НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ННК "Інститут прикладного системного аналізу"
(повна назва інституту/факультету)

Кафедра Системного проектування
(повна назва кафедри)

«До захисту допущено»

Завідувач кафедри
_____ А.І.Петренко
(підпис) (ініціали, прізвище)

"___"_____2016 р.

Дипломна робота

першого (бакалаврського) **рівня вищої освіти**
(першого (бакалаврського), другого (магістерського))

зі спеціальності7.05010102, 8.05010102 Інформаційні технології проектування
7.05010103, 8.05010103 Системне проектування
(код та назва спеціальності)

на тему: Побудова наближених до реального часу розподілених систем обробки даних із застосуванням Apache Storm

Виконав: студент 4 курсу, групи ДА-22
(шифр групи)

Кривокінь Сергій Миколайович _____
(прізвище, ім'я, по батькові) (підпис)

Керівник асистент, к.т.н., Свірін П.В _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант економічний професор, д.е.н.СеменченкоН.В _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Нормоконтроль ст. викладач Британов О.А _____
(підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2016 року

Національний технічний університет України
«Київський політехнічний інститут»

Факультет (інститут) ННК "Інститут прикладного системного аналізу"
(повна назва)

Кафедра Системного проектування
(повна назва)

Рівень вищої освіти Перший(Бакалаврський)
(перший (бакалаврський), другий (магістерський) або спеціаліста)

Спеціальність 7.05010102, 8.05010102 Інформаційні технології проектування 7.05010103, 8.05010103 Системне проектування
(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ А.І.Петренко
(підпис) (ініціали, прізвище)
«___»_____2016 р.

ЗАВДАННЯ
на дипломний проект (роботу) студенту
Кривоконю Сергію Миколайовичу
(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) Побудова наближених до реального часу розподілених систем обробки даних із застосуванням Apache Storm

керівник проекту (роботи) Свірін Павло Володимирович, к.т.н., асистент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від 12 травня 2016 р. № 50-ст

2. Строк подання студентом проекту (роботи) 13.06.2016

3. Вихідні дані до проекту (роботи)
Аналіз фреймворку Apache Storm

Реалізований програмний продукт з застосуванням Storm

Результати запуску та тестування створеного продукту

4. Зміст розрахунково-пояснювальної записки (перелік завдань, які потрібно розробити)

1. Дослідити підхід Apache Storm до вирішення задачі розподілених обчислень в реальному часі

2. Дослідити засоби Apache Storm для розробки розподілених систем реального часу.
3. Проаналізувати можливі застосування Storm та порівняти Storm з аналогами
4. Розробити прикладний додаток з використанням Storm.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслеників, плакатів тощо)

1. Принцип роботи паралелізму в Storm.
2. Архітектура спроектованої системи.
3. Топологія розробленої компоненти статистики.

6. Консультанти розділів проекту (роботи)*

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	професор, д.е.н.СеменченкоН.В		

7. Дата видачі завдання 01.02.2016

Календарний план

№ з/п	Назва етапів виконання дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	Отримання завдання	01.02.2016	
2	Збір інформації	15.02.2016	
3	Дослідження існуючих систем потокової обробки даних	28.02.2016	
4	Дослідження технології Apache Storm	10.03.2016	
5	Аналіз засобів Apache Storm та дослідження аналогів	15.03.2016	
6	Розробка додатку	25.03.2016	
7	Тестування додатку	25.04.2016	
8	Оформлення дипломної роботи	30.04.2016	
9	Оформлення дипломної роботи	31.05.2016	
10	Отримання допуску до захисту та подача роботи в ДЕК	13.06.2016	

Студент _____ Кривокінь С.М.
(підпис) (ініціали, прізвище)

Керівник проекту (роботи) _____ Свірін П.В.
(підпис) (ініціали, прізвище)

* Консультантом не може бути зазначено керівника дипломного проекту (роботи).

АНОТАЦІЯ

бакалаврської дипломної роботи Кривоконя Сергія Миколайовича
на тему: «Побудова наближених до реального часу розподілених систем обробки даних із застосуванням Apache Storm»

Дипломна робота присвячена використання програмного продукту Apache Storm для побудови систем обробки потоків інформації великого об’єму у режимі реального часу. У роботі наведено аналіз засобів, що надає фреймворк, архітектури типової системи на основі фреймворку та внутрішніх механізмів та архітектуру самого фреймворку Storm.

Актуальність роботи пов’язана зі зростанням навантаження на системи обробки даних. Коли необхідно працювати з даними, які надходять у великих об’ємах, для вирішення задачі використовують розподілені та масштабовані системи. Одним із засобів для побудови таких систем і є Apache Storm.

В якості прикладу застосування фреймворку було розроблено прототип системи для розподіленого ведення статистики. Така система може бути застосована під час виборів для відображення поточного стану та статистичних даних.

Шляхи подальшого розвитку предмета дослідження – встановлення системи на реальному розподіленому кластері та інтеграція із зовнішніми системами.

Загальний обсяг роботи: 68 сторінок, 23 рисунка, 8 таблиць, 1 додаток на 6 стор., 16 посилань.

Ключові слова: РОЗПОДІЛЕНІ СИСТЕМИ, СИСТЕМИ РЕАЛЬНОГО ЧАСУ, ПОТОКОВА ОБРОБКА ДАНИХ, APACHE STORM.

АННОТАЦИЯ

бакалаврской дипломной работы Кривоконя Сергея Николаевича

на тему: «Построение приближенных к реальному времени распределенных систем обработки данных с применением Apache Storm»

Дипломная работа посвящена использованию программного продукта Apache Storm для построения систем обработки потоков информации большого объема в режиме реального времени. В работе приведен анализ средств, предоставляемых фреймворком, архитектуры типичной системы на основе фреймворка, внутренних механизмов и архитектуру самого фреймворка Storm.

Актуальность работы связана с ростом нагрузки на системы обработки данных. Когда необходимо работать с данными, поступающими в больших объемах, для решения задачи используют распределенные и масштабируемые системы. Одним из средств для построения таких систем и является Apache Storm.

В качестве примера применения фреймворка был разработан прототип системы для распределенного ведения статистики. Такая система может быть применена во время выборов для отображения текущего состояния и статистических данных.

Пути дальнейшего развития предмета исследования – установка системы на реальном распределенном кластере и интеграция с внешними системами.

Общий объем работы 68 страниц, 23 рисунка, 8 таблиц, 1 приложение на 6 стр., 16 ссылок.

Ключевые слова: РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ, СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ, ПОТОКОВАЯ ОБРАБОТКА ДАННЫХ, APACHE STORM.

ABSTRACT

of the bachelor’s thesis of Serhii Krivokin M.

"Development of real-time distributed systems for big data processing using Apache Storm”

Thesis is devoted to the use of Apache Storm software for building systems of stream processing of large amounts of information in real time. The paper provides an analysis of development tools provided by the framework, the typical architecture of the system on the basis of the framework, the internal mechanisms of the framework and architecture of the Storm.

The relevance of the work is related to the increase of load on the data processing systems. When it is needed to process large amounts of streaming data, the problem solution is to build distributed real-time system. One of the means to build such systems is Apache Storm.

As an example of application of the framework has been developed a prototype system for distributed statistical purposes. Such a system can be used during elections to show the current status and statistics.

The total amount of work 68 pages, 23 figures, 8 tables, 1 appendix for 6 p., 16 references.

Ways of further development of the study of the subject - installation of the system on a real distributed cluster and integration with external systems.

Keywords: DISTRIBUTED SYSTEMS, REAL-TIME, STREAMING, APACHE STORM.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ І УМОВНИХ ПОЗНАЧЕНЬ

JVM – Java Virtual Machine

БД – база даних

ОС – операційна система

DRPC – distributed remote procedure call

Storm – фремворк Apache Storm

worker – процес, що виконує частину топології

task – логічна одиниця топології

executor – потік, що виконує task всередині worker процесу

ВСТУП

Метою роботи є дослідження засобів програмного продукту Apache Storm для побудови розподілених систем обробки даних в режимі реального часу. У ході роботи будуть розглянуті основні компоненти Storm, принципи його роботи та внутрішню архітектуру. Буде наведено порівняння за аналогами, та виділено можливі області застосування фреймворку.

Розподілена система – це система, яка функціонує на компонованій станції з певної кількості обчислювальних машин, які в поєднанні утворюють кластер. Таким чином, логіка системи одночасно виконується на різних машинах та на різних процесорах. Така система дозволяє досягти швидкодії завдяки паралельній обробці даних. Основна задача, яку виконує розподілена система – це поєднання фізичного ресурсу багатьох обчислювальних машин для проведення обчислень над складною задачею, або великим об’ємом даних.

Для того, щоб мати можливість оброблювати дані паралельно, треба провести сегментацію задачі – розділення на підзадачі, що можуть обчислюватися паралельно. Проте не кожна задача може бути сегментована, тому треба проектувати систему так, щоб сегментувати тільки підзадачі, які можна розподілити на певну кількість незалежних підзадач.

Система реального часу – це система, яка повинна реагувати на події у зовнішньому по відношенню до системи середовищі або впливати на середовище в межах необхідних тимчасових обмежень. Іншими словами, обробка інформації системою повинна проводитися за певний кінцевий період часу, щоб підтримувати постійну та своєчасну взаємодію з середовищем. Звісно, що масштаб часу контролюючої системи та контрольованого нею середовища повинен збігатися.

Під реальним часом розуміється кількісна характеристика, яка може бути виміряна реальними фізичними годинами, на відміну від логічного часу, який визначає лише якісну характеристику, відображену відносним порядком

проходження подій. Кажуть, що система працює в режимі реального часу, якщо для опису роботи цієї системи потрібні кількісні тимчасові характеристики[1].

Одним із засобів для побудови таких систем є програмний продукт Apache Storm. Його актуальність підтверджується тим, що продукт застосовують компанії світового рівня: YaHoo, Twitter, Spotify та інші[2]. Продукт має відкритий програмний код, тому щоб пересвідчитись, що над продуктом ведеться постійна робота, достатньо зайти на GitHub репозиторій продукту.

Головною перевагою використання фреймворку при розробці розподіленої системи є те, що великий об’єм роботи по забезпеченню синхронізації потоків, забезпеченню відмовостійкості, перебалансуванню фізичних ресурсів та інших необхідних властивостей розподіленої системи уже виконаний розробниками фреймворку. При використанні такого фреймворку використовується менше робочих ресурсів, тож розробка стає економічно ефективнішою. Крім того, значно зменшуються випадки допущення розробниками помилок при написанні складних механізмів, які уже реалізовані у фреймворку. Це дозволяє зосередитись на програмуванні бізнес-логіки, а не логіки роботи системи.

Можливими галузями застосування продукту є такі галузі, які потребують обробки великих об’ємів даних у режимі реального часу. Це можуть бути інтернет-сервіси магазини, для яких система буде визначати рекламні пропозиції для відвідувачів з певними параметрами, або портали новин, яким треба запропоновувати найбільш цікаві новини певному класу відвідувачів.

У роботі спроектована система для сервісу онлайн-держави, що дозволяє вести статистику під час виборів. Подібна система може бути використана у будь-якому сервісі, де потрібно вести статистику в режимі реального часу.

1. ПІДХІД APACHE STORM ДО ВИРІШЕННЯ ЗАДАЧІ РОЗПОДІЛЕНИХ ОБЧИСЛЕНЬ В РЕАЛЬНОМУ ЧАСІ

1.1 Актуальність задачі

На сьогоднішній день переважна частина усієї інформації оброблюється обчислювальними машинами. Існують області застосування, де обробка інформації має виконуватись у режимі реального часу, для того щоб надати корисну інформацію з короткою затримкою. Серед таких областей є транспортні сервіси, які мають розраховувати маршрут, відстежуючи стан доріг в реальному часі, сенсорні системи, що надсилають дані з датчиків у реальному часі, наприклад з інформацією про клімат або про пульс носія датчика, системи таргетингу, які на основі статистики відвідування групами користувачів певних сторінок, визначають які сторінки(товари, послуги, інший контент) запропонувати користувачам з подібними ознаками.

Сучасні компанії-лідери інформаційних технологій приділяють значну увагу системам реально часу. Так, наприклад компанія Google, ще з 2009р. впровадила пошук в реальному часі, який дозволяє знаходити інформацію, яка надходить у потоках постів в соціальних мережах, таких як Facebook, Twitter та ін.. Навігаційні сервіси Yandex та Google maps оброблюють великі масиви даних в реальному часі для відображення актуального стану мапи доріг та відстеження проблемних ділянок з заторами.

На даний час ведеться розробка декількох продуктів що призначені для побудови розподілених систем реально часу. Серед найбільш відомих є такі продукти як Spark Streaming і Apache Storm. В 2015 р. Twitter презентував новий продукт – Heron, який має замінити Storm, і демонструє значний прогрес у швидкодії та зменшенню затримки у порівнянні зі Storm[3].

1.2 Моделі систем розподіленого обчислення

Розподілена система являє собою модель, в якій компоненти, розташовані на об’єднаних в мережу комп'ютерах, які координувати свої дії шляхом передачі повідомлень. Компоненти взаємодіють один з одним для досягнення спільної мети. Серед основних характеристик розподілених систем виділяють: паралелізм компонентів, відсутність глобального таймеру, незалежність стабільності роботи компонентів, тобто при виході з ладу однієї компоненти, інші не мають виходити з ладу.

Виконання програм на розподілених системах потребує відповідної архітектури програмного продукту, де компоненти розділені між собою так, що можуть функціонувати паралельно і незалежно. Для взаємодії між компонентами використовується механізм передачі повідомлень, для реалізації якого існує декілька альтернатив, серед яких протокол HTTP, RPC- подібні конектори та черги повідомлень.

Немає абсолютно чіткого визначення розподіленої системи, і часто змішуються такі поняття як «розподілена» та «паралельна» система (Рис 1.1). Для визначення розподіленої системи часто використовують такі її властивості:

- Система складається з певної кількості автономних обчислювальних машин, кожна з яких має свою власну локальну пам’ять
- Обчислювальні машини комунікують між собою за допомогою обміну повідомленнями
- Кожна обчислювальна машина в системі має індивідуальну відмовостійкість
- Кожна машина має обмежений доступ до функціоналу системи, і відповідає за обробку лише певної частини вхідної інформації

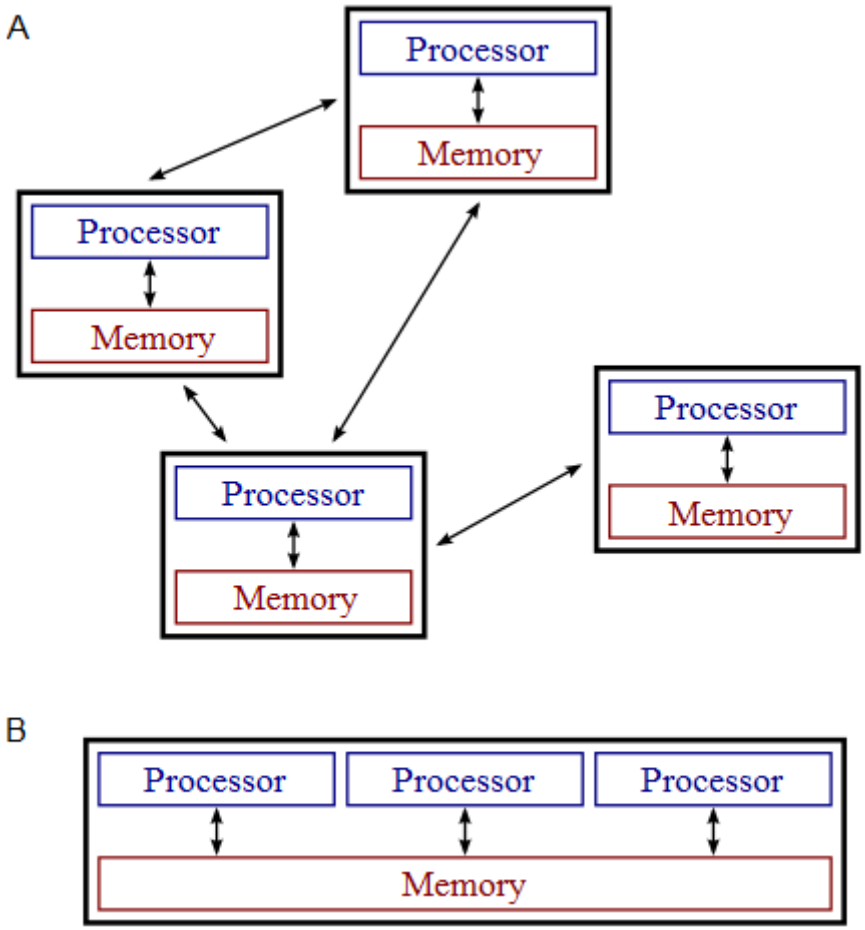

Рис 1.1. Розподілена (A) та паралельна (B) системи

Типова задача розподіленого програмного продукту, це така задача, що потребує великої кількості обчислень над великими об’ємами даних. В залежності від поставленої задачі, можуть проводитися пакетні обчислення або обчислення у реальному часі.

Для проведення пакетних обчислень часто використовується технологія MapReduce зі стеку Hadoop. Цей підхід дозволяє ефективно оброблювати великі масиви уже існуючих даних, для видачі результату одразу для всього масиву.

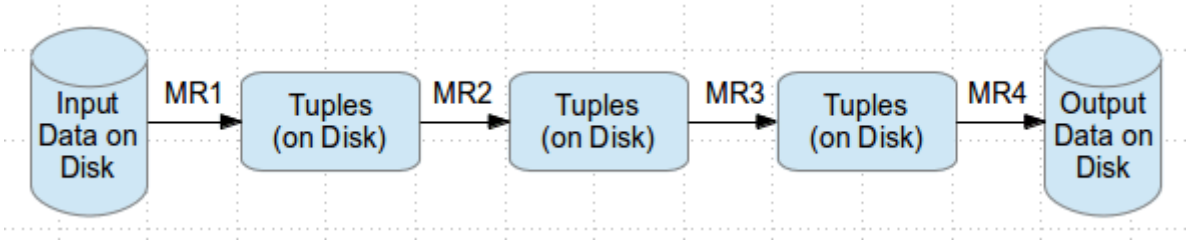

Рис 1.2. Схема роботи технології MapReduce

На кожному етапі в MapReduce відбувається розподілення(Map) або об’єднання даних(Reduce). Для забезпечення відмово стійкості дані записується на диск, що потребує багато додаткового часу на запис/зчитування інформації з диску (Рис 1.2).

Технологія MapReduce надає можливість масштабування системи, але оскільки потребує постійного зчитування/запису на жорсткий диск, має недостатньо коротку затримку для використання у системах реального часу.

Обчислення у реальному часі проводяться не над великим цілісним масивом даних, а над фрагментами, що надходять поступово у вигляді потоку. Для побудови систем моделі реального часу розроблені відповідні фреймворки, одним з яких є Apache Storm.

1.3 Засоби Apache Storm для розробки розподілених систем реального часу

1.3.1 Компоненти Storm кластера

Storm кластер за своєю структурою подібний до кластера Hadoop MapReduce. Аналогічно до того як в Hadoop виконуються так звані “MapReduce jobs”, в Storm виконуються «топології». "Job’и" і "топології" самі по собі дуже відрізняються – одна ключова різниць в тому, що MapReduce job в кінцевому підсумку закінчується, в той час як топологія оброблює повідомлення безперервно (або поки ви не буде явно завершена). Інша ключова різниця полягає в механізмі забезпечення відмовостійкості: в той час, як MapReduce job використовує запис розподіленої інформації на диск, елементи

топології не зберігають інформацію, натомість Spout відправляє дані ще раз, якщо на певному етапі відбувся збій. Ще однією відмінністю є те, що MapReduce job не працює в реальному часі і приймає на вхід одразу великий об’єм даних, в той час як Storm топологія оброблює кожну надіслану в повідомленні одиницю даних одразу після отримання.

У Storm кластері є 2 види вузлів: master-вузол (головний) і worker-вузли. Головний вузол запускає daemon (процес), званий " Nimbus", який відповідає за розподіл коду між машинами кластера, призначення завдань машинам, а також моніторинг збоїв у роботі кластера.

Кожен worker-вузол запускає daemon (процес), що зветься "Supervisor". Supervisor приймає від Nimbus’а роботу, що відведена для його машини і запускає/зупиняє worker процеси в міру необхідності на основі тієї задачі, яку присвоїв машині Nimbus. Кожен worker процес виконує певну частину топології; виконання топології відбувається на певній кількості робочих процесів, розподілених серед машин у кластері. Кількість робочих процесів і потоків конфігурується перед запуском топології.

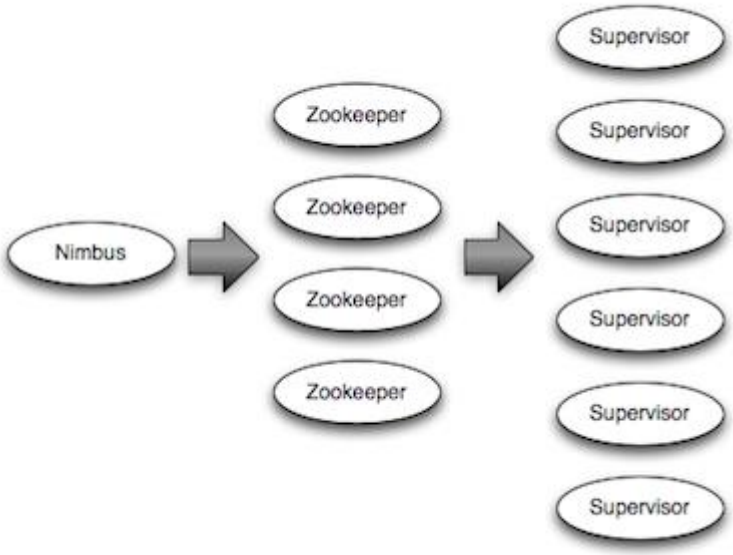

Рис 1.3. Структура Storm кластера

За координацію між Nimbus’ом та Supervisor’ами відповідає кластер Zookeeper, який також відповідає за збереження поточного стану вузлів та відновлює роботу Nimbus та Supervisor у разі незапланованого припинення, що робить систему відмовостійкою[4].

1.3.2 Storm топології

Для виконання обчислень в реальному часі, створюються так звані “топології”. Топологія представляє собою напрямлений граф за яким відбуваються обчислення. Кожна вершина графу містить певну частину логіки програми, а ребра графу вказують шлях по якому дані передаються між вузлами кластера.

Основною абстракцією передачі даних в Storm є потоки, що складаються з послідовності кортежів даних (Tuple). Storm надає можливість трансформувати потоки даних по мірі їх проходження через топологію.

Основними примітивами, що виконують трансформації потоків даних в Storm є Spout і Bolt. Для виконання логіки програми реалізуються інтерфейси цих примітивів. Spout – це джерело потоку даних, він може зчитувати кортежі даних з брокера повідомлень, або інтегруватись з API зовнішньої системи, після чого продукувати кортежі даних в топологію. Bolt приймає на вхід деяку кількість потоків, виконує задану логіку, та може також продукувати вихідні кортежі даних. Bolt може містити в собі таку логіку як, фільтрація кортежів даних, агрегування, об’єднання потоків, взаємодія з базою даних, тощо. Складна логіка перетворень потоку даних потребує декількох етапів обробки, які можна розподілити на відповідну кількість Bolt вузлів.

Граф, утворений з елементів Spout і Bolt представляє собою топологію і запускається на кластері для виконання логіки програми. Коли деякий вузол продукує потік даних, усі вузли, що підписані на цей потік отримують з нього дані.

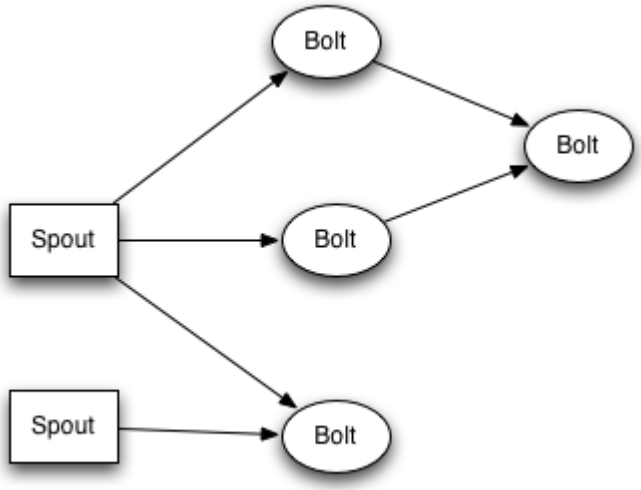

Рис 1.4. Storm топологія

Зв’язки між вузлами показують напрямки, в яких будуть розповсюджуватися кортежі даних. Наприклад, якщо існує зв’язок направлений від Spout A до Bolt B, зв’язок від Spout A до Bolt C, та зв’язок від Bolt B до Bolt C, тоді кожен вихідний кортеж зі Spout A буде надісланий до Bolt B та Bolt C. Також вихідні дані з Bolt B надійдуть до Bolt C.

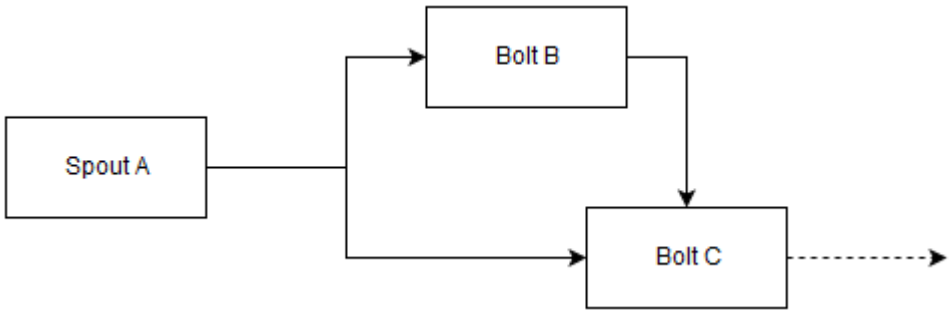

Рис 1.5. Приклад розповсюдження кортежів даних

Таким чином можна будувати ланцюжки з елементів Bolt, в яких попередній елемент виконує деякі обчислення та підготовлює дані для наступних елементів, передаючи їх через виклик методу emit().

1.3.3 Storm Trident

Trident – це високорівнева абстракція для проведень обчислень в реальному часі на основі Storm. Технологія дозволяє отримати баланс між високою пропускною здатністю (мільйони повідомлень в секунду), потоковою обробкою зі збереженням стану та розподіленим виконанням запитів з низькою затримкою. Trident має такі операції як поєднання, агрегація, групування, функції та фільтрація, для оперування над потоками. Також Trident додає абстракції для інкрементальної обробки зі збереженням стану поверх БД. Trident має семантику exactly-once, тому доцільного його використовувати коли необхідна така семантика.

Ключовою проблемою у функціонуванні системи зі збереженням стану з семантикою exactly-once є збереження стану рівно 1 раз. Якщо елемент дає збій після збереження стану, та до відправки підтвердження (ack), то запит на збереження стану буде відправлений ще раз. Для вирішення цієї проблеми в Trident вводиться додаткова збиткова інформація у вигляді ідентифікатору пакету, тоді виконується такий механізм:

1. Кожен пакет має свій ідентифікатор. Якщо пакет буде відправлений на обробку повторно, то він матиме такий же ідентифікатор.
2. Обновлення стану впорядковане між пакетами. Таким чином, обновлення стану для пакету 3 не буде виконано поки не буде успішно виконано обновлення стану для пакету 2.

Такий механізм дозволяє досягти семантики exactly-once, але призводить до збереження збиткової інформації у вигляді ідентифікаторів пакетів, а також сповільнює роботу системи у випадку збоїв, оскільки обновлення стану для пакетів відбувається послідовно.

Trident топології компілюються в Storm топології таким чином, що передача Tuple відбувається тільки при необхідності їх перегрупування. Приклад компіляції Trident топології в Storm топологію з елементів Spout та Bolt зображений на рисунку 1.6.

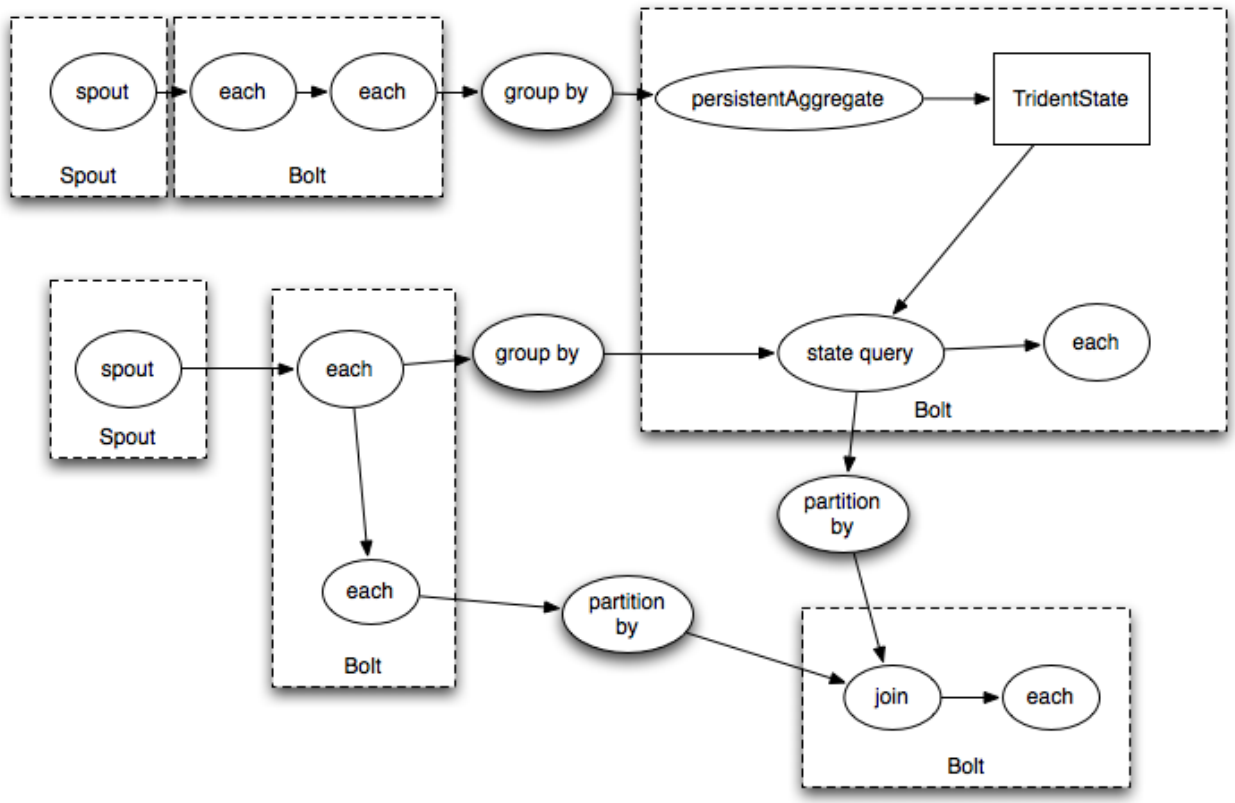

Рис 1.6 Приклад компіляції Trident топології

Для того, щоб обрати між Trident та Core Storm потрібно відповісти на такі питання:

- Чи потрібна семантика exactly-once для вирішення задачі
- Чи є головним критерієм найбільш коротка затримка на обробку даних
- Чи є головним критерієм пропускна здатність системи

Якщо головний критерій це коротка затримка на обробку даних, то краще використовувати Core Storm, оскільки він не створює додаткового навантаження та затримок для забезпечення семантики exactly-once. Якщо більш важливими є пропускна здатність системи або семантика exactly-once, то доцільно використовувати Trident, завдяки пакетній обробці та забезпеченню відповідної семантики.

1.3.4 Distributed RPC

Ідея на якій базується розподілений RPC (DRPC), це виконувати паралельне обчислення інтенсивних функцій в реальному часі використовуючи Storm. Топологія приймає на вхід потік аргументів функцій та породжує вихідний потік результатів обчислення для кожного виклику функції з відповідними аргументами.

DRPC існує не тільки в Storm, це лише патерн, що реалізований в Storm на основі його примітивів, таких як потоки, Spout, Bolt елементи і топології. Storm DRPC можна було б видавати окремою бібліотекою, але цей інструмент дуже часто використовується в розробці, тому його одразу додають в стандартний пакет інструментів Storm.

Координація DRPC відбувається через DRPC-сервер, імплементація якого поставляється разом зі Storm. Сервер проводить координацію в такі етапи:

1. Приймає вхідний RPC запит
2. Відсилає прийнятий запит на вхід Storm топології
3. Приймає результат обчислень від топології
4. Відправляє результат обчислень клієнту, що відправив запит

Для клієнта не є видимим той факт, що RPC функціонує розподілено. На стороні клієнта для виклику функцій використовується DRPCClient. Наприклад, так виглядає запит на до функції “reach” з аргументом “twitter.com”:

```
DRPCClient client = new DRPCClient("drpc-host", 3772);
String result = client.execute("reach", "http://twitter.com");
```

На стороні серверу, аргумент функції буде передано в топологію, яка може функціонувати паралельно на розподіленому кластері. Таким чином можна виконувати розподілені RPC операції.

Робочий процес DRPC можна представити так:

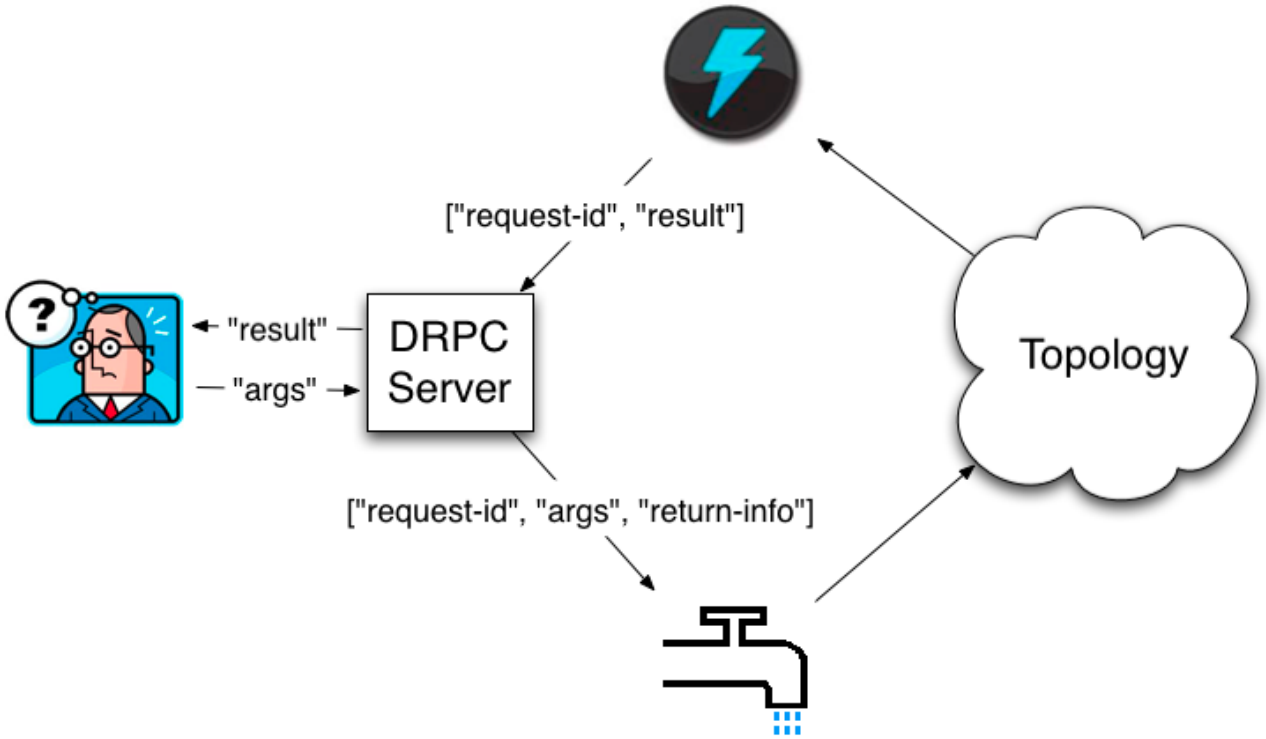

Рис 1.7 Схема роботи DRPC

Клієнт надсилає серверу ім’я функції для виконання та аргументи для цієї функції. Топологія, що реалізує цю функцію використовує DRPCSpout для отримання потоку аргументів від DRPC-сервера.

Кожен виклик функції позначається унікальним ідентифікатором на стороні DRPC-сервера. Топологія виконує обчислення, в кінці якого обов’язково спеціальний Bolt ReturnResults з’єднується з сервером і повертає йому результат обчислення функції. Після отримання результату обчислення функції від топології, DRPC-сервер порівнює ідентифікатор отриманого результату з ідентифікаторами викликаних функцій, щоб повернути результат потрібному клієнту, який був ініціатором виклику функції і очікує на результат.

1.4 Паралелізм в Apache Storm

Паралелізм в Storm базується на таких основних рівнях:

- worker процеси
- executors (потоки)
- task (логічна одиниця)

Взаємодію цих сутностей можна проілюструвати так:

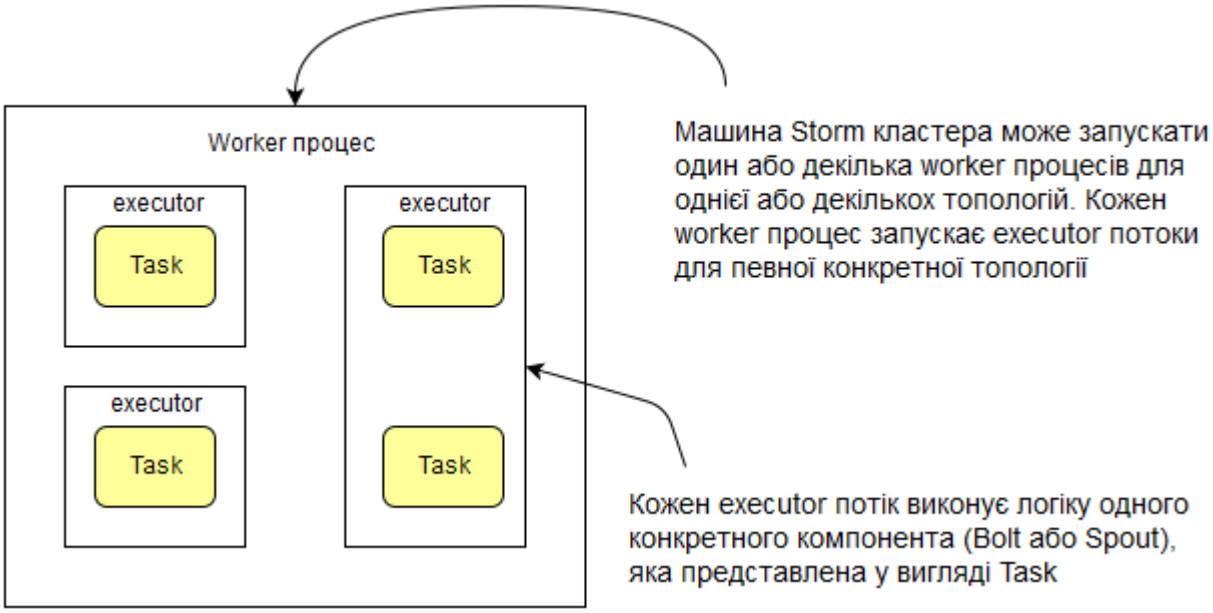

Рис 1.8. Взаємодія worker процесів, executor потоків та логічних одиниць Task

Worker процес виконує певну частину топології та запускає окрему JVM(Java Virtual Machine). Цей процес належить конкретній топології і може запускати executor процеси для одного або декількох компонентів (spout або bolt) топології . Діюча топологія складається з певної кількості таких процесів, що виконуються на машинах Storm кластера.

Executor це потік, створений worker процесом і функціонує в його JVM. Executor може виконувати одну або декілька логічних одиниць Task одного конкретного компонента топології (spout або bolt). Оскільки executor представляє собою однин потік, то task'и виконуються на ньому послідовно.

Task виконує безпосередню логіку програми і запускається всередині свого executor потоку. Кількість Task є сталою протягом усього функціонування топології. Для зміни кількості Task потрібен перезапуск топології з новими параметрами[5].

Перед запуском топології слід враховувати, що:

- Кількість executor потоків може бути змінена без перезапуску топології
- Кількість Task статична і може бути змінена тільки з перезапуском

Для налаштування паралелізму в Storm доступні такі методи:

Таблиця 1.1 – Налаштування паралелізму в Storm

назва налаштування	опція конфігуратора	метод для зміни значення
кількість worker процесів	Config# TOPOLOGY_WORKERS	Config#setNumWorkers
кількість executor потоків	-	TopologyBuilder#setSpout() TopologyBuilder#setBolt()
кількість логічних одиниць task	Config# TOPOLOGY_TASKS	ComponentConfigurationDeclarer# setNumTasks()

В звичайному випадку логічно мати конфігурацію, в якій на одну логічну одиницю Task є один executor потік. Така конфігурація також встановлюється за вмовчуванням, якщо не було явно вказано кількість Task.

Конфігурація відмінна від стандартної може знадобитися в тому випадку, якщо планується розширення певного елементу топології. Якщо задати початкову конфігурацію executors = 2 tasks = 4, тоді можна буде розширити елемент до конфігурації executors = 4 tasks = 4 без перезапуску топології. Виконати перебалансування можна за допомогою команди storm rebalance

наприклад:

Зміна конфігурації топології mytopology,

Встановимо кількість executors = 4 для "boltA"

$ storm rebalance mytopology -e boltA = 4

Таким чином буде досягнуто пере балансування системи і нові executor потоки зможуть зайняти додані в кластер обчислювальні ресурси.

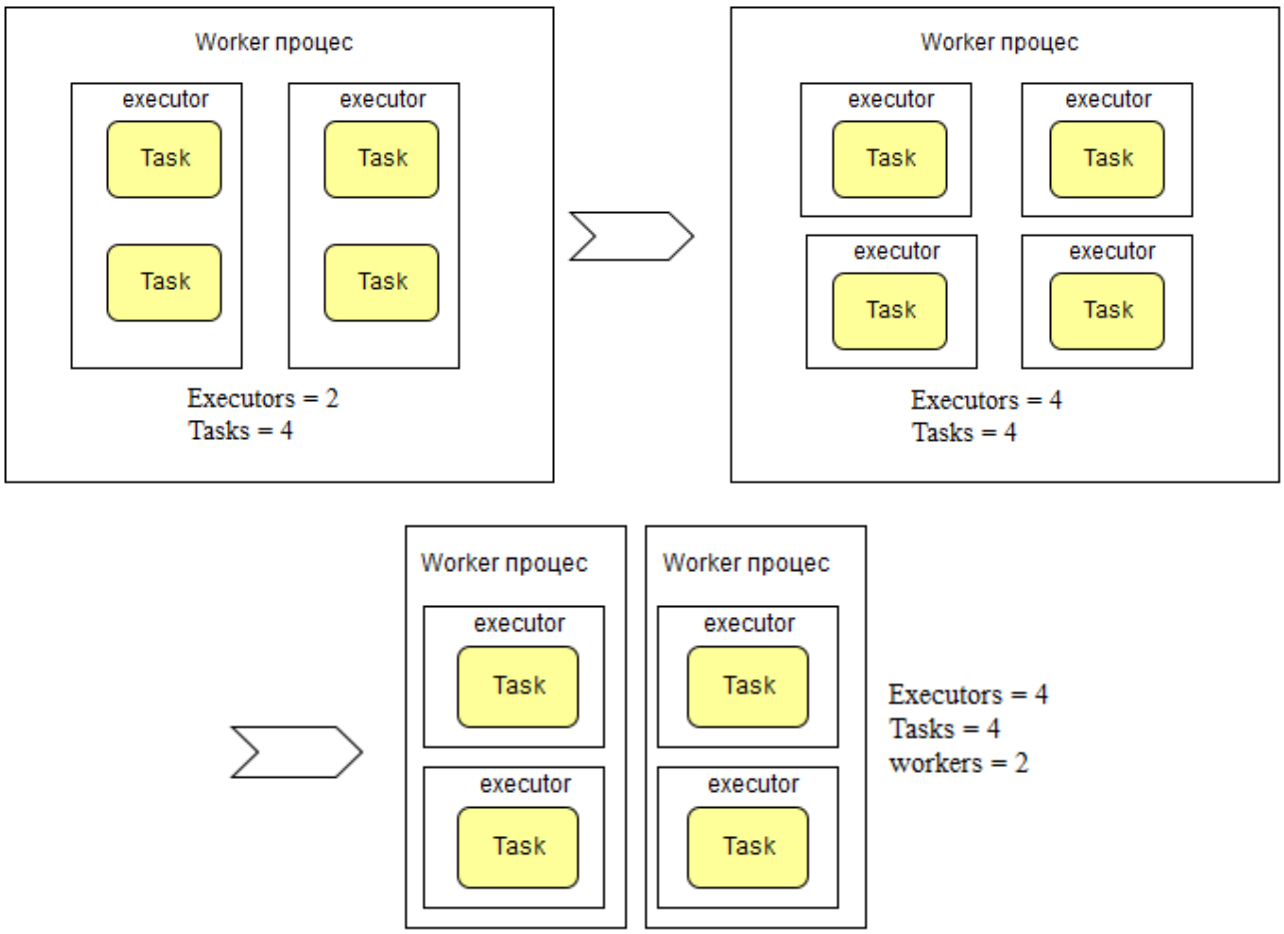

Рис. 1.9 Перебалансування топології

1.5 Аналіз продуктивності Storm

Дослідження продуктивності в цьому розділі базується на дослідженні опублікованих матеріалів компаніями, які використовують Storm у своїй роботі.

Щодо швидкодії Storm, розробники з Twitter рапортували результат в 1.64 мільйони кортежів в секунду на кожній машині в кластері[10] для версії Storm 0.8.0, з часу виходу якої було проведено багато роботи над продуктивністю. Ще немає опублікованих досліджень швидкодії останньої версії – 2.0.0, але вона має бути більшою ніж у попередніх версіях.

Розробники з keen.io опублікували матеріал в якому описані проблемні ділянки в продуктивності роботи Storm та шляхи вирішення цих проблем[11].

Однією з проблем являється розподілення запитів різної складності. Якщо запит, який потребує багато фізичних ресурсів, потрапляє у worker, то виконання інших, більш легких запитів теж сповільнюється, оскільки їм доводиться ділити фізичний ресурс машини. Таким чином не можна гарантувати сталу затримку на виконання запиту, оскільки цей час виконання залежить від інших запитів що оброблюються в той же час.

Для вирішення проблеми варіативності затримки на виконання одних і тих же запитів пропонується забезпечити виконання таких умов:

- Максимально зменшити варіативність між типами запитів які виконують worker (JVM) на певній машині
- В рамках однієї машини, не допустити єдиному вхідному запиту зайняти всі фізичні ресурси

Для зменшення варіативності між типами запитів для однієї машини рекомендується ізолювати усі worker процеси відповідної топології, тобто виконувати їх на виділених ізольованих машинах.

В поточній версії Storm 2.0.0 існує можливість ізольованого розподілу worker процесів між машинами за допомогою IsolationScheduler[12].

При стандартному розподілі worker процесів, різні процеси, що виконують одну й ту ж саму топологію розміщуються на кількох машинах (Рис 1.8). При такому розміщенні виникає дві проблеми:

1. Виникає боротьба за ресурс машини між топологіями. Більш важкі топології будуть сповільнювати роботу більш легких.
2. Для комунікації worker процесів між машинами виникає додатковий трафік, а також витрачається додатковий час на серіалізацію та передачу інформації між машинами.

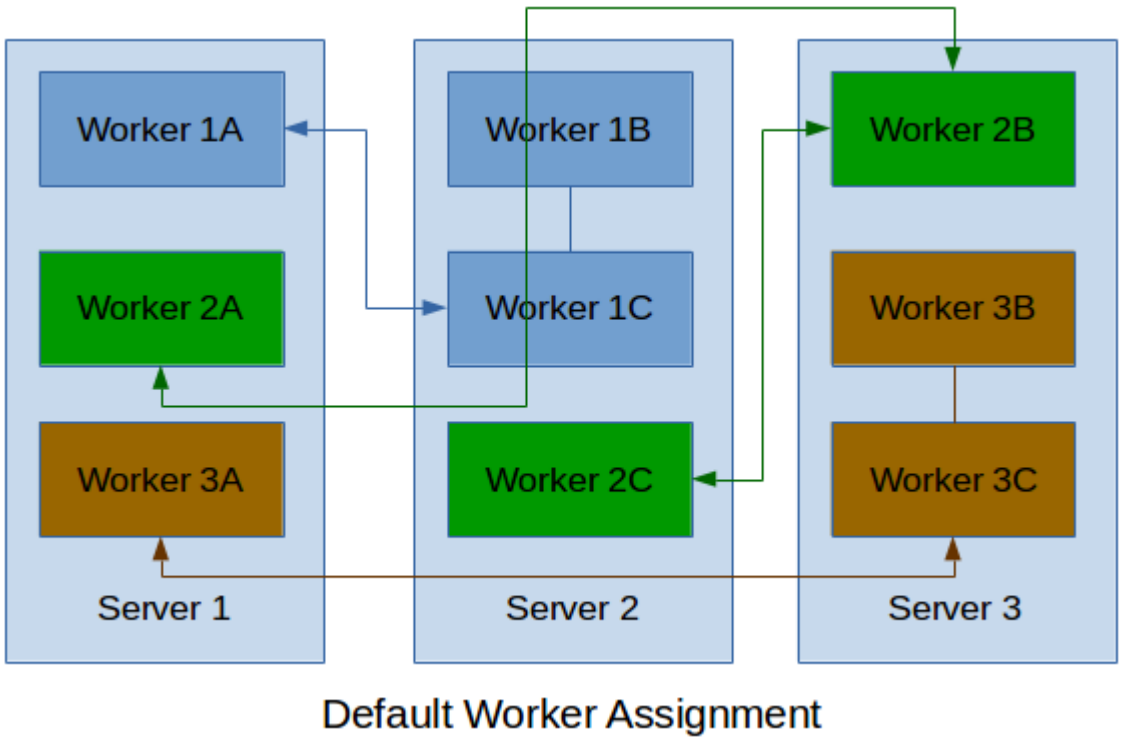

Рис 1.10 Стандартний розподіл worker процесів між машинами

У випадку ізольованого розподілу worker процесів, усі процеси, що належать певній топології будуть блоковані на задану кількість машин в кластері (Рис 1.11). Таким чином буде досягнуто однотипності задач відведених для певної машини, а також зникне боротьба за ресурси машини між різними топологіями.

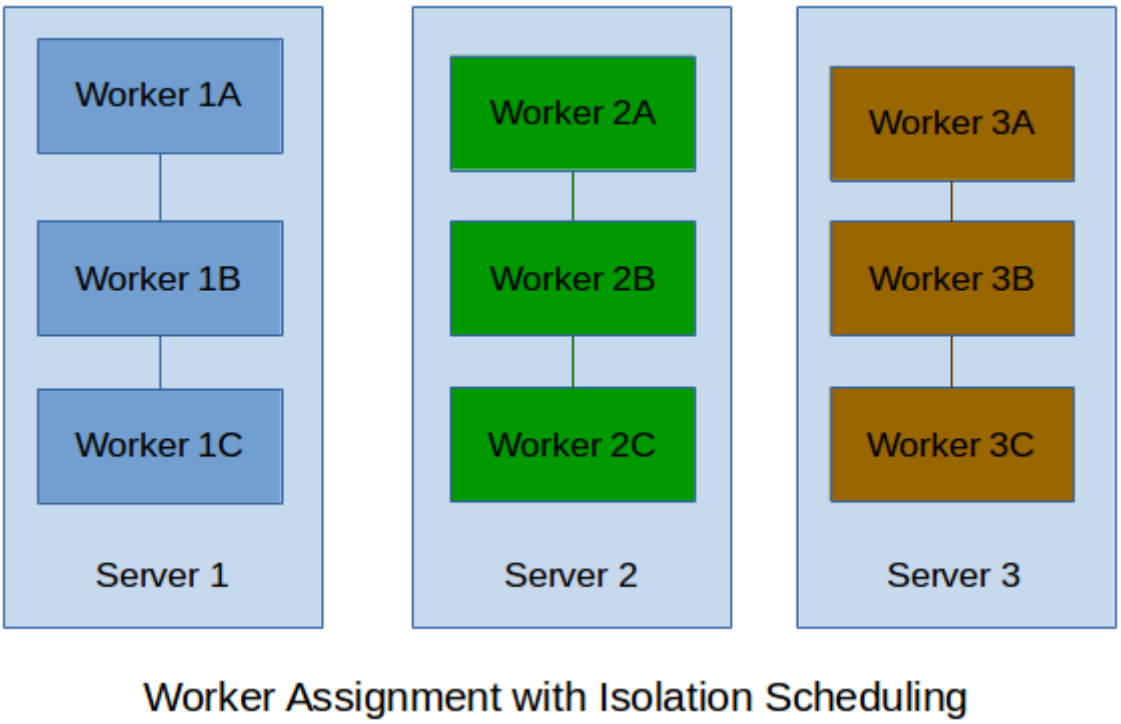

Рис 1.11 Ізольований розподіл worker процесів між машинами

При розподілі топологій між машинами в кластері, не завжди варто використовувати ізольований розподіл, оскільки він може призвести до наступних негативних наслідків:

- Якщо запити до топології на ізольованій машині не надходять, то ресурс машини простоює. У випадку стандартного розподілу, Nimbus виконає перерозподіл процесів, щоб використовувати ресурс всіх машин наявних у кластері.
- У випадку збоїв на ізольованій машині, топологія буде перенесена на іншу машину, яку потрібно буде ізолювати, а всі процеси розподілити між іншими машинами. Такий процес займає багато часу

Тож ізольований розподіл слід використовувати для важких топологій, запити до яких надходять постійно. Більш легкі топології доцільніше розподіляти стандартно, для ефективнішого використання ресурсів кластера.

1.6 Аналіз можливих застосувань та порівняння з аналогами

Розробники Storm вказують на головній сторінці сайту такі можливі області застосування[6]:

- аналіз потоку даних в реальному часі
- побудова систем розподілених обчислень в реальному часі
- побудова machine-learning систем реального часу

Засоби, що дозволяють оброблювати дані в реальному часі та будувати розподілені системи були розглянуті в попередніх пунктах. Але засоби для роботи з machine-learning у попередніх пунктах не згадувались, тому що machine-learning напрям не розвинутий у Storm і інформація щодо розробки machine-learning застосувань відсутня в документації фреймворку.

1.6.1 Apache Spark

Spark – це фреймворк призначений для розподіленої обробки даних з використанням спеціалізованих примітивів, що дозволяють проводити рекурентну обробку в даних в оперативній пам’яті. Такий підхід дозволяє збільшити швидкість пакетної обробки даних у 100 разів порівняно з MapReduce, в якому відбувається запис даних на диск. Така швидкість, а також можливість багатократного доступу до занесених в пам’ять даних робить Spark привабливим для розробників machine-learning алгоритмів[13].

Spark має розвинуту бібліотеку для роботи з алгоритмами machine-learning – Spark MLib, яка широко застосовується у світі[7]. В той же час, серед списку компаній, які використовують Storm жодна не вказала таку ціль використання як machine-learning реального часу[2].

Таким чином можна зробити висновок, що для розробки систем з застосуванням machine-learning краще використовувати Spark, аніж Storm.

Для потокової обробки даних розробники Spark створили окрему гілку розвитку продукту – Spark Streaming. Фреймворк відрізняється від Storm тим,

що проводить пакетну обробку, але з дуже коротким інтервалом прийому даних, таким чином наближуючи обробку до реального часу, але на практиці такий підхід дає більше навантаження на систему і збільшує затримки на обробку даних. Такий підхід в Spark Streaming аналогічний до підходу Storm Trident, який також проводить пакетну обробку з малими затримками. Серед дискусій на тему який фреймворк кращий існують різні точки зору, але можна виділити той факт, що Storm більше застосовується великими компаніями для обчислень в реальному часі, а Spark більше використовується компаніями та науковцями, що працюють з machine-learning.

Крім того, оскільки Spark Streaming побудований таким чином, що дані зберігаються в оперативній пам’яті, можливі сценарії з втратою певної частини даних у разі збоїв машин у кластері[14].

Один з відомих дослідників Big-Data, Rassul Fazelat, надає таку таблицю з порівнянням характеристик Storm Core, Triden та Spark Streaming[15]

Storm Core vs. Storm Trident vs. Spark Streaming

	Core Storm	Storm Trident	Spark Streaming
Community	> 100 contributors	> 100 contributors	> 280 contributors
Language Options	Java, Clojure, Scala, Python, Ruby, ...	Java, Clojure, Scala	Java, Scala Python (coming)
Processing Models	Event-Streaming	Micro-Batching	Micro-Batching Batch (Spark Core)
Processing DSL	No	Yes	Yes
Stateful Ops	No	Yes	Yes
Distributed RPC	Yes	Yes	No
Delivery Guarantees	At most once / At least once	Exactly Once	Exactly Once
Latency	sub-second	seconds	seconds
Platform	Storm Cluster, YARN	Storm Cluster, YARN	YARN, Mesos Standalone, DataStax EE

Рис 1.12 Порівняльна таблиця Storm Core, Trident та Spark Streaming

1.6.2 Java Akka

Більш базовою альтернативою Storm є фреймворк Java Akka. Фреймворк призначений для спрощення побудови паралельних розподілених систем, що функціонують на JVM. Базові принципи паралельності у Java Akka подібні до мови програмування Erlang та базуються на моделі акторів.

Відповідно до означення моделі акторів, кожна обчислювальна одиниця, що представляє собою актора, у відповідь на повідомлення може виконувати такі паралельні операції[17]:

- надсилати певну кількість повідомлень іншим акторам
- створювати певну кількість нових акторів
- встановлювати певну поведінку, яка буде використана при отримані актором наступного повідомлення

Актори в Akka обмінюються повідомленнями завдяки тому, що мають посилання на інших акторів системи, або можуть отримати таке посилання знаючи ім’я необхідного актора.

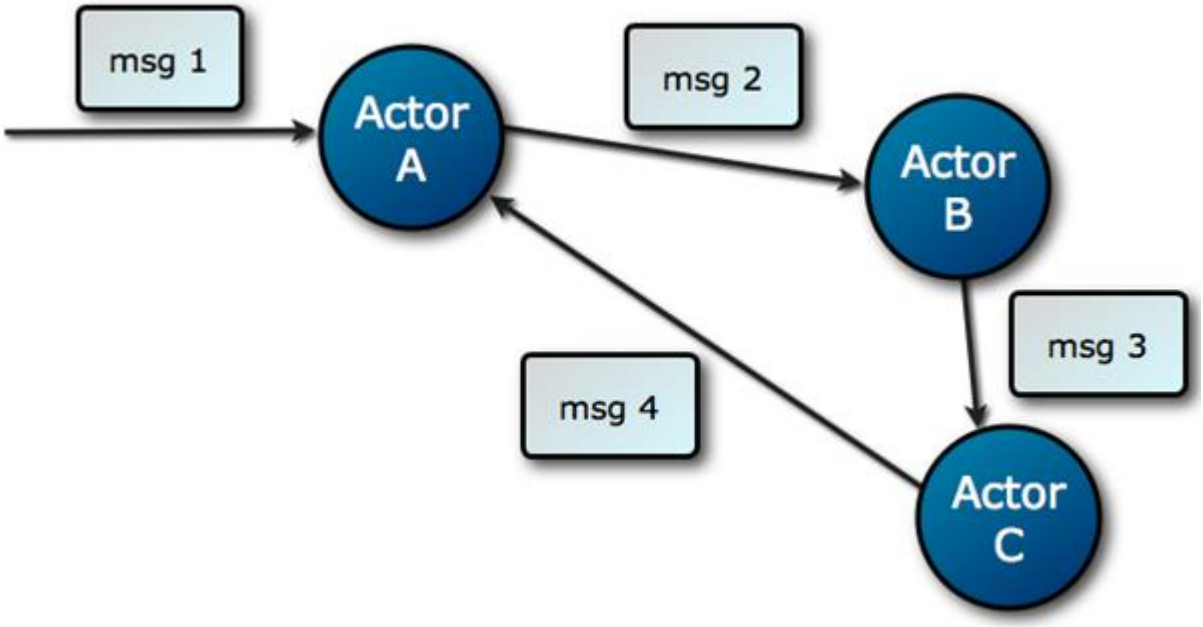

Рис 1.13 Комунікація між акторами в Akka

Повідомлення можуть бути відправлені безпосередньо з одного актору в інший. На відміну від Akka, Storm має чітку структуру, що представляє собою напрямлений граф, в напрямі якого розповсюджуються повідомлення. На основі Akka можна побудувати Storm-подібну систему, але крім того можна побудувати систему з архітектурою відмінною від Storm. Мінусом у порівнянні зі Storm є кількість затрачених зусиль на побудову Storm-подібної архітектури. Якщо архітектура Storm задовольняє потребам задачі, то використання Storm є більш доцільним ніж використання Akka, оскільки в Storm уже зроблена велика кількість роботи по забезпеченню стабільності роботи системи, яку в Akka доведеться реалізовувати з нуля[9].

Таким чином Akka краще підходить для побудови більш гнучких та специфічних систем, в яких всю логіку взаємодії між акторами можна задати самому розробнику. Якщо структура Storm топології не задовольняє вирішенню задачі, то доцільним буде побудувати власну структуру з використанням Akka.

1.7 Приклади застосування Storm у світі

У цьому розділі буде розглянуто декілька прикладів застосування Storm відомими у світі компаніями. Список компаній, які використовують Storm приведений на powered-by сторінці сайту фреймворка.

Yahoo:

- потокова обробка подій від користувачів
- обробка потоків контенту для користувачів
- обробка логів

В Yahoo Storm використовується як частина платформи для обробки великих об’ємів даних, що відповідає за обробку даних з короткими затримками. Також в Yahoo використовується Hadoop для пакетної обробки[2].

Twitter:

- аналіз даних в реальному часі
- пошук
- оптимізація прибутку

Storm інтегрується з іншими компонентами інфраструктури Twitter, серед яких бази даних Cassandra, Memchached, Mesos, системи моніторингу та оповіщення[2].

Infochimps:

- інтеграція з зовнішніми джерелами даних
- імпорт даних із зовнішніх систем, та організація транспортування даних
- виконання ETL (extract, transform, load) операцій

Storm використовується як компонент хмарної системи, що забезпечує відмовостійку, лінійно масштабовану систему для потокового збору, транспортування та обробки даних[2].

Загалом, використання Storm у різних компаніях прив’язане до різних конкретних задач, але ці задачі можна класифікувати як такі, що потребують потокової обробки даних. Критерії щодо затримки на обробку відрізняються, але завжди є наближеними до реального часу. Застосування фреймворку на практиці відповідає його декларованим можливостям.

1.8 Висновки

В даному розділі було розглянуто та досліджено підхід Apache Storm до вирішення задачі розподілених обчислень в реальному часі. В ході дослідження було описано механізми та засоби Storm для побудови систем розподілених обчислень в реальному часі.

Було розглянуто принцип побудови Storm топологій на основі базових елементів Spout та Bolt, описано механізм взаємодії між ними.

Було розглянуто паралелізм в Storm, описано основні рівні на яких він базується – workers, executors, tasks, пояснено та зображено на рисунку принцип їх взаємодії.

Досліджено декларовані можливості Storm та використання його на практиці відомими світовими компаніями. Виявлено, що в цілому використання Storm збігається з декларованими задачами та можливостями і використовується у системах потокової обробки даних.

Було розглянуто альтернативні продукти, які мають переваги над Storm в деяких напрямах. Так, наприклад Spark краще підходить для задач з machine-learning, завдяки своїй бібліотеці MLib, а Java Akka дозволяє будувати більш гнучкі системи.

2. РОЗРОБКА ПРИКЛАДНОГО ПРОГРАМНОГО ПРОДУКТУ

2.1 Постановка завдання

Метою розробки прикладної системи є наглядна демонстрація можливостей фреймворку Apache Storm. Створена система має застосовувати основні інструменти фреймворку, висвітлювати основні принципи та підходи до розробки систем з використанням Storm.

Для конкретизації завдання необхідно провести такі етапи дослідження:

1. Розглянути існуючі розподілені системи обробки даних в реальному часі. Де і для чого вони використовуються.
2. Розглянути існуючі системи, що побудовані з використанням Apache Storm. Якими компаніями використовується фреймворк так у яких цілях.
3. Проаналізувати розглянуті приклади, та виявити типові задачі, які розв’язують за допомогою Storm у світі.
4. Виявити області використання в Україні, де може знадобитись така система.
5. На основі попередніх пунктів сформувати прикладну задачу, для вирішення якої знадобиться система, яку можна побудувати з використанням Apache Storm.

У результаті конкретизації завдання має бути знайдено прикладну область, де можна застосувати систему для виконання обчислень в реальному часі, яку доцільно буде реалізувати з використанням фреймворку Storm.

2.2 Конкретизація завдання

В результаті дослідження областей застосування фреймворку Apache Storm, було виявлено, що здебільшого фреймворк використовується для побудови розподілених систем реального часу, які не містять “Batch” робіт, і дозволяють отримати результат одразу після надходження вхідних даних. Storm використовується для систем з RPC семантикою “at least once”, але також дозволяє отримати семантику “exactly once” завдяки топології Trident, але цей підхід веде до програшу у продуктивності функціонування системи.

З огляду на можливість практичного застосування в Україні, було обрано таку область застосування як сервіс системи онлайн-держави. Оскільки в Україні ще відсутня можливість проводити голосування онлайн, а також явка на вибори є досить низькою, така система може бути актуальною і вирішувати такі задачі:

1. Ведення статистики під час голосування в режимі реального часу
    - відображення статистики по кількості голосів за кандидата
    - відображення статистики по розподілу голосів відданих певному кандидату жителями певного населеного пункту
    - відображення статистики розподілу голосів за віковими категоріями виборців
2. Збільшення явки виборців завдяки підвищенню зацікавленості. Якщо виборець зайде на портал, щоб перегляну статистику, з великою вірогідністю він і сам захоче залиши голос
3. Надання відкритого доступ до статистичних даних для всіх виборців. Це дозволить знаходити такі аномалії, як наприклад голосування цілого невеликого населеного пункту за одного кандидата, або різке зростання кількості голосів за кандидата у короткий проміжок часу.

Розроблена система має вести статистику під час голосування. Основний сервер надсилає в систему пару <Виборець. Кандидат>, яка означає, що виборець проголосував за кандидата. Система має обробити надіслану інформацію та обновити статистичні дані у режимі реального часу.

2.3 Загальна архітектура системи

Загальна архітектура системи включає в себе усі компоненти, що дозволять організувати процес електронних виборів. У роботі буде розроблено компоненту, яка зображена на рисунку як «Сервер статистики». Інші компоненти не будуть розроблені у цій дипломній роботі.

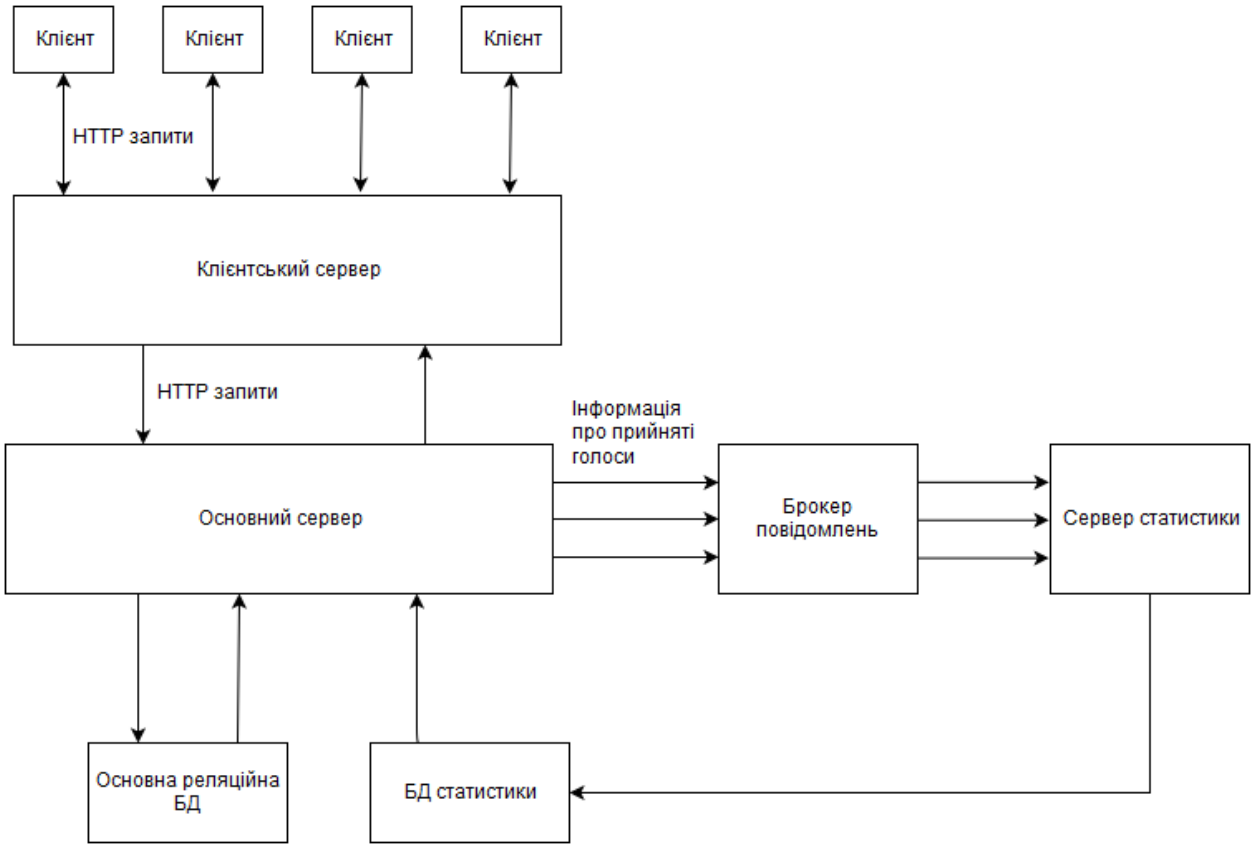

Рис 2.1. Загальна архітектура системи

Пояснення до рисунку:

Клієнт – пристрій з якого виборець здійснює голосування через веб-портал.

Кієнтський сервер – сервер, що відповідає за взаємодію з клієнтом, відображенням контенту веб-порталу та виконання логіки веб-порталу.

Основний сервер – сервер що виконує основну логіку системи, взаємодіє з базою даних та зберігає надіслану з клієнтського серверу інформацію про прийняті голоси.

Брокер повідомлень – система, яка відповідає за комунікацію головного серверу з сервером статистики, передає інформацію про прийняті голоси з основного серверу до серверу статистики

Сервер статистики – виконує обробку надісланої інформації про прийняті голоси та обновлює статистичні дані. Сервер отримує повідомлення від брокера повідомлень та зберігає результат обробки.

Така архітектура забезпечить слабку зв’язаність компонент, тобто при наявності збоїв в одній з компонент, інші зможуть продовжити свою роботу. Також така архітектура дозволить вести розробку компонент паралельно декільком командам, кожна з якої буде займатись однією з компонент, і лише обговорювати інтерфейси взаємодії з командами, які займаються іншими компонентами.

Для розробки компоненту статистики необхідно спроектувати внутрішню архітектуру компоненти, яка буде представляти собою Storm топологію, налаштувати середовище розробки та написати програмний код компоненти. Після запуску написаної програми потрібно перевірити правильність її виконання в багато поточному режимі, а також дослідити швидкодію.

2.4 Архітектура компоненту статистики

Компонент статистики представляє собою топологію фреймворку Storm, яка складається з одного елементу Spout, який продукує пари <Виборець, Кандидат> та трьох елементів Bolt, які обновлюють статистичні дані.

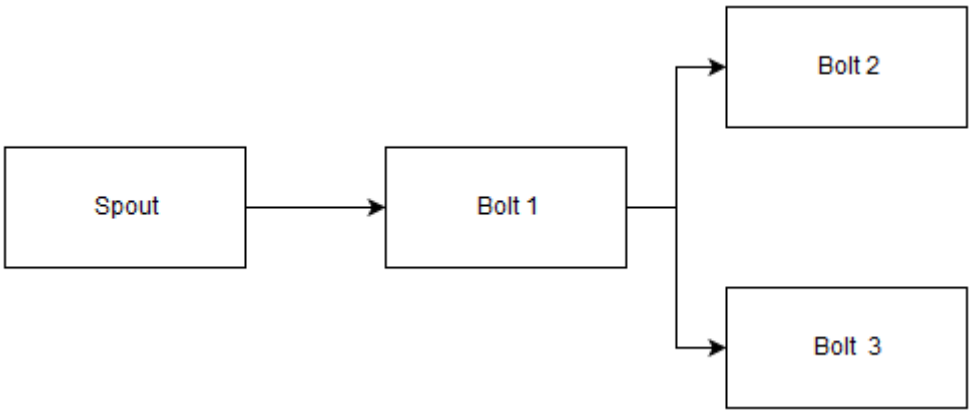

Рис 2.2. Топологія для збору статистики

Розглянемо детальніше функціональне призначення кожного компоненту:

Spout – елемент, який є кореневим джерелом потоку інформації у зображеній топології. Цей елемент інтегрується з брокером повідомлень, таким чином повідомлення з брокеру продукуються в топології через Spout у вигляді набору структурних одиниць Tuple, які по суті є масивами Java об’єктів.

Bolt 1 – елемент, що веде статистику по загальній кількості голосів, відданих кожному кандидату, а також генерує Tuple для наступних Bolt елементів. Після надходження Tuple зі Spout в елемент Bolt 1, виконується логіка по інкрементації кількості голосів для кандидата, вказаного в прийнятому Tuple, далі дістається детальна інформація про виборця, вказаного в прийнятому Tuple, після чого генеруються потоки даних для двох наступних елементів Bolt 1 та Bolt 2.

Bolt 2 – елемент, що веде статистику по кількості голосів за кандидата по певному населеному пункту, а також вираховує кількість голосів по групам населених пунктів, що об’єднані у області.

Bolt 3 – елемент, що веде статистику по кількості голосів відданих кандидату виборцями певної вікової категорії, розраховує середній вік серед усіх виборців, що голосували за кандидата.

2.5 Налаштування середовища розробки

Для того, щоб почати безпосередньо написання коду топології, потрібно спершу налаштувати всі необхідні компоненти середовища.

У роботі розробка велась на типовому наборі технологій:

- JDK(Java Development Kit) – набір інструментів та бібліотек для написання та компіляції Java коду
- Maven – програма, що дозволяє виконувати збірку проекту, а також завантажувати необхідні бібліотеки з репозиторіїв
- Intellij IDEA – інтегроване середовище розробки, яке має широкий набір інструментів для написання коду та інтегрує в себе 2 попередні компоненти

2.5.1 Налаштування JDK та Maven

Інструкції щодо встановлення JDK та Maven легко знайти в інтернеті. Перевірити успішність встановлення можна виконавши з консолі такі команди

```
java –version – має виводити встановлену версію Java
mvn --version – має виводити встановлену версію Maven
```

Якщо компоненти були успішно встановлені, то на консоль буде виведено відповідну інформацію про їх версії (Рис 2.3).

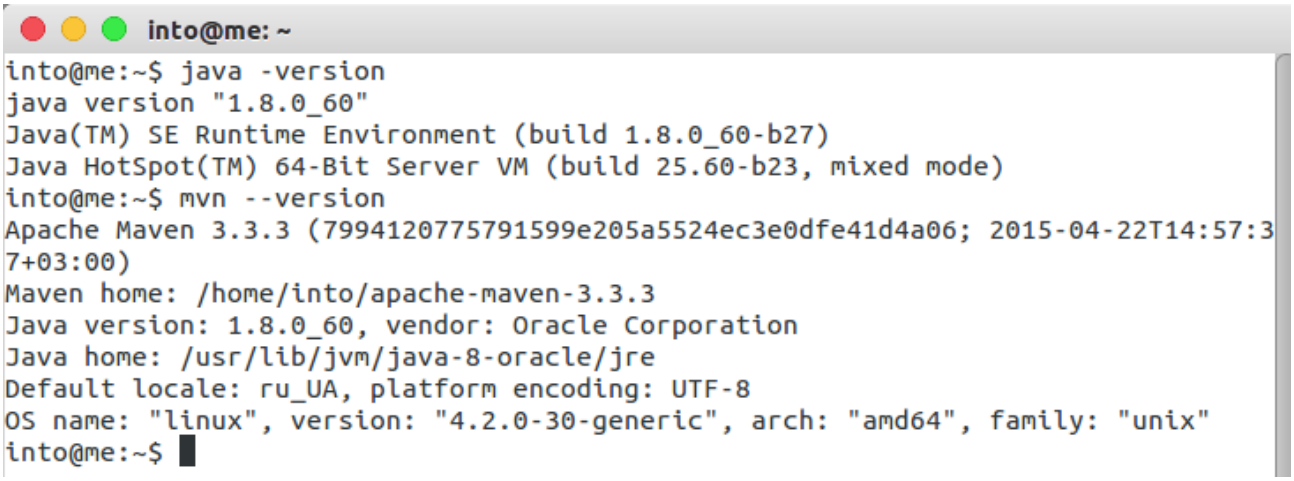

Рис 2.3 Перевірка встановлення JDK та Maven

2.5.2 Збірка фреймворку Storm

Для того, щоб отримати доступ до інструментів фреймворку Storm останньої версії, потрібно завантажити його код з git репозиторію https://github.com/apache/storm, після чого зібрати проект за допомогою Maven. Це потрібно для того, щоб бібліотеки зібрані з коду фреймворку потрапили до локального Maven репозиторію, звідки до них можна буде отримати доступ з IDE під час написання коду.

Для запуску збірки фреймворку Storm потрібно виконати відповідну Maven команду з кореню директорії з кодом Storm (Рис 2.4)

```
mvn clean install -DskipTests=true
```

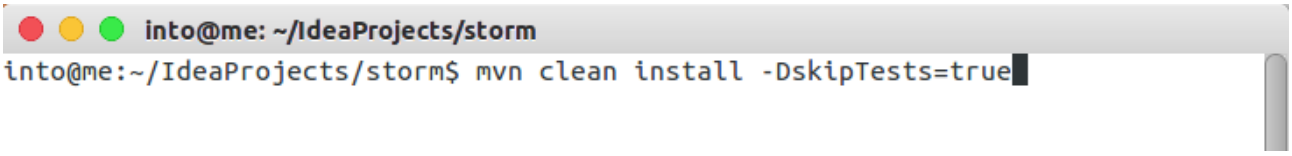

Рис 2.4 Запуск збірки фреймворку Storm

Після успішної збірки на консоль має бути виведено повідомлення Build SUCCES (Рис 2.5).

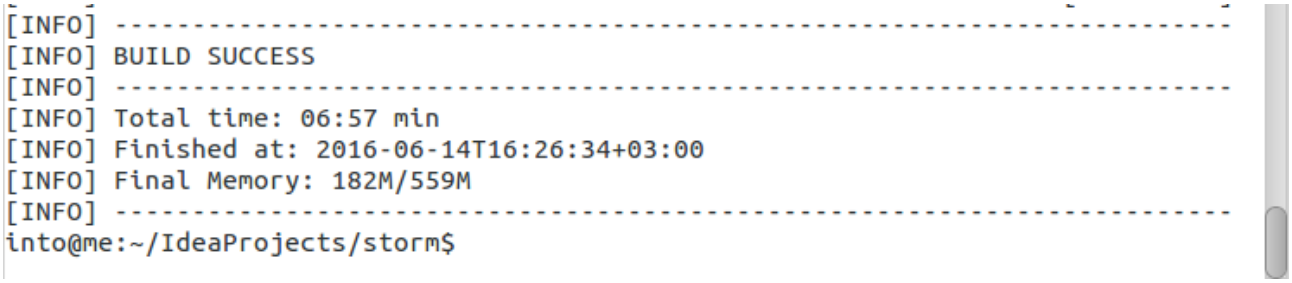

Рис 2.5 Успішна збірка фреймворку Storm

2.6 Написання програмного коду

Після налаштування середовища бібліотеки фреймворку є доступними з IDE, де буде проходити написання коду.

2.6.1 Написання коду для джерела даних

Для генерації та збереження даних створено спеціальний Java клас, що використовує спеціальні контейнери для безпечного зчитування та запису з декількох потоків:

- ConcurrentHashMap – контейнер типу Map, що зберігає пару <ключ, значення>
- BlockingQueue – черга, з якою можуть працювати декілька потоків. Будемо використовувати її для збереження черги повідомлень і зчитування їх Spout елементами топології.
- AtomicInteger – дозволяє виконувати атомарну операцію інкрементації

Для представлення пари <Виборець, Кандидат> створимо відповідний клас VoterCandidateEntry (додаток А).

При ініціалізації класу сховища даних генерується задана кількість вхідних пар <Виборець, Кандидат>.

Згенеровані дані заносяться в чергу

```
List<VoterCandidateEntry> voterCandidateEntries = generateVoterCandidateEntries();
очередь сюда
```

2.6.2 Написання коду топології

Для зчитування даних з джерела та передачі їх на оброку в топологію потрібно реалізувати відповідну логіку в Spout елементі. Spout елемент буде звертатись до черги, яка містить записи пар <Виборець, Кандидат>, зчитувати дані та передавати в потоках до елементів Bolt.

```
entry = VotingData.entriesQueue.take();
Long voterId = entry.voterId;
Long candidateId = entry.candidateId;
_collector.emit(new Values(voterId, candidateId));
```

Обробка даних здійснюється в Bolt елементах. Для отримання даних Bolt має бути підписаним на потік даних від Spout. Для цього задається групування потоків по назві створеного елементу Spout:

```
builder.setSpout("votingSpout", new VotingSpout(), 1);
builder.setBolt("candidateVotesCountBolt", new
CandidateVotesCountBolt(),1).shuffleGrouping("votingSpout");
```

Виконання логіки елементу Bolt відбувається в методі execute, відповідно в елементі, що відповідає за оновлення кількості голосів за кожного кандидата, задана логіка, що збільшує кількість голосів за кандидата на 1 кожного разу, коли надходить запис з ідентифікатором відповідного кандидата і зберігає цей стан в базу

```
Long candidateId = input.getLong(1);
Long voterId = input.getLong(0);
AtomicInteger count = VotingData.candidateVotes.get(candidateId);
if (count == null) {
   count = new AtomicInteger(0);
   AtomicInteger old = VotingData.candidateVotes.putIfAbsent(candidateId, count);
   if (old != null) {
      count = old;
   }
}
count.incrementAndGet();
```

2.7 Запуск та тестування топології

Для запуску топології потрібно виконати такі етапи:

- Задати елементи топології методами об’єкту класу TopologyBuilder: setSpout та setBolt
- Задати параметри конфігуратора методами об’єкту класу Config
- Ініціалізувати кластер LocalCluster, якщо запуск відбуватиметься на локальній машині
- Запустити топологію викликом методу submitTopology() від об’єкту LocalCluster для локальної машини або від класу StormSubmitter для кластера

Для перевірки правильності виконання логіки розробленої топології виконаємо її запуск і звіримо отриманий результат з очікуваним. Очікуваний результат розрахуємо на основі згенерованих даних, а результат роботи зчитаємо з бази після того як топологія опрацює усі згенеровані записи.

Згенеруємо 10^7 пар <Виборець, Кандидат>, де голоси будуть рівномірно розподілені між 3 кандидатами.

Запуск виконаємо з такими налаштуваннями паралелізму: 2 потоки для елементів Spout і 4 потоки для елементів Bolt.

Очікуваний та реальний результат виведемо на консоль:

```
Candidate 1 expected to get 3333146 votes
Candidate 1 actually got 3333146 votes
Candidate 2 expected to get 3333113 votes
Candidate 2 actually got 3333113 votes
Candidate 3 expected to get 3333741 votes
Candidate 3 actually got 3333741 votes
```

Очікуваний результат збігається з реальним, тому можна зробити висновок, що логіка програми виконується правильно і не відбувається збоїв при паралельній роботі кількох елементів топології

Для тестування швидкодії в залежності від налаштувань паралелізму, проведемо заміри часу між двома подіями: першим зчитування з черги елементом Spout та моментом коли в черзі більше не залишиться даних.

Таблиця 2.1 Час обробки при різних налаштуваннях паралелізму

потоків Spout	потоків Bolt	час обробки 10^7 записів
1	1	120 с.
2	4	78 с.
4	8	90 с.

Тестування відбувалось на машині з наступною конфігурацією:

Процесор: Intel Core i5-3220M 2.6GHz

ОЗУ: 4 Gb DDR-3

ОС: Ubuntu 14.04

Storm: 2.0.0-SNAPSHOT

З отриманих результатів можна зробити висновок, що паралелізм в розробленій топології працює правильно і дає приріст у швидкодії. Оскільки тестування проводилося на локальній машині, то оптимальної швидкодії було досягнуто на досить невеликій кількості потоків, але результати тестування на кластері відрізнялись би від отриманих на локальній машині, тому що обробка на кластері відбувалась би одразу на кількох процесорах, а тому була би доцільною більша кількість потоків.

2.8 Висновки

В даному розділі на основі аналізу відомостей з розділу 1 було спроектовано прототип системи сервісу онлайн-держави, в якій одну з компонентів, що веде статистику голосування в реальному часі, доцільно реалізувати з використанням фреймворку Storm

Було описано архітектуру загальної системи, в якій функціонує компонент статистики, а також архітектуру самого компоненту. В основі архітектури компоненту статистики лежить розподіл обчислень над вхідними даними між різними машинами, що дозволить вести обробку даних паралельно і будувати розподілену систему.

Було проведено запуск та тестування розробленої топології. Очікуваний результат роботи топології збігається з реальним результатом, що свідчить про правильність логіки роботи топології та відсутності помилок при паралельному функціонуванні декількох потоків Spout та Bolt. Можна зробити висновок, що така топологія не буде давати збоїв і при роботі на кластері.

Тестування часу обробки в залежності від параметрів паралелізму показало, що паралельна обробка скорочує час обчислень, порівняно з обробкою в одному потоці. Оптимальна кількість потоків виявилась досить низькою тому, що тестування проводилося на локальній машині.

3. ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

У даному розділі проводиться оцінка основних характеристик програмного продукту, що представляє собою розподілену систему, здатну оброблювати великі об’єми інформації в режимі реального часу, а також здатну працювати в умовах великих навантажень.

Програмний продукт призначений для використання на кластерних серверах, на які надходять великі об’єми даних.

Нижче наведено аналіз різних варіантів реалізації системи з метою вибору оптимальної, з огляду при цьому як на економічні фактори, так і на характеристики продукту, що впливають на продуктивність роботи і на його сумісність з апаратним забезпеченням. Для цього було використано апарат функціонально-вартісного аналізу.

Функціонально-вартісний аналіз (ФВА) – це технологія, яка дозволяє оцінити реальну вартість продукту або послуги незалежно від організаційної структури компанії. Як прямі, так і побічні витрати розподіляються по продуктам та послугам у залежності від потрібних на кожному етапі виробництва обсягів ресурсів. Виконані на цих етапах дії у контексті метода ФВА називаються функціями.

Мета ФВА полягає у забезпеченні правильного розподілу ресурсів, виділених на виробництво продукції або надання послуг, на прямі та непрямі витрати. У даному випадку – аналізу функцій програмного продукту й виявлення усіх витрат на реалізацію цих функцій.

Фактично цей метод працює за таким алгоритмом:

визначається послідовність функцій, необхідних для виробництва продукту. Спочатку – всі можливі, потім вони розподіляються по двом групам:

ті, що впливають на вартість продукту і ті, що не впливають. На цьому ж етапі оптимізується сама послідовність скороченням кроків, що не впливають на цінність і відповідно витрат.

для кожної функції визначаються повні річні витрати й кількість робочих часів.

для кожної функції наоснові оцінок попереднього пункту визначається кількісна характеристика джерел витрат.

після того, як для кожної функції будуть визначені їх джерела витрат, проводиться кінцевий розрахунок витрат на виробництво продукту.

3.1 Постановка задачі функціонально-вартісного аналізу

У роботі застосовується метод ФВА для проведення аналізу розробки.

Відповідно цьому варто обирати і систему показників якості програмного продукту.

Технічні вимоги до продукту наступні:

забезпечувати високу швидкість обробки великих об’ємів даних у реальному часі;

забезпечувати взаємодію з кластерним середовищем;

передбачати мінімальні витрати на впровадження програмного продукту.

3.1.1 Обґрунтування функцій програмного продукту

Головна функція F_0 – розробка розподіленої системи для потокової обробки великих об’ємів даних у реальному часі з застосуванням фреймворку Apache Storm:

F_1 – вибір ОС на якій буде встановлюватись система.;

F_2 – вибір версії фреймворку;

F_3 – мови програмування.

Кожна з основних функцій може мати декілька варіантів реалізації.

Функція F_1:

а) Windows;

б) Linux(Ubuntu)

Функція F_2:

а) більш нова(остання версія);

б) більш стабільна.

Функція F_3:

а) мова програмування Java;

б) мова програмування Python;

3.1.2 Варіанти реалізації основних функцій

Варіанти реалізації основних функцій наведені у морфологічній карті системи (рис. 3.1). На основі цієї карти побудовано позитивно-негативну матрицю варіантів основних функцій (таблиця 3.1).

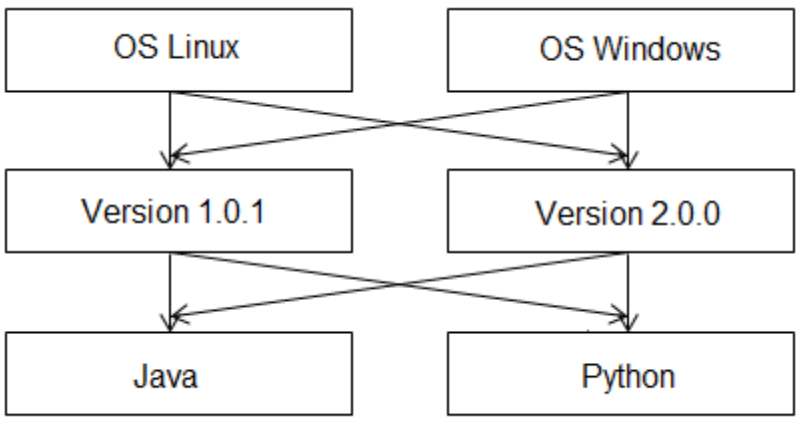

Рисунок 3.1 – Морфологічна карта

Варіанти реалізації основних функцій зведено у позитивно-негативну матрицю варіантів основних функцій (таблиця 3.1).

Таблиця 3.1 – Позитивно-негативна матриця

Основні функції	**Варіанти реалізації**	**Переваги**	**Недоліки**
F3	*А*	Більш зручна розробка	ОС не розвинена як серверна платформа
	Б	Менш зручна розробка	ОС використовується як серверна поатформа
F2	А	Більше можливостей, більший потенціал на майбутнє	Можливість наявності помилок, недостача мануалів
	Б	Вірогідно менше помилок, більше інформації та мануалів	Потенційно поступиться місцем новій версії у майбутньому
F3	*А*	Більш поширена мова	Більш складний та об’ємний код
	Б	Більш простий і короткий код	Час на вивчання мови, недостача мануалів

На основі аналізу позитивно-негативної матриці робимо висновок, що при розробці програмного продукту деякі варіанти реалізації функцій варто відкинути, тому, що вони не відповідають поставленим перед програмним продуктом задачам. Ці варіанти відзначені у морфологічній карті.

Функція *F1:*

Оскільки Windows є гіршою ОС для серверу, тому оберемо Ubuntu, варіант Б.

Функція *F2:*

Оскільки фреймворк розвивається і не стоїть на місці, доцільно обрати більш нову версію, яка має більше можливостей і має замінити стару, тому залишаємо варіант А.

Функція *F3:*

Java є більш поширеною мовою та має більше наявних методичних матеріалів щодо її застосування зі Storm, але Python має кращі бібліотеки для обробки даних, а також код на Python простіший і менший за об’ємом. Будемо розглядати обидва варіанти.

Таким чином, будемо розглядати такі варіанти реалізації ПП:

1. F1б – F2а – F3а
2. F1б – F2а – F3б

Для оцінювання якості розглянутих функцій обрана система параметрів, описана нижче.

3.2 Обґрунтування системи параметрів ПП

3.2.1 Опис параметрів

На підставі даних про основні функції, що повинен реалізувати програмний продукт, вимог до нього, визначаються основні параметри виробу, що будуть використані для розрахунку коефіцієнта технічного рівня.

Для того, щоб охарактеризувати програмний продукт, будемо використовувати наступні параметри:

– *X1* – швидкодія мови програмування;

– *X2* – об’єм пам’яті для збереження даних;

– *X3* – час обробки даних;

– *X4* – потенційний об’єм програмного коду.

X1: Відображає швидкодію операцій залежно від обраної мови програмування.

X2: Відображає об’єм пам’яті в оперативній пам’яті персонального комп’ютера, необхідний для збереження та обробки даних під час виконання програми.

X3: Відображає час, який витрачається на дії.

X4: Показує розмір програмного коду який необхідно створити безпосередньо розробнику.

3.2.2 Кількісна оцінка параметрів

Гірші, середні і кращі значення параметрів вибираються на основі вимог замовника й умов, що характеризують експлуатацію ПП як показано у табл. 3.2.

Таблиця 3.2 – Основні параметри ПП

Назва Параметра	**Умовні позначення**	**Одиниці виміру**	**Значення параметра**		
			гірші	**середні**	**кращі**
Швидкодія мови програмування	X1	Оп/мс	19000	11000	2000
Об’єм пам’яті для збереження даних	X2	Мб	32	16	8
Потенційний об’єм програмного коду	X3	кількість строк коду	2000	1500	1000
Час обробки запитів користувача	X4	мс	200	100	50

За даними таблиці 3.2 будуються графічні характеристики параметрів – рис. 3.2 – рис. 3.5.

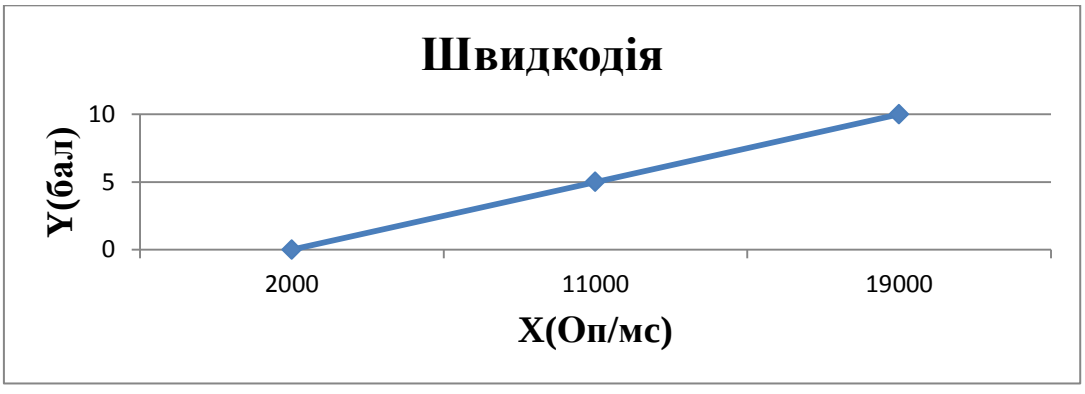

Рисунок 3.2 – X1, швидкодія мови програмування

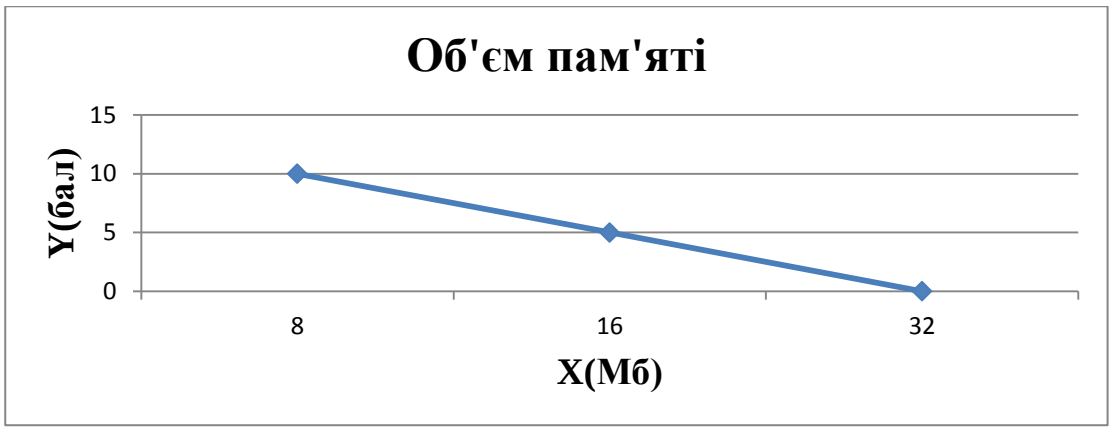

Рисунок 3.3 – X2, об’єм пам’яті для збереження даних

Рисунок 3.4– X3, потенційний об’єм програмного коду

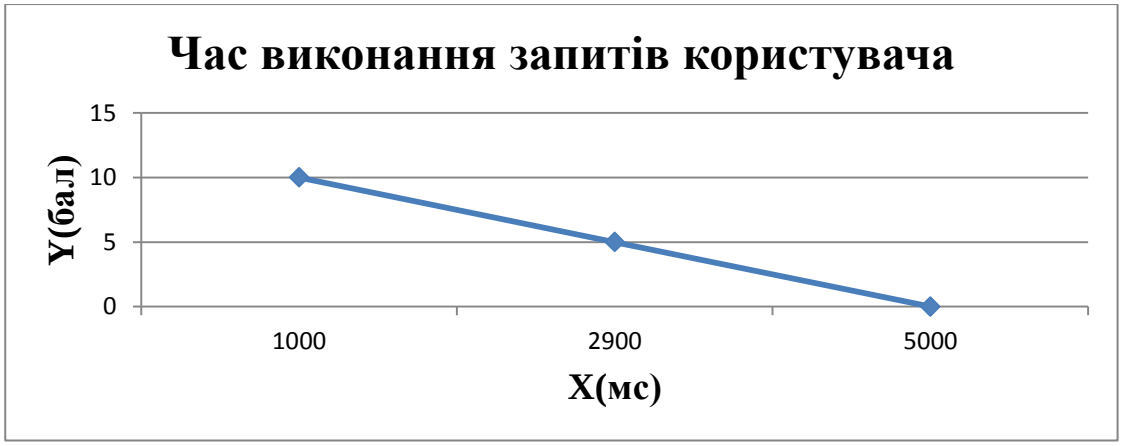

Рисунок 3.5 – X4,час обробки запитів користувача

3.2.3 Аналіз експертного оцінювання параметрів

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка програмного продукту, який має найбільш зручний інтерфейс.

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 7 людей. Визначення коефіцієнтів значимості передбачає:

– визначення рівня значимості параметра шляхом присвоєння різних рангів;

– перевірку придатності експертних оцінок для подальшого використання;

– визначення оцінки попарного пріоритету параметрів;

– обробку результатів та визначення коефіцієнту значимості.

Результати експертного ранжування наведені у таблиці 3.3.

Таблиця 3.3 – Результати ранжування параметрів

Позначення параметра	Назва параметра	Одиниці виміру	Ранг параметра за оцінкою експерта							Сума рангів R_i	Відхилення Δ_i	Δ_i^2
			1	2	3	4	5	6	7			
X1	Швидкодія мови програмування	Оп/мс	3	2	2	1	1	2	2	13	-4,5	20,25
X2	Об’єм пам’яті для збереження даних	Мб	1	1	1	2	2	1	1	9	-8,5	72,25
X3	Час обробки запитів користувача	Мс	2	3	3	3	3	3	4	21	3,5	12,25
X4	Потенційний об’єм програмного коду	кількість строк коду	4	4	4	4	4	4	3	27	9,5	20,25
	Разом		10	10	10	10	10	10	10	70	0	195

Для перевірки степені достовірності експертних оцінок, визначимо наступні параметри:

а) сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^{N} r_{ij} R_{ij} = \frac{Nn(n+1)}{2} = 70,$$

де N – число експертів, n – кількість параметрів;

б) середня сума рангів:

$$T = \frac{1}{n} R_{ij} = 17{,}5.$$

в) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T$$

Сума відхилень по всім параметрам повинна дорівнювати 0;

г)загальна сума квадратів відхилення:

$$S = \sum_{i=1}^{N} \Delta_i^2 = 195.$$

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 195}{7^2(4^3 - 4)} = 0{,}8 > W_k = 0{,}67$$

Ранжування можна вважати достовірним, тому що знайдений коефіцієнт узгодженості перевищує нормативний, котрий дорівнює 0,67.

Скориставшись результатами ранжирування, проведемо попарне порівняння всіх параметрів і результати занесемо у таблицю 3.4.

Таблиця 3.4 – Попарне порівняння параметрів

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	<	<	<	>	>	<	<	<	0,5
X1 і X3	<	>	>	>	>	>	>	>	1,5
X1 і X4	>	>	>	>	>	>	>	>	1,5
X2 і X3	>	>	>	>	>	>	>	>	1,5
X2 і X4	>	>	>	>	>	>	>	>	1,5
X3 і X4	>	>	>	>	>	>	<	>	1,5

Числове значення, що визначає ступінь переваги *i*–го параметра над *j*–тим, a_{ij} визначається по формулі:

$$a_{ij} = \begin{cases} 1{,}5 \text{ при } X_i > X_j \\ 1.0 \text{ при } X_i = X_j \\ 0.5 \text{ при } X_i < X_j \end{cases}$$

З отриманих числових оцінок переваги складемо матрицю $A = \| a_{ij} \|$.

Для кожного параметра зробимо розрахунок вагомості $K_{вi}$ за наступними формулами:

$$K_{\text{вi}} = \frac{b_i}{\sum_{i=1}^{n} b_i}, \text{де } b_i = \sum_{i=1}^{N} a_{ij}.$$

Відносні оцінки розраховуються декілька разів доти, поки наступні значення не будуть незначно відрізнятися від попередніх (менше 2%).На другому і наступних кроках відносні оцінки розраховуються за наступними формулами:

$$K_{\text{вi}} = \frac{b_i'}{\sum_{i=1}^{n} b_i'}, \text{де } b_i' = \sum_{i=1}^{N} a_{ij} b_j.$$

Як видно з таблиці 3.5, різниця значень коефіцієнтів вагомості не перевищує 2%, тому більшої кількості ітерацій не потрібно.

Таблиця 3.5 – Розрахунок вагомості параметрів

Параметриx_i	Параметриx_j				Перша ітер.		Друга ітер.		Третя ітер	
	X1	X2	X3	X4	b_i	$K_{\text{ві}}$	b_i^1	$K_{\text{ві}}^1$	b_i^2	$K_{\text{ві}}^2$
X1	1,0	0,5	1,5	1,5	4,5	0,281	16,25	0,275	59,125	0,274
X2	1,5	1,0	1,5	1,5	5,5	0,344	21,25	0,36	77,875	0,361
X3	0,5	0,5	1,0	1,5	3,5	0,219	12,25	0,208	44,875	0,207
X4	0,5	0,5	0,5	1,0	2,5	0,156	9,25	0,157	34,125	0,158
Всього:					16	1	59	1	216	1

3.3 Аналіз рівня якості варіантів реалізації функцій

Визначаємо рівень якості кожного варіанту виконання основних функцій окремо.

Абсолютні значення параметрів *X2*(об’єм пам’яті для збереження даних) та *X1* (швидкодія мови програмування) відповідають технічним вимогам умов функціонування даного ПП.

Абсолютне значення параметра *X3 (*потенційний об’єм програмного коду) обрано не найгіршим (не максимальним), тобто це значення відповідає або варіанту а) 2000 або варіанту б) 1500

Абсолютне значення параметра *X4 (ч*ас обробки запитів користувача) обрано не найкращим (не мінімальним), тобто це значення відповідає або варіанту а) 50 мс або варіанту б) 100 мс

Коефіцієнт технічного рівня для кожного варіанта реалізації ПП розраховується так (таблиця 3.6):

$$K_K(j) = \sum_{i=1}^{n} K_{вi,j} B_{i,j} ,$$

де n – кількість параметрів;$K_{вi}$– коефіцієнт вагомості i–го параметра;B_i – оцінка i–го параметра в балах.

Таблиця 3.6 – Розрахунок показників рівня якості варіантів реалізації основних функцій ПП

Основні функції	Варіант реалізації функції	Параметри, що беруть участь у реалізації функцій	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1	Б	X1	100	5	0,274	1,37
F2	А	X2	500	6,7	0,361	2,42
F3	А	X3	1500	2,5	0,158	0,395
	Б		800	6,4	0,158	1,01
	А	X4	50	7,5	0,207	1,55
	Б		100	5,5	0,207	1,14

За даними з таблиці 3.6 за формулою

$$K_K = K_{\text{ТУ}}[F_{1k}] + K_{\text{ТУ}}[F_{2k}] + \ldots + K_{\text{ТУ}}[F_{zk}],$$

визначаємо рівень якості кожного з варіантів:

$K_{K1} = 1{,}37 + 2{,}42 + 1{,}55 + 0{,}395 = 5{,}725$

$K_{K2} = 1{,}37 + 2{,}42 + 1{,}14 + 1{,}01 = 5{,}95$

Як видно з розрахунків, кращим є другий варіант, для якого коефіцієнт технічного рівня має найбільше значення.

3.4 Економічний аналіз варіантів розробки ПП

Для визначення вартості розробки ПП спочатку проведемо розрахунок трудомісткості.

Всі варіанти включають в себе два окремих завдання:

1. Розробка проекту програмного продукту;

2. Розробка програмної оболонки;

Завдання 1 за ступенем новизни відноситься до групи А, завдання 2 – до групи Б. За складністю алгоритми, які використовуються в завданні 1 належать до групи 1; а в завданні 2 – до групи 3.

Для реалізації завдання 1 використовується довідкова інформація, а завдання 2 використовує інформацію у вигляді даних.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань. Загальна трудомісткість обчислюється як

$$T_O = T_P \cdot K_П \cdot K_{СК} \cdot K_M \cdot K_{СТ} \cdot K_{СТ.М}, \quad (5.1)$$

де T_P – трудомісткість розробки ПП;$K_П$ – поправочний коефіцієнт;$K_{СК}$ – коефіцієнт на складність вхідної інформації; K_M – коефіцієнт рівня мови програмування;$K_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм;$K_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру степеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: T_P =90 людино-днів. Поправочний коефіцієнт, який враховує вид нормативно-довідкової інформації для першого завдання: $K_П$ = 1.7. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації для всіх семи завдань рівний 1: $K_{СК}$ = 1. Оскільки при розробці першого завдання використовуються стандартні модулі, врахуємо це за допомогою коефіцієнта $K_{СТ}$ = 0.8. Тоді, за формулою 5.1, загальна трудомісткість програмування першого завдання дорівнює:

$T_1 = 90 \cdot 1.7 \cdot 0.8 = 122.4$ людино-днів.

Проведемо аналогічні розрахунки для подальших завдань.

Для другого завдання (використовується алгоритм третьої групи складності, степінь новизни Б), тобто T_P =27 людино-днів, $K_П$ =0.9,$K_{СК}$ = 1,$K_{СТ}$ =0.8:

$T_2 = 27 \cdot 0.9 \cdot 0.8 = 19.44$ людино-днів.

Складаємо трудомісткість відповідних завдань для кожного з обраних варіантів реалізації програми, щоб отримати їх трудомісткість:

$T_I = (122.4 + 19.44 + 4.8 + 19.44) \cdot 8 = 1328{,}64$ людино-годин;

$T_{II} = (122.4 + 19.44 + 6.91 + 19.44) \cdot 8 = 1345.52$ людино-годин;

Найбільш високу трудомісткість має варіант II.

В розробці бере участь один програміст з окладом 25000 грн., один фінансовий аналітик з окладом 20000грн. Визначимо зарплату за годину за формулою:

$$С_ч = \frac{М}{T_m \cdot t} грн.,$$

де М – місячний оклад працівників;T_m – кількість робочих днів тиждень;t – кількість робочих годин в день.

$$С_ч = \frac{25000 + 20000}{2 \cdot 21 \cdot 8} = 134 грн.$$

Тоді, розрахуємо заробітну плату за формулою

$$С_{ЗП} = С_ч \cdot Т_i \cdot К_Д,$$

де $С_ч$– величина погодинної оплати праці програміста;$Т_i$ – трудомісткість відповідного завдання; $К_Д$ – норматив, який враховує додаткову заробітну плату.

Зарплата розробників за варіантами становить:

I. $С_{ЗП} = 134 \cdot 1328.64 \cdot 1.2 = 213645,312$ грн.

II. $С_{ЗП} = 134 \cdot 1345.52 \cdot 1.2 = 216359,616$ грн.

Відрахування на єдиний соціальний внесок в залежності від групи професійного ризику (II клас) становить 22%:

I. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 213645,312 \cdot 0.22 = 47002$ грн.

II. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 67281,38 \cdot 0.22 = 47600$ грн.

Тепер визначимо витрати на оплату однієї машино-години. ($С_М$)

Так як одна ЕОМ обслуговує одного програміста з окладом 25000 грн., з коефіцієнтом зайнятості 0,2 то для однієї машини отримаємо:

$С_Г = 12 \cdot М \cdot К_З = 12 \cdot 25000 \cdot 0,2 = 60000$ грн.

З урахуванням додаткової заробітної плати:

$С_{ЗП} = С_Г \cdot (1 + К_З) = 60000 \cdot (1 + 0.2) = 72000$ грн.

Відрахування на єдиний соціальний внесок:

$С_{ВІД} = С_{ЗП} \cdot 0.22 = 72000 \cdot 0,22 = 15840$ грн.

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 30000 грн.

$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР} = 1.15 \cdot 0.25 \cdot 30000 = 8625$ грн.,

де $К_{ТМ}$– коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача; $К_А$– річна норма амортизації; $Ц_{ПР}$– договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо як:

$С_Р = К_{ТМ} \cdot Ц_{ПР} \cdot К_Р = 1.15 \cdot 30000 \cdot 0.05 = 1725$ грн.,

де $К_Р$– відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$Т_{ЕФ} = (Д_К – Д_В – Д_С – Д_Р) \cdot t_З \cdot К_В = (365 – 104 – 8 – 16) \cdot 8 \cdot 0.9 = 1706.4$ годин,

де $Д_К$ – календарна кількість днів у році; $Д_В$, $Д_С$ – відповідно кількість вихідних та святкових днів; $Д_Р$ – кількість днів планових ремонтів устаткування; t –кількість робочих годин в день; $К_В$– коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$С_{ЕЛ} = Т_{ЕФ} \cdot N_С \cdot К_З \cdot Ц_{ЕН} = 1706{,}4 \cdot 0{,}156 \cdot 0{,}9733 \cdot 2{,}0218 = 523.83$ грн.,

де $N_С$ – середньо-споживча потужність приладу; $К_З$– коефіцієнтом зайнятості приладу; $Ц_{ЕН}$ – тариф за 1 КВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

$С_Н = Ц_{ПР} \cdot 0.67 = 8000 \cdot 0{,}67 = 5360$ грн.

Тоді, річні експлуатаційні витрати будуть:

$$С_{ЕКС} = С_{ЗП} + С_{ВІД} + С_А + С_Р + С_{ЕЛ} + С_Н$$

$С_{ЕКС} = 72000 + 15840 + 8625 + 1725 + 523.83 + 5360 = 104073{,}83$ грн.

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$С_{М\text{-}Г} = С_{ЕКС} / Т_{ЕФ} = 104073{,}83/1706{,}4 = 61$ грн/час.

Оскільки в даному випадку всі роботи, які пов‘язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу, в залежності від обраного варіанта реалізації, складає:

$$С_М = С_{М-Г} \cdot Т$$

I. $С_М = 61 \cdot 1328,64 = 81047,04$ грн.;

II. $С_М = 61 \cdot 1345.52 = 82076,72$ грн.;

Накладні витрати складають 67% від заробітної плати:

$$С_Н = С_{ЗП} \cdot 0,67$$

I. $С_Н = 213645,312 \cdot 0,67 = 143142,36$ грн.;

II. $С_Н = 216359,616 \cdot 0,67 = 144960,94$ грн.;

Отже, вартість розробки ПП за варіантами становить:

$$С_{ПП} = С_{ЗП} + С_{ВІД} + С_М + С_Н$$

I. $С_{ПП} = 213645,312 + 47002 + 81047,04 + 143142,36 = 484836,712$ грн.;

II. $С_{ПП} = 216359,616 + 47600 + 82076,72 + 144960,94 = 490997,276$ грн.;

3.5 Вибір кращого варіанта ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$К_{ТЕРj} = К_{Кj} / С_{Фj},$$

$К_{ТЕР1} = 5,325 / 484836,712 = 1,098 \cdot 10^{-5}$;

$К_{ТЕР2} = 6,35 / 490997,276 = 1,29 \cdot 10^{-5}$;

Як бачимо, найбільш ефективним є другий варіант реалізації програми з коефіцієнтом техніко-економічного рівня $К_{ТЕР1} = 1,29 \cdot 10^{-5}$.

3.6 Висновки

В даному розділі проведено повний функціонально-вартісний аналіз ПП, який було розроблено в рамках дипломного проекту. Процес аналізу можна умовно розділити на дві частини.

В першій з них проведено дослідження ПП з технічної точки зору: було визначено основні функції ПП та сформовано множину варіантів їх реалізації; на основі обчислених значень параметрів, а також експертних оцінок їх важливості було обчислено коефіцієнт технічного рівня, який і дав змогу визначити оптимальну з технічної точки зору альтернативу реалізації функцій ПП.

Другу частину ФВА присвячено вибору із альтернативних варіантів реалізації найбільш економічно обґрунтованого. Порівняння запропонованих варіантів реалізації в рамках даної частини виконувалось за коефіцієнтом ефективності, для обчислення якого були обчислені такі допоміжні параметри, як трудомісткість, витрати на заробітну плату, накладні витрати.

Після виконання функціонально-вартісного аналізу програмного комплексу що розроблюється, можна зробити висновок, що з альтернатив, що залишились після першого відбору двох варіантів виконання програмного комплексу оптимальним є другий варіант реалізації програмного продукту. У нього виявився найкращий показник техніко-економічного рівня якості $К_{ТЕР} = 1,29 \cdot 10^{-5}$.

Цей варіант реалізації програмного продукту має такі параметри:

– мова програмування – Java

– остання версія фреймворку Storm;

– розробка для ОС Linux(Ubuntu).

Така конфігурація дозволить вести розробку найбільш ефективно з фінансової точки зору, а також дасть найбільш типовий продукт, який побудований відповідно до реальних тенденцій.

ВИСНОВКИ

Дипломна робота присвячена вирішенню задачі розподілених обчислень великих об’ємів даних із застосуванням фреймворку Apache Storm.

Було досліджено підхід Storm до розв’язання задачі та виявлено основні області застосування фреймворку, а також типові задачі які вирішують із застосуванням фреймворку. Було виявлено такі особливості фреймворку у порівнянні з його аналогами, як більш коротка затримка на обробку вхідних даних порівняно зі Spark Streaming та менша необхідна кількість зусиль на побудову системи порівняно з Akka. В той же час, в порівнянні з цими аналогами, Storm програє в розробці machine-learning систем першому, та програє в гнучкості побудови системи другому аналогу.

Було розглянуто приклади застосувань Storm великими компаніями світового рівня. Виявилось, що Storm досить часто використовується на практиці для задач розподіленого обчислення потоків даних та їх аналізу в реальному часі.

Було розглянуто засоби Storm для побудови розподілених систем реального часу, серед яких, крім базових примітивів Bolt та Spout, є засоби для пакетної обробки подібно до Spark Streaming, що реалізовані в Storm Trident. Використання цієї технології дозволяє отримати семантику exactly-once і досягти більшої пропускної здатності, але мінусом технології є збільшення навантаження на систему, порівняно зі Storm Core, та збільшення затримок на обробку даних.

У ході аналізу паралелізму та продуктивності Storm були описані основні рівні, на яких базується паралелізм в Storm та визначено методи розподілу задач між машинами кластера, які дозволять пришвидшити роботу системи. Було виявлено, що збитковість розподілу задач між worker процесами можна використати для подальшого пере балансування системи у разі розширення кластера. Також було приведено метод ізольованого розподілу топологій, який

дозволяє запобігти боротьбі за ресурси між процесами топологій на одній машині.

У другому розділі дипломної роботи було спроектовано систему, в якій один з компонентів, компонент статистики, є доцільним реалізувати з використанням Storm. Був описаний процес налаштування середовища розробки та реалізації компоненту.

Тестування розробленого компоненту на локальній машині показали, що логіка компоненту виконується правильно і не відбувається порушення логіки через паралельне виконання обробки даних. Тому можна зробити висновок, що розроблений компонент функціонує правильно і виконує задану логіку без помилок.

Тестування швидкості обробки даних у розмірі 10^7 записів дозволяють побачити приріст швидкості обробки в паралельному режимі, порівняно з послідовним режимом. Оптимальний результат швидкодії був досягнений на досить низькій кількості потоків, тому що тестування проводилося на локальній машині. Розміщення системи на потужному розподіленому кластері дозволить збільшити пропускну здатність, тому не виникне затримок при отриманні одночасно декількох мільйонів вхідних записів на обробку.

Наступним етапом розробки є встановлення компоненту статистики на реальний кластер та інтеграція з зовнішніми системами, такими як брокер повідомлень Kafka та розподіленою базою даних Cassandra або HBase. Також для використання системи в реальних умовах потрібно розробити компоненти головного та фронтального серверів, які дозволять отримувати дані безпосередньо від користувачів та передавати їх на обробку в компонент статистики. Таку систему можна запровадити в якості сервісу інтернет-держави в Україні, що має збільшити явку виборців та дозволить вести статистику в реальному часі.

ПЕРЕЛІК ПОСИЛАНЬ

1. Jean J. Labrosse. DSP in Embedded Systems / Jean J. Labrosse. – Newnes, 2007. – 792 p.
2. Офіційна сторінка Apache Storm – PoweredBy. – Режим доступу: http://storm.apache.org/Powered-By.html.
3. Офіційна сторінка Twitter – Heron. – Режим доступу: https://blog.twitter.com/2015/flying-faster-with-twitter-heron. – Дата доступу : 29.04.2016.
4. Офіційна сторінка Apache Storm – Tutorial. – Режим доступу: http://storm.apache.org/releases/current/Tutorial.html. – Дата доступу : 29.04.2016.
5. Офіційна сторінка Apache Storm – Paralellism. – Режим доступу: http://storm.apache.org/releases/1.0.1/Understanding-the-parallelism-of-a-Storm-topology.html. – Дата доступу : 29.04.2016.
6. Офіційна сторінка Apache Storm – Why to use storm? – Режим доступу: http://storm.apache.org/index.html. – Дата доступу : 29.04.2016.
7. Офіційна сторінка Apache Spark – MLib guide. – Режим доступу: http://spark.apache.org/docs/latest/mllib-guide.html. – Дата доступу : 29.04.2016.
8. Офіційна сторінка Apache Storm – Trident tutorial. – Режим доступу: http://storm.apache.org/releases/current/Trident-tutorial.html. – Дата доступу : 29.04.2016.
9. Публікація Adam Warski – Akka vs Storm. – Режим доступу: http://www.warski.org/blog/2013/06/akka-vs-storm/. – Дата доступу : 29.04.2016.
10. Публікація розробників з twitter – storm benchmark. – Режим доступу: https://twitter.com/nathanmarz/status/207989068519317505. – Дата доступу : 29.04.2016.

11. Публікація розробників з keen.io blog – storm performance. – Режим доступу: http://highscalability.com/blog/2016/3/8/performance-tuning-apache-storm-at-keen-io.html. – Дата доступу : 29.04.2016.
12. Офіційна сторінка Apache Storm – Storm scheduler. – Режим доступу: http://storm.apache.org/releases/1.0.1/Storm-Scheduler.html. – Дата доступу : 29.04.2016.
13. Офіційна сторінка Apache Spark. – Режим доступу: http://spark.apache.org/docs. – Дата доступу : 29.04.2016.
14. Публікація Tayor Goez – Stortm vs Spark. – Режим доступу: http://www.slideshare.net/ptgoetz/apache-storm-vs-spark-streaming. – Дата доступу : 29.04.2016.
15. Публікація Rassul Fazelat – Storm vs Spark. – Режим доступу: https://www.linkedin.com/pulse/comprehensive-analysis-data-processing-part-deux-apache-fazelat. – Дата доступу : 29.04.2016.
16. Gul Agha. Actors: A Model of Concurrent Computation in Distributed Systems. / Gul Agha – MIT Press, 1985.

ДОДАТОК А

Код програми

VotingSpout.java

```
package voting;


import org.apache.log4j.Logger;
import org.apache.storm.spout.SpoutOutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseRichSpout;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;
import java.util.*;
public class VotingSpout extends BaseRichSpout{
    SpoutOutputCollector _collector;
    private static final Logger LOG = Logger.getLogger(VotingSpout.class);
    private boolean done = false;
    private boolean first = true;
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("voterId", "candidateId"));
    }
    @Override
    public void open(Map conf, TopologyContext context, SpoutOutputCollector
collector) {
        _collector = collector;
    }
    @Override
    public void nextTuple() {
        VotingData.VoterCandidateEntry entry = null;
```

```
//        if (VotingData.entriesQueue.isEmpty()){
//            if (!done){
//                LOG.info("Queue is empty");
//                done = true;
//            }
//        } else {
            try {
                entry = VotingData.entriesQueue.take();
                Long voterId = entry.voterId;
                Long candidateId = entry.candidateId;
                _collector.emit(new Values(voterId, candidateId));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
//        }
    }
}
```

CandidateVotesCountBolt.java

```
package voting;
import org.apache.commons.collections.map.HashedMap;
import org.apache.log4j.Logger;
import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.task.OutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.TopologyBuilder;
import org.apache.storm.topology.base.BaseRichBolt;
import org.apache.storm.tuple.Tuple;
```

```
import org.apache.storm.utils.Utils;

import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.atomic.AtomicInteger;

public class CandidateVotesCountBolt extends BaseRichBolt {
    OutputCollector _collector;
    private static final Logger LOG = Logger.getLogger(CandidateVotesCountBolt.class);

    @Override
    public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
        _collector = collector;
    }

    @Override
    public void execute(Tuple input) {
        Long candidateId = input.getLong(1);
        Long voterId = input.getLong(0);
        AtomicInteger count = VotingData.candidateVotes.get(candidateId);
        if (count == null) {
            count = new AtomicInteger(0);
            AtomicInteger old = VotingData.candidateVotes.putIfAbsent(candidateId, count);
            if (old != null) {
                count = old;
            }
        }
        count.incrementAndGet();
    }
    @Override
```

```
        public void declareOutputFields(OutputFieldsDeclarer declarer) {


        }
        private static void printExpected(){
            Map<Long, Integer> candidateVotes = new HashMap<>();
            for (VotingData.VoterCandidateEntry vce : VotingData.voterCandidateEntries){
                Integer votesCount = candidateVotes.get(vce.candidateId);
                if (votesCount == null) {
                    candidateVotes.put(vce.candidateId, 1);
                } else {
                    candidateVotes.put(vce.candidateId, votesCount + 1);
                }
            }
            for (int i = 1; i < VotingData.CANDIDATES_COUNT + 1; i++) {
                Long id = (long) i;
                System.out.println("Candidate " + i + " expected to get " + candidateVotes.get(id) + " votes");
                System.out.println("Candidate " + i + " actually got " + VotingData.candidateVotes.get(id) + " votes");
            }
        }
        public static void main(String[] args) throws Exception {
            TopologyBuilder builder = new TopologyBuilder();
            builder.setSpout("votingSpout", new VotingSpout(), 2);
            builder.setBolt("candidateVotesCountBolt", new CandidateVotesCountBolt(), 4).shuffleGrouping("votingSpout");


            Config conf = new Config();
            conf.setDebug(false);


            if (args != null && args.length > 0) {
                conf.setNumWorkers(3);
```

```
                StormSubmitter.submitTopologyWithProgressBar(args[0],           conf,
builder.createTopology());
            }
            else {
                LocalCluster cluster = new LocalCluster();
                cluster.submitTopology("voting", conf, builder.createTopology());
                Utils.sleep(100000);
                cluster.killTopology("voting");
                cluster.shutdown();
                printExpected();
                return;
            }
        }
    }
```

VotingData.java

```
    package voting;
    import com.google.common.util.concurrent.AtomicDouble;
    import java.util.ArrayList;
    import java.util.List;
    import java.util.concurrent.*;
    import java.util.concurrent.atomic.AtomicInteger;
    public class VotingData {
        static class VoterCandidateEntry{
            long voterId;
            long candidateId;
            public VoterCandidateEntry(long voterId, long candidateId) {
                this.voterId = voterId;
                this.candidateId = candidateId;
            }
            @Override
```

```
            public boolean equals(Object o) {
                if (this == o) return true;
                if (o == null || getClass() != o.getClass()) return false;
                VoterCandidateEntry that = (VoterCandidateEntry) o;
                if (voterId != that.voterId) return false;
                return candidateId == that.candidateId;
            }
            @Override
            public int hashCode() {
                int result = (int) (voterId ^ (voterId >>> 32));
                result = 31 * result + (int) (candidateId ^ (candidateId >>> 32));
                return result;
            }
        }
        public static ConcurrentHashMap<Long, AtomicInteger> candidateVotes = new
ConcurrentHashMap<>();
        public static List<VoterCandidateEntry> voterCandidateEntries =
generateVoterCandidateEntries();
        public static final int VCE_SIE = (int) 1e7;
        public static BlockingQueue<VoterCandidateEntry> entriesQueue = new
ArrayBlockingQueue<VoterCandidateEntry>(VCE_SIE, false, voterCandidateEntries);
        public static final int CANDIDATES_COUNT = 3;
        private static List<VoterCandidateEntry> generateVoterCandidateEntries(){
            List<VoterCandidateEntry> entries = new ArrayList<>(VCE_SIE );
            for (int i = 1; i < VCE_SIE + 1; i++) {
                entries.add(new                                VoterCandidateEntry(i,
ThreadLocalRandom.current().nextInt(1, CANDIDATES_COUNT +1)));
            }
            return entries;
        }
    }
```