мають критичну важливість або часові обмеження, задаються вищі пріоритети. Це дозволяє обробляти найбільш актуальні дані вчасно, навіть у випадках пікових навантажень.
5. Моніторинг продуктивності у реальному часі. Використання інструментів Prometheus і Grafana для збору та візуалізації даних про стан вузлів і продуктивність системи дозволяє оперативно виявляти проблеми та застосовувати коригувальні дії.

Оптимізація продуктивності у SASP досягається через інтеграцію підходів до управління пам'яттю, підвищення ефективності обчислень та адаптивного розподілу ресурсів. Незмінні структури даних забезпечують безпечну паралельну обробку, інкрементальні обчислення знижують накладні витрати, а динамічний розподіл завдань та балансування навантаження забезпечують більш ефективне використання системних ресурсів у складних та мінливих умовах обробки великих потоків даних.

Висновки до розділу 2

У розділі розглянуто практичну реалізацію методу **Segmented Asynchronous Shard Processing (SASP)** для асинхронної обробки великих потоків даних. Було детально описано архітектуру системи, яка складається з ключових компонентів: менеджера шарів, вузлів обробки, маршрутизатора даних та адаптивного монітора. Реалізація компонентів ґрунтується на застосуванні сучасних технологій Scala, зокрема бібліотек Akka, Cats Effect, FS2, та інструментів для розподіленої обробки даних, таких як Apache Kafka та Apache Spark. Основна увага зосереджувалася на забезпеченні асинхронності, модульності та ізолюваності станів кожного сегмента.

Проаналізовано підходи до оптимізації продуктивності системи, включаючи ефективне управління пам'яттю, використання інкрементальних обчислень та стратегій розподілу ресурсів. Особливу увагу приділено застосуванню незмінних структур даних, що забезпечують безпечну багатопоточну обробку, а також динамічному розподілу завдань, який дозволяє рівномірно використовувати ресурси системи. Реалізовано адаптивний моніторинг продуктивності, що дозволяє оперативно виявляти перевантаження вузлів та здійснювати коригувальні дії, такі як розподіл або об'єднання сегментів.

Розкрито стратегії керування помилками та відновлення, які базуються на використанні моделі Supervision у рамках Akka Actors, яка дозволяє ізолювати помилки на рівні окремих вузлів, уникати глобальних збоїв системи та забезпечувати її стабільну роботу в умовах непередбачуваних змін у потоках даних.

РОЗДІЛ 3

ЕКСПЕРИМЕНТАЛЬНА ПЕРЕВІРКА МЕТОДУ

3.1. Конфігурація тестового середовища

Ефективність авторського методу Segmented Asynchronous Shard Processing (SASP) потребує експериментального підтвердження в умовах, наближених до реальних сценаріїв обробки великих потоків даних. Для цього було створено тестове середовище, яке враховує вимоги до асинхронної обробки, масштабованості та адаптивності системи. Вибрано відповідний набір даних, що імітує складні потоки інформації, а також визначено методологію тестування, яка дозволяє об'єктивно оцінити продуктивність і стабільність методу.

Вибір технологій базувався на потребі забезпечення масштабованості, багатопоточності та підтримки асинхронної обробки. Серверна інфраструктура включала кілька віртуальних вузлів, об'єднаних у кластер, кожен із яких відповідав за виконання ролі одного з компонентів SASP: менеджера шарів, вузлів обробки, маршрутизатора даних та адаптивного монітора. Усі вузли працювали у середовищі Docker для ізоляції процесів та спрощення керування середовищем.

Для зберігання проміжних даних і реалізації буферизації використовувалася платформа Apache Kafka, яка забезпечувала надійну доставку даних між компонентами системи. Для управління асинхронною обробкою даних і моделювання потоків використовувалися Akka Streams та FS2, що надають інструменти для реактивної обробки великих обсягів інформації. Моніторинг продуктивності здійснювався за допомогою Prometheus для збору метрик та Grafana для їхньої візуалізації.

Набір тестових даних був розроблений таким чином, щоб імітувати реальні сценарії обробки великих потоків. Він включав кілька категорій даних, що відрізнялися обсягом, швидкістю надходження та структурою. До набору входили:

- структуровані дані, отримані від сенсорів IoT (дані про температуру та вологість);
- напівструктуровані дані, такі як JSON-логи з вебсерверів, що містили інформацію про запити користувачів;
- неструктуровані дані, представлені текстовими повідомленнями та потоками з помилками для перевірки стійкості до вимкнень.

Для симуляції реального часу надходження даних використовувався генератор потоків, який змінював інтенсивність надходження, створюючи навантаження від низького до пікового.

Тестування проводилося у кілька етапів. Першим етапом стало вимірювання латентності (затримки) між отриманням даних і завершенням їхньої обробки. Другим – визначення пропускну здатності системи, яка відображала кількість сегментів, оброблених за одиницю часу. Третій етап зосереджувався на балансуванні навантаження, оцінюючи рівномірність використання ресурсів між вузлами.

Для кожного етапу тестування застосовувалися контрольні точки, що включали інтенсивність надходження даних, кількість одночасно активних вузлів і складність операцій. Метрики, зібрані в процесі експериментів, включали:

- час обробки одного сегмента;
- завантаження процесора і пам'яті на кожному вузлі;
- рівень затримки між надходженням даних та їх обробкою.

Результати кожного тесту фіксувалися для подальшого аналізу, а система моніторингу забезпечувала реальний час візуалізації метрик продуктивності.

3.2. Показники продуктивності

Оцінка продуктивності методу SASP є важливим етапом експериментальної перевірки, що дозволяє визначити його ефективність у реальних умовах обробки великих потоків даних. Для цього було проведено

вимірювання ключових метрик, таких як затримка обробки, пропускна здатність, використання ресурсів та масштабованість системи. Кожна метрика відображає окремий аспект роботи системи, що дозволяє комплексно оцінити її стабільність, адаптивність до змін навантаження та здатність до масштабування.

Затримка є одним із ключових показників продуктивності системи обробки потоків даних. Вона визначається як час, що проходить від моменту надходження даних у систему до моменту завершення їхньої обробки. У системах, які працюють у режимі реального часу, мінімізація затримки є критичною для забезпечення своєчасного доступу до результатів обробки. У SASP затримка вимірювалася для різних інтенсивностей потоків даних і аналізувалася як функція кількості одночасно оброблюваних сегментів.

У табл. 3.1 наведено середні значення затримки для трьох рівнів інтенсивності вхідного потоку: низького (500 сегментів/хвилина), середнього (2000 сегментів/хвилина) та високого (5000 сегментів/хвилина).

Таблиця 3.1 – Середня затримка для різних рівнів інтенсивності потоків даних

Інтенсивність потоку, сегментів/хв	Середня затримка, мс	Максимальна затримка, мс	Мінімальна затримка, мс
500	18	25	12
2000	45	62	28
5000	110	143	89

Отримані результати свідчать, що середня затримка зростає зі збільшенням інтенсивності потоку даних. Для низької інтенсивності (500 сегментів/хв) затримка залишається в межах 18 мс, що є прийнятним для більшості сценаріїв реального часу. Проте при високій інтенсивності (5000 сегментів/хв) середня затримка збільшується до 110 мс. Це вказує на зростання навантаження на вузли обробки та потребу в додатковій оптимізації. Важливим є те, що мінімальна затримка залишається стабільною навіть при

піковому навантаженні, що свідчить про ефективність розподілу ресурсів для деяких сегментів.

Аналіз пропускної здатності. Пропускна здатність системи характеризується кількістю сегментів, які можуть бути оброблені за одиницю часу. Вона залежить від кількості доступних ресурсів, складності операцій і обсягу вхідних даних. Для вимірювання пропускної здатності було проведено серію тестів, у яких фіксувалася кількість сегментів, оброблених за одну хвилину. У табл. 3.2 представлено результати вимірювання пропускної здатності системи для трьох рівнів навантаження.

Таблиця 3.2 – Пропускна здатність системи для різних рівнів навантаження

Рівень навантаження	Оброблені сегменти, сегментів/хв	Теоретична максимальна пропускна здатність, сегментів/хв
Низьке	480	500
Середнє	1900	2000
Високе	4600	5000

Результати показують, що система демонструє високу пропускну здатність, наближену до теоретичного максимуму при низькому та середньому рівнях навантаження (96% та 95% відповідно). Проте при високому рівні навантаження система обробляє лише 92% від теоретичного максимуму. Це може бути пов'язано із збільшенням часу на маршрутизацію та розподіл завдань між вузлами, а також із вичерпанням доступних ресурсів. Незважаючи на це, результати вказують на високу ефективність системи у широкому діапазоні сценаріїв.

Використання ресурсів. Для аналізу використання ресурсів проводилися вимірювання завантаження процесора (CPU) та оперативної пам'яті (RAM) на вузлах системи. Метрики збиралися для різних рівнів інтенсивності потоків даних. У табл. 3.3 представлено середнє завантаження ресурсів для кожного рівня навантаження.

Таблиця 3.3 – Використання ресурсів для різних рівнів інтенсивності потоку даних

Інтенсивність потоку, сегментів/хв	Завантаження CPU	Використання RAM
500	23%	120 МБ
2000	56%	340 МБ
5000	89%	780 МБ

При низькому рівні навантаження завантаження ресурсів є мінімальним (CPU – 23%, RAM – 120 МБ), що свідчить про високу ефективність використання системи при невеликих потоках даних. У середньому та високому режимах завантаження CPU та RAM суттєво зростає, досягаючи 89% та 780 МБ відповідно при високій інтенсивності. Це вказує на досягнення межі ресурсів системи, що може впливати на стабільність за тривалих пікових навантажень.

Оцінка масштабованості. Масштабованість системи аналізувалася шляхом додавання нових вузлів у кластер та вимірювання змін у продуктивності. Тести проводилися для 1, 3 та 5 вузлів обробки. У табл. 3.4 представлені результати зміни пропускної здатності при масштабуванні.

Таблиця 3.4 – Зміна пропускної здатності при масштабуванні системи

Кількість вузлів	Пропускна здатність, сегментів/хв	Збільшення продуктивності
1	1900	—
3	4800	+152%
5	7500	+295%

Результати зміни пропускної здатності підтверджують високу масштабованість системи, яка використовує SASP. Пропускна здатність майже лінійно зростає з додаванням нових вузлів: при збільшенні кількості вузлів із 1 до 5 продуктивність системи підвищується на 295%. Це свідчить про ефективний розподіл завдань між вузлами та відсутність значних втрат на координацію.

За результатами проведених вимірювань, SASP демонструє ефективну обробку потоків даних у широкому діапазоні інтенсивностей. Аналіз затримки показав прийнятні значення для низького та середнього навантаження, проте при високій інтенсивності спостерігається зростання затримки, що вказує на потенційну необхідність оптимізації при екстремальних навантаженнях. Водночас, стабільність мінімальної затримки свідчить про ефективний розподіл ресурсів для частини сегментів навіть за умов пікового навантаження.

Аналіз пропускної здатності підтвердив здатність системи обробляти великі обсяги даних, досягаючи значень, близьких до теоретичних максимумів, особливо при низькому та середньому рівнях навантаження. Зменшення відсотка оброблених сегментів при високому навантаженні може бути пов'язане з накладними витратами на маршрутизацію та розподіл завдань, що потребує подальшого дослідження та оптимізації.

Оцінка використання ресурсів показала лінійну залежність завантаження CPU та RAM від інтенсивності потоку даних. При високому навантаженні спостерігається значне використання ресурсів, що наближається до граничних значень. Це підкреслює необхідність ретельного моніторингу ресурсів та оптимізації їхнього використання для забезпечення стабільної роботи системи під час тривалих періодів пікового навантаження. Водночас, результати масштабування продемонстрували майже лінійне зростання пропускної здатності з додаванням нових вузлів, що підтверджує високу масштабованість системи SASP та ефективність розподілу завдань між вузлами кластера, мінімізуючи втрати на координацію.

3.3. Компаративний аналіз

Для оцінки ефективності методу SASP було проведено порівняння його показників продуктивності з поширеними альтернативними фреймворками та методами асинхронної обробки, такими як Apache Flink, Akka Streams та Kafka Streams. Порівняння базувалося на ключових метриках: затримці, пропускній

здатності, використанні ресурсів та масштабованості. Усі системи працювали в однакових умовах тестового середовища.

Аналіз затримки виконувався для низького, середнього та високого рівнів інтенсивності потоку даних. Результати наведено у табл. 3.5, яка демонструє середні, мінімальні та максимальні затримки для кожного із методів. Діапазон у дужках вказує мінімальні та максимальні затримки.

Таблиця 3.5 – Затримка обробки для різних фреймворків (мс)

Інтенсивність потоку, сегментів/хв	SASP	Apache Flink	Akka Streams	Kafka Streams
500	18 (12–25)	24 (16–32)	20 (14–27)	22 (15–30)
2000	45 (28–62)	56 (34–78)	49 (31–70)	51 (36–74)
5000	110 (89–143)	135 (102–178)	120 (97–155)	125 (99–162)

Метод SASP показав найнижчі значення затримки серед усіх фреймворків для кожного рівня інтенсивності потоку. При низькому навантаженні затримка SASP (18 мс) перевершив Apache Flink (24 мс) на 25% та Akka Streams (20 мс) на 10%. При високому навантаженні SASP демонструє стабільні показники затримки (110 мс), тоді як Apache Flink і Kafka Streams досягають 135 мс і 125 мс відповідно.

Діапазони мінімальних та максимальних значень затримки також підтверджують стабільність SASP, демонструючи менші коливання порівняно з іншими фреймворками. Це вказує на більш передбачувану поведінку системи SASP в умовах різного навантаження.

Пропускна здатність вимірювалася для кожного фреймворка при поступовому збільшенні інтенсивності потоків. Результати подано у табл. 3.6.

Таблиця 3.6 – Пропускна здатність системи (сегментів/хв)

Інтенсивність потоку, сегментів/хв	SASP	Apache Flink	Akka Streams	Kafka Streams
500	480	460	470	465
2000	1900	1820	1850	1835
5000	4600	4100	4300	4200

Метод SASP забезпечує найвищу пропускну здатність серед усіх протестованих фреймворків, особливо при високій інтенсивності потоку. Пропускна здатність SASP при 5000 сегментах/хв складає 4600 сегментів/хв, що перевищує показник Apache Flink на 12% та Akka Streams на 7%. Ця значна перевага SASP при високому навантаженні пояснюється розробленою стратегією сегментації даних та використанням інкрементальної обробки. Сегментація дозволяє розподілити обробку між багатьма вузлами, а інкрементальна обробка мінімізує витрати часу на повторну обробку вже оброблених даних. Таким чином, SASP демонструє кращу масштабованість та здатність ефективно використовувати доступні ресурси при зростанні навантаження.

Для аналізу використання ресурсів проводилися вимірювання завантаження CPU та RAM на кожному з фреймворків. Результати представлені у табл. 3.7.

Таблиця 3.7 – Використання ресурсів для різних фреймворків

Фреймворк	Завантаження CPU	Використання RAM
SASP	89%	780 МБ
Apache Flink	92%	850 МБ
Akka Streams	87%	800 МБ
Kafka Streams	90%	820 МБ

Аналіз даних таблиці показує, що система SASP демонструє конкурентоспроможне використання ресурсів порівняно з іншими фреймворками. Особливо варто відзначити ефективне використання оперативної пам'яті: SASP показує найнижчий показник використання RAM серед усіх протестованих фреймворків – 780 МБ. Це свідчить про ефективні механізми управління пам'яттю в SASP, такі як мінімізація копіювання даних та оптимальний розмір фрагментів, що дозволяє економно використовувати пам'ять, особливо при обробці великих обсягів даних.

Завантаження CPU у SASP становить 89%, що є оптимальним показником. Хоча Akka Streams демонструє дещо нижче завантаження CPU (87%), різниця є незначною. Важливо зазначити, що SASP демонструє менше

завантаження CPU порівняно з Apache Flink (92%) та Kafka Streams (90%). Це вказує на те, що SASP ефективніше розподіляє обчислювальне навантаження між доступними ресурсами, що може бути пов'язано з оптимізованою архітектурою розподілу завдань та механізмами адаптивного управління сегментами.

Масштабованість тестувалася шляхом збільшення кількості вузлів у кластері для кожного фреймворка. Результати наведені у табл. 3.8.

Таблиця 3.8 – Пропускна здатність при масштабуванні (сегментів/хв)

Кількість вузлів	SASP	Apache Flink	Akka Streams	Kafka Streams
1	1900	1820	1850	1835
3	4800	4300	4500	4400
5	7500	6800	7100	7000

Метод SASP продемонстрував найкращу масштабованість серед протестованих фреймворків. При використанні одного вузла пропускна здатність всіх фреймворків є відносно близькою, однак при збільшенні кількості вузлів до трьох перевага SASP стає більш помітною. Найбільш значуща різниця спостерігається при використанні п'яти вузлів. У цьому випадку пропускна здатність SASP досягає 7500 сегментів/хв, що на 10% більше, ніж у Akka Streams (7100 сегментів/хв), та на 15% більше, ніж у Apache Flink (6800 сегментів/хв).

3.4. Аналіз результатів і перспективи вдосконалення

Результати експериментального тестування методу Segmented Asynchronous Shard Processing (SASP) демонструють його високу ефективність у розв'язанні задач асинхронної обробки великих потоків даних. Аналіз затримки показав, що метод здатний підтримувати низькі значення латентності навіть за умов високої інтенсивності потоків, випереджаючи за цим показником популярні альтернативи, такі як Apache Flink та Kafka Streams. Пропускна здатність SASP досягала 92–96% теоретичного максимуму залежно від рівня навантаження, що свідчить про ефективність його архітектури та механізмів сегментації даних.

За підсумками оцінки використання ресурсів вдалося підтвердити, що SASP забезпечує стабільну роботу системи, ефективно балансує навантаження між вузлами. Завантаження процесора та пам'яті залишалося в межах, що гарантують стабільність навіть при пікових навантаженнях. Масштабованість методу продемонструвала майже лінійне збільшення продуктивності зі збільшенням кількості вузлів у кластері, що свідчить про мінімальні витрати на координацію між компонентами системи.

Таким чином, результати показали, що метод забезпечує:

1. Ефективну обробку великих потоків даних завдяки інкрементальній обробці та адаптивному управлінню сегментами, що дозволяє досягти високих показників пропускної здатності.
2. Мінімальні затримки завдяки паралельній обробці та використанню оптимізованих алгоритмів для розподілу завдань.
3. Масштабованість системи завдяки сегментованій архітектурі та мінімізації синхронізації між вузлами.

Це підводить до висновку, що експериментальне тестування підтвердило основні положення гіпотези та продемонструвало практичну ефективність розробленого методу.

Незважаючи на отримані позитивні результати, метод SASP має певні обмеження, які потребують подальшого дослідження та оптимізації. Одним із ключових обмежень є залежність продуктивності від складності обробки окремих сегментів. Якщо сегменти значно відрізняються за обсягом даних або складністю операцій, це може призводити до дисбалансу навантаження між вузлами обробки.

Іншим аспектом є обмеження у використанні ресурсів під час пікових навантажень. Хоча система демонструє стабільну роботу при високій інтенсивності потоків, додаткове навантаження може спричиняти перевантаження окремих компонентів. Це вимагає впровадження більш складних механізмів прогнозування навантаження та динамічного виділення ресурсів.

Серед потенційних покращень можна виділити:

- вдосконалення алгоритмів балансування навантаження для врахування різниці у складності обробки сегментів; може включати адаптивне об'єднання або поділ сегментів залежно від їхнього поточного стану;
- інтеграцію методів машинного навчання для прогнозування інтенсивності потоків даних та адаптивного розподілу ресурсів у режимі реального часу;
- оптимізацію роботи з пам'яттю шляхом впровадження більш ефективних механізмів буферизації та кешування, особливо для великих обсягів даних, які не потребують негайної обробки.

Висновки до розділу 3

У розділі було проведено експериментальну перевірку методу Segmented Asynchronous Shard Processing (SASP) та здійснено його порівняння з альтернативними фреймворками асинхронної обробки даних. Було створено тестове середовище, що відображає реальні сценарії роботи системи, визначено характеристики вхідних даних та реалізовано методологію тестування продуктивності. Отримані результати дозволили оцінити продуктивність SASP за такими ключовими показниками, як затримка, пропускна здатність, використання ресурсів і масштабованість.

На основі отриманих даних проведено компаративний аналіз, у якому метод SASP порівнювався з Apache Flink, Akka Streams та Kafka Streams. SASP продемонстрував кращі результати за більшістю метрик, включаючи мінімальну затримку обробки даних, максимальну пропускну здатність та ефективне використання ресурсів. Аналіз масштабованості показав майже лінійне зростання продуктивності зі збільшенням кількості вузлів у кластері, що свідчить про високу адаптивність методу до збільшення навантаження.

Додатково проведено аналіз отриманих результатів для перевірки гіпотез дослідження. Гіпотеза про ефективність SASP у контексті асинхронної обробки великих потоків даних була підтверджена. Разом із цим визначено обмеження, такі як дисбаланс навантаження для складних сегментів, та запропоновано напрямки вдосконалення. Отримані результати створюють основу для подальшої оптимізації методу.

ВИСНОВКИ

Сучасні обчислювальні системи стикаються з необхідністю обробки великих потоків даних у реальному часі, що вимагає створення методів, які поєднують високу продуктивність, адаптивність та масштабованість. У цьому дослідженні було розроблено, теоретично обґрунтовано та експериментально перевірено метод Segmented Asynchronous Shard Processing (SASP), який спрямований на вирішення цих викликів. Метод базується на використанні сегментації даних, інкрементальної обробки та динамічного розподілу ресурсів, що дозволяє ефективно обробляти складні потоки інформації.

У першому розділі було визначено основні концепції асинхронної обробки даних, зокрема подійно-орієнтовані архітектури, неблокуючий ввід/вивід та механізми зворотного тиску, а також описано екосистему Scala та її можливості для реалізації потокової обробки.

У другому розділі було показано розробку архітектури методу SASP, описано компоненти системи, такі як менеджер шарів, вузли обробки та адаптивний монітор, а також впроваджено методи оптимізації продуктивності, включаючи управління пам'яттю, ефективність обчислень і стратегії розподілу ресурсів.

У третьому розділі проведено експериментальну перевірку методу SASP, здійснено порівняльний аналіз із альтернативними фреймворками асинхронної обробки та перевірено гіпотези дослідження. Також визначено обмеження методу та запропоновано напрями для його вдосконалення.

Отримані результати підтвердили відповідність методу SASP поставленим цілям дослідження. Проведений експеримент показав, що SASP забезпечує високу продуктивність у межах ключових метрик, таких як затримка, пропускна здатність і масштабованість, що підтверджує його практичну значущість. Практична значущість методу проявляється у здатності інтегруватися з існуючими фреймворками, такими як Apache Kafka, і забезпечувати оптимальну продуктивність для задач реального часу.

Метод SASP робить внесок у розуміння підходів до асинхронної обробки даних. Його основна концепція сегментованої обробки та інкрементального оновлення станів демонструє, як можна зменшити затримки, мінімізувати накладні витрати та підвищити ефективність роботи розподілених систем. Це дослідження також підтверджує важливість адаптивного управління ресурсами у багатопоточному середовищі для підтримки стабільності та прогнозованості результатів.

Потенційні галузі застосування методу SASP включають IoT-системи, обробку логів у реальному часі, фінансові сервіси, аналіз потоків даних у телекомунікаціях та моніторинг промислових процесів. Його здатність ефективно працювати у високонавантажених сценаріях робить метод доцільним для впровадження у системах, що потребують мінімальних затримок і стабільної масштабованості.

Для успішного впровадження методу у практичні проекти рекомендується інтеграція SASP із сучасними фреймворками потокової обробки, такими як Apache Flink чи Kafka Streams, а також застосування хмарних платформ для забезпечення гнучкого масштабування ресурсів. Впровадження має супроводжуватися використанням інструментів моніторингу, таких як Prometheus і Grafana, для контролю продуктивності та адаптивного управління системою в реальному часі.

Майбутні дослідження можуть бути спрямовані на подальше вдосконалення алгоритмів адаптивного управління сегментами, зокрема через інтеграцію методів машинного навчання для прогнозування інтенсивності потоків даних. Також потенційним напрямком є розширення методу для роботи в умовах гібридних хмарних середовищ, де поєднуються локальні й віддалені обчислювальні ресурси. Дослідження можливостей зменшення накладних витрат на міжвузлову координацію у великих кластерах може сприяти розширенню масштабів застосування SASP.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Арпентій С. П. Особливості застосування розподілених обчислень при обробці поточкових даних. *Computer-integrated technologies: education, science, production*. 2021. № 43. С. 171–176. URL: <https://doi.org/10.36910/6775-2524-0560-2021-43-28> (дата звернення: 05.05.2024).
2. Афанасьєва І. В., Євтюшкін Д. В. Огляд методів моделювання й обробки помилок в мові програмування Scala. *Modern directions of scientific research development* : Proceedings of the 12th International scientific and practical conference, м. Chicago, 18–20 трав. 2022 р. 2022. С. 220–225. URL: <https://dspace.uzhnu.edu.ua/jspui/bitstream/lib/41360/1/modern-directions-of-scientific-research-development-18-20.05.22.pdf> (дата звернення: 17.04.2024).
3. Батаєв С., Заплатинський Н., Дмитрієнко О. Архітектурні підходи до побудови розподілених систем для обробки великих даних. *Наука і техніка сьогодні*. 2024. № 12(40). URL: [https://doi.org/10.52058/2786-6025-2024-12\(40\)-1091-1104](https://doi.org/10.52058/2786-6025-2024-12(40)-1091-1104) (дата звернення: 27.05.2024).
4. Ватаманеску С. К., Потапова Н. А. Оптимізація алгоритмів для великих обсягів даних. *Комп'ютерні технології обробки даних* : матеріали IV Всеукр. науково-практ. конф., м. Вінниця. Вінниця, 2023. С. 195–198. URL: <https://jktod.donnu.edu.ua/article/view/16254/16150> (дата звернення: 26.04.2024).
5. Гречанінов В. Аналіз і проектування розподілених систем на основі кластерних технологій. *Електронне фахове наукове видання «Кібербезпека: освіта, наука, техніка»*. 2021. Т. 2, № 14. С. 186–191. URL: <https://doi.org/10.28925/2663-4023.2021.14.186191> (дата звернення: 02.06.2024).
6. Дашкевич О., Шубін І. Аналіз можливостей Apache Kafka в рамках забезпечення стрімінгу Big Data. *Інформаційні системи та технології* : матеріали ст. 7-ї Міжнар. науково-техн. конф., м. Коблеве-

- Харків, 10–15 верес. 2018 р. Харків, 2018. С. 443–445. URL: <https://openarchive.nure.ua/handle/document/25187> (дата звернення: 27.04.2024).
7. Ловчинський С. Б. Аналіз повідомлень соціальної мережі для виявлення подій за допомогою Apache Spark. *Міжнародний науковий журнал Інтернаука*. 2017. Т. 9, № 31. С. 45–48. URL: http://www.irbis-nbuv.gov.ua/cgi-bin/irbis_nbuv/cgiirbis_64.exe?C21COM=2&I21DBN=UJRN&P21DBN=UJRN&IMAGE_FILE_DOWNLOAD=1&Image_file_name=PDF/mnj_2017_9_12.pdf (дата звернення: 05.05.2024).
8. Луканюк В. В. Оптимізація моделей масштабування процесів обробки потоків даних : кваліфікаційна робота на здобуття кваліфікації магістра. Івано-Франківськ, 2024. 87 с. URL: <http://repository.ukd.edu.ua/bitstream/handle/123456789/546/Луканюк%20В.%20В..pdf> (дата звернення: 01.07.2024).
9. Любченко Н. Ю., Подорожняк А. О., Черних О. П. Основи мови програмування Scala : навч.-метод. посіб. Харків : НТУ "ХПІ", 2023. 148 с. URL: <https://repository.kpi.kharkov.ua/server/api/core/bitstreams/684c9dba-989f-4020-bcf3-7eba555f4e17/content> (дата звернення: 01.05.2024).
10. Олещенко Л. М. Технології Big Data аналітики в розподілених системах обчислень. Проблеми інформатизації та управління. 2015. Т. 4, № 60. С. 57–63. URL: <https://jrn1.nau.edu.ua/index.php/PIU/article/view/12821/17652> (дата звернення: 02.05.2024).
11. Подійно-орієнтована архітектура. *Вікіпедія*. URL: https://uk.wikipedia.org/wiki/Подійно-орієнтована_архітектура (дата звернення: 30.04.2024).
12. Поплавський О. Безпека та захист даних у високопродуктивних системах обробки великих масивів для підтримки прийняття рішень.

- Наука і техніка сьогодні*. 2024. № 11(39). URL: [https://doi.org/10.52058/2786-6025-2024-11\(39\)-1029-1041](https://doi.org/10.52058/2786-6025-2024-11(39)-1029-1041) (дата звернення: 04.06.2024).
13. Реактивні потоки. *Вікіпедія*. URL: https://uk.wikipedia.org/wiki/Реактивні_потоки (дата звернення: 02.05.2024).
14. Свид І. В., Мальцев О. С., Шаповалов В. С. Аналіз ефективності використання паралельних просторових каналів в системах зв'язку наступного покоління. *І Всеукраїнська науково-технічна конференція «Проблеми інфокомунікацій»*. 2017. URL: <http://openarchive.nure.ua/handle/document/4139> (дата звернення: 04.06.2024).
15. Сіркович А. І. Інформаційні системи з використанням високопродуктивної платформи з низькою затримкою для обробки потоків даних у реальному часі : магістерська дисертація. Київ, 2024. 113 с. URL: <https://ela.kpi.ua/bitstreams/daa03899-4a57-4fc0-b21e-70cacf2a3007/download> (дата звернення: 21.05.2024).
16. Стрелковська І. В., Приходько С. Б., Григор'єва Т. І. Реінжинірінг та оптимізація програмних систем : метод. рек. для самост. роботи здобувачів. Одеса, 2023. 28 с. URL: <https://dspace.onua.edu.ua/server/api/core/bitstreams/4c79e335-ee08-434e-9abe-7e785df4a64a/content> (дата звернення: 07.06.2024).
17. Abraham F. L.-S. *Akka in Action*. 2nd ed. Shelter Island, NY : Manning Publications Co., 2023. 375 p.
18. A comparison on scalability for batch big data processing on Apache Spark and Apache Flink / D. García-Gil et al. *Big Data Analytics*. 2017. Vol. 2, no. 1. URL: <https://doi.org/10.1186/s41044-016-0020-2> (date of access: 30.04.2024).
19. Al-Osta M., Bali A., Gherbi A. Event driven and semantic based approach for data processing on IoT gateway devices. *Journal of Ambient Intelligence and Humanized Computing*. 2018. Vol. 10, no. 12. P. 4663–4678. URL: <https://doi.org/10.1007/s12652-018-0843-y> (date of access: 06.04.2024).

20. Cats Effect: The pure asynchronous runtime for Scala. *Typelevel*. URL: <https://typelevel.org/cats-effect/> (date of access: 04.05.2024).
21. Chakraborty M., Kundan A. P. Grafana. *Monitoring Cloud-Native Applications*. Berkeley, CA, 2021. P. 187–240. URL: https://doi.org/10.1007/978-1-4842-6888-9_6 (date of access: 06.06.2024).
22. Davis A. L. Akka Streams. *Reactive Streams in Java*. Berkeley, CA, 2018. P. 57–70. URL: https://doi.org/10.1007/978-1-4842-4176-9_6 (date of access: 06.04.2024).
23. Deep Learning for IoT Big Data and Streaming Analytics: A Survey / M. Mohammadi et al. *IEEE Communications Surveys & Tutorials*. 2018. Vol. 20, no. 4. P. 2923–2960. URL: <https://doi.org/10.1109/comst.2018.2844341> (date of access: 11.06.2024).
24. Event-driven architecture. *Wallarm*. URL: https://cdn.prod.website-files.com/5ff66329429d880392f6cba2/661fbe4c588e5f74dfdd0bb2_553-min.jpg (date of access: 19.04.2024).
25. FS2: Functional Streams for Scala. *GitHub*. URL: <https://github.com/typelevel/fs2> (date of access: 24.04.2024).
26. Gurcan F., Berigel M. Real-Time Processing of Big Data Streams: Lifecycle, Tools, Tasks, and Challenges. *2018 2nd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, Ankara, 19–21 October 2018. 2018. URL: <https://doi.org/10.1109/ismsit.2018.8567061> (date of access: 05.04.2024).
27. Hueske F., Kalavri V. Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications. O'Reilly Media, 2019. 310 p.
28. Introduction to FS2: Functional Streams for Scala. *Baeldung*. URL: <https://www.baeldung.com/scala/fs2-functional-streams> (date of access: 24.04.2024).
29. Kleppmann M. Designing data-intensive applications. 4th ed. Sebastopol, CA : O'Reilly Media, 2019. 137 p. URL: <https://www.scylladb.com/wp->

- content/uploads/ScyllaDB-Designing-Data-Intensive-Applications.pdf
(date of access: 10.06.2024).
30. Kolajo T., Daramola O., Adebisi A. Big data stream analysis: a systematic literature review. *Journal of Big Data*. 2019. Vol. 6, no. 1. URL: <https://doi.org/10.1186/s40537-019-0210-7> (date of access: 03.04.2024).
 31. Manchana R. Event-Driven Architecture: Building Responsive and Scalable Systems for Modern Industries. *International Journal of Science and Research (IJSR)*. 2021. Vol. 10, no. 1. P. 1706–1716. URL: <https://doi.org/10.21275/sr24820051042> (date of access: 30.04.2024).
 32. Monix: Asynchronous Programming for Scala and Scala.js. Monix. URL: <https://monix.io/> (date of access: 10.04.2024).
 33. Netty project - an event-driven asynchronous network application framework. *GitHub*. URL: <https://github.com/netty/netty> (date of access: 04.05.2024).
 34. Online Anomaly Detection over Big Data Streams / L. Rettig et al. *Applied Data Science*. Cham, 2019. P. 289–312. URL: https://doi.org/10.1007/978-3-030-11821-1_16 (date of access: 03.04.2024).
 35. Reddy Kommera A. The Power of Event-Driven Architecture: Enabling RealTime Systems and Scalable Solutions. *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*. 2020. Vol. 11, no. 1. P. 1740–1751. URL: <https://doi.org/10.61841/turcomat.v11i1.14928> (date of access: 01.05.2024).
 36. Tantalaki N., Souravlas S., Roumeliotis M. A review on big data real-time stream processing and its scheduling techniques. *International Journal of Parallel, Emergent and Distributed Systems*. 2019. Vol. 35, no. 5. P. 571–601. URL: <https://doi.org/10.1080/17445760.2019.1585848> (date of access: 10.01.2025).
 37. The Scala Programming Language. URL: <https://www.scala-lang.org/> (date of access: 02.04.2024).
 38. Turnbull J. Monitoring with Prometheus. Turnbull Press, 2018. 381 p.

- 39.Varatharaj M. Scalable event-driven architectures for real-time data processing: a framework for distributed systems. *International journal of computer engineering and technology*. 2024. Vol. 15, no. 6. P. 1952–1965. URL: https://doi.org/10.34218/ijcet_15_06_167 (date of access: 02.05.2024).
- 40.Wunder C., Bahati H. Overview of Blocking vs Non-Blocking. *Node.js – Run JavaScript Everywhere*. URL: <https://nodejs.org/en/learn/asynchronous-work/overview-of-blocking-vs-non-blocking> (date of access: 01.05.2024).