КВАЛІФІКАЦІЙНА РОБОТА

Група МІПЗз-22

Луканюк В.В.

2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

на правах рукопису

Луканюк Володимир Володимирович

УДК 004.4

Оптимізація моделей масштабування процесів обробки потоків даних

Спеціальність 121 – «Інженерія програмного забезпечення»

Кваліфікаційна робота на здобуття кваліфікації магістра

Нормоконтроль	Студент
_____________Стисло О.В.	_____________ Луканюк В.В.
(підпис, дата, розшифрування підпису)	(підпис, дата, розшифрування підпису)
Допускається до захисту Завідувач кафедри	Керівник роботи
______________к.т.н., доц. Ващишак С.П.	______________ к.т.н., доц. Демчина М.М.
(підпис, дата, розшифрування підпису)	(підпис, дата, розшифрування підпису)

Івано-Франківськ – 2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

Освітній ступінь: «магістр»

Спеціальність: 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

____

« 19 » лютого 2024 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Луканюка Володимира Володимировича

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи

Оптимізація моделей масштабування процесів обробки потоків даних

керівник роботи:

Демчина Микола Миколайович, кандидат технічних наук, доцент

затверджена наказом вищого навчального закладу від « 26 » червня 2023 року

№ 32/1 с

2. Термін подання студентом роботи 16.02.2024

3. Вихідні дані роботи: Формальні моделі, методи та алгоритми.

4. Зміст кваліфікаційної роботи (перелік питань, які потрібно розробити)

   1. Огляд та аналіз підходів до побудови та контролю потоків даних.
   2. Моделювання послідовностей станів потоків даних.
   3. Дослідження процесів виконання завдань ітеративних процесів
   4. Реалізація підтримки ітераційних процесів на основі Apache Flink

5. Дата видачі завдання 29.06.2023

КОНСУЛЬТАНТИ РОЗДІЛІВ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Розділ	Консультант (прізвище, ініціали та посада)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1.	Огляд та аналіз підходів до побудови та контролю потоків даних.	26.09.2023	Виконано
2.	Моделювання послідовностей станів потоків даних	20.10.2023	Виконано
3.	Дослідження процесів виконання завдань ітеративних процесів	15.11.2023	Виконано
4.	Реалізація підтримки ітераційних процесів на основі Apache Flink	30.11.2023	Виконано
5.	Формування висновків	09.12.2023	Виконано
6.	Оформлення пояснювальної записки	22.12.2023	Виконано
7.	Оформлення графічного матеріалу та підготовка до захисту роботи	11.01.2024	Виконано

Студент ____________________ Луканюк В.В.

(підпис) (прізвище та ініціали)

Керівник роботи ____________________ Демчина М.М.

(підпис) (прізвище та ініціали)

Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Сторінка	Опис графічного матеріалу	Сторінка	Опис графічного матеріалу
13	Map-Reduce архітектура	49	Розподіл стану та альтернативи метаданих
17	Програмний стек Apache Flink	52	Походження програми зі знімками
20	Програма Flink Scala для аналізу датчиків температур	60	Огляд суворо скоординованих ітераційних процесів у системах пакетних обчислень
22	Рівні представлення у Flink	64	Огляд ітераційної обробки довгострокових завдань
23	Логічне, оптимізоване та фізичне представлення прикладу програми	65	Глобальне та децентралізоване відстеження прогресу

27	Приклад не інкрементного вікна визначення середнього	70	Зображення помічених low watermarking і похідної метрики прогресу
28	Приклад інкрементного вікна визначення середнього	71	Алгоритм для водяних знаків
29	Приклад періодичного багатовіконного середнього значення	73	Приклад low watermarking з інтервалом оновлення 10 с
32	Огляд потокової обробки на основі епохи	75	Алгоритм для відстеження прогресу
34	Приклад фіксації синхронної епохи	77	Повідомлення та low watermarking для кожного контексту в структурованому циклі
36	Асинхронно узгоджені епохи	79	Завершення контексту в структурованому циклі
38	Детальний огляд середовища виконання Flink	80	Огляд системних доповнень на основі Apache Flink
40	Розробка компонентів фізичних завдань	81	Змінні, доступні в контексті циклу (ctx)
42	Огляд Epoch Commit зі Snapshots	82	Внутрішні елементи оператора ітерації вікна
44	Приклад процедури відкату		

АНОТАЦІЯ

Кваліфікаційна робота присвячена оптимізації моделей масштабування процесів обробки потоків даних шляхом виконання порівняння продуктивності графових алгоритмів в найсучаснішій системі пакетної обробки, спеціалізованій системі аналізу графів і власному потоку даних системи, яка підтримує як масові, так і поетапні ітерації.

В першому розділі проаналізовано підходи до побудови, контролю та масштабування потоків даних, наведено опис платформи Apache Flink для обробки потоків даних. Здійснено огляд моделі та підходу обробки великих потоків даних на основі Apache Flink, наведено основні аспекти проектування систем контролю потоків даних.

В другому розділі виконано моделювання послідовностей станів для процесів масштабування обробки потоків даних, приведена концепція обробки потоків на основі епох та досліджено процес моделювання станів даних засобами розподіленого середовища. Приведені процеси реконфігурації системи для масштабування обробки потоків даних та ізоляції доступу для керованого стану даних.

В третьому розділі здійснена розробка моделей оптимізації ітеративних потоків даних, описана сутність ітераційної обробки та опис базової моделі ітеративних процесів. Наведені процеси виконання завдань ітеративних процесів, модель масштабування для позачергової ітеративної обробки даних і представлена реалізація підтримки ітераційних процесів на основі Apache Flink.

КЛЮЧОВІ СЛОВА: ПОТІК ДАНИХ, ІТЕРАТИВНИЙ ПРОЦЕС, КОНТРОЛЬ ОБЧИСЛЕНЬ, МОДЕЛЬ ПРОЦЕСУ, РОЗПОДІЛЕНЕ СЕРЕДОВИЩЕ, МАСШТАБУВАННЯ, МОДЕЛЮВАННЯ СТАНУ

SUMMARY

The qualification work is devoted to the optimization of scaling models of data flow processing processes by performing a comparison of the performance of graph algorithms in a state-of-the-art batch processing system, a specialized graph analysis system, and the system's own data flow, which supports both mass and stepwise iterations.

The first chapter analyzes approaches to building, controlling, and scaling data streams, and describes the Apache Flink platform for processing data streams. A practical overview of Apache Flink-based models and approach to processing large data flows, the main aspects of designing a data flow control system are given.

In the second chapter, stateful modeling of data stream processing scaling processes is performed, the concept of epoch-based stream processing is presented, and the process of data state modeling using a distributed environment is explored. System reconfiguration processes for scaling data flow processing and access isolation for managed data state are given.

In the third section, the development of optimization models for iterative data flows is carried out, the essence of iterative processing and the description of the basic model of iterative processes are described. Iterative task execution processes, a scaling model for out-of-order iterative data processing, and an implementation of supported iterative processes based on Apache Flink are presented.

KEY WORDS: DATA FLOW, ITERATIVE PROCESS, COMPUTATION CONTROL, PROCESS MODEL, DISTRIBUTED ENVIRONMENT, SCALING, STATE SIMULATION

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 9
ВСТУП 10
РОЗДІЛ 1. АНАЛІЗ ПІДХОДІВ ДО ПОБУДОВИ, КОНТРОЛЮ ТА МАСШТАБУВАННЯ ПОТОКІВ ДАНИХ 13
1.1 Аналіз підходу обробки та масштабування великих потоків даних 13
1.2 Опис платформи Apache Flink для обробки потоків даних 16
1.3 Огляд моделі та підходу обробки великих потоків даних на основі Apache Flink 19
1.3 Основні аспекти проектування систем контролю потоків даних 24
Висновки до розділу 1 31
РОЗДІЛ 2. МОДЕЛЮВАННЯ ПОСЛІДОВНОСТЕЙ СТАНІВ ДЛЯ ПРОЦЕСІВ МАСШТАБУВАННЯ ОБРОБКИ ПОТОКІВ ДАНИХ 32
2.1 Концепція обробки потоків на основі епох 32
2.2 Дослідження процесу моделювання станів даних засобами розподіленого середовища 37
2.2.1. Розробка архітектури 38
2.2.2. Розробка завдання та модель процесу 40
2.2.3. Реалізація протоколу 41
2.2.4. Процедура відкату 44
2.2.5. Серверна інтеграція 46
2.3 Процеси реконфігурації системи для масштабування обробки потоків даних 48
2.4 Ізоляція доступу для керованого стану даних 50
Висновки до розділу 2 53
РОЗДІЛ 3. РОЗРОБКА МОДЕЛЕЙ ОПТИМІЗАЦІЇ ІТЕРАТИВНИХ ПОТОКІВ ДАНИХ 54

3.1 Сутність ітераційної обробки та опис базової моделі ітеративних процесів 54
3.1.1. Дослідження процесу ітеративної обробки 54
3.1.2. Представлення базової моделі для ітеративних процесів 56
3.1.3. SGD і обмежена асинхронність 58
3.2 Опис процесів виконання завдань ітеративних процесів 60
3.2.1. Короткотривале виконання завдань ітераційних процесів 60
3.2.2. Довготривале виконання завдань ітераційних процесів 63
3.3 Модель масштабування для позачергової ітеративної обробки даних 67
3.3.1. Позачергова обробка потоку 67
3.4 Децентралізоване відстеження прогресу для часткового порядку подій обробки даних 73
3.5 Реалізація підтримки ітераційних процесів на основі Apache Flink 80
Висновки до розділу 3 83
ВИСНОВКИ 84
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 85

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

DSMS - Data Stream Management System

DSEL - Domain Specific Language

OOP - Out-of-Order Processing

MVCC - Multiversion Concurrency Control-Enabled

WAL - Write-Ahead-Log

SSSH - Single Source Shortest Paths

SGD - Stochastic Gradient Descent

BSP - Bulk Synchronous Processing

ВСТУП

Актуальність теми дослідження. Системи паралельних потоків даних стають все більш популярним рішенням для аналізу великих обсягів даних. Вони пропонують просте програмування абстракцій, засноване на орієнтованих ациклічних графах, що звільняє програміста від роботи зі складними завданнями планування обчислень, передачі проміжних результатів і роботи з помилками. Найважливіше те, що вони дозволяють поширювати програми потоків даних на великій кількості машин, що є обов’язковим, якщо мати справу із сьогоднішніми обсягами даних. Окрім паралельних баз даних, MapReduce є найвідомішим представником, популярним завдяки своїй застосовності за межами реляційних даних.

Хоча системи потоку даних спочатку створювалися для таких завдань, як індексування, фільтрація, перетворення чи агрегування даних, їхній простий інтерфейс і потужна абстракція зробили їх популярними для інших види додатків, наприклад машинне навчання або аналіз графів. Багато з цих алгоритмів є ітеративними або рекурсивними, повторюючи деякі обчислення, доки умова не буде виконана. Звичайно, ці завдання створюють виклик для систем потоку даних, якщо потік даних більше не є ациклічним. Протягом останніх років з’явилася низка рішень для визначення та виконання ітераційних алгоритмів як потоків даних. MapReduce розширення, як-от Twister або HaLoop, і такі фреймворки, як Spark здатні ефективно виконувати певний клас ітерацій алгоритмів. Однак багато алгоритмів машинного навчання та графів все ще працюють погано через нездатність цих систем використовувати (розріджені) обчислювальні залежності, присутні в цих завданнях.

Будемо називати повторно обчислений стан – частковим рішенням ітерації, і надалі розрізнятимемо наступні типи ітерацій:

- Масові ітерації: кожна ітерація обчислює абсолютно нове часткове рішення з результату попередньої ітерації; необов'язкове використання додаткових наборів даних, які залишаються постійними в курсі ітерації. Яскравими прикладами є машинне навчання, такі алгоритми як Batch Gradient Descend (Стохастичний градієнтний спуск), багато алгоритмів кластеризації (наприклад, K-Means), і добре відомий алгоритм PageRank.
- Поступові ітерації: результат кожної ітерації лише частково відрізняється від результату попередньої ітерації. Розріджені обчислювальні залежності існують між елементами часткового рішення: оновлення одного елемента має прямий вплив лише на невелику кількість інших елементів, таких, що різні частини розрідження можуть сходитися з різною швидкістю. Приклад – це алгоритм підключених компонентів, де зміна та членство компонентів вершини безпосередньо впливає лише на членство своїх сусідів.

Мета і завдання дослідження. Метою кваліфікаційної роботи є дослідження моделей та методів масштабування процесів обробки великих потоків даних та оптимізації процесів надійного виконання зі збереженням стану обробки, що надаються системою потокової обробки за наявності часткових збоїв або потребою реконфігурації даних.

Об'єктом дослідження є системи паралельної обробки потоків даних, як частини аналітичних конвеєрів великих даних.

Предметом дослідження є сукупність теоретичних, аналітичних підходів, методів та моделей обробки потоків даних з метою оптимізації процесів їх масштабування.

Методи дослідження базуються на використанні методів ітеративного моделювання та обробки потоків даних, методів та засобів графових представлень, методів часткового упорядкування даних, методів моделювання множин станів та масштабування потоків даних.

Для досягнення поставленої мети необхідно розв'язати такі задачі:

- виконати аналіз підходів до побудови, контролю та масштабування потоків даних;
- провести моделювання послідовностей станів даних;
- дослідити концепцію обробки потоків на основі епох;
- виконати розробку моделей ітеративних потоків даних;
- представити модель масштабування для позачергової ітеративної обробки даних.

Наукова новизна одержаних результатів полягає представленні методу оптимізації на основі розробленої моделі для виконання розподіленого потоку, яка ділить обчислення на епохи, кожна з яких завершується атомарно, і відповідно після кожної епохи повний стан системи доступний для цілого ряду потреб управління станом, таких як усунення несправностей і реконфігурація.

Практичне значення одержаних результатів полягає в дослідженні концепції ітераційних процесів із визначенням універсального набору примітивів програмування, які використовуються для їх опису та представлення проблеми реалізації цих примітивів у розподіленій архітектурі як розширення моделі потокової обробки поза порядком для циклічних обчислень.

Апробація результатів дослідження. Матеріали дослідження було представлено у матеріалах I Всеукраїнської науково-практичної інтернет конференції “IT екосистема: цифровізація бізнес-процесів в умовах війни”, у тезах доповіді “Хмарне масштабування процесів обробки потоків даних”.

Структура. Кількість розділів – 3. Загальний обсяг основної частини – 90 сторінок. Список використаних джерел містить – 52 позиції.

РОЗДІЛ 1. АНАЛІЗ ПІДХОДІВ ДО ПОБУДОВИ, КОНТРОЛЮ ТА МАСШТАБУВАННЯ ПОТОКІВ ДАНИХ

1.1 Аналіз підходу обробки та масштабування великих потоків даних

Останні досягнення в розподілених і хмарних обчислювальних системах привели сферу управління даними в еру «великих даних», яка відома критичними змінами парадигми. Значне зниження вартості зберігання та обчислювальних ресурсів, а також поява службових обчислень сприяли масовій розробці та прийняттю систем, що масштабуються.

Map-Reduce (рис. 1.1) став визначним етапом переходу до управління даними, оскільки він перетворив робочі навантаження масової обробки даних із монолітного виконання запитів до бази даних на розподілену модель виконання завдань, що підходить для системних архітектур без спільного використання.

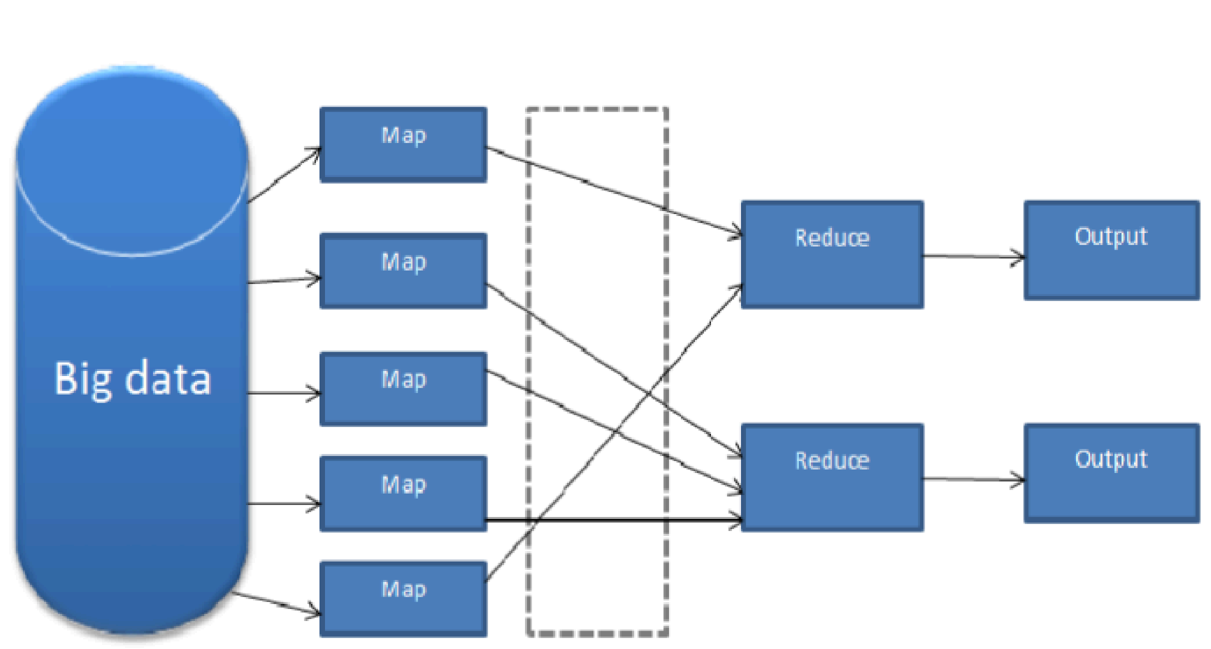


Рисунок 1.1 – Map-Reduce архітектура

Незважаючи на велику увагу до масштабованості та відмовостійкості, Map-Reduce критикували за використання незручних обмежень, зокрема щодо забезпечення поетапного виконання між кожним обчислювальним кроком, а також обмеження програм ациклічними обчислювальними графами завдань без стану. Останній зробив ітераційні програми, такі як інтерактивне виконання запитів, машинне навчання та робочі навантаження обробки графіків, недоступними без будь-якої форми обміну даними між завданнями та відкрив область масштабованих систем спеціального призначення, які усувають деякі з цих обмежень. Apache Spark можна вважати безпосереднім наступником технологій, пов’язаних із Map-Reduce, оскільки він доповнив існуючі основні властивості дизайну, такі як масштабованість і відмовостійкість, одночасно подолавши обмеження моделі програмування, які існували виключно як вибір дизайну. Зокрема, застосування Spark лінійних графів продемонструвало, що обмін даними може бути досягнутий без додаткових витрат у розподіленій архітектурі, тоді як етапи можуть використовувати реплікацію в пам’яті замість розподіленої файлової системи.

Наступний радикальний зсув в управлінні даними, який є основним напрямком є "масштабованість потокової обробки". Сама обробка потоку бере свій початок у системах, заснованих на подіях [1], як-от проміжне програмне забезпечення для публікації та підписки, а також управління потоком даних (DSMS - Data Stream Management System) із галузі дослідження баз даних. Відмінною характеристикою цього системного класу є поняття самих даних, тобто безперервних, можливо, нескінченних замість «фактів та статистичних даних, організованих та зібраних разом для подальшого використання або аналізу». Попередні форми потокової обробки можна спостерігати у відповідних областях, таких як мережеве програмування на байтових потоках, функціональне програмування, комплексна обробка подій і матеріалізовані представлення бази даних. Управління потоками також було активною дослідницькою сферою протягом десятиліть [2, 3, 4]. Тим не менш, деякі з цих ідей лише нещодавно були

послідовно об’єднані для створення нових стеків програмного забезпечення (включаючи зберігання, доставку, обробку та предметно-спеціальні мови) для написання масштабованих програм, зосереджених навколо поняття даних як необмеженого ресурсу.

Декілька популярних розробок із відкритим кодом у технології потокової обробки були поступовими підходами, які в першу чергу мали на меті запропонувати альтернативу MapReduce із низькою затримкою для обробки в реальному часі, де надійність і потужність виразності віддавали пріоритет швидкості. Система Storm від Twitter [5] є прикладом масштабованої потокової системи даних з обмеженими гарантіями, яка стала популярною приблизно в той час, коли почалася робота над цією дисертацією. Подібним чином академічні дослідження, пов’язані з обробкою потоків, зосереджувалися на наближених структурах даних [6] і спеціалізованих областях застосування (наприклад, обробка складних подій), що також сприяло тому ж неправильному уявленню. Це призвело до створення догми, яка називала потокове передавання даних приблизною технікою аналізу даних. Лямбда-архітектура [7] підкреслила цю ідею, розділивши проблеми надійної великомасштабної обробки даних і потокової обробки з використанням «пакетного» і «швидкісного» рівня відповідно. Тим не менш, нещодавні зусилля, включно з роботою, представленою в цій дисертації, переглянули цю технологію та повернули її на поверхню не як нішевий чи додатковий підхід до сучасного рівня, а як радикально загальну пропозицію до надійного програмування, орієнтованого на дані., здатний включити існуючі моделі програмування (наприклад, MapReduce [8]). Насправді потокова обробка розширила, а не обмежила сферу управління даними від ретроспективного аналізу даних до безперервної, необмеженої та масштабованої обробки в поєднанні з постійним станом.

У цій роботі ми досліджуємо раціональний дизайн системи для вирішення деяких найбільш помітних відкритих проблем у масштабованому потоковому передачі даних:

- Відмовостійкість і масштабоване управління станом;
- Спільне використання обчислень і семантика даних;
- Ітераційне потокове передавання даних.

Хоча часткові рішення є вже відомими із цих проблем, натомість ми шукаємо механізми, які не накладають непотрібних компромісів у продуктивності та відокремлюють питання часу виконання від будь-якої моделі програмування, орієнтованої на користувача.

У цій роботі ми інкапсулюємо ці вимоги через набір прийнятих властивостей дизайну: уникнення блокування-координації, прозорість програмної моделі та композиційність. Також розглянемо ядро Apache Flink, системи обробки потоків з відкритим кодом, яка мала великий успіх завдяки широкому застосуванню в даній галузі.

1.2 Опис платформи Apache Flink для обробки потоків даних

Apache Flink є проектом Apache верхнього рівня з відкритим вихідним кодом. Flink забезпечує стек для програмування, компіляції та запуску розподілених безперервних програм. Програми у Flink оцінюються після надсилання до середовища виконання для виконання. По суті, програма Flink — це декларативний засіб розподіленої програми та зазвичай складається з низки перетворень потоку даних. У свою чергу, виконувані програми є реконфігурованими, розподіленими графіками довгострокових виконуваних завдань, які плануються та постійно контролюються Flink. У порівнянні з іншими відомими обчислювальними фреймворками [8, 27], які керуються короткостроковим виконанням, керованим планувальником, Flink виділяє ресурси обчислювального кластера для програм, поки ці програми виконуються, також відомий як «тривале виконання завдання». На рисунку 1.2 ми розбиваємо стек програмного забезпечення Flink від його бібліотек високого домену до виконання під час виконання. Також ми представимо

огляд Apache Flink і далі зосереджуємося на компонентах, щодо обробки даних.

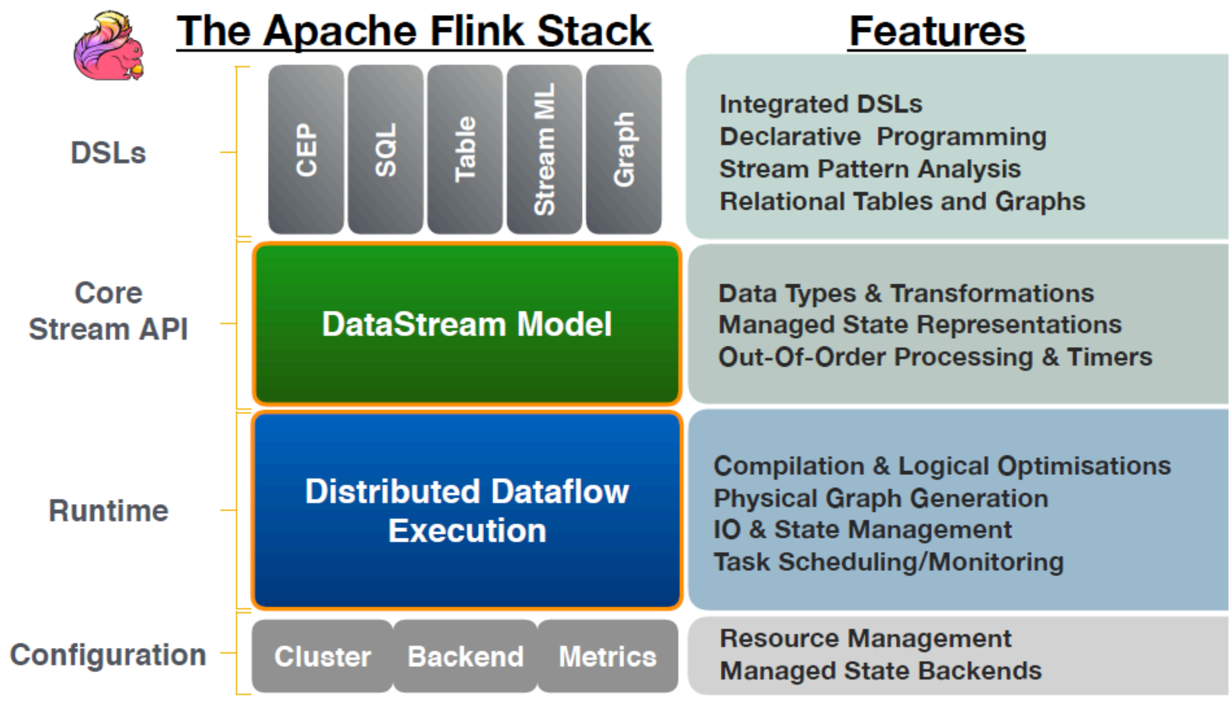


Рисунок 1.2 – Програмний стек Apache Flink

Flink забезпечує підтримку програмування на основі потоку як «неглибокої» доменно-специфічної мови (DSEL - Domain Specific Language) через свій API, наданий у Scala та Java. Його інтерфейс програмування складається з основного API та ряду спеціалізованих бібліотек, побудованих на його основі, які забезпечують абстракції програмування спеціального призначення, такі як підтримка SQL. Основний Stream API надає основні примітиви для створення додатків на основі подій, таких як керований стан додатків і таймери, а також наскрізне створення додатків із перетвореннями потоку за допомогою DataStream API. DSL від Flink включають Gelly для обробки графів, FlinkML для моделей пакетного навчання та онлайн-обслуговування моделей, Table та Stream SQL для постійних реляційних запитів із представленнями таблиць, а також CEP декларативний

DSL для вираження комплексної обробки подій із збереженням стану та зіставлення шаблонів у потоках подій.

Середовище виконання Flink — це розподілена система, яка планує та підтримує додатки та відповідні їм завдання. Як було сказано вище, Flink використовує багатофункціональні архітектурні структури. Це означає, що завдання розподіляються за доступними слотами та постійно виконуються у виділених потоках, доки програма не вимкнеться вручну або не буде потрібно змінити конфігурацію. Існує три типи компонентів середовища виконання, які керують кластером Flink: Client, JobManager і TaskManagers.

Клієнтський модуль компілює та оптимізує кожну програму користувача в логічне графічне представлення, яке можна надіслати в JobManager для виконання.

JobManager є головним процесом у розгортанні Flink і повністю контролює програму, а також відповідає за фізичний переклад, розгортання та моніторинг кожного логічного графіка, поданого на виконання.

TaskManager — це процес, розгорнутий на кожному хості, який виконує все керування локальними ресурсами (наприклад, мережа/диск ввід-вивід, буфери пам'яті, розподіл слотів, конфігурація), необхідні для локально запущених завдань. Середовище виконання Flink періодично передає прогрес системи до зовнішнього стабільного сховища та інших систем за допомогою транзакційного протоколу атомарної фіксації, який виконується одночасно з кожною розподіленою програмою.

Flink дозволяє конфігурувати певні зовнішні модулі на вимогу, як правило, для кожного розгортання Flink або окремого додатка за допомогою файлів конфігурації. Сюди входять бекенди стану, які є підсистемами, спеціалізованими для матеріалізації таких операцій стану (наприклад, Rocksdb [47]). Інші модулі, зокрема, призначені для збору метрик і керування кластерами (наприклад, YARN і Mesos [48,49]).

У цьому розділі ми зосереджуємося на семантиці програмування Flink, компіляції та перекладі на графіки виконання завдань. У наступному розділі

ми надамо детальний опис певних компонентів середовища виконання та механізмів, що представляють інтерес, таких як Epoch Commit Protocol, який керує керуванням станом Flink.

1.3 Огляд моделі та підходу обробки великих потоків даних на основі Apache Flink

Потокові програми в Apache Flink нагадують плавний функціональний синтаксис програмування Apache Spark [27]. Основний API Flink надає набір базових абстрактних типів, таких як DataStream і WindowedStream, кожен з яких підтримує перетворення, які є функціями вищого порядку, такими як filter, map, flatmap і reduce. В принципі, кожне перетворення представляє оператор логічного графа, який має залежності даних від інших операторів у концептуальному графі (потоку даних). На прикладі ми продемонструємо основні можливості моделі програмування Flink, а також шлях від компіляції до розподіленого виконання.

Приклад: розглянемо варіант використання, коли набір різних датчиків періодично генерує та надсилає в розділений журнал (наприклад, чергу Kafka) свою поточну температуру. Припустімо, що ми хочемо створити надійну службу, яка зчитує ці події та виконувати наступне:

1) кожні 8 хвилин обчислює середню температуру, виміряну кожним датчиком протягом 1 години;

2) повідомляє попередження, якщо середня температура перевищує 40 градусів;

3) ігнорування початкових 5 вимірювань датчика протягом періоду його калібрування.

У лістингу (рис. 1.3) ми представляємо програму Flink, яка реалізує вищезгаданий сервіс аналізу подій датчика, використовуючи різні основні стилі програмування та функції, надані у рамках. Програма Flink починається з оголошення одного або кількох джерел даних. Джерелом даних може бути

обмежений ресурс даних, наприклад текстовий файл, або необмежений, наприклад активний журнал повідомлень. Фреймворк Flink уже надає набір інтегрованих з’єднувачів, які можна використовувати з коробки, у цьому випадку ми використовуємо з’єднувач Kafka, який споживає повідомлення з черги повідомлень, ідентифікованих як «sensorTemperatures». Основна логіка програми Flink складається з набору конвеєрних перетворень.

```
1  case class SensorEvent(sensorID: Long, temperature: Double);
2  case class TemperatureWarning(sensorID: Long, temperature: Double);
3
4  //STREAM SOURCE
5  val sensors : DataStream[SensorEvent] = env.addSource(KafkaConsumer("sensorTemperatures"));
6  //----------------------------------------------------------
7  //STREAM TRANSFORMATIONS
8  val filteredEvents :DataStream[SensorEvent] =
9    //Distribute Computation by SensorID
10   sensors.keyBy{_.sensorID}
11   //Ignore the first 5 temperatures per sensor
12   .filterWithState((evt:SensorEvent, count: Option[Int]) =>
13       count match{
14         case Some(c) => (c > 5, Some(c+1))
15         case None => (true, Some(1))
16       });
17 val avgTempStream: DataStream[SensorEvent] =
18   //Compute 1h rolling average of temperature/sensor every 8min
19   filteredEvents.timeWindow(Time.hours(1), Time.minutes(8))
20   .aggregate(AverageTemperature());
21   //Output warnings for average temperatures above 40 degrees
22 val warnings: DataStream[TemperatureWarning] =
23   avgTempStream.flatMap((evt, collector) => if(evt.temperature > 40)
24     collector.collect(TemperatureWarning(evt.sensorID, evt.temperature)));
25 //----------------------------------------------------------
26 //STREAM SINK
27 warnings.addSink(KafkaProducer("tempWarnings"));
28
29 //EXECUTION
30 env.execute()
```

Рисунок 1.3 – Програма Flink Scala для аналізу датчиків температур

Операція keyBy (рядок 10) оголошує ключ, який використовується для розділення обчислень і стану. Незважаючи на те, що його семантика виглядає схожою на реляційну groupBy, keyBy не групує події, а скоріше визначає, як події будуть перемішуватися в різні екземпляри оператора (події одного ключа обробляються одним екземпляром) і як установлюється стан, за допомогою цього ключа. Ефект keyBy стає більш зрозумілим у перетворенні

filterWithState (рядок 12), яке параметризовано за допомогою визначеної користувачем функції типу (event,state) —> (bool,newState), яка дає значення true для подій, які залишаються в потоці поряд із а новий стан (інстанційований ключ події). Оголошена функція підтримує кількість подій на датчик як стан і відфільтровує події з кількістю менше 5 (рядок 14). Під час виконання програми Flink автоматично керує постійним станом кожного ключа (ідентифікатора датчика), який у цьому випадку є лише цілим значенням (допускаються також складні типи).

Розділені відфільтровані події потім надходять у віконну агрегатну трансформацію (рядки 19 і 20). Трансформація timeWindow дискретизує потік подій на набори на основі часу застосування, який вони генерують, і застосовує функцію середнього значення до цих наборів, щоб отримати об’єкт SensorEvent кожного датчика на вікно, що містить середнє значення температури. Після цього функція flatMap (рядок 24) генерує подію TemperatureWarning для кожного агрегату, температура якого перевищує порогове значення (40 градусів). Зауважимо, що у функції flatMap, яка дозволяє відображати кожну вхідну подію на нуль або більше вихідних подій. У цьому випадку вихідний колектор передається до визначеної користувачем функції, яка представляє вихідний буфер, який використовується для вибіркового очищення вихідних подій у логіці імперативного потоку керування (у межах літералів функціонального програмування). Нарешті, визначається приймач потоку Kafka (рядок 27) для фіксації виходу обчислення, тобто згенерованого попередження.

Огляд компіляції та виконання. Кожна програма у Flink проходить трифазову компіляцію перед виконанням у розподіленому середовищі виконання, що створює 1) логічне, 2) оптимізоване та 3) графове (фізичне) представлення відповідно як показано на рисунку 1.4.

Логічне представлення: кожне перетворення в програмі нагадує логічний оператор, який виконує логіку на основі подій. Коли програма досягає команди виконання (рядок 30) , середовище виконання Flink створює

на стороні клієнта графічне представлення всіх логічних операторів у програмі. Логічне представлення (рис. 1.4 (a)) є орієнтованим ациклічним графом, вузли джерела та приймача якого представляють ті, що оголошені в програмі. Оператор може інкапсулювати логіку одного перетворення (наприклад, фільтр, плоска карта, вікно). Однак стандартні оптимізації логічного потоку даних, такі як злиття, також застосовуються на проміжному етапі, що дозволяє об’єднати кілька операторів в один (рис. 1.4 (b)).

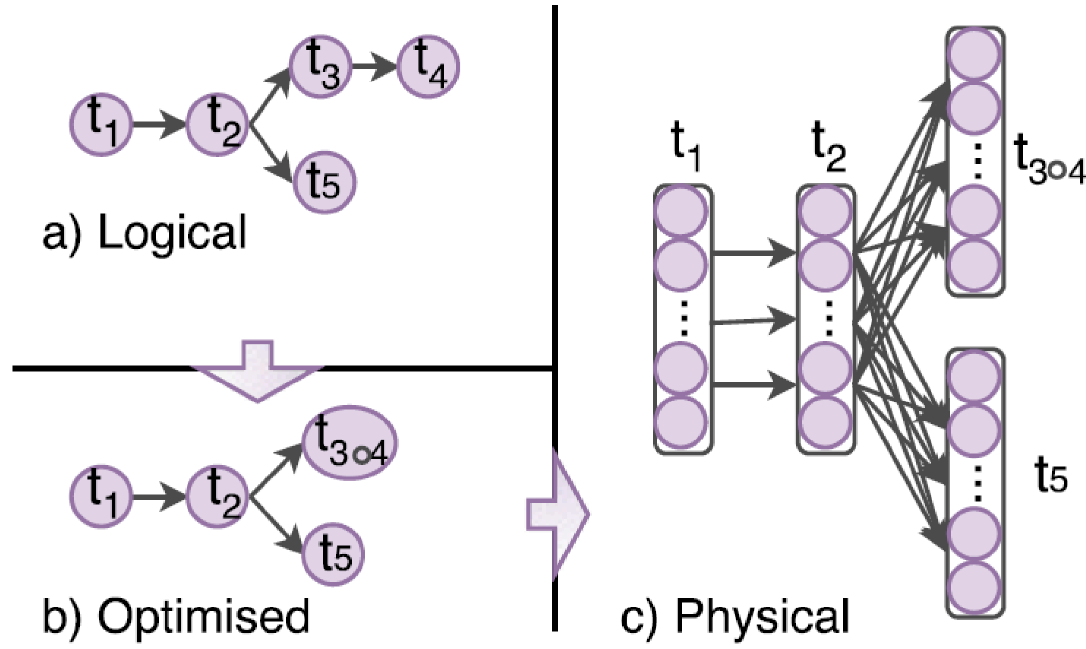


Рисунок 1.4 – Рівні представлення у Flink

Фізичне представлення: Відображення логічного графа, фізичного, розподіленого графа виконання (Рис. 1.4 (c)) відбувається, коли конвеєр розгортається, під час його початкового запуску або після реконфігурації (наприклад, для масштабування). На цьому етапі кожен логічний оператор є відображається на певну кількість фізичних одиниць, які ми називаємо «задачами», кожен з яких розгортається в доступні контейнери по всьому кластеру (наприклад, за допомогою YARN [48] або Mesos [49]) відповідно до вирішеного (або передбачуваного/доступного) ступеня паралелізму. Залежності даних між розгорнутими завданнями в представлені

налаштованими каналами, які дозволяють завданням надсилати та отримувати повідомлення під час їх виконання (наприклад, за допомогою збирача в логіці оператора).

Приклад виконання. На рисунку 1.5 подано кожну фазу прикладу програми, показаного раніше в лістингу (рис. 1.3), починаючи від її логічного представлення до кожного фізичного розгортання та виконання.

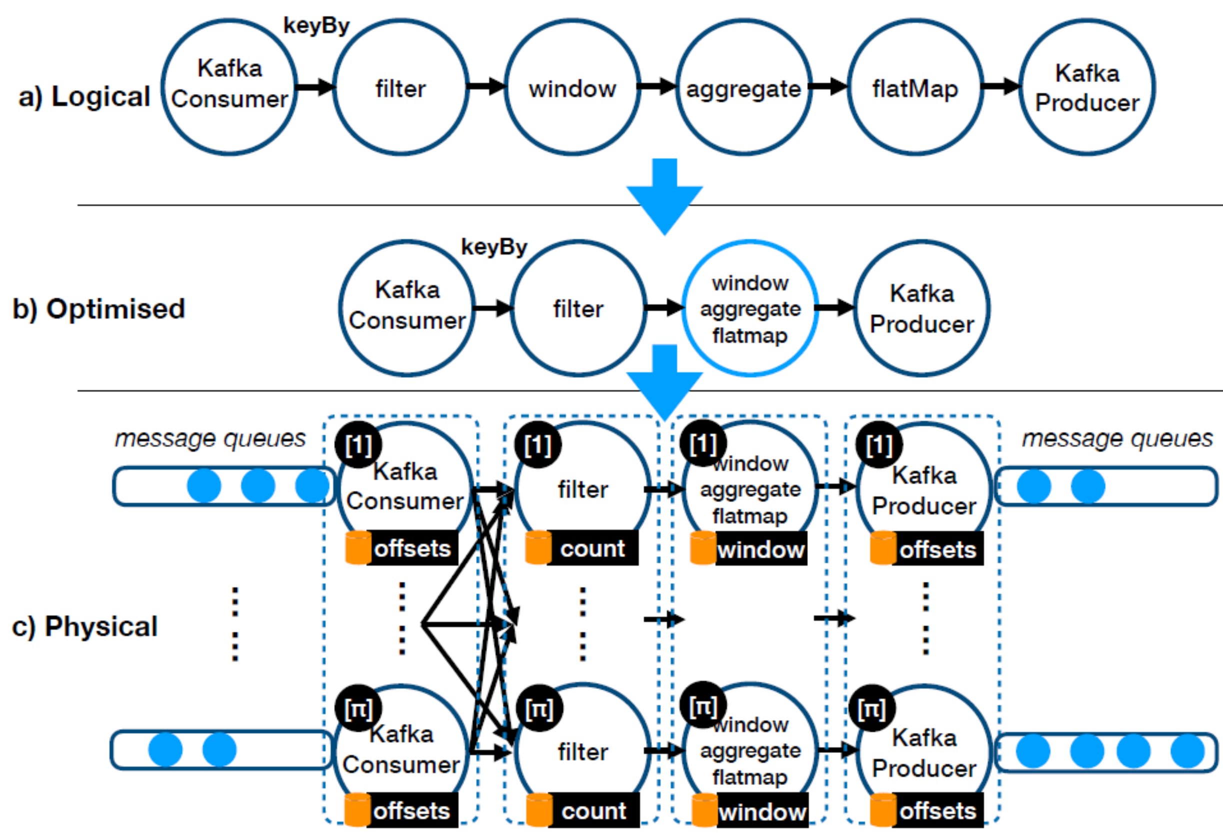


Рисунок 1.5 – Логічне, оптимізоване та фізичне представлення прикладу програми

У оптимізованому графі оператори window, aggregate і flatMap об’єднані в єдиний оптимізований оператор (рис. 1.5 (b)). Поєднання логіки агрегації вікна з його дискретизацією, як правило, необхідно, щоб уникнути непотрібних і надлишкових обчислень. Оператор flatMap також включено в той самий об’єднаний оператор, оскільки між ними не визначено жодної підписки keyBy чи інших паралельних даних.

Кінцевий фізичний граф (рис. 1.5 (c)) складається з кількох екземплярів кожного оператора, кожен з яких ми зазвичай називаємо завданням. Кожне завдання викликається після вхідного повідомлення в одному зі своїх каналів, запускає локальне обчислення, яке може отримати доступ до його стану для читання та запису, а потім генерує вихідні повідомлення в одній дії обробки. Прикладами стану з нашого зразка програми є підрахунки, які зберігаються в операторі filterWithState, або стан кожного вікна (згруповані записи та їхні агрегати). Виробники та споживачі Kafka також зберігають внутрішній стан (наприклад, зміщення журналу), який є необхідним для протоколу транзакційної фіксації Flink. Стан зазвичай розкривається для кожного ключа, однак кожне завдання статично виділяється групою ключів, за які воно відповідає. Це означає, що:

1) усі записи відповідного ключа спрямовуються середою виконання системи до відповідного фізичного завдання;

2) записи стану всіх ключів розділені в одному фізичному розділі, де знаходиться відповідальне завдання.

Точна специфікація базової моделі виконання, яка використовується в кожному завданні, детально буде описана а наступному розділі разом із загальносистемною моделлю транзакційної обробки.

В даній роботі ми будемо використовувати термін «завдання» для позначення фізичних завдань (виконання), а термін «оператор» — для позначення або логічного оператора або логіки, яка виконується в межах завдання. Наприклад, оператор вікна розгортається в завданнях, які виконують логіку оператора вікна (або просто оператор вікна) відповідно до кожного завдання.

1.3 Основні аспекти проектування систем контролю потоків даних

Кожне перетворення потоку у Flink може оголошувати стан, який є постійним і завжди доступним для операцій читання та запису. Стан є

основним блоком програми, оскільки він інкапсулює в будь-який час контекст кожного обчислення.

Типи станів: концептуально існують два масштаби, в яких працює керований стан. Для чисто паралельних потокових операцій з даними, таких як, наприклад, середнє значення за ключем, обчислення, його стан і пов'язані потоки можуть мати логічну область і працювати незалежно для кожного ключа (тобто, коли використовується keyBy). Ми називаємо цей стан Keyed-State. Для локальних обчислень для кожного завдання, таких як, наприклад, навчальна модель машинного навчання або споживання розділеного журналу на основі зміщення (наприклад, джерела даних Kafka), стан також можна оголошувати на рівні фізичного завдання, відомому як Operator-State. І Keyed-State і Operator-State прозоро розділені та керовані середовищем виконання системи та в більшості випадків локально доступні для кожного завдання.

Вбудований стан: вибір вбудованого стану був одним із перших проектних рішень, прийнятих на початку створення системи. У роботах [10, 11] було доведено, що цей конкретний вибір дизайну добре працює завдяки уникненню виклику віддаленого доступу до стану, однак він також створив проблеми. Спочатку найбільш критичною проблемою у Flink була потреба у відмовостійкості. Програми Flink призначені для безперервної роботи протягом довільних періодів часу, від хвилин до років, тому ймовірність відмови машини під час життя програми надзвичайно висока. Наявність набору неузгоджених, незалежно виконуваних завдань і відсутність глобальної одиниці чи «кроку» в обчисленнях, таких як дуже приваблива модель RDD Spark [12], ускладнювала та невизначила роботу Flink (під час його фази прототипу). На додачу до локального стану, надійна фіксація наскрізних побічних ефектів (наприклад, вихідних повідомлень із приймачів) також була відкритою проблемою протягом багатьох років. Найбільші користувачі Flink також вимагали підтримки масштабування у своїх програмах, маючи можливість змінювати конфігурацію своїх завдань на

льоту та змінювати паралелізм або просто переміщувати завдання в нові контейнери.

Загальна робота над управлінням станом мала на меті запропонувати універсальний підхід до всіх цих питань управління станом, доповнюючи оригінальну детальну модель виконання завдань Flink, не жертвуючи при цьому коректністю програм. У наступному розділі досліджується формальна системна специфікація та опис базової моделі виконання «Асинхронного епохального коміту», яка базується на здатності фіксувати повну конфігурацію (тобто стани завдань) розподіленої програми без зупинки обчислювальних завдань або застосування будь-яких засобів блокування синхронізації. У той час як існуючі протоколи реалізують миттєві знімки асинхронно (наприклад, алгоритм Чанді та Лемпорта [28]), жодне з попередніх рішень не вирішувало проблему захоплення розподілених станів на повних попередньо визначених обчислювальних межах раніше, більше обмежена формою знімка. Також ми дослідимо як працювати з циклами в графі обробки фізичного потоку, а також визначимо необхідні властивості безпеки та живучості моментального знімка на основі епох навчання.

Спільне використання віконних обчислень: у Flink та інших потокових процесорах [42, 39], вікна потоку служать єдиним способом групувати безперервні дані в набори, що розвиваються і виконують обчислення на цих наборах. У попередньому прикладі лістингу 1.1 ми коротко продемонстрували використання вбудованих періодичних вікон програми. Періодичні вікна зазвичай оголошуються за допомогою діапазону. У цьому ж прикладі ці параметри становили 1 годину та 8 хвилин відповідно. Більш уважне спостереження за цією конкретною операцією може виявити надзвичайно зайву кількість базових обчислень, задіяних у вікнах, що накладаються. Наприклад, якщо ми розглянемо середню швидкість 1000 записів на секунду під час застосування, це означатиме, що 52 із 60 хвилин на вікно відповідають обчисленню, яке було повторно обчислено в минулому, приблизно 52000 агрегацій. Насправді, це лише поверхневий аналіз, коли

мова заходить про основні потреби спільного використання віконних обчислень, які ми можемо узагальнити, відсортувавши за складністю (зростанням) наступним чином:

1. Часткове агрегування вікон: ця проблема обмежена тим, як ми оцінюємо кожне окреме вікно. Існує дві стратегії: а) зберігати всі записи вікна, доки воно не буде завершено, а потім запускати обчислення для всього вікна; б) використовувати частковий агрегат, який оновлюється поступово для кожного вхідного запису.

Вікна є обмеженими наборами, але потенційно можуть бути як завгодно великими (наприклад, кожне містити мільярди записів), тому стратегія б) зазвичай є кращою, оскільки вона усуває мету підтримки довільного стану вікна та гарантує постійні вимоги до простору. Це можливо, коли функція агрегації є розподільною та соціативною. Модель програмування Flink забезпечує підтримку обох стратегій за допомогою ProcessFunction і AggregationFunction відповідно. На рисунках 1.6 і 1.7 ми показуємо приклад обох альтернатив відповідно для того самого середнього вікна температур датчика, представленого раніше на рисунку 1.3. Інкрементна версія дозволяє користувачеві розбити агрегацію на тип часткової сукупності (у цьому випадку (Double, Long), що відповідає сумі та підрахунку) та остаточний результат, витягнутий із цього типу (середнє значення).

```
1  val avgTempStream: DataStream[SensorEvent] =
2    filteredEvents.timeWindow(Time.hours(1), Time.minutes(8))
3    .process(AverageTemperature());
4
5  class AverageTemperature extends ProcessWindowFunction[SensorEvent, SensorEvent, Long,
       TimeWindow] {
6    def process(key: Long, context: Context, input: Iterable[SensorEvent], out:
         Collector[SensorEvent]): () = {
7      var sum = 0.0;
8      var count = 0L;
9      for (evt <- input) {
10       sum = sum + evt.temperature;
11       count = count + 1;
12     }
13     out.collect(SensorEvent(key, sum / count))
14   }
15 }
```

Рисунок 1.6 – Приклад не інкрементного вікна визначення середнього

```
1  val avgTempStream: DataStream[SensorEvent] =
2    filteredEvents.timeWindow(Time.hours(1), Time.minutes(8))
3    .aggregate(AverageTemperature(), ResultEvent());
4
5  class AverageTemperature extends AggregateFunction[SensorEvent, (Double, Long), Double] {
6    override def createAccumulator() = (0.0, 0);
7
8    override def add(evt: SensorEvent, partial: (Double, Long)) =
9      (partial._1 + evt.temperature, partial._2 + 1L);
10
11   override def getResult(partial: (Double, Long)) = partial._1 / partial._2;
12
13   override def merge(a: (Double, Long), b: (Double, Long)) =
14     (a._1 + b._1, a._2 + b._2);
15 }
16
17 class ResultEvent extends ProcessWindowFunction[Double, SensorEvent, Long, TimeWindow] {
18   def process(key: Long, context: Context, averages: Iterable[Double], out:
         Collector[SensorEvent]): () = {
19     val average = averages.iterator.next();
20     out.collect(SensorEvent(key, average));
21   }
22 }
```

Рисунок 1.7 – Приклад інкрементного вікна визначення середнього

2. Періодичне агрегування вікон: ми бачили, як часткове агрегування оптимізує обчислення для кожного вікна, однак цей підхід не вирішує загальної проблеми. Це пов’язано з тим фактом, що навіть часткове, поступове обчислення все одно буде дублюватися кілька разів, по одному на кожне вікно. У попередньому прикладі діапазону 60 хвилин і слайда 8 хвилин у будь-який час буде активним близько 8 вікон (можливо, навіть більше, якщо вікна оцінюватимуться не за порядком). Щоб вирішити цю проблему, більшість існуючих систем використовують клас неявних методів спільного використання обчислень, які називаються «зрізанням» [13,14], які ефективно попередньо обчислюють частки, що не перекриваються, поверх сегментів потоку, відомих апріорі під час компіляції (оскільки вікна є періодичними).

3. Періодичний спільний доступ до багатовіконної агрегації: досить часто програма містить агрегати за декількома специфікаціями вікон (наприклад, під час побудови заходів сховища даних або аналізу змодельованих наукових моделей). Для прикладу, на рисунку 1.8 ми маємо

три віконних середніх значення, визначені для різних, але дуже перекриваючих часових вікон.

```
1
2  val avgTempStream1: DataStream[SensorEvent] =
3    filteredEvents.timeWindow(Time.hours(1), Time.minutes(8))
4    .aggregate(AverageTemperature(), ResultEvent());
5
6  val avgTempStream2: DataStream[SensorEvent] =
7    filteredEvents.timeWindow(Time.hours(2), Time.minutes(30))
8    .aggregate(AverageTemperature(), ResultEvent());
9
10
11 val avgTempStream3: DataStream[SensorEvent] =
12   filteredEvents.timeWindow(Time.minutes(20), Time.minutes(5))
13   .aggregate(AverageTemperature(), ResultEvent());
14
15 ...
```

Рисунок 1.8 – Приклад періодичного багатовіконного середнього значення

Це хороша проблема оптимізації, яку також ефективно вирішували, головним чином для періодичних вікон із складеними схемами оптимізації зрізу, які вирішувалися під час компіляції [14]. Втім, якщо мова йде про неперіодичних випадках, то існуючі рішення повертаються до активних структур даних перед агрегацією [32, 15], які потребують великого обсягу пам’яті.

4. Спільне використання агрегації будь-якого вікна: сьогодні семантика вікон стає дедалі складнішою. На додаток до часових, такі системи, як Flink, Beam або Apex [46, 42, 39], дозволяють ще більше вбудованих специфікацій вікон, таких як вікна сесії [21]. Крім Flink і Beam, такі системи, як IBM Streams і його мова SPL [46, 47] пропонують користувачам можливість писати призначувачі для визначеної користувачем логіки вікон або користувацьких політик, які визначають, як записи можуть бути розміщені у вікнах виключно на основі на стан виконання програми. Вікна також часто визначаються на основі тенденцій даних, які можуть змінюватися з часом. Суть тут полягає в тому, що більшість складних вікон неможливо знати апріорі, і тому системі важко вибрати правильну стратегію оптимізації для

спільного використання обчислень у специфікаціях з одним або кількома вікнами. В даному випадку пропонується універсальне рішення для всіх вищезгаданих проблем. Цей підхід є подвійним: спочатку ми представляємо поняття визначених користувачем детермінованих вікон, які можна використовувати для створення спільних часткових агрегатів під час виконання для безлічі специфікацій вікон. Потім ми надаємо спільний доступ до всіх фрагментів, специфічних для середовища виконання, у структурі даних перед агрегацією, яку можна використовувати для безперешкодного отримання будь-якого повного віконного агрегату.

Багатопрохідне агрегування вікон. Ациклічного потоку керування в програмі не завжди достатньо для вирішення обчислювальної задачі. Той самий принцип, відомий з імперативного програмування, застосовується до більшості парадигм програмування, включаючи обробку потоку даних. Інфраструктури потокової обробки, такі як та, що зараз надається Flink, дозволяють виконувати складні операції (наприклад, агрегації ковзних вікон) і примітиви синхронізації (наприклад, глобальні таймери за часом події) у потоках, але вони не можуть інтегрувати циклічні обчислення як з точки зору програмування. підтримка семантики та виконання. У нашому дослідженні ми починаємо шукати правильну модель циклічного виконання, маючи за базову існуючу архітектуру Flink.

Обробка поза порядком: вікна потоку є привабливою відправною точкою для використання ітераційних процесів поверх. Головним чином це пов’язано з наявною вбудованою підтримкою паралельного виконання та детермінованості, особливо в контексті вікна подій. Час події перехресно пов’язує події з загальним порядковим відношенням, тобто часом їх генерації у відповідному джерелі, відображаючи кожну подію на певний земний час. Очевидно, оскільки локальні годинники не синхронізовані з глобальними атомними годинами, а мережі реального світу стоять між (і всередині) систем, події можуть оброблятися не в порядку [23]. Flink та більшість інших систем потокової обробки в даний час реалізують проміжну модель для відстеження

прогресу часу, яка називається моделлю обробки поза порядком (OOP - out-of-order processing), яка базується на монотонній прогресії часу до попередньо визначеного обмеженого ступеня невпорядкованості. Це дозволяє замінити локальні метриками прогресу, які є монотонними показниками, що використовуються для реалізації бар'єрів синхронізації на основі часу події між паралельними завданнями (наприклад, вікна часу події). Показники прогресу широко використовуються в системах виробничого класу і зазвичай реалізуються за допомогою використання децентралізованої техніки, яка правильно працює на моделі вбудованого стану. Застосування показників прогресу та OOP зосереджено на визначенні того, коли певні обчислення можуть бути ініційовані, а не на тому, як вони оптимізовані чи загальні, тому OOP не становив інтересу дослідницької роботи щодо спільного використання віконної агрегації.

Висновки до розділу 1

В даному розділі проведено аналіз літературних джерел та документів платформи Flink - провідної системи з відкритим вихідним кодом у сфері обробки та аналізу великих потоків даних. Також описано сучасні технології масштабування процесів обробки потоків даних.

РОЗДІЛ 2. МОДЕЛЮВАННЯ ПОСЛІДОВНОСТЕЙ СТАНІВ ДЛЯ ПРОЦЕСІВ МАСШТАБУВАННЯ ОБРОБКИ ПОТОКІВ ДАНИХ

2.1 Концепція обробки потоків на основі епох

Концепція обробки потоків на основі епох пропонує єдине рішення для надійного потокового передавання та відповідає всім відповідним потребам управління станом. Розглянемо концепцію та опишемо кілька підходів до її матеріалізації, одночасно прагнучи задовольнити основні властивості дизайну - нескоординоване та безблокове виконання, прозорість і композиційність.

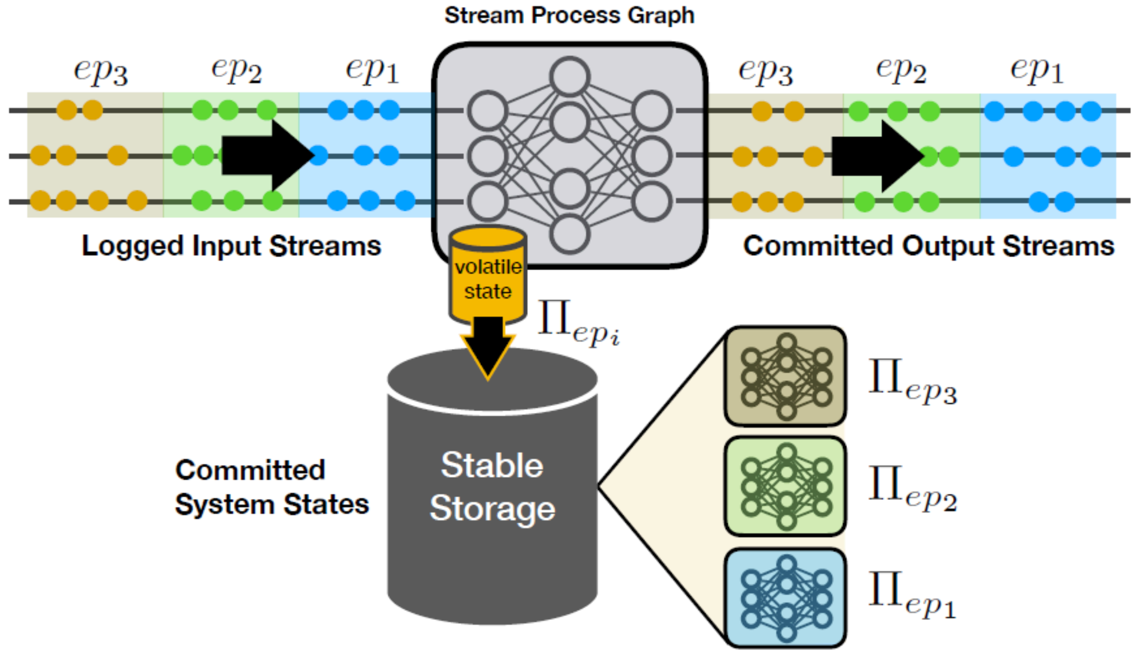


Рисунок 2.1 – Огляд потокової обробки на основі епохи

На рисунку 2.1. ми візуалізуємо основну ідею обробки на основі епохи на концептуальному рівні. Ми припускаємо, що граф потокового процесу G з моделлю припинення відмов і детермінованою послідовністю подій у його джерелах (наприклад, зареєстровані потоки). Основна мета полягає в тому,

щоб викрити безперервне виконання, яке задовольняє надійну обробку, тобто збереження всіх подій і дотримання причинно-наслідкового зв'язку. Однак замість того, щоб міркувати про окремі події, ми виділяємо дискретні етапи, або «епохи», які представляють кінцеві проміжні підмножини безперервного виконання E як такого:

$$E^{ep_1} \subset E^{ep_2} \subset E^{ep_3} \ldots \subset E.$$

Концептуально кожна епоха представляє частину обчислення, яка атомарно обробляється (завершується або перезапускається). Це дозволяє нам зняти всі проблеми щодо надійності з рівня окремих записів або виробництв на грубий рівень епох. Одного разу епоха здійснила всі стани процесу, граф на додаток до зовнішніх вихідних потоків відображає повний стан виконання до цього моменту. У разі збою під час виконання епохи ep_n ми можемо просто відкотити детермінований вхід і глобальний стан графа процесу до попередньої завершеної епохи (наприклад, ep_{n-1}). Основна вимога для використання виконання на основі епохи полягає в тому, щоб мати можливість фіксувати всі окремі стани завдань системи після завершення епохи в стабільному сховищі. По суті, це конфігурація системи в той момент, коли введення епохи ep_i було повністю оброблено. Таким чином, знімок S_{epi} такої конфігурації складатиметься лише з локальних станів завдання $S_{epi} = \{\Pi_{epi}\}$. Тут важливо зазначити, що існує багато можливих виконань до завершення епохи, наприклад, $E_{epi} \neq E'_{epi}$, оскільки загальний порядок обробки не гарантується у виконанні розподіленого потоку. Проте вчинена епоха включає в себе результат однієї можливої страти до епохи. Це поняття стосується неформального використання семантики наскрізної обробки точно один раз [9, 29], яка в даному випадку стосується конкретно «рівно однієї» фіксації епохи.

Здійснення синхронної епохи. У найпростішому випадку обробка потоку на основі епох може бути досягнута шляхом постановки основного виконання. Постановка зазвичай реалізується за допомогою протоколу атомарної фіксації між процесом-координатором і завданнями графа.

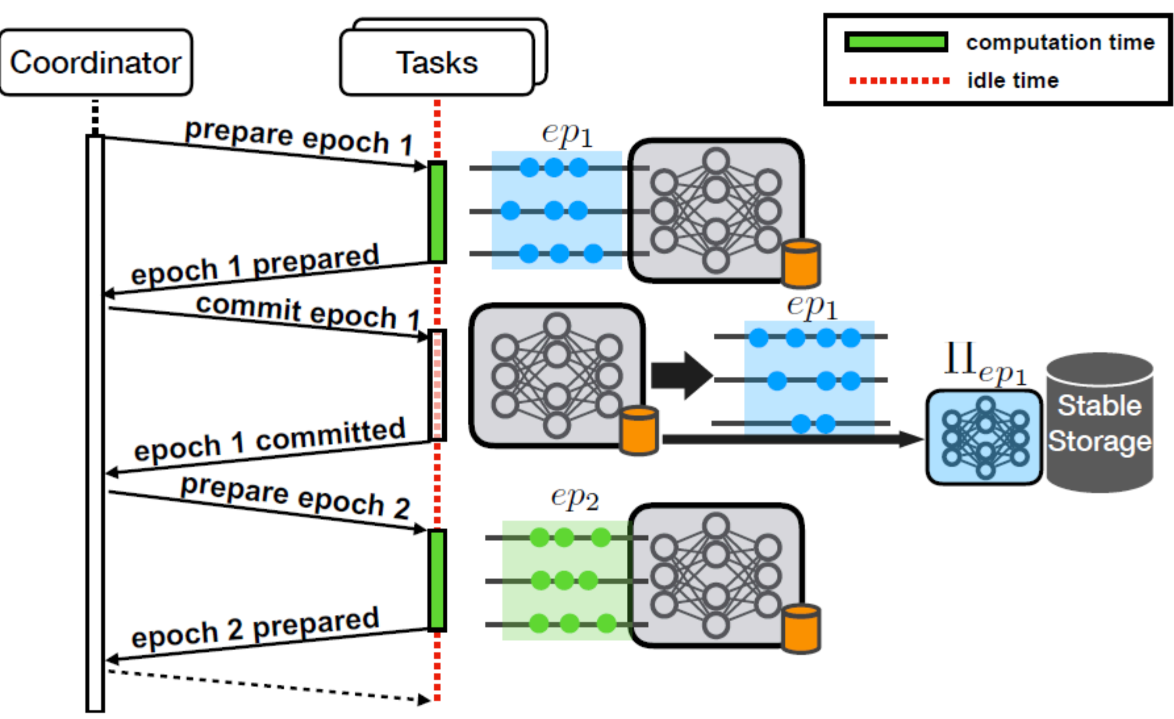


Рисунок 2.2 – Приклад фіксації синхронної епохи

На рисунку 2.2 представлено, як синхронний (двофазний) протокол атомарної фіксації можна використовувати для постановки епох у виконанні обробки потоку. Перша фаза гарантує, що всі обчислення завершено, а друга фаза завершена, коли всі побічні ефекти зберігаються в стабільному сховищі (тобто стани завдань і вихід потоку). Незважаючи на те, що цей підхід працює на практиці, є один серйозний недолік: він передбачає застосування протоколу координації блокування. Суть проблеми постановки полягає в тому, що більшість завдань повинні залишатися в режимі очікування доки кожне інше завдання не завершить обчислення і поки кожне інше завдання передає свій стан і побічні ефекти в стабільне сховище. Останнього потенційно можна уникнути за допомогою асинхронних методів копіювання,

однак, якщо епохи часті, комунікаційні витрати самого протоколу часто можуть перевищувати фактичний час обчислення. З іншого боку, якщо епохи трапляються рідко, завдання менше простоюють, але наскрізна затримка між надходженням вхідних потоків і часом фіксації результатів може бути занадто великою, а отже, занадто пізньою для багатьох програм (наприклад, критичних моніторинг подій).

Потокове мікродозування [12] – це механізм, який емулює можливості потокової обробки в системах пакетної обробки, які використовують короткочасне виконання завдань (Spark [27], MapReduce [8]). По суті, мікро дозування (мікропакетування) еквівалентно протоколу фіксації синхронної епохи. Основна ідея полягає в тому, щоб мати координуючий процес (тобто драйвер у Spark), який періодично ділить потік на пакети, а потім планує завдання (граф процесу короткочасних завдань) для кожного пакета. У випадку мікропакетування новий набір завдань буде заплановано, щоб випереджати обчислення для кожної епохи. У разі збою пакет повторно виконується тим самим або новим набором завдань (паралельне відновлення [12]). Це забезпечує гнучкість перерозподілу та реконфігурації обчислень для кожного пакету, однак це також вводить додаткові накладні витрати на планування на додаток до витрат синхронної епохи, згаданих вище. Крім того, оригінальна версія дискретизованих потоків передбачала використання дискретизованих у часі пакетів у своїй моделі програмування, роблячи всю архітектуру непрозорою для моделі програмування, спрямованої на користувача. Цю проблему можна пояснити початковим вибором дизайну в контексті пакетного виконання, а не невід'ємною властивістю потокової обробки на основі епохи, оскільки нові версії Spark Streaming пропонують більш плавну модель програмування з урахуванням стану (наприклад, Structured Streaming).

Якщо ми можемо забезпечити можливість скорочення епох протягом тривалого безперервного виконання, тоді можна підготувати та зафіксувати епохи асинхронно, як показано на рисунку 2.3. Це можна зробити за

допомогою знімків, однак потрібен протокол, який реалізує не просто послідовне скорочення, а епохальне скорочення.

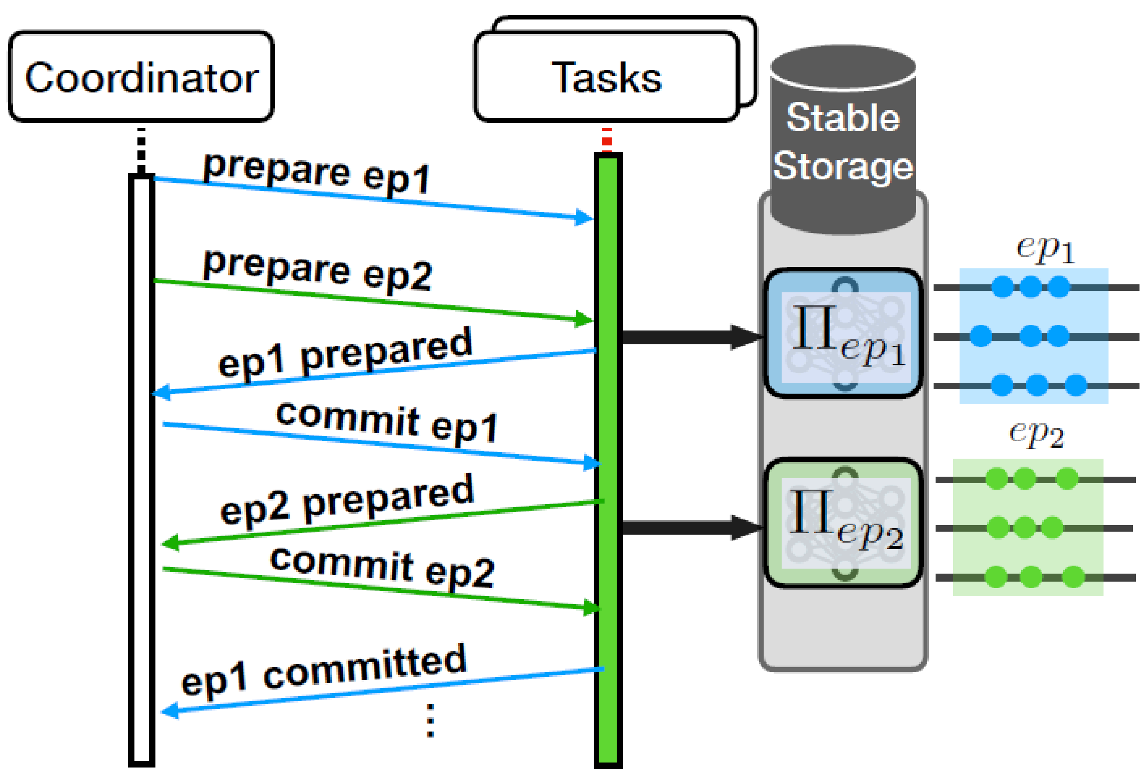


Рисунок 2.3 – Асинхронно узгоджені епохи

Під повнотою епохи ми маємо на увазі поняття захоплення (локального) виконання епохи ep_n у процесі p, також позначеного як Ep. З огляду на те, що кожен повний епоховий знімок реалізує епоховий розріз, усі повідомлення включаються до моментального знімка, тому жодні повідомлення, що передаються, не розглядаються в захопленій конфігурації:

Event Interface:

Indication: $\langle \mathrm{record} | p, n, s_p^i \rangle$

Properties:

ESNAP1: *Termination*: $\Lambda = \Pi$ in $E^{ep_n} \implies \langle \mathrm{record} | p, n, _ \rangle \in E_p, \forall p \in \Pi$

ESNAP2: *Validity*: Configuration $S_{\mathcal{C}_{ep_n}} = \{\Pi_{ep_n}\}$ is valid

ESNAP3: *Epoch-Completeness*: Configuration $S_{\mathcal{C}_{ep_n}}$ is epoch-complete

Важливим є той факт, що в цій специфікації миттєві знімки є не одиночними, а повторюваними і мають задовольняти припинення якщо не відбувається жодних збоїв до повного виконання епохи.

Підхід до створення миттєвих знімків це механізм створення миттєвих знімків на основі маркерів який демонструє, як захопити послідовний знімок одночасно з виконанням без необхідності координації блокування. Однак у контексті обробки на основі епохи він не здатний запропонувати епохальні скорочення. Для кращого розуміння даний процес розбивають на три частини: 1) ініціювання знімка, 2) вирівнювання епохи та 3) циклічний стан.

2.2 Дослідження процесу моделювання станів даних засобами розподіленого середовища

Розподілене середовище виконання Flink має архітектуру головний-підлеглий, подібну до Spark і Hadoop MapReduce. Проте, на відміну від пакетно-орієнтованого керування додатками [12], яке базується на поетапних, короткочасних обчисленнях, Flink використовує одноразовий розподіл завдань за розкладом.

Тим не менш, завдяки використанню епох і відповідних серверних модулів стану, які абстрагують операції над станом, система здатна переконфігурувати програми (наприклад, масштабування) і перерозподіляти стан програми на вимогу, пропонуючи надійні гарантії обробки (через атомарну фіксацію епохи).

Цей підхід мінімізує накладні витрати на керування, одночасно дозволяючи подальшу адаптацію до змін апаратного чи програмного забезпечення або часткових збоїв, які потенційно можуть виникнути. Спочатку ми пояснюємо загальний вибір дизайну, а потім детальніше зосередимося на кожному відповідному механізмі, пов’язаному зі знімками на основі епох.

2.2.1. Розробка архітектури

На рисунку 2.4 надано огляд усіх компонентів середовища виконання Flink та їх призначення, яке ми описуємо більш детально нижче.

Client: клієнтська бібліотека забезпечує підтримку статичної перевірки типів і компіляції програм Flink. Операція відповідає унарній (наприклад, карта, фільтр, згортання, вікно) або n-арній функції вищого порядку (наприклад, об’єднання, співкарта, співплоска карта) і параметризована за допомогою визначених користувачем функціональних літералів. Клієнт Flink компілює операції в логічний граф завдань, інкапсулюючи кожен незалежний компонент програми та його залежності даних від інших компонентів.

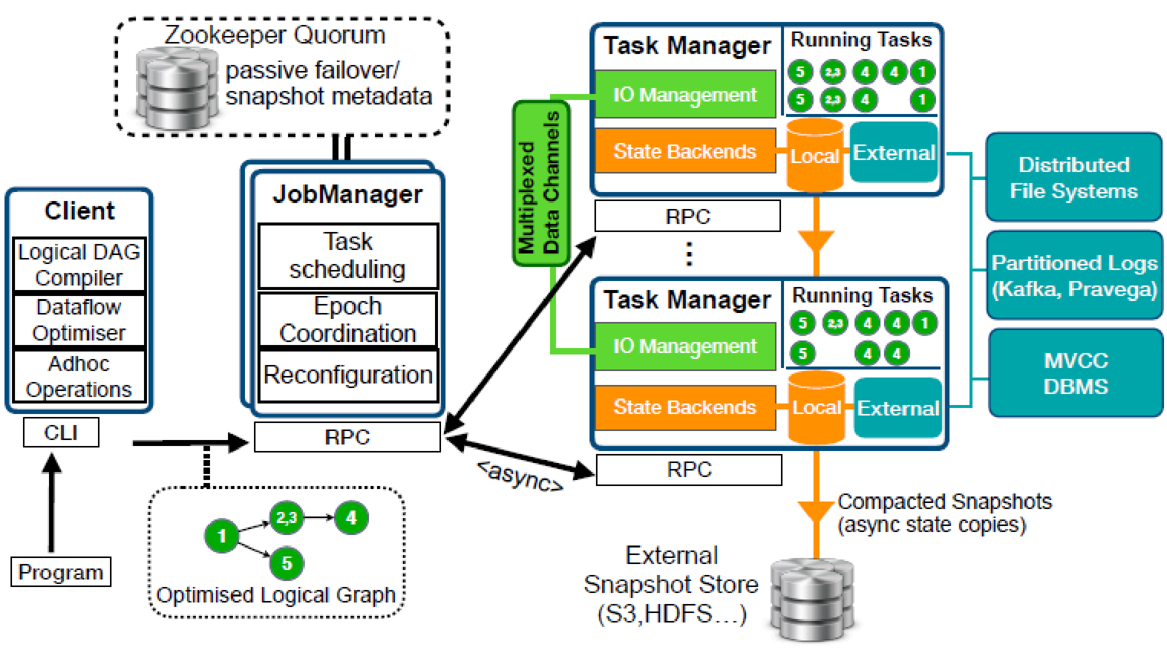


Рисунок 2.4 – Детальний огляд середовища виконання Flink

Завдання в остаточному оптимізованому графі відповідає одному або декільком (ланцюжковим) логічним операторам, скомпільованим генератором і оптимізатором графів Flink. Клієнт забезпечує інтерфейс командного рядка (CLI) для компіляції, надсилання, моніторингу та переналаштування

запущених програм і, таким чином, служить повним інтерфейсом для розподіленого середовища виконання Flink.

JobManager: JobManager — це процес JVM, який підтримує глобальний перегляд і контроль кожної запущеної програми, включаючи відповідні завдання та їхні локально керовані стани. Роль JobManager має вирішальне значення для виконання на основі епохи, оскільки він є посередником, який ініціює події зміни епохи, збирає записані знімки та повідомляє зворотні завдання про завершені епохи, як частину протоколу асинхронної епохи. Крім того, він використовує визначення збоїв на основі пульсу та моніторинг усіх ресурсів кластера Flink. Кожна дія менеджера завдань, яка змінює метадані програми (наприклад, планування, реконфігурація, завершення епохи тощо), спочатку фіксується в Zookeeper [47]. Це дозволяє використовувати пасивне резервне розгортання, яке може гарантувати послідовне виконання системи, яка може справлятися з усіма типами збоїв, включаючи збої головного вузла. Уся комунікація з клієнтом і локальними працівниками (TaskManagers) здійснюється за допомогою асинхронного протоколу на основі RPC, який не заважає каналам даних, які використовуються завданнями програми.

TaskManager: кожне логічне завдання програми планується та фізично виконується паралельно кількома працівниками. Фізичним завданням надається доступ до двох основних ресурсів: мережевих каналів і стану. Загальні ресурси, доступні в робочому файлі, повинні розподілятися ефективно й ізольовано кількома фізичними завданнями (зазвичай призначаються виділеним контейнерам за допомогою YARN [48] або Mesos [49]). Це робота процесу JVM TaskManager, який також служить основним проксі між фізичними завданнями та JobManager. Процеси JobManager не мають статусу та використовують мережеві зв’язки на основі політики та доступ до стану під час виконання запитів, отриманих від JobManager. Канали даних між потоками завдань мультиплексуються в спільні TCP-з’єднання зі спільними буферами для потреб серіалізації/десеріалізації, а взаємоблокувань зазвичай уникають за допомогою керування потоком у

польоті. Сервери стану — це модульний спосіб зробити операції стану прозорими для розташування та представлення фізичного стану. Flink підтримує плагіни для локально вбудованих баз даних (наприклад, RocksDB), зовнішніх розділених журналів (Kafka, Pravega, Kinesis тощо) і зовнішніх баз даних із підтримкою багатоверсійного керування паралелізмом (MVCC - Multiversion Concurrency Control-Enabled), не вимагаючи жодних змін у програмі користувача. Найважливіше те, що фізичний стан майже завжди зберігається поза купою та керується ззовні JVM для масштабованості та продуктивності (без збирання сміття).

2.2.2. Розробка завдання та модель процесу

Фізичні завдання у Flink використовують модель потокового процесу, а тому, слідують суворому потоку керування на основі повідомлень, маніпулюючи станом і здійснюють генерування вихідних записів у рамках атомарної дії, викликаної вхідним повідомленням. У виконанні на основі епох із надійною обробкою гарантується відсутність непрямого зв'язку чи зовнішніх з'єднань у цьому критичному шляху. Однак зовнішні асинхронні механізми зв'язку використовуються для допоміжних цілей, таких як протокол фіксації епохи, представлений пізніше.

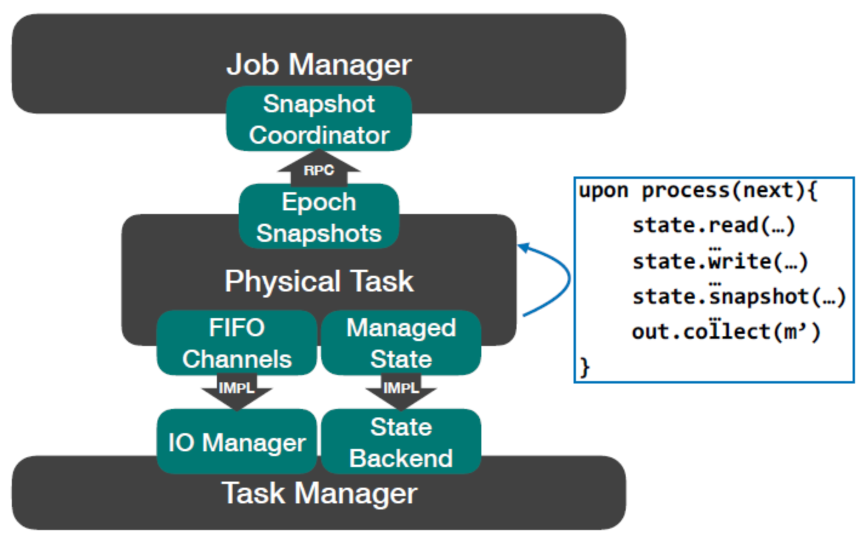


Рисунок 2.5 – Розробка компонентів фізичних завдань

На рисунку 2.5 показано, як фізичні завдання моделюються з точки зору проектування компонентів. Логіка в межах фізичного завдання викликається для кожного вхідного повідомлення, отриманого з одного з каналів FIFO.

Завдання може читати, записувати або створювати операції знімка свого керованого стану, а також надсилати вихідні повідомлення через інтерфейс збирача ("collect" знімає повідомлення за потоком). Обидві реалізації каналів FIFO та керованого стану надаються TaskManager. Крім того, наприкінці повної епохи кожне завдання надає посилання на свій знімок стану, який збирається за допомогою асинхронного RPC, виданого диспетчером завдань. Для зручності логіка пріоритезації каналів, необхідна для фази вирівнювання протоколу на основі епохи, реалізована за допомогою каналів FIFO, які надаються компонентом IOManager.

2.2.3. Реалізація протоколу

Flink використовує протокол Asynchronous Epoch Commit, що гарантує, що всі події у виконанні до епохи та їхні операції внутрішнього та зовнішнього стану атомарно передані в стабільне зберігання. Реалізований протокол включає всі етапи асинхронного зв’язку між процесом Snapshot Coordinator (JobManager), Physical Tasks (TaskManager) і відповідними серверними частинами стану. Екземпляр протоколу запускається на зміну епохи та ініціюється координатором, підтримуючи одночасні екземпляри протоколу.

Якщо екземпляр переривається (наприклад, через частковий збій), система відкочує всі попередньо внесені зміни, і виконання перезапускається з останньої епохи фіксації. Найважливіше те, що протокол фіксації епохи виконується одночасно з виконанням фізичного завдання і не впливає на критичну частину обчислення. На рисунку 2.6 подано кожен крок зв’язку протоколу фіксації епохи та докладніше пояснюємо кожен крок нижче.

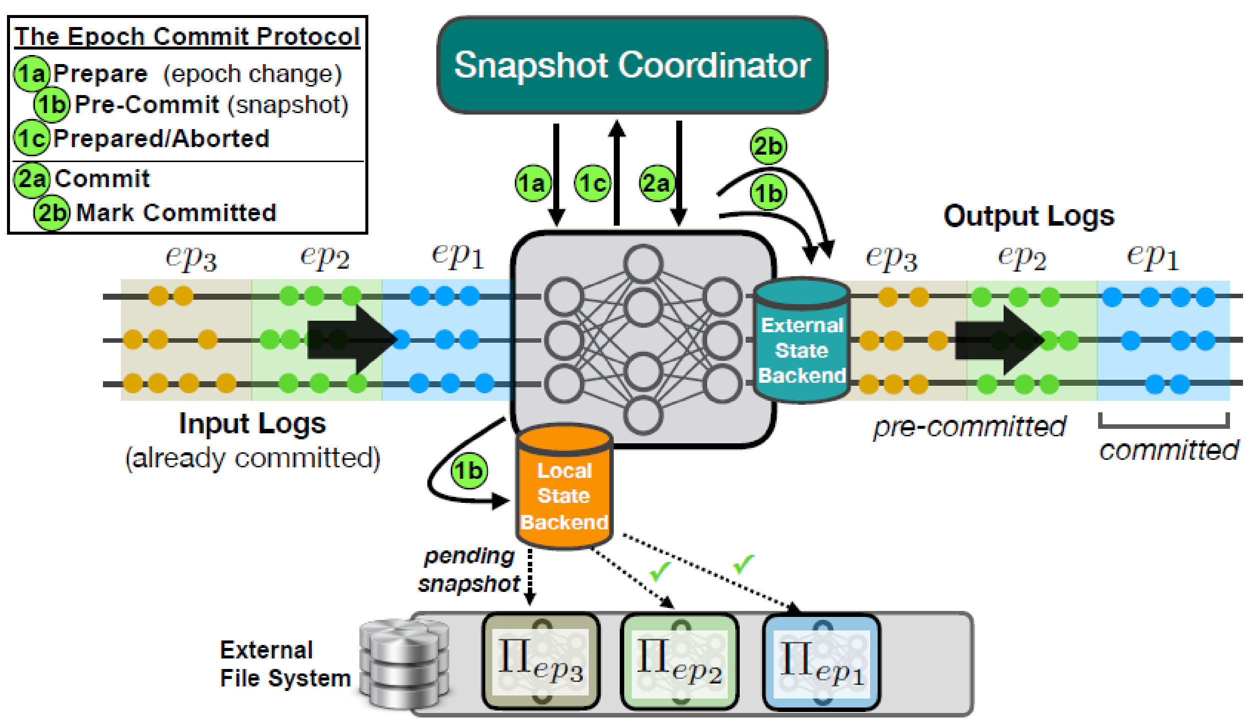


Рисунок 2.6 – Огляд Epoch Commit зі Snapshots

Фаза підготовки: фаза підготовки починається, коли координатор знімка видає зміну епохи. Це може відбуватися періодично (наприклад, кожні 20 секунд) або як спеціальний запит від користувача. В обох випадках координатор транслює повідомлення про зміну епохи (1a) до всіх вузлів TaskManager, що, у свою чергу, ініціює протокол створення знімків на основі епохи для всіх вихідних завдань, включаючи розповсюдження маркера та вирівнювання епохи. Зрештою, якщо збоїв не відбувається, кожне фізичне завдання запускає локальний знімок свого керованого стану, а підтвердження надсилається назад координатору (повідомлення про запис). Ми називаємо кожну локальну операцію створення моментального знімка кроком перед фіксацією (1b). У випадку операторів із локальним бекендом стану попередня фіксація — це операція копіювання стану до зовнішньої файлової системи. Однак у випадку зовнішнього бекенда стану попередня фіксація блокує віддалені зміни, щоб не виконувати подальших операцій і підготувати її до остаточної фіксації. Тим не менш, ані зроблені моментальні знімки станів, ані зовнішні попередньо зафіксовані стани не повинні бути доступними на цьому

етапі, оскільки епоха ще не зафіксована. Фаза підготовки завершується, коли всі завдання сповіщають координатора про «підготовлене» підтвердження (1c). Це може статися лише після завершення алгоритму знімка. Якщо під час процесу виникає часткова помилка (повідомлення про скасування) або глобальний тайм-аут, протокол переривається.

Фаза фіксації: Фаза фіксації починається, коли координатор отримує всі «Підготовлені» команди. Метою етапу фіксації є підтвердження всіх попередніх фіксованих станів для зовнішнього доступу. Координатор моментальних знімків ініціює фазу фіксації, транслюючи повідомлення «Епоха фіксації» всім завданням (2a), які, у свою чергу, фіксують епохи, що очікують, на серверних частинах (2b). Ця дія може бути викликана диспетчером завдань одночасно з виконанням завдання, таким чином не впливаючи на продуктивність критичного шляху виконання. Фаза фіксації особливо важлива для зовнішніх серверних модулів стану, щоб зробити всі зовнішні зміни видимими зовні (наприклад, попередньо зафіксовані вихідні потоки). Під час цієї фази потенційно можуть виникнути збої, які можуть призвести до неповних прийнятих змін в епосі. Однак це не порушує жодних системних гарантій. З огляду на те, що на цьому етапі принаймні гарантовано, що всі зміни будуть попередньо зафіксовані, незавершені коміти можуть бути врешті-решт повторно видані під час механізму відкату (пояснено нижче), щоб завершити процес.

Підсумок: Epoch commit — це двофазний протокол фіксації спеціального призначення, який забезпечує гарантії обробки транзакцій. Використання асинхронних знімків на основі епох для попередньої реєстрації всіх побічних ефектів дає кілька важливих спостережень. По-перше, відсутність операції фіксації впливає на продуктивність під час виконання (тобто на пропускну здатність) системи. Протокол фіксації просто робить усі побічні ефекти епохи видимими зовні, коли гарантується, що все зберігається в певній формі стабільного сховища.

По-друге, це дозволяє конвеєрувати кілька одночасних екземплярів фіксації епохи з використанням асинхронних знімків на основі епохи. У прикладі на рисунку 2.6 ми можемо побачити випадок трьох, можливо, одночасних епох. Для епохи ep і всі побічні ефекти фіксуються, тоді як ep_2 було попередньо фіксовано, а фаза фіксації очікує. Нарешті, ep_3 знаходиться на етапі перед фіксацією, що означає, що наразі виконуються алгоритми створення знімків. У тому ж прикладі можна безпечно відкотитися від 2-го етапу, оскільки в цей момент усі стани зберігаються в стабільному сховищі.

2.2.4. Процедура відкату

Механізм відкату Flink ініціюється, коли виявляється часткова помилка під час нормальної роботи (модель зупинки збою), або коли запитується повторна конфігурація, або після перерваного екземпляра фіксації епохи. Відкат виконує процедуру «зупинити, перепланувати та відновити», незалежно від того, причиною є відновлення після збою чи зміна конфігурації (рис. 2.7).

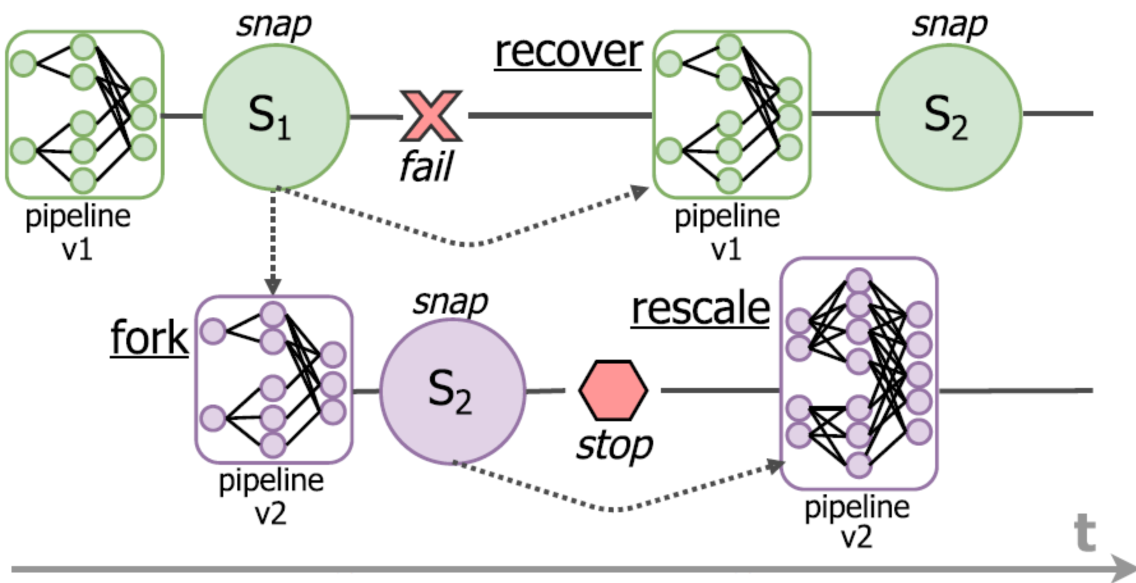


Рисунок 2.7 – Приклад процедури відкату

У всіх випадках система (JobManager) вибирає останній підготовлений (але не обов'язково зафіксований) знімок для повторного запуску виконання. Згодом усі стани в конвеєрі поступово витягуються та застосовуються для відображення точного, дійсного розподіленого виконання у відновлену епоху. Нижче ми виділимо кілька особливих випадків відкату завдання:

- Завдання Loop Head: у випадку завдань Loop Head усі записи, зареєстровані під час моментального знімку відновлюються та скидаються на вихідні канали перед їхньою звичайною логікою пересилання запису. Таким чином гарантується, що стан циклу повернеться назад до моментального виконання (тобто до суми записів у дорозі).

- Стан у зовнішніх серверах: щоб назавжди обійти регістр незафіксовані зовнішні стани, усі серверні модулі зовнішнього стану видають превентивне фіксування на початку відкату, щоб гарантувати, що жодні незафіксовані зміни не збережуться після відновленої точки виконання.

- Звичайні джерела: усі джерела даних повинні відновити своє (детерміноване) виконання до поточного зсуву кожного потоку, коли відбувся знімок. Джерела даних Flink надають цю функціональність нестандартно, зберігаючи зсуви до останнього запису, обробленого до епохи із зовнішніх систем реєстрації. Після відновлення сукупний стан цих джерел відображає точний прогрес розподіленого прийому, досягнутий до епохи відновлення. Цей підхід припускає, що зовнішні системи журналювання, з якими джерела взаємодіють, індексують і впорядковують дані між розділами надійним способом (наприклад, Kafka, Kinesis, PubSub і розподілені файлові системи).

- Вибірковий відкат: залежно від причини відкату можна застосувати певні оптимізовані схеми відновлення. Наприклад, під час повного перезапуску або масштабування всі завдання перерозподіляються, тоді як після збою переконфігуруються лише завдання, що належать ураженому підключеному компоненту (графу виконання), якщо існує більше одного підключеного компонента.

По суті, відомі методи інкрементального відновлення з мікропакетної обробки [12] є ортогональними до підходу відкату Flink і також можуть бути використані. Епоха знімка діє як точка синхронізації, подібно до мікропакету або розділення вхідних даних. Після відновлення плануються нові екземпляри завдань, які після ініціалізації повертають розподілений їм спільний стан назад до відповідних серверних модулів із знімків, що зберігаються назовні.

2.2.5. Серверна інтеграція

У Flink внесено декілька реалізацій серверної частини, які бездоганно інтегруються з протоколом Flink Epoch Commit. У таблиці 2.1 ми підсумовуємо деякі з існуючих і можливих серверних модулів і необхідні операції для інтеграції з Epoch Commit від Flink.

Таблиця 2.1

Приклади зіставлення внутрішніх операцій із протоколом Epoch Commit Flink

	Start	Pre-Commit	Commit	Abort
Local (RocksdB)	new MemTable	flush MemTable / Snapshot SSTAbles	-	delete snapshot
Local (Heap)	-	deep copy (full snapshot)	-	delete snapshot
Kafka ≥0.11	new transaction	close/new transaction	commit transaction	abort transaction
Pravega	new Segment	seal Segment	commit Segment	delete Segment
HDFS	create File in tmp dir	close File (no writes)	move (atomic) file to committed Dir	truncate/delete file
DBMS (non-MVCC)	new WAL	snapshot WAL	execute WAL as transaction	drop WAL
DBMS (MVCC)	-	new version	incr version	decr version

Загалом кожен сервер підтримує чотири операції:

1. Запуск.
2. Попереднє закріплення.
3. Закріплення.
4. Переривання.

Для кількох серверних модулів, які дозволяють певну форму транзакційного запису, операційні потреби задовольняються з коробки. До них входять останні версії розділених журналів, такі як Kafka і Pravega, а також СУБД з багатоверсійним керуванням паралелізмом. Наприклад, Apache Kafka представив підтримку одноразової доставки для виробників. Приймач транзакцій Kafka від Flink фактично створює нову транзакцію для кожної епохи яку Kafka може зафіксувати атомарно. Pravega є ще одним прикладом новішої системи, яка має вбудовану підтримку транзакційного запису між розділами вже у своїй моделі. Сегменти Pravega представляють автономні розділені журнали, які можуть бути атомарно запущені та запечатані, тісно інтегровані з епохальною обробкою Flink, оскільки сегмент для кожної епохи може бути запущений, запечатаний (закріплений) і видалений у разі переривання епохи.

У решті випадків впровадження двофазного коміту має вирішувати опосередковано розробник серверної частини. Прикладами нетранзакційних серверних програм є бази даних і файлові системи, що не належать до MVCC. Наприклад, приймачі транзакцій HDFS у Apache Flink зберігають усі операції додавання стану до тимчасових файлів і покладаються на скорочення HDFS для припинення епохи та операції атомарного переміщення для переміщення закритого файлу HDFS епохи до каталогу, «збереженого для читання». У випадку СУБД або інших зовнішніх систем зберігання загальною стратегією, яка працює з додатковою затримкою та вартістю зберігання, є підтримка журналу попереднього запису (WAL - Write-Ahead-Log) для всіх незафіксованих зовнішніх операцій у локальному стані. Після фази фіксації епохи (або відновлення) WAL може бути виконано та вилучено з локального стану.

Загалом, локальні серверні модулі потрібні для доповнення знімка всіх метаданих, необхідних для збереження зовнішніх двофазних комітів. Наприклад, WAL, зміщення журналів, ідентифікатори транзакцій тощо є важливими метаданими, які вимагають бухгалтерського обліку, щоб

остаточно завершити будь-які незавершені асинхронні протоколи обміну повідомленнями, які були ініційовані зовнішніми системами.

2.3 Процеси реконфігурації системи для масштабування обробки потоків даних

Асинхронні знімки епохи та процедура відкату покривають потреби реконфігурації, але лише частково. Типовою потребою в будь-якому розгортанні програми з інтенсивним використанням даних є можливість змінювати масштаб (тобто паралельність) певних логічних завдань. Для завдань, які мають оголошений керований стан, нам потрібно послідовно розподіляти дані, виконувати потокові розділи та подальший перерозподіл у разі реконфігурації. Фактично, більшість масштабованих операцій і відповідний стан охоплюються визначеним користувачем ключем із простором ключів K. Для балансування навантаження обидва потоки та відповідні стани розділяються на сегменти в просторі узгодженої функції хешування h : K → N+.

Flink роз'єднує розділення ключового простору та розподіл стану подібно до Dynamo. Середовище виконання відображає ключі в проміжному циклічному хеш-просторі «ключ-групи»: K* Є N+ з урахуванням максимального паралелізму π -max і хеш-функції h як такої:

$$K^* = \{h(k) \bmod \pi\text{-max} \mid k \in K, \pi\text{-max} \in \mathbb{N}^+, h : K \to \mathbb{N}^+\}$$

З огляду на те, що знімки мають містити всю інформацію, необхідну для пошуку та повторного розподілу стану, існує очевидний компроміс між накладними витратами на відкат (введення/виведення під час сканування стану) та метаданими знімка, необхідними для повторного розподілу стану для різної кількості екземплярів s. З одного боку кожне паралельне завдання

може сканувати весь стан (часто віддалено), щоб отримати значення всіх призначених йому ключів. Це призводить до значної кількості непотрібного введення-виведення (рис. 2.8 (a)). З іншого боку, знімки можуть містити посилання на кожен окремий ключ-значення, і кожне завдання може вибірково отримувати доступ до призначених йому станів ключів (рис. 2.8(b)). Однак цей підхід збільшує витрати на індексацію (пропорційно кількості окремих ключів) і накладні витрати на зв'язок для кількох віддалених зчитувань стану, таким чином не приносячи переваги грубому зчитуванню стану. Групи ключів (рис. 2.8(c)) пропонують суттєвий компроміс: читання обмежується необхідними даними, а групи ключів зазвичай достатньо великі для детального читання (якщо п -max встановлено відповідним чином низьким). У непоширеному випадку де |K| < п-max можливо, що деякі екземпляри завдання просто не отримують стану. Нарешті, це відображення гарантує, що одне паралельне фізичне завдання оброблятиме всі стани в кожній призначеній групі, роблячи ключову групу атомарною одиницею для повторного розподілу.

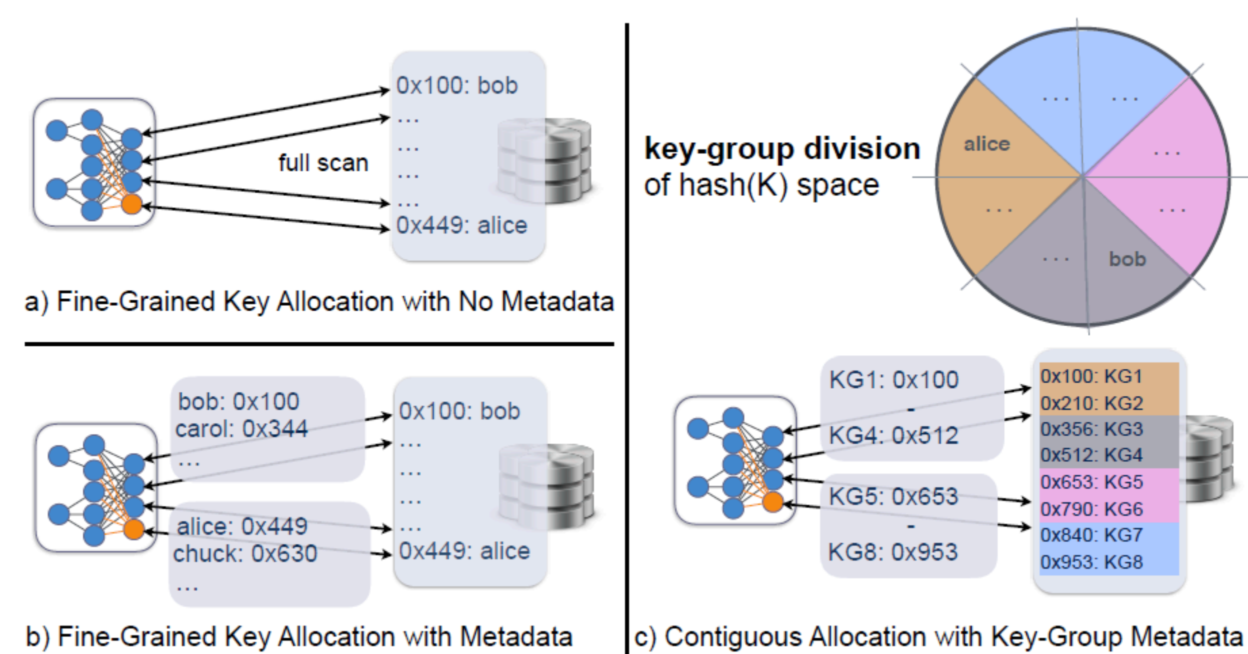


Рисунок 2.8 – Розподіл стану та альтернативи метаданих

Щоб повторно призначити стан, ми використовуємо розподіл діапазону груп ключів однакового розміру. Для n паралельних екземплярів кожен екземпляр $t = 1 \in \Pi$, $0 < i < n$ отримує діапазон груп ключів. Пошук є дорогим, особливо в розподілених файлових системах. Тим не менше, призначаючи суміжні групи ключів, ми усуваємо непотрібний пошук і перевантаженість читання, забезпечуючи низьку затримку після повторного розподілу. Записи Operator-State, які не можуть бути обмежені ключем, зберігаються послідовно (комбінуючи потенційні більш детальні атомарні стани, визначені в різних завданнях), для кожного оператора, у знімках і повторно призначених на основі їхнього шаблону перерозподілу, наприклад, у циклічному режимі або шляхом широкомовної передачі об'єднання всіх записів стану до всіх примірників оператора.

2.4 Ізоляція доступу для керованого стану даних

Відмовостійкість і реконфігурація є лише підмножиною потенційних потреб і переваг, охоплених епохальним знімком. У цьому розділі ми представляємо ще два нових приклади використання знімків, а саме зовнішні гарантії ізоляції доступу для керованого стану та походження програми, обидва з яких були досліджені та прототиповані в Apache Flink.

Flink дозволяє прямі adhoc запити до свого керованого стану ззовні системи. Таким чином зовнішні системи або користувачі можуть отримати доступ до ключового стану Flink подібним чином до сховища ключів/значень, надаючи доступ лише для читання до останніх значень, обчислених потоковим процесором. Ця особливість мотивована двома спостереженнями. По-перше, багатьом програмам потрібно надавати спеціальний доступ до стану програми для швидшого аналізу. По-друге, охоча публікація стану в зовнішніх системах часто стає вузьким місцем у додатку, оскільки віддалений запис у зовнішні системи не встигає за продуктивністю локального стану Flink у високопродуктивних потоках.

Доступ до стану запиту можна отримати через API на основі підписки. По-перше, у вихідній програмі оголошено керований стан, який дозволяє доступ для запитів. Після оголошення стану можна дозволити доступ із зовнішніх запитів, просто встановивши позначку в дескрипторі, який використовується для створення фактичного стану, маючи призначену унікальну назву для цього конкретного стану, до якого потрібно отримати доступ, як такий:

```
1  //stream processing application logic
2  val descriptor: ValueStateDescriptor[MySchema] = ...
3  descriptor.setQueryable("myKV")
4  ...
5  val mutState: ValueState[MySchema] = ctx.getState(descriptor)
```

Після розгортання служба реєстру ініціюється та виконується одночасно із завданням, яке має доступ для запису до цього стану. Клієнт, який бажає прочитати стан для певного ключа, може в будь-який час подати асинхронний запит (отримання майбутнього) до цієї служби, вказавши ідентифікатор завдання, зареєстровану назву стану та ключ, як показано нижче:

```
1  //client logic
2  val client = QueryableStateClient(cfg);
3  var readState: Future[_] = client.getKVState(job, "myKV", key);
```

Поточна реалізація запитуваного стану підтримує точкові пошуки значень за ключем. Клієнт запиту запитує у майстра Flink (JobManager) розташування екземпляра оператора, який містить розділ стану для запитуваного ключа. Потім клієнт надсилає запит до відповідного TaskManager, який отримує значення, яке наразі зберігається для цього ключа, із серверної частини стану.

З точки зору рівня ізоляції запитів бази даних, такі запити отримують доступ до незафіксованого стану, таким чином дотримуючись рівня ізоляції читання-незафіксованого. Однак за допомогою знімків можна запропонувати

підтримку ізоляції з підтвердженням читання , дозволяючи диспетчерам завдань зберігати стан зафіксованих знімків і використовувати цей стан для виконання запитів adhoc.

Виконання на основі епохи роблять можливим походження додатків і, що більш важливо, тривіальним. Це пов'язано з тим, що миттєві знімки позначають чіткий ланцюжок залежностей між епохами та діями реконфігурації, застосованими протягом всієї історії тривалої програми обробки потоку. Як показано на рисунку 2.9, діаграма залежності виконання програми нагадує діаграму системи контролю версій (наприклад, git), де її чітка зміна інкапсулюється у знімки епохи.

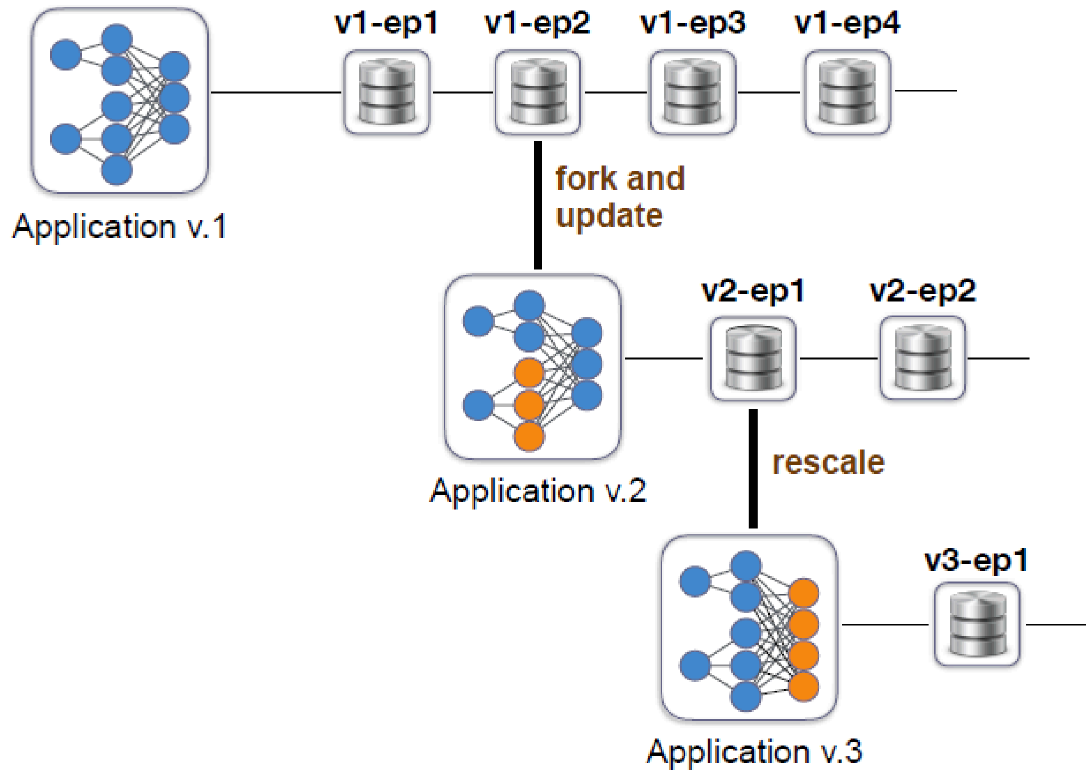


Рисунок 2.9 – Походження програми зі знімками

Крім того, програма може мати кілька версій, створених із існуючих знімків, які виконуються в різних конфігураціях (наприклад, різний паралелізм, кластер, логіка тощо). Це може ще більше спростити розробку та підтримку безперервних програм, а також надати свободу випускати

виправлення помилок, які можуть повернутися до минулого (наприклад, на знімках старої епохи) і, таким чином, повторно узгоджувати помилкову функціональність, починаючи з того часу, коли вона фактично була сталося.

Іншим важливим аспектом знімків є те, що вони роблять безперервне виконання програми суто переносним. Перенести повний конвеєр так само просто, як відкотити програму до встановленої епохи. З огляду на те, що знімки самі по собі є блоками, які можна переміщувати в різні місця, це надає важливу гнучкість розробникам додатків, оскільки робить повний перехід до різних хмарних постачальників або локальних кластерів тривіальним.

Висновки до розділу 2

В даному розділі представлено базові примітиви моделі виконання та специфікації надійної обробки потоку даних. Наш аналіз стосується набору універсальних проблем, коли йдеться про надійну обробку потоку, включаючи здатність відновлюватися після збоїв, змінювати конфігурацію виконання безперервної обробки потоку. З цією метою ми запропонували модель потокової обробки на основі епох, згідно з якою обчислення потоку поділяють на серію епох, які атомарно фіксують стан виконання

РОЗДІЛ 3. РОЗРОБКА МОДЕЛЕЙ ОПТИМІЗАЦІЇ ІТЕРАТИВНИХ ПОТОКІВ ДАНИХ

3.1 Сутність ітераційної обробки та опис базової моделі ітеративних процесів

3.1.1. Дослідження процесу ітеративної обробки

Ітеративна обробка за своєю суттю добре підтримується системами пакетної обробки, використовуючи їх вбудоване централізовано скоординоване виконання та використання зовнішнього спільного стану в розподілених файлових системах [17]. Однак застосування багатопрохідних обчислень у потоковій обробці створює декілька проблем, які необхідно вирішити, беручи до уваги існуючі варіанти дизайну, які вже використовуються, такі як виконання завдань без координації, асинхронне знімання та керування потоком. З цією метою ми визначаємо набір доповнень середовища виконання, а також примітивів моделі програмування, які є важливими для ітераційної обробки, але поточні потокові процесори виробничого рівня не можуть забезпечити їх.

На рівні середовища виконання необхідно скласти довільне обчислення вкладеного потоку даних. Це означає, що ми повинні мати можливість включати конкретні циклічні структури в графіки потокових процесів і чітко визначати їхні області, щоб мати можливість визначати та відстежувати конкретно, як прогресує багатоетапне розподілене обчислення та коли воно завершується. Наявність додаткових рівнів інкапсуляції в графах процесів створює додаткові проблеми, такі як здатність пов’язувати окремі записи до відповідних обчислювальних областей і масштабований спосіб збору, відстеження та сповіщення про хід кожного циклічного обчислення для його базових розподілених завдань, уникаючи центральної координації та

взаємоблокувань через перевантаження мережі в циклічних підкомпонентах. Важливі проблеми також виникають на рівні програмування, щоб запропонувати користувачам зручну семантику для вираження циклічних обчислень, вкладених у безперервний конвеєр. По суті, ми шукаємо способи залучити добре відомі ітераційні примітиви обробки з попередніх спеціалізованих систем, такі як масова або застаріла суперкрокова синхронізація та завершення фіксованої точки в контексті потокової передачі. Модель, орієнтована на користувача, має бути композиційною та добре інтегруватися з керованим станом та існуючими однопрохідними агрегаціями (наприклад, у вікнах потоку).

Попередні підходи, які розглядають ітерації потоків даних надмірно спеціалізовані на конкретній області (наприклад, аналіз графів). Єдиним винятком є модель Timely Dataflow [46] реалізована системою Naiad [20], яка використовує асинхронну модель збору та розсіювання для спостереження за ітераційним та іншим ієрархічним прогресом за допомогою проміжних процесів відстеження. Хоча підхід Timely Dataflow є гарним прикладом загального підходу, він погано інтегрується з асинхронними епоховими комітами та керуванням потоком у польоті, що вимагає, щоб уся обчислювальна логіка та стан залишалися в графі потокового процесу.

Ми пропонуємо розширення однопрохідної потокової обробки, яке забезпечує повну підтримку довільних вкладених ітерацій і ґрунтується на низьких водяних знаках [22], домінуючому механізмі передачі даних у системах потоку виробничих даних (наприклад, Apache Flink, Beam [42, 9], Spark's Structured Streaming, Kafka Streams і Storm [38]).

Ми досліджуємо, як області можна використовувати в поєднанні з часовими мітками прогресу, розширенням низьких водяних знаків для вкладених монотонних показників прогресу для виклику завдань щодо різних типів оновлень прогресу. Крім того, ми реалізуємо оператор агрегації вікон із кількома параметрами, який демонструє можливості системи для мультиплексування ітерацій у різних вікнах потоку, одночасно надаючи

доступ до основного постійного та керованого стану для кожного вікна. Нарешті, ми демонструємо застосовність віконних багатопрохідних агрегацій для забезпечення графоцентричної моделі програмування для динамічних графіків і показуємо, як ми можемо реалізувати класичні стандартні алгоритми, такі як PageRank (PR), Connected Components і Single Source Shortest Paths (SSSP).

3.1.2. Представлення базової моделі для ітеративних процесів

Перш ніж ми почнемо дослідження семантики ітераційної обробки в потоковій передачі даних, ми спочатку опишемо ітеративний процес як просту абстракцію програмування та розглянемо її основні структурні властивості, які ми хочемо інкапсулювати в потоковій передачі даних.

Концепція ітерації є стандартною основою для багатьох алгоритмів у областях обчислювальної статистики, машинного навчання, графічного та логічного програмування. Зазвичай він складається з блоку k операцій, які повторюються, щоразу генеруючи оновлене рішення, доки умова не буде задоволена. Ітераційний процес подібний до загальних циклічних потоків керування в мовах програмування, але це абстракція вищого рівня.

Наприклад, ітераційний процес може бути використаний для мінімізації диференційованої цільової функції або наближеного розташування центроїдів кластера на повторюваних кроках, які враховують повний доступ до даних або стану на кожному кроці. Це відрізняється від перерахування елементів списку або змінення масиву в циклі. У цьому розділі ми представимо просту модель для створення ітераційних процесів, починаючи з моделі непаралельного виконання та пізніше розширюючи її для розподіленої обробки.

Ітерація зазвичай може бути розкладена на дві функції: функцію циклу та функцію завершення. Функція циклу $L : X \rightarrow X$ інкапсулює блок повторюваних операцій над заданим станом або «розв’язком» типу X, тоді як

функція завершення c : (X, X) → bool видає true, коли умова завершення задовольняється. Враховуючи ці два примітиви та початковий стан x_{init} Є X ітераційне обчислення може бути складено таким чином:

```
1  def iterate(c,L)(x_init)
2      x = x_init
3      do{
4          x = L(x)
5      }while(¬c(x,L(x)))
6      x
```

Наведеної вище композиції ітераційного процесу достатньо, коли критерій завершення та функція циклу залежать лише від стану обчислення. Наприклад, ітерація з фіксованою точкою, яка завершується, коли рішення сходиться (тобто без змін на кроці), може бути визначена за допомогою наступного критерію завершення на основі наведеної вище моделі:

```
1  def c(x,x′) = x==x′
```

Інші варіанти обчислення з фіксованою точкою, такі як критерій збіжності на основі помилки на основі заданої помилки ϵ , можна визначити таким же чином:

```
1  def c(ϵ)(x,x′) = |x−x′| ≤ ϵ
```

Крім того, наше визначення вище також підтримує композицію через використання часткового застосування. Більш конкретно, функція циклу сама по собі може бути вкладеною ітерацією (на довільних рівнях) з умовою завершення c' і функцією циклу L':

```
1  val x = iterate(c,iterate(c′,L′))(x_init)
```

Незважаючи на простоту нашої моделі, є випадки, коли обчислення циклу або фактична умова завершення потребують відстеження поточного кроку обчислення.

Типовий спосіб інкапсулювати це поняття прогресу в ланцюжку послідовних ітераційних кроків: $L^0(x)$, $L^1(x)$,..., $L^p(x)$ — це лічильник кроків p Є N, який діє як контроль. змінна, доступна в контексті ітерації. У такій моделі ми можемо підтримувати функцію циклу з усвідомленням прогресу L : (N, X) → X, а також умову завершення c:(N,X,X) → bool у межах ітерації, вираженої таким чином:

```
1  def iterate(c,L)(x_init)
2      (p,x) = (0,x_init)
3      do{
4          (p,x) = (inc(p), L(p,x))
5      }while(¬c(p,x,L(x)))
6      x
```

Цієї моделі достатньо для опису всіх поширених типів ітераційної обробки. Наприклад, типовий фіксований ітераційний процес з k кроками може бути реалізований наступною умовою завершення, яка задовольняється, коли лічильник циклу вищий або дорівнює k:

```
1  def c(k)(p,_,_) = p >= k
```

3.1.3. SGD і обмежена асинхронність

Іншою областю ітераційної обробки, яка нещодавно отримала широке поширення, є машинне навчання. Стохастичний градієнтний спуск [113] (SGD - Stochastic Gradient Descent) і його паралельна інкрементальна реалізація є найпопулярнішим на сьогодні методом оптимізації диференційованих цільових функцій. Однією з відмінних властивостей SGD і загалом ітераційної оцінки опуклих функцій є їхня здатність сходитися

швидше, зберігаючи при цьому точність у розслаблених моделях розподіленої синхронії [19].

Застаріла синхронна паралельна модель, один із домінуючих методів синхронізації, дозволяє завданням виконувати обчислення незалежно до обмеженого ступеня асинхронності, тобто на m Є N кількість кроків попереду від найповільнішого завдання в системі, відомого як зависання. Це забезпечує кращий рівень використання та конвергенції при передбачуваних втратах точності.

Моделі паралельних пакетних обчислень даних, такі як MapReduce [8] і відповідні системні реалізації в Hadoop і Spark, усі мають важливе спільне, вони є втіленнями Bulk Synchronous Processing Valiant (BSP [16]) моделі для розподілених обчислень. Відповідно до моделі BSP розподілене обчислення організовано в конкретні надкроки (паралельні кроки) і складається з наступних трьох основних елементів:

- розподілених компонентів/процесів, що виконують локальні інструкції, змінюючи локальний стан;
- абстракцію маршрутизатора який маршрутизує повідомлення між будь-якими двома компонентами;
- примітивом синхронізації (sync()), який дозволяє прогрес лише тоді, коли суперкрок завершено для всіх компонентів.

Наша ітераційна модель програмування BSP може дозволити обмежену асинхронність, враховуючи модифікацію примітиву синхронізації, який враховує завислий параметр s Є N так, що синхронізаці дозволяє прогрес, коли інші паралельні ітераційні процеси відстають не більше ніж на s суперкроків від поточного завершеного суперкроку.

Для стислості ми опустимо визначення функції прогресу, але розглянемо можливість виражати та виконувати обмежену асинхронність у запропонованій нами моделі.

3.2 Опис процесів виконання завдань ітеративних процесів

3.2.1. Короткотривале виконання завдань ітераційних процесів

Реалізація групових синхронних паралельних ітераційних моделей на існуючих короткочасних розподілених обчислювальних системах була активною темою досліджень, насамперед у контексті популярної архітектури MapReduce [8]. Існуючі підходи доповнюють існуюче централізовано скоординоване поетапне виконання завдань систем на основі MapReduce (операції зіставлення, перетасування та зменшення) семантикою ітераційного процесу. Більш конкретно, такі системи, як Haloop, Twister, а також Spark [27] демонструють, що ітераційний процес можна розгорнути на серію етапів. На рисунку 3.1 ми узагальнюємо основні ідеї, де кожен надетап ітераційного процесу передбачає планування набору завдань без стану для його виконання, тоді як відстеження ітераційного прогресу та умови завершення оцінюються глобально на рівні програми, у рамках головного процесу завдання.

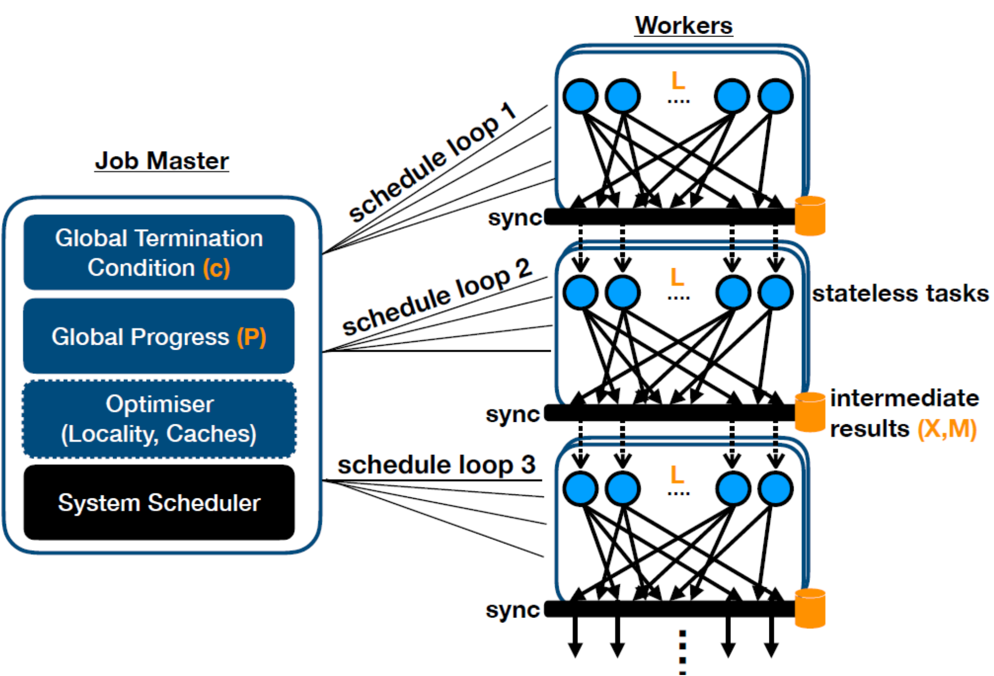


Рисунок 3.1 – Огляд суворо скоординованих ітераційних процесів у системах пакетних обчислень

Отже, кожен з примітивів ітераційного процесу, який ми визначили раніше, можна підтримувати наступним чином:

- **Обчислення циклу планування (L) і синхронізація:** Обчислення циклу зазвичай планується для запуску на спеціальному етапі Map-Reduce на кожну ітерацію, а точніше в межах фази зменшення, за якою слідує фаза відображення та перемішування для маршрутизації повідомлень, які використовуватимуться в наступному турі. Масова синхронізація за своєю суттю підтримується транзакційною обробкою послідовних етапів як у розширеннях MapReduce, так і в завданнях Spark. Це пов’язано з тим, що синхронізація вимагає маршрутизації всіх повідомлень і завершення обчислення перед виконанням наступного суперкроку. У всіх випадках суперкрок планується лише після завершення попереднього (тобто фази карти, перетасування та скорочення).
- **Маршрутизація та стан повідомлень (M, X):** Маршрутизація повідомлень вимагає, щоб набір згенерованих повідомлень направлявся до наступного екземпляра суперкроку ітерації. Фаза перемішування MapReduce може достатньо реалізувати цю функціональність, оскільки вона передбачає маршрутизацію та групування повідомлень за ключем. Подібним чином стан ітерації (якщо підтримується) також має бути зафіксовано в розподіленій файловій системі або відтворено в пам’яті (у випадку Spark), щоб використовуватися на наступному суперкроці ітерації, оскільки заплановані завдання в усіх моделі пакетного виконання за своєю суттю не мають стану. Це створює додаткову складність зв’язку, однак у більшості підходів центральний оптимізатор може переконатися, що локальність даних зберігається на максимальному рівні, перепланувавши завдання через ітерації в одному фізичному розділі для доступу до тих самих даних (на Spark це явно досягається за допомогою операції cache(). Крім того, для розділів з інваріантним станом через ітерації (наприклад, в ітераціях фіксованих точок)

може використовуватися кешування, щоб уникнути завантаження та перемішування без потреби.

- **Умова завершення та прогрес (c, P):** Зазвичай компонент відстеження прогресу викликається після завершення етапу, потім він збільшує глобальний лічильник суперкроків і викликає глобальну перевірку умови завершення. Для фіксованих ітерацій умова тривіально базується на лічильнику циклу, однак для завершення фіксованої точки умова має враховувати фактичні дані. Децентралізованим, але дорогим підходом було б додати ще один крок карти/зменшення для порівняння станів останніх двох ітерацій. Натомість Haloop пропонує використання кешування та індексування локального виводу редукторів. Цей підхід уникає необхідності спеціального кроку зменшення карти для реалізації перевірки завершення фіксованої точки, оскільки перевірка може бути виконана безпосередньо на редукторах, які завершують результат ітерації. Нарешті, зрештою, усі завдання скорочення повідомляють про свої рішення головному вузлу завдання через асинхронний зв'язок, а остаточне рішення щодо планування приймається на головному вузлі.

Було запропоновано багато оптимізацій для підвищення продуктивності ітераційних процесів на додаток до короткочасного розподіленого виконання завдань. Однак ми визначаємо кілька критичних обмежень. Незважаючи на те, що кілька систем можуть усунути одне з цих обмежень, жоден підхід, який базується на наших знаннях, не враховує всі з них за допомогою універсального дизайну системи. Першим обмеженням є можливість забезпечувати довільні потоки керування, такі як планування вкладених ітераційних процесів. Це часто відбувається через і без того високий ступінь перемикання контексту, необхідного для передачі (або кешування) стану та повідомлень від одного етапу до іншого. Введення вкладених ітераційних станів і повідомлення повинні включати всі дані процесів інкапсуляції ітерації, що робить прийняття вкладених структур складним і неефективним. По-друге, критерії прогресу та завершення є централізованими. Це

фундаментальне обмеження короткочасних структур виконання завдань, враховуючи те, що завдання не мають можливості підтримувати постійне уявлення про виконання разом із показниками прогресу та довільним змінним станом для прийняття локальних рішень, тому централізована координація та бухгалтерія неминучі в багато випадків. Нарешті, важко реалізувати примітиви слабкої синхронізації, такі як застарілі синхронні ітерації поверх системи пакетної обробки, якщо повністю не змінити її внутрішні механізми. Наявність чітких етапів виконання в усіх цих системах, включаючи Apache Spark, робить відображення глобального суперкроку в один етап необхідною незручністю.

3.2.2. Довготривале виконання завдань ітераційних процесів

Концепція довготривалого виконання завдань тісно пов’язана з потоковою обробкою, і її можна підсумувати як поняття розгортання та запуску графа завдань, які виконують усі етапи обчислення, замість планування кожного набору завдань на окремих етапах. Цей підхід в ітераційній обробці зазвичай застосовується в обчислювально інтенсивних областях аналізу даних, оскільки він може забезпечити кращу продуктивність завдяки використанню асинхронності та постійного локального стану (тобто без перемикання контексту). Основні зусилля для забезпечення безперервного ітеративного виконання були зроблені у відповідних областях застосування. У обробці графів такі системи, як Kineograph і Chronos [34] полегшують виконання алгоритмів ітераційних графів із сильними вимогами до локальності. Подібним чином, подібні зусилля були зроблені в галузі машинного навчання та статистичної апроксимації для мультиплексування ітераційних процесів і поступових оновлень протягом тривалого виконання завдань. Tornado [48] є прикладом системи, яка може обмінюватися наближеними результатами стохастичного градієнтного спуску (SGF) через різні довгострокові ітераційні наближення. Хоча всі ці підходи вирішують

певні ітераційні проблеми, їм бракує загальності, коли не можна зробити певних припущень щодо критеріїв завершення (конвергенції) і властивостей функції циклу (наприклад, чи є він опуклим). Кілька заслуговують на увагу зусиль [10, 20], спрямованих на створення загальнішої моделі програмування та виконання для ітерацій довгострокових завдань, які мають декілька спільних характеристик, зображених на рисунку 3.2, які ми підсумувати нижче.

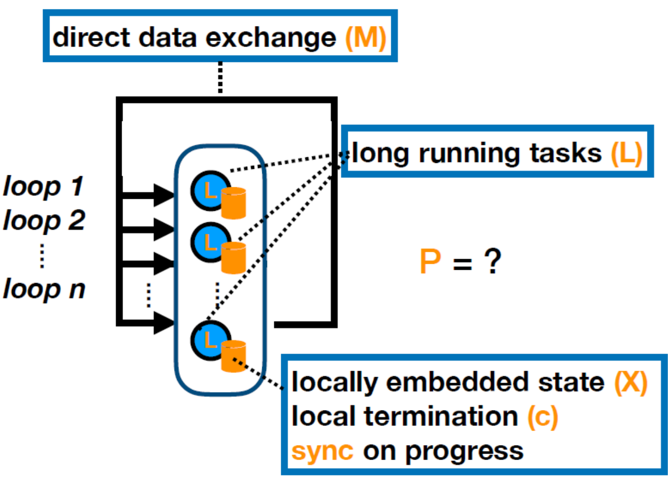
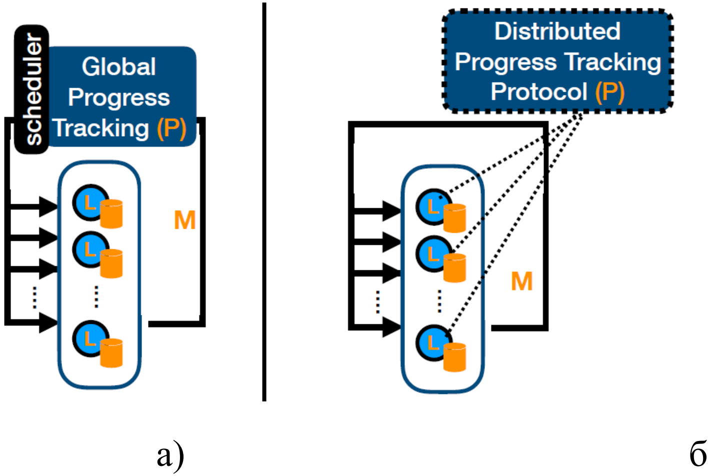


Рисунок 3.2 – Огляд ітераційної обробки довгострокових завдань

- **Обчислення циклу, стан і завершення (L, X):** обчислення та стан під час тривалого виконання тісно інтегровані. Як ми вже розглядали в контексті нашої основної моделі потокового процесу, кожен виклик завдання керується повідомленнями та призводить до згенерованого виводу та переходу стану. Це природна модель для реалізації ітераційних процесів, враховуючи розділений стан X і його поступові перетворення, що відбуваються на етапах ітерації. Завдання взаємопов'язані, а цикли – це фактичні канали даних, які повертають повідомлення в топології потокової обробки (у SEEP [10] цикли також можуть бути локально оцінені неявно, коли в обчисленнях не беруть участь паралельні залежності даних, однак ітераційний процес примітиви та синхронізація завдань не є основним напрямком цієї роботи). Враховуючи те, що повідомлення перемішуються під час виконання через цикли (порівняно з тим, що вони зберігаються або

кешуються під час короткочасного виконання завдання), можна вбудувати спеціальну логіку (наприклад, локальні оператори), яка викликає умову завершення на основі даних, якими обмінюються протягом ітерацій.

- **Маршрутизація повідомлень (M):** важливою відмінністю від короткочасного виконання є те, що повідомлення (M) безперервно створюються та споживаються. Як ми вже бачили, залежності даних у графі потокового процесу відповідають фізичній конфігурації мережі, де мережеві канали з'єднують різні частини обчислень. Це дає велику перевагу для ітераційної обробки, дозволяючи менш синхронну, конвеєрну обробку логіки циклу, що може полегшити застаріле синхронне виконання, парадигму, яку інакше було важко реалізувати в короткострокових системах виконання.
- **Синхронізація та прогрес (P):** Залишилася та найскладніша вимога до асинхронного тривалого виконання — це можливість відстежувати прогрес паралельних підзадач та зробити висновок, коли обчислення завершено. Загальна відсутність центрального координатора робить цю проблему нетривіальною, оскільки прогрес, як правило, потрібно наближати децентралізовано без будь-яких глобальних знань. Ця проблема, яку ми неофіційно називаємо «відстеженням прогресу», включає в себе потребу в механізмі розподіленої синхронізації, оскільки він може надавати інформацію про те, коли обчислювальний крок завершено на рівні кожного окремого завдання.



а) б)

Рисунок 3.3 – Глобальне та децентралізоване відстеження прогресу

Розрізняють два відповідних підходи до відстеження розподіленого прогресу для довгострокової синхронізації завдань, що зображено на рисунку 3.3 : a) глобальний і b) розподілений або децентралізований.

Глобальне відстеження прогресу відомо із системи Naiad [20] та її моделі Timely Dataflow. Naiad надає примітиви програмування та архітектуру часу виконання для підтримки довільних вкладених ітерацій у тривалому виконанні завдання. Її модель виконання має деякі варті уваги відмінності порівняно з пропонованою моделлю потокового процесу. По-перше, весь обмін повідомленнями контролюється глобальним (для кожного комп'ютера) блоком планування повідомлень, який перехоплює зв'язок і оновлює спільну структуру даних показників прогресу (на основі кількості незавершених записів на унікальну позначку часу), яку ми можемо назвати «трекером». Роль трекера полягає в тому, щоб підтримувати остаточно послідовне уявлення про «годинник», який, у будь-який час може дати наближену оцінку прогресу обчислення (відомого як кордон) у всіх частинах графіка потокового процесу. Потім локальний планувальник може планувати події сповіщень для завдань на основі цього прогресу та дозволяти емуляцію бар'єрів синхронізації. Глобальне відстеження прогресу показало, що добре працює, враховуючи грубі метрики, такі як обчислювальні епохи та кроки ітерації. Однак його адаптація до повністю децентралізованого виконання графа потокового процесу була б нереальною, враховуючи локальне незалежне виконання останнього від завдань, а також необхідність включення існуючих децентралізованих механізмів, таких як наддрібні часові вікна подій [22] і знімки на основі маркерів [29, 30].

Мало зусиль було докладено для відстеження ітераційного прогресу довгострокових завдань у децентралізованому режимі. Основна ідея такого підходу полягає в тому, щоб дозволити обчислювальним завданням виводити розподілений прогрес ітерації (наприклад, завершення суперкроків) просто за допомогою існуючого засобу обміну повідомленнями в графі потокового процесу без будь-якого зовнішнього обходу. Такий підхід може бути

багатообіцяючим для більш широкого впровадження ітераційної потокової обробки на існуючих платформах виробничого масштабу, які не враховують жодного зовнішнього зв'язку чи блоку керування у виконанні. Серед існуючих досліджень підхід Flying Fixpoint (FFP) показує хороші результати прийняття протоколу під час польоту для відстеження прогресу в контексті підтримки рекурсивних запитів Datalog під час виконання потокової обробки.

3.3 Модель масштабування для позачергової ітеративної обробки даних

Нашою задачею є процес підтримки довільної вкладеної ітераційної обробки на розгортаних графах потокових процесів довгострокових завдань із збереженням стану. Більш конкретно, ми прагнемо реалізувати всю семантику ітераційного процесу поверх загальної моделі потокового процесу. Забезпечення повністю децентралізованого ітераційного виконання без блокування та координації є основним принципом розробки нашого підходу та загальною вимогою нашої основної моделі потокового процесу.

Наш підхід базується на понятті позачергової обробки (OOP), яка використовується для обчислення блокування на основі прогресу. Ми визначаємо ітераційну обробку як особливий випадок OOP і надаємо повне децентралізоване рішення для інтеграції показників циклічного прогресу в схему OOP.

3.3.1. Позачергова обробка потоку

Обчислення в графах потокових процесів можуть призвести до різних розподілених виконання, оскільки завдання працюють незалежно одне від одного, перетягуючи записи через свої вхідні канали та обробляючи їх без дотримання будь-яких гарантій загального порядку. З одного боку, це робить потокову обробку швидкою архітектурою без координації для обробки з

низькою затримкою. З іншого боку, це вимагає, щоб потенційні механізми синхронізації були створені на основі доставки записів поза порядком через різні вхідні потоки.

Позачергова обробка [22, 23] (OOP) — це парадигма обробки, яка будується на основі типової (асинхронної) моделі потокового процесу, щоб запропонувати примітиви прогресу та синхронізації. Початкова потреба в OOP полягала у відсутності узгодженого механізму міркування про загальний порядок обробки для обчислень на основі часу (наприклад, часові вікна програми) у потоковому виконанні. Модель потоку даних Google [21], яка наразі розроблена в рамках Apache Beam [42], популяризувала поняття OOP, яке пізніше було прийнято моделлю програмування Flink серед інших і реалізовано за допомогою використання низьких водяних знаків [22, 9]. У цьому розділі ми пояснимо модель OOP з наголосом на децентралізованому механізмі відстеження прогресу для OOP, який наразі обмежений ациклічними потоковими графами процесів, але може бути розширений для підтримки ітеративної обробки.

Незважаючи на те, що потокові завдання постійно викликаються для кожного вхідного запису, логіку, яка виконується в цих завданнях (яку ми будемо називати «оператор»), можна класифікувати як блокуючу або неблокуючу. Неблокуючі оператори – це ті, чия логіка не чутлива до порядку. Наприклад, простий фільтр або об'єднання це оператори потоку, які викликають локальне обчислення та створюють незалежний вихід для кожної події введення.

З іншого боку, існує клас блокуючих операторів, виконання яких залежить від загальної міри порядку. Наприклад, вікно потоку часу застосування буде завершено, коли всі записи до певного часу будуть гарантовано оброблені. З огляду на відсутність атомарного годинника «програми», неможливо в будь-який час отримати доступ до цього точного показника для кожного окремого фізичного завдання. Фактично, загальне замовлення на обробку цього заходу було б нездійсненним без суворо

скоординованого виконання. Натомість парадигма позачергової обробки, як випливає з назви, дозволяє невпорядковану обробку з використанням «метрики прогресу c1» , слабшого поняття, ніж повний порядок обробки, але достатнього для потреб операторів потоку блокування. В означенні 3.1 ми представляємо поняття метрики прогресу для попередньо визначеного загального порядку записів у виконанні обробки потоку, визначеного у зв'язку з нашою основною моделлю процесу потоку.

Означення 3.1. Метрика прогресу: Нехай R — потік даних, а час події ts — загальне відношення порядку ts : R → N. Враховуючи перехід системи потоковим завданням p Є П зі станом області S, метричне відношення прогресу T : S → N задовольняє таке:

- $T(s_p^i) \leq \mathbf{ts}(m^i)$
- $T(s_p^i) \leq T(s_p^{i+1})$ (monotonicity)

Суть метрики прогресу полягає в тому, що різні операції блокування можуть бути асинхронно викликані записами, які надходять не за порядком (тобто, коли $ts(m^i) > ts(m^{i+1})$). Однак певна логіка оператора вимагає лише монотонної метрики для завершення. Враховуючи те, що потоки можуть бути довільно невпорядкованими ще до того, як вони будуть прийняті системою обробки потоків, потрібно встановити визначене користувачем обмеження щодо максимального очікуваного ступеня невпорядкованості або затримки часу, що також впливає на те, як довго система повинна чекати, поки видає оновлення своєї метрики прогресу.

Ми називаємо «відстеження прогресу» основним механізмом, який реалізує метрику прогресу виключно шляхом спостереження та взаємозв'язку часових позначок записів у системі потокової обробки. Low watermarking [22] є широко використовуваним механізмом сьогодні для відстеження прогресу розподілених подій. Основна інтуїція, що стоїть за Low watermarking, подібна до нашого алгоритму знімка, представленого в розділі 2, який враховуючи

ациклічний графік пов’язаного потокового процесу, можна розповсюджувати через оновлення метрики прогресу у вигляді записів пунктуації, починаючи з джерел. Повна доступність графа потокового процесу в поєднанні з каналами FIFO гарантує, що такі події пунктуації досягнуть кожного завдання в тому ж порядку, в якому вони були видані в джерелах.

Low watermarking можуть бути видані за допомогою різних стратегій (наприклад, на періодичних інтервалах часу [23,21]) у вихідних завданнях графа, однак їх базові гарантії та логіка виведення показників прогресу є загальною для всіх потоків. На рисунку 3.4 ми показуємо приклад основного механізму з точки зору фізичного завдання, яке з часом споживає різні відповідні вхідні потоки.

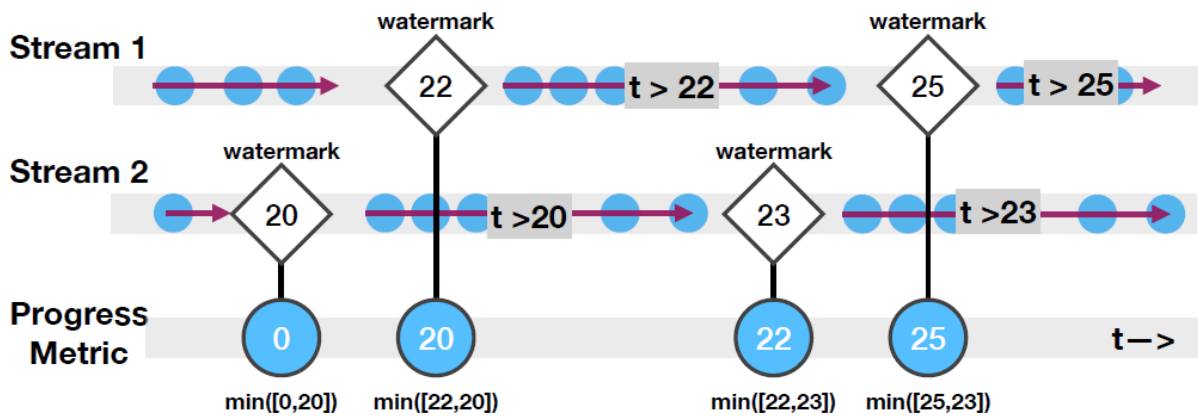


Рисунок 3.4 – Зображення помічених low watermarking і похідної метрики прогресу

Кожен потік даних збагачується періодичними водяними знаками, які вказують на метрику прогресу, яка задовольняється для решти конкретного потоку. Наприклад, якщо watermarking має значення 20, це означає, що з цього моменту у відповідному потоці не спостерігатиметься запис із міткою часу $t < 20$. Завдання, яке використовує кілька потоків, оновлює свою метрику прогресу на основі найконсервативнішого значення метрики прогресу (тобто мінімальної кількості останніх watermarking, помічених у вхідних потоках на

даний момент). Враховуючи початкове значення t = 0 у прикладі, локальна метрика прогресу оновлюється, коли мінімальний водяний знак у вхідних потоках стає вищим за 0. Це відбувається, коли спостерігається водяний знак із t = 22 і метрика оновлюється до 20 (оскільки min([20,22])). Механізм продовжує працювати таким же тривіальним способом, зберігаючи останній watermarking, спостережуваний для кожного каналу і оновлюючи його локальну метрику прогресу.

Ми додаємо відстеження прогресу та обробку показників прогресу як просте розширення до нашої початкової моделі процесу. Спеціальну логіку для подій водяних знаків узагальнено в алгоритмі 1 (рис. 3.5):

```
Algorithm 1 : Process Logic for Low Watermarking
1:  (I_p, O_p) ← configured_channels;
2:  s_p ← ∅;                                   ▷ volatile local state
3:  T ← 1_W;                                   ▷ the local progress metric
4:  watermarks ← {};                           ▷ last watermarks received per-channel
5:  foreach in ∈ I_p do
6:    watermarks(in) ← 1_W;
7:  /* Common Task Logic                                                         */
8:  Upon ⟨rcvd, W(ts)⟩ on in ∈ I_p
9:    tmp ← T;
10:   watermarks(in) ← ts;
11:   T ← min(watermarks.vals);
12:   if tmp ≠ T then
13:     s_p ← processOnTime(T, s_p, O_p);
14:     foreach out ∈ O_p do
15:       out → ⟨send, W(T)⟩;
16: Upon ⟨rcvd, m⟩ on in ∈ I_p
17:   ...;
```

Рисунок 3.5 – Алгоритм для водяних знаків

Розширення включає такі важливі зміни: на додаток до логіки процесу за замовчуванням у вхідний запис ми включаємо processOnTime, який вказує програмі на (монотонну) зміну показника прогресу (T). Подібно до функції

process, processOnTime може однаково призводити до вихідних повідомлень, а також зміни стану volatile відповідно до логіки оператора. Враховуючи, що водяні знаки надходять у порядку на канал FIFO, але не впорядковано між каналами, цей метод гарантує, що завдання завжди містить найбільш консервативне наближення метрики прогресу. Спочатку для кожного watermarking встановлюється ідентифікаційне (мінімальне) значення, а для кожного вхідного значення показник прогресу оновлюється до відповідного мінімуму. Після зміни T викликається processOnTime (рядок 13) і новий вихідний watermarking (W(T)) транслюється на всі вихідні канали, щоб решта графа обробки оновлювала своє значення (рядок 15) .

Приклад використання: ми можемо продемонструвати використання low watermarking за допомогою вікна із зміщенням 10 секунд, і ми припускаємо, що для спрощення всі записи містять мітки часу, які представляють секунди. Початкова метрика прогресу нашого оператора вікна (Win) наразі встановлена на 0 і обчислення відбувається не в порядку. На рисунку 3.6 (a) оператор отримує запис із t = 12 і починає нове вікно потоку для інтервалу [11–20], незважаючи на те, що вікно [1–10] ще не завершено. Обчислення продовжується у непорядковому режимі навіть у точці, коли оператор отримує low watermarking з одного зі своїх каналів, що вказує W = 22 (рис. 3.6 (b)). Його метрика прогресу залишається як W = 0, оскільки watermarking вище нуля ще не отримано від іншого каналу. Нарешті, зміна метрики прогресу відбувається на рисунку 3.6 (c), коли оператор отримує low watermarking від іншого незавершеного каналу, що вказує W = 11. Оскільки min([22,11]) = 11, його локальна метрика прогресу оновлюється до 11 і це запускає завершення роботи його вікна в діапазоні [1-10], оскільки 10 < 11 з гарантією того, що в майбутньому більше не надійде записів нижче часу 11. Потім оператор виводить результат обчислення (з тегом часу завершення t = 11), очищає стан вікна і, нарешті, транслює low watermarking для W = 11. Це допомагає наступним операторам зробити висновок про зміну прогресу як пояснювалося раніше в алгоритмі 1 (рис. 3.5).

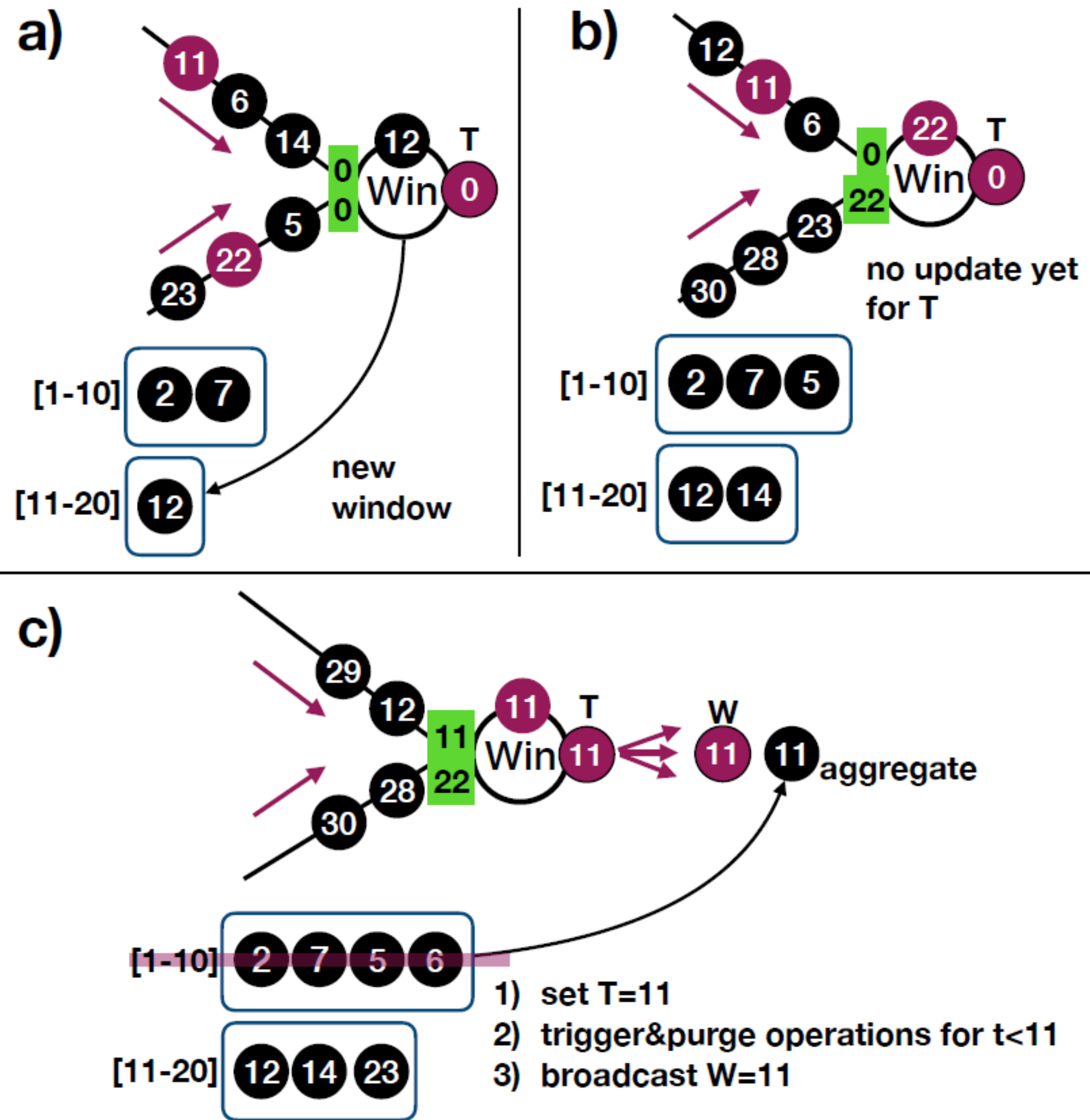


Рисунок 3.6 – Приклад low watermarking з інтервалом оновлення 10 с

3.4 Децентралізоване відстеження прогресу для часткового порядку подій обробки даних

Загальний підхід зосереджується на концепції часткового ітераційного порядку виконання, яку ми опишемо в цьому підрозділі.

Ми визначили показники прогресу як достатній засіб фіксації прогресу повністю впорядкованого атрибута, такого як час події у виконанні розподіленого потоку. З кожним вкладеним структурованим циклом ми фактично запроваджуємо нову метрику прогресу, яка фіксує глобальний

частковий порядок повідомлень для кожної окремої мітки часу, яка входить до його області. Це дає перспективу досягнення позачергової обробки для різних часових позначок, а також для паралельних суперкроків, що виконуються для кожної часової позначки. Ми стверджуємо, що часткового порядку достатньо для відстеження прогресу в контексті реалізації ітераційних процесів на потоках даних. Це, по-перше, через те, що масові ітерації потребують фіксованого набору даних для роботи (тобто набору входів X) замість необмеженого потоку змін. Очевидно, що достатньо охопити ітерації, наприклад обмежити їх обчислення рівнем кожного вікна потоку (або «епохи», як визначено в проекті Naiad [20]) і оскільки вікна можуть оброблятися поза порядком, частковий порядок метрики відстеження достатньо для цієї мети. Нарешті, ще одна причина, чому ми не розглядаємо загальний порядок, це необмежений характер ітераційних процесів. Реалізація ітераційних процесів є основною метою цієї роботи, але кожен ітераційний процес може приймати довільну кількість кроків для завершення Якщо ми блокуємо виконання, доки не завершиться кожен послідовний ітераційний процес, ми фактично перетворюємо наше асинхронне виконання на послідовне блокуюче виконання, яке не відповідає парадигмам потокової передачі даних і обробки поза порядком загалом.

Порядок часової позначки часткового виконання. Ми розрізняємо дві частини в кожній мітці часу прогресу: 1) його поточний прогрес P^T і його контекст P^{ctx} :

$$P = [\overbrace{p^n}^{P^T}, \underbrace{p^{n-1}, p^{n-2}, \ldots, p^0}_{P^{ctx}}]$$

Зазвичай нам потрібно відстежувати прогрес повідомлень щодо певного контексту. Цей тип відношення часткового порядку описано в означенні 3.2.

Означення 3.2. Порядок часткової мітки часу: повідомлення в дорозі в межах того самого діапазону частково впорядковуються за міткою часу виконання $P \in N^n$ таким чином:

$$P, P' \in \mathbb{N}^n : P \geq P' \text{ iff } (P^{ctx} = P'^{ctx}) \wedge (P^T \geq P'^T)$$

```
Algorithm 2 : Context-Based Low Watermarking
1:  (I_p, O_p) ← configured_channels;
2:  s_p ← ∅;                                        ▷ volatile local state
3:  T ← {};                          ▷ the local progress metric per context
4:  watermarks ← dict();  ▷ latest metrics received per context and channel
5:  /* Common Task Logic                                                */
6:  Upon ⟨rcvd, W(P)⟩ on in ∈ I_p
7:      if T(P^ctx) = nil then
8:          initialize(P^ctx);
9:      tmp ← T(P^ctx);
10:     watermarks(P^ctx)(in) ← P^T;
11:     T(P^ctx) ← min(watermarks(P^ctx).vals);
12:     if tmp ≠ T(P^ctx) then
13:         (s_p, c_pq) ← processOnTime(T(P^ctx), P^ctx);
14:         foreach out ∈ O_p do
15:             out → ⟨send, W(T(P^ctx) :: P^ctx)⟩;
16:     if T(P^ctx) = ø then
17:         watermarks(P^ctx) ← T(P^ctx) ← nil;          ▷ progress purging
18: Fun initialize(ctx)
19:     watermarks(ctx) ← {};
20:     foreach in ∈ I_p do
21:         watermarks(ctx)(in) ← 1_W;
22:     T(ctx) ← 1_W;
```

Рисунок 3.7 – Алгоритм для відстеження прогресу

Low watermarking можна використовувати для відстеження прогресу часткового порядку, просто запустивши новий екземпляр протоколу для унікального спостережуваного контексту. Це означає, що кожному оператору потрібно буде підтримувати метрику прогресу для кожного контексту, а також видавати водяні знаки для кожного оновлення прогресу, яке відбувається в конкретному контексті. Крім того, коли обчислення в певному контексті завершилося (що позначається символом завершення прогресу 0), оператору

доведеться очистити відповідну метрику прогресу. Спочатку ми обговоримо оновлену логіку для low watermarking у вкладених областях, а потім доведемо, як вбудовані графи, які ми представили раніше, забезпечують необхідні неявні гарантії, пов'язані з прогресом.

В алгоритмі 2 (рис. 3.7) ми підсумовуємо оновлену логіку, що виконується кожним завданням у системі для відстеження прогресу. Щоб дозволити часткове виконання порядку у вкладених областях, ми розширюємо всі базові операції на час, щоб вони були обмежені контекстом. Починаючи з локальної метрики прогресу, кожне завдання тепер потребуватиме збереження окремої метрики прогресу, ініціалізованої для кожного контексту (рядок 18) і випуску водяних знаків (рядок 15) , які містять повні мітки часу виконання W(P), де P інкапсулює контекст (P^{ctx}) . і показник прогресу для цього контексту (P^{T}). Крім того, ми розширюємо processOnTime, щоб включити контекст кожного оновлення прогресу, щоб оператори могли охоплювати обчислення для певного контексту (наприклад, для розділення та очищення стану для кожного контексту). Крім того, коли локальна метрика прогресу певного контексту досягає значення завершення (позначеного як Ø), усі стани очищаються (після запуску обчислення за допомогою виклику processOnTime). Нарешті, у звичайному випадку завдань, які не є частиною структурованого циклу (у загальному обсязі), логіка ідентична вихідному алгоритму, оскільки дефакто розглядається лише один контекст, контекст порожнього вектора.

Правильність підходу. Схема watermarking на основі контексту разом із операціями вбудованого графіка може гарантувати, що метрика часткового порядку, яка відстежується в кожній області, не буде порушена. Тобто для кожного значення прогресу v Є R, зазначеного в контексті a Є N^{n-1}, це має гарантувати, що жоден запис із часовою міткою прогресу P, де $P^{T} \leq v$ і $P^{ctx} = a$, не буде оброблено в решті виконання.

Оскільки правильність пов'язана з частковим порядком, давайте розглянемо лише події, які знаходяться в контексті a. У підмножині

виконання, яке має відношення до контексту, протокол має ідентичну логіку оригінального алгоритму low watermarking тобто «найменше, найбільш консервативне значення для останніх low watermarking, отриманих через вхідні канали (потоки), визначає фактичне значення метрики прогресу». Єдина відмінність полягає в тому, що він виконує паралельний екземпляр для кожного контексту. Крім того, ядро доводить, що стратегія low watermarking гарантує коректність у розподіленому ациклічному графі [22] з каналами FIFO, оскільки watermarking ніколи не передує запису, який має нижчу позначку часу. Тут нам потрібно довести, що спеціальна вбудована логіка в структурованому циклі все ще може підтримувати цей інваріант.

Розглянемо повний набір повідомлень M з P = a, які надходять у структурований цикл M з різних потоків. Враховуючи логіку низьких водяних знаків за умовчанням, ці повідомлення завжди передуватимуть водяному знаку W(a). Якщо ми тепер розіб’ємо розподілене виконання в рамках структурованого циклу, події слідуватимуть за чіткою послідовністю кроків, як показано на рисунку 3.8. Ми досліджуємо, як наш інваріант не порушується через індукцію на транзитивне закриття повідомлень, отриманих під час ітераційного виконання.

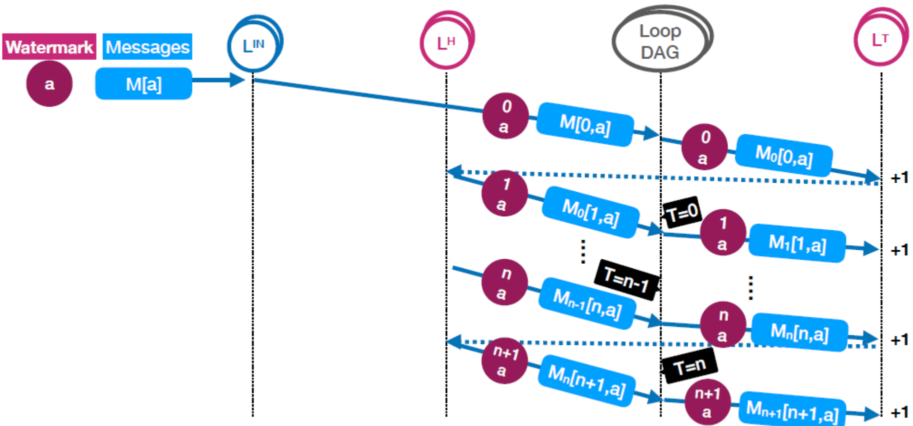


Рисунок 3.8 – Повідомлення та low watermarking для кожного контексту в структурованому циклі

Крок 0: кожне повідомлення проходить через оператор запису зі збереженням порядку (L^{IN}), який перетворює a на контекст і додає метрику прогресу зі значенням 0 до кожного повідомлення, яке пересилається, після чого ставиться водяний знак, що позначає завершення введення за допомогою значення вкладеної метрики 0. Обчислення на першому ітераційному кроці відбувається в межах наданого користувачем графа, який сам по собі можна вважати DAG (вкладені структуровані цикли також поводяться як один оператор). Отже, враховуючи початкову логіку протоколу, транзитивне закриття виводу, згенерованого циклом DAG, є не чим іншим, як набором повідомлень M_0 у момент часу 0, за якими слідують переслані водяні знаки часу 0, які закінчуються в хвості циклу. Наш інваріант поки що гарантований, доки кожне повідомлення кроку 0 не прийде в хвіст.

Крок 1: Хвіст циклу збільшує мітки часу повідомлення та пересилає їх назад до DAG через оператор head у строгому порядку FIFO. Це означає, що фактично транзитивне закриття повідомлень, що надходять на другий крок ітерації, буде M_0, а прогрес 1 супроводжуватиметься low watermarking часу 1. Порядок у межах M_0 такий неважливий (і зазвичай не підтримується через перетасування каналів у DAG), якщо watermarking не вказують на незавершені кроки (наприклад, на цьому етапі, якщо W(t>1 :: a)). Процес продовжується, як на кроці 0, зберігаючи інваріант.

Крок n: Подібним чином, якщо ми розглянемо M_n, повний вихідний набір повідомлень, який цикл DAG генерує на кроці n, він, природно, передує watermarking часу n, які, у свою чергу, знову подаються в DAG із збільшеними часовими мітками.

За допомогою індукції на транзитивне закриття повідомлень, створених під час розподіленого виконання і кроків індукції, обговорених вище, гарантується, що watermarking W(n :: a) завжди слідуватиме за транзитивним закриттям повідомлень $\{M_o,..., M_n\}$. Оскільки локальні показники прогресу не можуть перевершити вхідні водяні знаки, наш основний інваріант завжди гарантується внутрішньо в структурованому циклі для кожного контексту.

Наша схема структурованого циклу повинна гарантувати завершення на основі визначеного користувачем критерію. Припинення відбувається, коли всі завдання в рамках структурованого циклу отримують повідомлення про завершення обчислення для визначеного контексту a. У нашій моделі ми використали спеціальний показник прогресу (Ø), щоб поширити поняття завершення.

Після сповіщення про завершення вбудовані та спеціальні оператори, які зберігають прогрес (L^T, L^{OUT}) можуть ініціювати незавершені обчислення, а також очистити збережені показники прогресу та водяні знаки введення для завершеного контексту a. Припускаючи, що критерій завершення гарантовано врешті-решт буде виконано на всіх екземплярах заголовка циклу, ми почнемо з цього, щоб перевірити, як розповсюджуються повідомлення про завершення, також зображене на рисунку 3.9.

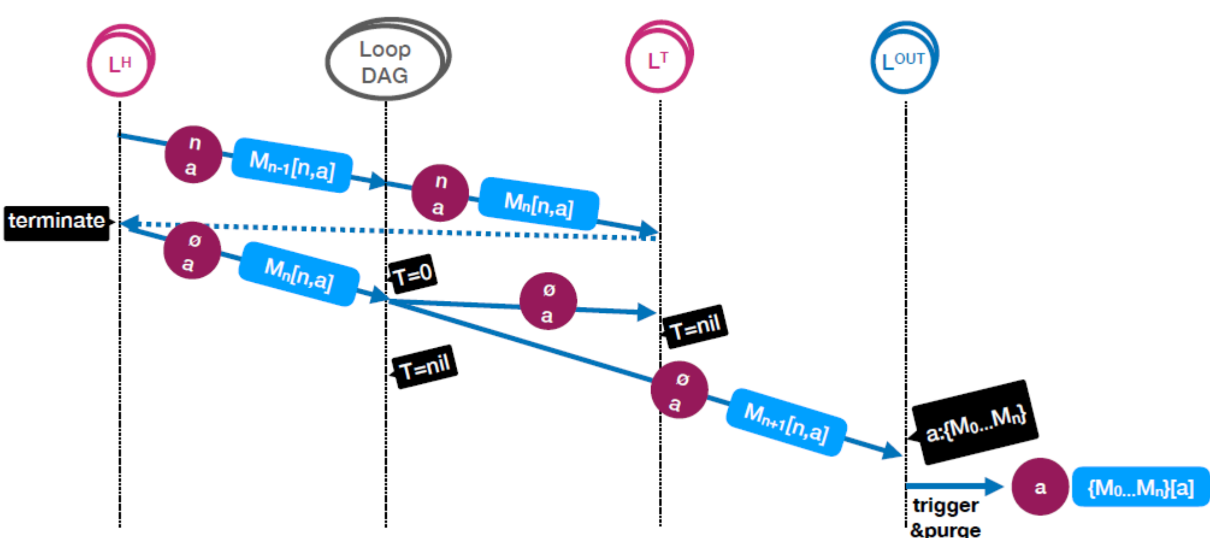


Рисунок 3.9 – Завершення контексту в структурованому циклі

Відповідно до логіки, яку виконує L^H, критерій завершення визначається операторами заголовка циклу під час отримання watermarking (loopFun викликається як запис для створення залежних від даних критеріїв завершення з даних).

3.5 Реалізація підтримки ітераційних процесів на основі Apache Flink

У цьому розділі ми представляємо модель програмування, яка забезпечує підтримку ітераційних процесів у вікнах потоку, реалізовану як розширення фреймворку Apache Flink. З огляду на структуру програмування, орієнтовану на користувача, ітераційні процеси можна розглядати як особливий випадок віконної агрегації, який вимагає багаторазових проходів даних.

Представлений дизайн розширює кілька рівнів стеку Apache Flink, як показано на рисунку 3.10. На рівні середовища виконання ми забезпечили підтримку системи для структурованих циклів і часових позначок прогресу відповідно до точної специфікації.

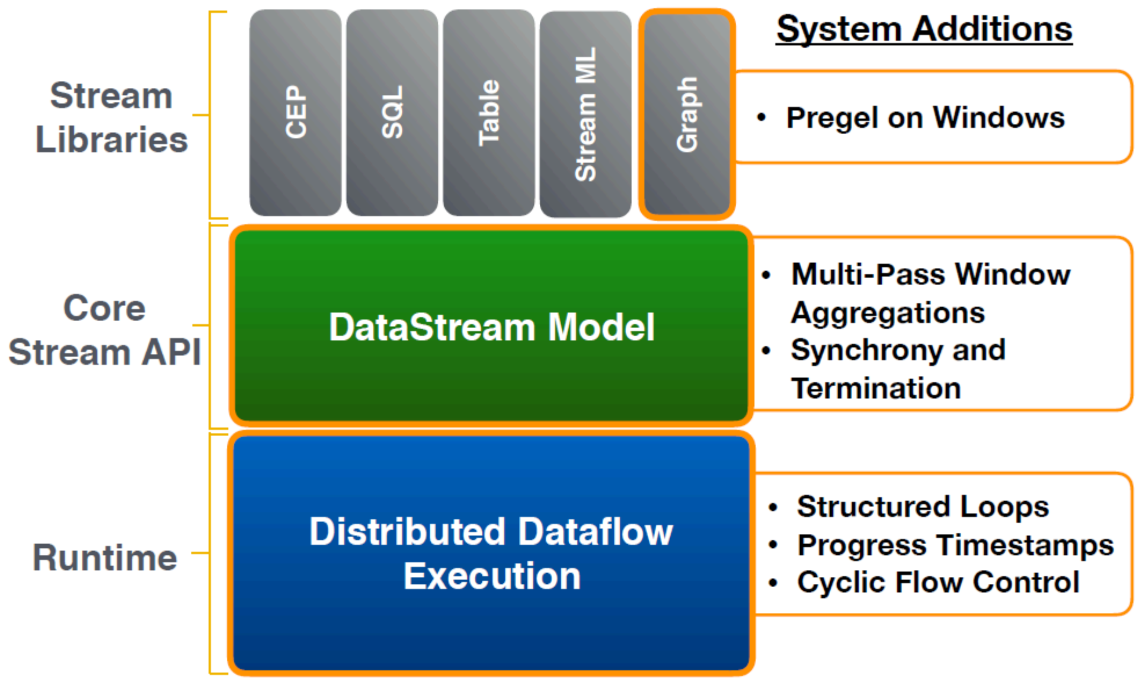


Рисунок 3.10 – Огляд системних доповнень на основі Apache Flink

Крім того, ми розширили логіку керування потоком ядра Flink, щоб усунути можливість тупикових блокувань у циклах. Як доказ концепції,

поверх структурованих циклів ми реалізували ітераційний оператор для вікон потоку. Наш дизайн інкапсулює всі базові примітиви, а саме різні умови завершення (c), синхронізацію (sync) і визначені користувачем покрокові функції (L), побудовані на основі структурованих циклів і керованого стану Flink. На додаток до бібліотек Flink, специфічних для домену (тобто комплексна обробка подій і Tables-Stream SQL), ми склали бібліотеку потоку графів для домену, яка використовує віконні ітерації для підтримки вершинно-центричного API графа, подібного до Pregel [17].

Модель віконної ітерації інкапсулює всі примітиви ітераційного процесу, представленого раніше, і застосовна до вікон паралельного потоку Flink. Кожен ітераційний процес можна описати за допомогою критерію завершення, опції синхронізації, ключа, який розділяє дані і набору з трьох основних функцій агрегації: вхід, крок і фіналізація, як підсумовано нижче:

```
1 val stream: DataStream[IN] = ...
2 stream.keyBy(...).window(...)
3    .iterate(<Termination>,<Synchrony>,<Key>,<Entry>,<Step>,<Finalize>)
```

На рисунку 3.11 ми підсумовуємо опис цих примітивів, включаючи надані системою властивості конфігурації для завершення та синхронізації.

LoopContext Properties	**Definition**
key: K	The current key of the computation
localProgress: ℕ	The current superstep of the local computation
globalProgress: ℕ	The last parallel complete superstep
eventTime: ℕ	The event time (ctx) of local computation
loopState: ManagedState	keyed state, purgeable per iterative process
persistentState: ManagedState	keyed state, persistent across iterative processes

Рисунок 3.11 – Змінні, доступні в контексті циклу (ctx)

Подібно до існуючої API Flink, доступ до керованого стану та метаданих агрегації надається через спеціальний об’єкт LoopContext (ctx на

рисунку 3.11) , який надає доступ до ключа обчислення, показників прогресу та двох типів читання та запис керованого стану:

- loopState для визначення змінних, які очищаються в кінці ітераційного процесу вікна;
- постійний стан для змінних, які спільно використовуються вікнами.

На рисунку 3.11 надає зведення всіх цих доступних властивостей. Слід зазначити, що оскільки керуються обидва типи стану, знімок системи і протокол фіксації епохи забезпечують його узгоджену обробку.

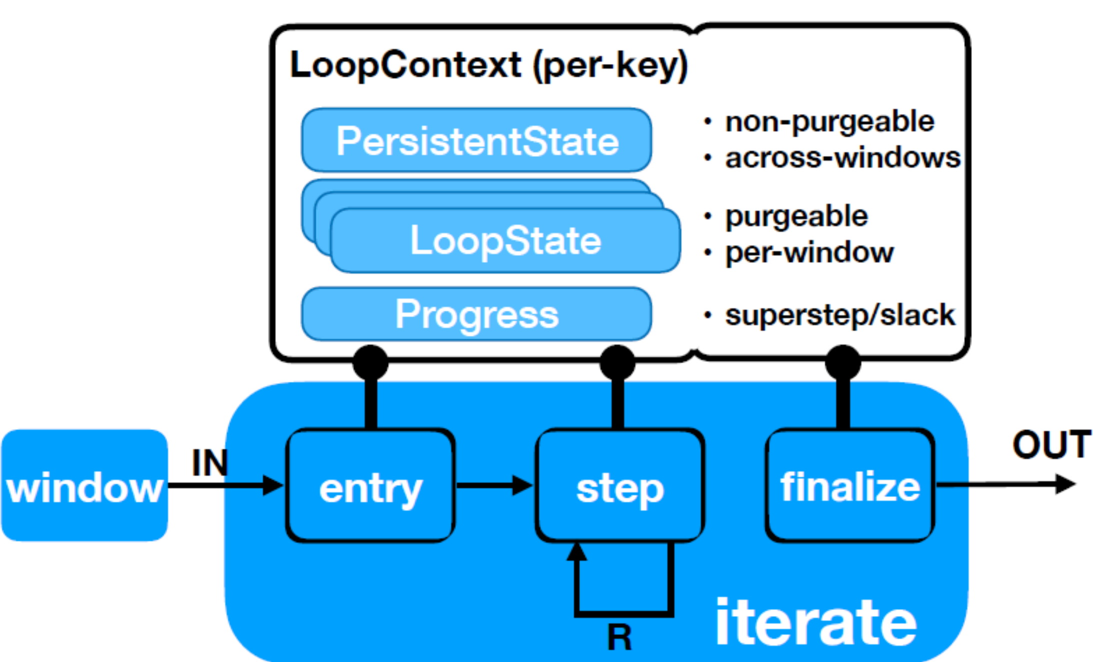


Рисунок 3.12 – Внутрішні елементи оператора ітерації вікна

На рисунку 3.9 представлено логічний потік між функціями агрегації в операторі віконної ітерації та далі описуємо їх використання нижче.

Entry Function: викликається один раз (для кожного ключа), коли запускається вікно введення і дозволяє ініціалізувати стан ітераційного процесу (наприклад, визначення початкового стану і збереження його в контексті циклу). Крім того, все ітераційне обчислення запускається у функції

входу шляхом видачі повідомлень [R], які будуть передані назад до першого суперкроку.

Step Function: отримує всі оновлення зворотного зв’язку, створені для кожного ключа на попередньому суперкроку, і генерує нові оновлення для наступного суперкроку. Крім того, він має доступ до LoopContext ітерації вікна, маючи можливість читати та записувати змінні як у чистому, так і в постійному керованому стані.

Finalize Function: викликається, коли задовольняється критерій завершення (наприклад, коли на останньому кроці під час агрегації фіксованих точок не внесено жодних змін). У цьому випадку кінцевий результат агрегації має бути отриманий із контексту, або шляхом читання останнього стану вікна перед його очищенням, або з постійного стану. Це також дає зручний спосіб послідовного внесення будь-яких подальших необхідних змін до постійного стану після завершення ітерації вікна.

Висновки до розділу 3

Отже, в цьому розділі представлено модель для ітераційних процесів крок за кроком від її базового представлення до семантики розподіленого виконання та масової синхронної обробки даних в різних середовищах виконання. Описані всі запропоновані базові абстракції потокової моделі, необхідні для інтеграції ітераційних процесів у довгострокове виконання потоку незалежних завдань та прототип бібліотеки агрегації вікон, яка використовує запропоновані абстракції моделі потоку даних.

ВИСНОВКИ

В магістерській роботі розглянуто та досліджено процес оптимізації моделей масштабування процесів обробки потоків даних. Розглянуто випадок потокового процесора як обчислювальної платформи загального призначення, яка може підтримувати довільні постійні робочі навантаження, керовані станом. Парадигма потокової обробки є привабливим засобом оголошення постійної логіки додатків із станом над змінними даними, дозволяючи створювати надійні служби та складні додатки за межами масштабованої аналітики даних.

Під час цієї роботи було розглянуто обробку потоку даних, щоб забезпечити чітку модель виконання системи через вирішення низки фундаментальних проблем відкритої системи: надійне виконання зі збереженням стану, спільне використання обчислень і підтримка структурованих ітерацій. Крім того, розглянуто підходи, які дотримуються чітких цілей дизайну: уникнення координації, композиційність і прозорість моделі програмування.

Проблема надійного виконання зі збереженням стану пов’язана з гарантіями обробки, що надаються системою потокової обробки за наявності часткових збоїв або необхідності реконфігурації, і досі була однією з найбільших проблем. Пропонований підхід моделює розподілене довгострокове виконання на серію фаз, які називаються епохами, кожна з яких має завершувати та фіксувати зміни атомарно. Щоб узгоджувати стани між епохами без координації, ми використовуємо асинхронні епохові скорочення, які прагнуть використовувати легкі знімки стану для всієї програми після попередньо визначених епох виконання. Миттєві знімки епохальних скорочень можуть служити основою для побудови складних залежностей походження додатків і дозволити міграцію та реконфігурацію додатків, які виходять за рамки необхідності відмовостійкості, як це було продемонстровано в системі Apache Flink.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," ACM Computing Surveys (CSUR), vol. 44, no. 3, p. 15, 2012.

2. D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," VLDBJ, 2003.

3. A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The stanford data stream management system," Book chapter, 2004.

4. S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, "TelegraphCQ: continuous dataflow processing," in Proceedings of the 2003 ACM SIGMOD international conference on Management of data. ACM, 2003,

5. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in Proceedings of the twenty-first ACM SIGMODSIGACT-SIGART symposium on Principles ofdatabase systems. ACM, 2002.

6. J. Dean and S. Ghemawat, 'Mapreduce: simplified data processing on large clusters," Communications of the ACM, vol. 51, no. 1, pp. 107—113, 2008.

7. T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "MillWheel: Fault-tolerant stream processing at internet scale," in VLDB, 2013.

8. R. C. Fernandez, M. Migliavacca, l". Kalyvianaki, and I). Pietzuch, "Making state explicit for imperative big data processing," in 2014 USENIX Annual Technical Conference (USENIX ATC 14), 2014.

9. R. Castro Fernandez, M. Migliavacca, Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state

management," in Proceedings ofthe ACM SIGMOD international conference on ACM, 2013, pp. 725-736.

10. M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing. USENIX Association, 2012.

11. J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams," ACM SIGMOD Record, 2005.

12. S. Krishnamurthy, C. Wu, and M. Franklin, "On-the-fly sharing for streamed aggregation," in AMC SIGMOD, 2006.

13. K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu, "General incremental sliding-window aggregation," in VLDB, 2015.

14. L. G. Valiant, "A bridging model for parallel computation," Communications of the ACM, vol. 33, no. 8, pp. 103-111, 1990.

15. G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, 1. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in Proceedings of the 2010 ACM SIGMOD International Conference on Management ofdata. ACM, 2010, pp. 135-146.

16. C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," Proceedings of the Hadoop Summit. Santa Clara, 2011.

17. M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server.' in OSDI, vol. 1, no. 10.4, 2014, p. 3.

18. D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, ,'Naiad: a timely dataflow system," in ACM SOSP, 2013.

19. T. Akidau, IR. Bradshaw, C. Chambers, S. Chernyak, R. J. FernåndezMoctezuma, R. Lax, S. McVeety, D. Mills, Perry, Schmidt Ill., "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," VLDB, 2015.

20. 'Out-of-order processing: a new architecture for high-performance stream .systems," Proceedings ofthe VLDB Endowment, vol. T, no. T, pp. 274—288, 2008.

21. U. Srivastava and J. Widom, "Ilexible time management in data stream systems," in Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of diltnbnse systems. ACM, 2004.

22. P. D. Bailis, "Coordination avoidance in distributed databases," Ph.D. dissertation, UC Berkeley, 2015.

23. L. Lamport et al., "Paxos made simple," ACM Sigact News, vol. 32, no. 4, pp. 18-25, 2001.

24. D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm." in USENIX Annual Technical Conference, 2014.

25. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." HotCloud, 2010.

26. K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," ACM Transactions on Computer Systems (TOCS), vol. 3, no. 1, pp. 63-75, 1985.

27. P. Carbone, S. Ewen, G. Fora, S. Haridi, S. Richter, and K. Tzoumas, "State management in apache flink: Consistent stateful distributed stream processing," Proceedings ofthe VLDB Endowment, 2017.

28. P. Carbone, G. F6ra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," arXiv preprint arXiv:1506.08603, 2015.

29. P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl, "Cutty: Aggregate sharing for user-defined windows," in Proceedings of the 25th ACM International on Conference on Information and Knowledge Management. ACM, 2016.

30. Arasu and J. Widom, "Resource sharing in continuous sliding-window aggregates,' in VLDB, 2004.

31. "Blink: How Alibaba Uses Apache IGlink," http://data-artisans.com/blinkflink-alibaba-search/, 2016.

32. "Stream processing with IGlink at Netflix," http://sf.flink-forward.org/ kb_sessions/keynote-stream-processing-with-flink-at-netflix/ , 2017.

33. "StreamlNG models, how ING adds models at runtime to catch fraudsters," http://sf.flink-forward.org/kb_sessions/streaming-models-how-ingadds-models-at-runtime-to-catch-fraudsters/ , 2017.

34. 'Real-time monitoring with Flink, Kafka and HB," http://2016.flinkforward.org/kb_sessions/a-brief-history- of- time- with- apache-flink-realtime-monitoring-and-analysis-with-flink-kafka-hb/, 2017.

35. 'Windowing and State in Storm," https://community.hortonworks.com/ articles / 14171 /windowing- and-state-checkpointing- in- apache-storm. html, 2017.

36. P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," IEEE Data Engineering Bulletin, p. 28, 2015.

37. P. Carbone, G. E. Gévay, G. Hermann, A. Katsifodimos, J. Soto, V. Markl, and S. Haridi, "Large-scale data stream processing systems," in Handbook of Big Data Technologies. Springer, 2017.

38. B. Hindman, A. Konwinski, M. Zaharia, A. (Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in NSDI, 2011.

39. M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," ACM Computing Surveys (CSUR), vol. 46, no. 4, 46, 2014.

40. C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: easy, efficient data-parallel pipelines," in ACM Sigpl(l/l Notices. ACM, 2010.

41. M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé et al, "IBM Streams Processing Language: Analyzing big data in motion," IBM Journal of Research and Development, vol. 57, no. 3/4, pp. 7—1, 2013.

42. M. Hirzel, H. Andrade, B. Gedik, V. Kumar, G. Losa, M. Nasgaard, R. soule, and K. Wu, "SPL stream processing language specification," NewYork: IBMResearchDivisionTJ. WatsonResearchCenter, IBM ResearchReport: RC24897 (W0911 044), 2009.

43. D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos, "Semantics of data streams and operators," in International Conference on Database Theory. Springer, 2005.

44. E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," ACM Computing Surveys (CSUR), vol. 34, no. 3, pp. 375408, 2002.

45. Y.-H. Feng, N.-IG. Huang, and Y.-M. Wu, "lifficient and adaptive stateful replication for stream processing engines in high-availability cluster," IEEE Transactions on Parallel & Distributed Systems, no. IT, pp. 1788—1796.

46. M. Balazinska, I l. Balakrishnan, S. R. Madden, and M. Stonebraker, "Faulttolerance in the Borealis distributed stream processing system," ACM Tmnsactions on Systems (TODS), vol. 33, no. l , p. 3, 2008.

47. M. Balazinska, J.-II. Ilwang, and M. A. Shah, "Fault-tolerance and high availability in data stream management systems," in Encyclopedia of Database Systems. Springer, 2009.

48. G. Jacques-Silva, F. Zheng, D. Debrunner, K.-L. Wu, V. Dogaru, E. Johnson, M. Spicer, and A. E. Sariyüce, "Consistent regions: guaranteed tuple processing in ibm streams," Proceedings of the VLDB Endowment, vol. 9, no. 13, pp. 1341-1352, 2016.

49. R. Strom and S. Yemini, "Optimistic recovery in distributed systems," ACM Transactions on Computer Systems (TOCS), vol. 3, no. 3, pp. 204—226, 1985.

50. L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Communications of the ACM, vol. 21, no. 7, pp. 558—565, 1978.

51. R. Guerraoui and L. Rodrigues, Introduction to reliable distributed programming. Springer Science & Business Media, 2006.

52. S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream processing at scale," in ACM SIGMOD, 2015.

StrikePlagiarism.com King Danylo University

Дата звіту 12/21/2023
Дата редагування ---

метадані

Заголовок
Оптимізація моделей масштабування процесів обробки потоків даних

Автор
Луканюк В.В. Науковий керівник / Експерт

підрозділ
King Danylo University

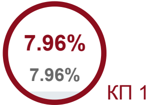
Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

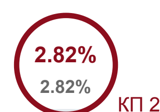
Заміна букв	ß	5
Інтервали	A→	0
Мікропробіли	·	0
Білі знаки	ß	0
Парафрази (SmartMarks)	a	68

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.



КП 1



КП 2

25 Довжина фрази для коефіцієнта подібності 2

17177 Кількість слів

130034 Кількість символів

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз

Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)	
1	https://www.sciopen.com/article/10.26599/TST.2021.9010029	45	0.26 %
2	https://dl.acm.org/doi/10.14778/2536222.2536229	44	0.26 %
3	https://dl.acm.org/doi/10.14778/2536222.2536229	42	0.24 %
4	http://repository.ukd.edu.ua:8080/bitstream/handle/123456789/379/%D0%91%D0%B0%D0%BB%D0%B0%D0%BD%D1%8E%D0%BA_%D0%86_%D0%86_%D0%9A%D0%92%D0%90%D0%9B%D0%86%D0%A4%D0%86%D0%9A%D0%90%D0%A6%D0%86%D0%98%CC%86%D0%9D%D0%90_%D0%A0%D0%9E%D0%91%D0%9E%D0%A2%D0%90.pdf?sequence=1	38	0.22 %