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 спеціальності 7.05010102, 8.05010102 Інформаційні технології проектування
_____7.05010103, 8.05010103 Системне проектування_____
(код та назва напряму підготовки)

_____

на тему: _____Хмарна платформа для Інтернету речей_____
_____

Виконав (-ла): студент (-ка) __4__ курсу, групи __ДА-22__
(шифр групи)

_____Ткаченко Дмитро Анатолійович_____ _____
(прізвище, ім'я, по батькові) (підпис)

Керівник _____доцент, к.т.н. Харченко К.В._____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант економічний розділ проф., д.е.н. Семенченко Н.В.___ _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент _____проф., д.т.н. Стіренко С.Г._____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Нормоконтроль _____ст.. викладач Британов О.А._____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (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
Ткаченку Дмитру Анатолійовичу
(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) Хмарна платформа для Інтернету речей

керівник проекту (роботи) Харченко Костянтин Васильович, к.т.н., доцент
(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) 10.06.2016

3. Вихідні дані до проекту (роботи) _____

Хмарна платформа для Інтернету речей, яка:

- вирішує основні задачі, що висуваються до подібних платформ (збереження, аналіз даних тощо);
- розроблена з урахуванням кращих практик і підходів до архітектури програмного забезпечення;
- має специфічну функціональність, якої не мають інші системи;
- містить засоби для розгортання на хмарних хостингах, а також інструменти керування групою серверів.

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

1. Розглянути предметну область та існуючі рішення, виділити шляхи їх покращення.
2. Сформулювати функціональні та нефункціональні вимоги до системи.
3. Розробити архітектуру системи; описати протоколи та формати обміну даними між компонентами.
4. Виділити вимоги до складових частин системи; обґрунтувати вибір сторонніх рішень.
5. Виконати програмну реалізацію платформи. Описати важливі аспекти її реалізації.
6. Розробити схему розгортання системи та описати інструменти керування хмарним кластером.
7. Навести приклад використання системи.

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

1. Архітектура платформи – плакат.
2. Діаграми потоків даних – плакат.
3. Діаграма розгортання. Формати даних і протоколи – плакат.

6. Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	Семенченко Н.В., проф.		

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 предметної області та існуючих рішень. Формулювання вимог до системи	28.02.2016	
4	Розробка архітектури системи	10.03.2016	
5	Вибір сторонніх рішень для складових частин системи	15.03.2016	
6	Опис аспектів хмарного розгортання системи	25.03.2016	

7	Виконання програмної реалізації системи	25.04.2016	
8	Розробка прикладу використання платформи	30.04.2016	
9	Оформлення дипломної роботи	31.05.2016	
10	Отримання допуску до захисту та подача роботи в ДЕК	10.06.2016	

Студент _____ Ткаченко Д.А.
(підпис) (ініціали, прізвище)

Керівник роботи _____ Харченко К.В.
(підпис) (ініціали, прізвище)

АНОТАЦІЯ

бакалаврської дипломної роботи Ткаченка Дмитра Анатолійовича

на тему: «Хмарна платформа для Інтернету речей»

Дана дипломна робота присвячена дослідженню серверних систем, що забезпечують роботу Інтернету речей, та розробці хмарної платформи.

В роботі розглянуто побудову платформи за допомогою новітніх підходів до розробки розподілених систем прийому, зберігання та обробки великих обсягів даних. Для побудови платформи використано сучасні бібліотеки та бази даних із відкритим вихідним кодом. Досліджено розгортання платформи на кластері у хмарному хостингу. Запропоновано архітектуру платформи, яка забезпечує її високу масштабованість і ефективність, а також дозволяє реалізувати рішення для моніторингу мережі пристроїв і роботи з адміністративними даними, які є більш функціональними за існуючі.

Використання платформи продемонстровано на прикладі. Показано також підключення реального пристрою до системи.

Загальний обсяг роботи: 112 сторінок, 22 рисунки, 10 таблиць, 32 посилання, 1 додаток на 12 сторінках.

Ключові слова: Інтернет речей, хмарна платформа, розподілена система, обробка великих обсягів даних, моніторинг мережі пристроїв, Apache Spark, Apache Kafka, Apache Cassandra, MongoDB.

АННОТАЦИЯ

бакалаврской дипломной работы Ткаченко Дмитрия Анатольевича

на тему: «Облачная платформа для Интернета вещей»

Данная дипломная работа посвящена исследованию серверных систем, обеспечивающих работу Интернета вещей, и разработке облачной платформы.

В работе рассмотрено построение платформы с помощью новейших подходов к разработке распределенных систем приема, хранения и обработки больших объемов данных. Для построения платформы использованы современные библиотеки и базы данных с открытым исходным кодом. Исследовано развертывание платформы на кластере в облачном хостинге. Предложена архитектура платформы, которая обеспечивает ее высокую масштабируемость и эффективность, а также позволяет реализовать решения для мониторинга сети устройств и работы с административными данными, являющиеся более функциональными, чем существующие.

Использование платформы продемонстрировано на примере. Показано также подключение реального устройства к системе.

Общий объем работы: 112 страниц, 22 рисунка, 10 таблиц, 32 библиографических наименования, 1 приложение на 12 страницах.

Ключевые слова: Интернет вещей, облачная платформа, распределенная система, обработка больших объемов данных, мониторинг сети устройств, Apache Spark, Apache Kafka, Apache Cassandra, MongoDB.

ABSTRACT

of the bachelor's thesis of Tkachenko Dmytro Anatoliyovych

"A cloud platform for the Internet of things"

This thesis studies server-side systems of the Internet of things and is aimed at developing a cloud platform.

This work describes the usage of a novel approaches to development of a distributed systems that receive, store and process big amounts of data. The platform was built using state-of-the-art open source libraries and databases. Cloud deployment of a system was studied. This thesis proposes a platform architecture, which ensures its high scalability and efficiency, and allows to implement solutions for monitoring a network of devices and management of an administrative data, that are better than the ones provided by the existing systems.

Application of the platform was demonstrated by an example. Connection of a real device to the system was also shown.

The thesis contains 112 pages, 22 figures, 10 tables, 32 references and 1 appendix (12 pages).

Key words: Internet of things, cloud platform, distributed system, big data processing, device network monitoring, Apache Spark, Apache Kafka, Apache Cassandra, MongoDB.

ЗМІСТ

ВСТУП

Інтернет речей є одним із найбільш перспективних напрямків розвитку інформаційно-комунікаційних технологій. Кількість підключених до мережі Інтернет пристроїв бурхливо зростає, і це вимагає сучасних підходів до побудови високонавантажених серверних систем. Платформи Інтернету речей мають забезпечувати можливість аналізувати різні аспекти даних, що потрібно для оптимізації різноманітних виробничих та інших процесів.

Проблеми та задачі у сфері побудови серверних систем для Інтернету речей можна поділити на *загальні*, які притаманні багатьом іншим системам обробки великих даних, та *специфічні*, що виникають лише в цій області. Загальними задачами є побудова та використання ефективних, відмовостійких, масштабованих і розподілених систем роботи з даними. Специфічними для Інтернету речей проблемами є забезпечення надійного керування та моніторингу пристроїв.

У цій роботі визначаються вимоги до побудови хмарних платформ для Інтернету речей, досліджуються існуючі платформи та виокремлюються шляхи їх покращення.

Вивчаються й описуються архітектурні рішення в галузі обробки великих обсягів даних та підходи до розробки програмного забезпечення, які дозволяють реалізувати платформу таким чином, аби вона задовольняла означеним функціональним і нефункціональним вимогам.

Досліджуються можливості сторонніх рішень, які можна використати для побудови платформи. Обґрунтовується вибір протоколів і форматів даних, що дають змогу забезпечити найкращі параметри системи.

Кінцевою метою роботи є побудова хмарної платформи для Інтернету речей, що реалізує основні підсистеми, які потрібні подібним системам, а саме: приймання, зберігання, обробки операційних даних; автентифікації та безпеки; роботи з адміністративними даними; моніторингу.

Також розглядаються аспекти розгортання системи на кластері у хмарному хостингу, такі як балансування навантаження, обробка відмов вузлів, забезпечення роботи з середовищем із динамічно змінюваною конфігурацією.

У роботі продемонстровано використання платформи на прикладі, що моделює роботу з мережею метеорологічних станцій. Підключено реальний мікрокомп’ютер і показано його взаємодію з системою, якій він надсилає дані зі свого сенсора.

1. ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ Й ІСНУЮЧИХ РІШЕНЬ. МЕТА РОБОТИ

1.1 Предметна область і актуальність роботи

Інтернет речей (Internet of Things – IoT) – концепція обчислювальної мережі фізичних об’єктів (“речей”), оснащених вбудованими технологіями для взаємодії один з одним або із зовнішнім середовищем, що розглядає організацію таких мереж як явище, яке здатне перебудувати економічні та суспільні процеси та виключити з частини дій та операцій необхідність участі людини.

Індустрія інтернету речей постійно збільшувалася в останні роки, і її ріст лише прискорюватиметься. Дослідницька і консалтингова компанія Gartner у своїй доповіді [1] зробила прогноз, що у 2016 році у світі налічуватиметься 6,4 мільярди “речей” – 30-відсоткове зростання порівняно із 2015 роком (4,9 мільярдів пристроїв). У грошовому вимірі, згідно доповіді, розмір індустрії зросте із 1183 мільярдів доларів у 2015 році до 1414 мільярдів доларів у 2016-му. У 2020-му році підключених пристроїв буде вже 20,7 мільярдів, а загальні витрати на них становитимуть 3 трильйони доларів (з них близько половини – споживачі, інша половина – бізнес). Найбільший у світі виробник мережевого обладнання, компанія Cisco [2], дає ще більш оптимістичний прогноз щодо розміру ринку IoT: 14,4 трильйонів доларів до 2022 року.

Інтернет речей проникає у всі сфери життя і забезпечує роботу багатьох систем: від суто споживацьких (таких як різноманітні побутові сенсори, носима електроніка, розумні будинки тощо) до індустріальних (керування і моніторинг виробничих процесів, розумні енергосистеми, розумні міста, автономні автомобілі тощо). За допомогою “речей” додана вартість збільшуватиметься [2] за рахунок:

- покращення клієнтського досвіду (customer experience);

- зменшення часу, що проходить від задуму продукту до його появи у продажу (time-to-market);
- поліпшення ланцюжків постачання та логістики;
- збільшення продуктивності працівників;
- більш ефективного використання активів (зменшення витрат).

Розвиток інтернету речей стимулює прогрес у багатьох галузях інженерних наук. Очевидно, що зростання попиту на апаратне забезпечення для IoT сприяє інноваціям в галузі електроніки, яка забезпечує індустрію інтернету речей електронними платами, сенсорами, акумуляторами тощо.

Специфіка портативних пристроїв накладає певні обмеження на програмне забезпечення, яке керує ними. Обмеженість обчислювальних можливостей компактних плат, а також вимоги щодо використання енергії (особливо для “речей”, які живляться від акумулятора), вимагає економного ставлення до ресурсів, а також переносу майже всіх операцій по обробці даних на бік сервера.

Серверні платформи, які обробляють показання сенсорів, також повинні обов’язково враховувати те, що дані неминуче містять похибки, а час від часу пристрої можуть виходити з ладу та функціонувати зовсім неправильно.

Питання безпеки також набуває все більшого значення в світі, де велика кількість пристроїв збирає, обробляє та передає різноманітні дані, які мають певну індустріальну чи бізнесову цінність, або містять персональну інформацію про користувачів. Також необхідно реагувати на виклики, пов’язані з постійно зростаючою кількістю інформаційних загроз.

Розробка програмних рішень, що керують великою кількістю пристроїв, вимагає застосування новітніх підходів у галузі хмарних обчислень, обробки і зберігання великих даних (Big Data). Такі серверні рішення повинні горизонтально масштабуватися, мати велику пропускну спроможність, відмовостійкість, короткий час відповіді на запити.

Особливе значення для ефективного використання даних в індустрії інтернету речей має наявність різноманітних інструментів їх аналізу. Наразі більшість пристроїв є суто інформаційними, тобто такими, які лише збирають і

передають дані та не виконують, наприклад, функцій керування. Тому розумна обробка даних є ключовою потребою таких систем.

Інформація, зібрана пристроями інтернету речей, приміром, може використовуватися для оптимізації певних виробничих процесів, контролю за якістю виробів, моніторингу працездатності інших систем. Вона зазвичай проходить процес формальної обробки, наприклад, агрегації, а потім досліджується спеціалістами. Подальший розвиток інструментів інтелектуального аналізу даних за допомогою методів машинного навчання може зменшити або взагалі прибрати необхідність залучення людей. За наявності ефективних механізмів автоматичного прийняття рішень можливо буде збільшити долю пристроїв, які самостійно здійснюють функції керування певними процесами.

Принагідно також зауважити, що методи та інструменти, що використовуються в серверних рішеннях для інтернету речей, є поширеними та можуть застосовуватися і в інших сферах, які збирають і обробляють великі обсяги даних. Такими сферами є: аналіз соціальних мереж, аналіз поведінки користувачі веб-сайтів, аналіз стану інфраструктури дата-центрів, рекомендаційні системи, таргетування реклами, веб-пошук та інші.

1.2 Мета роботи. Функціональні та нефункціональні вимоги до системи

У даній роботі досліджується побудова серверних рішень, які забезпечують роботу Інтернету речей. Зокрема, розглядається побудова таких підсистем:

- **Прийняття та зберігання даних.** На швидкості надходження даних (тобто з мінімальними затримками) приймає та зберігає їх в умовно постійне сховище даних (черга повідомлень).
- **Обробка даних.** Виконує потокову (stream) обробку даних і аналітичні запити. Потокова обробка даних відбувається між умовно постійним

сховищем даних та базою даних (постійним (persistent) сховищем даних). Вироджений (no-op) випадок обробки даних, якщо вона не потрібна – просте переписування інформації з черги повідомлень у базу даних.

- **Автентифікація та безпека.** Забезпечує перевірку автентифікаційних даних відправників (пристроїв) та адміністраторів. Здійснює безпечну передачу інформації між компонентами системи.
- **Передавання адміністративних даних.** Виконує перенесення даних від пристрою до адміністратора (статусна інформація) та від адміністратора до пристрою (команди). При цьому дані повинні бути збережені в проміжній підсистемі, оскільки, наприклад, команди треба зберігати до їх запитання пристроєм, бо він може бути тимчасово недоступним для прямого надсилання даних до нього.
- **Моніторинг.** Виконує перевірку статусів пристроїв та надісланої ними інформації на основі певних заданих правил.

Метою роботи є розробка хмарної платформи для Інтернету речей. Показником успіху проекту є робоча система, яка:

- вирішує основні задачі, що висуваються до подібних платформ (збереження, аналіз даних тощо), тобто реалізує вищевказані підсистеми, з урахуванням загальноприйнятих підходів до розробки розподілених систем обробки великих даних [3-7];
- розроблена з урахуванням кращих практик і підходів до архітектури програмного забезпечення [8-13], що дають можливість забезпечити функціональні та нефункціональні вимоги до системи;
- має специфічну функціональність, якої не мають інші системи (наприклад, надійна доставка повідомлень між пристроями й адміністраторами та засоби моніторингу пристроїв);
- містить засоби для розгортання на хмарних хостингах, а також інструменти керування групою серверів.

З урахуванням зазначений цілей, опишемо прецеденти (use case) системи, що розробляється. Наочно зобразімо їх на діаграмі (рис. 1.1).

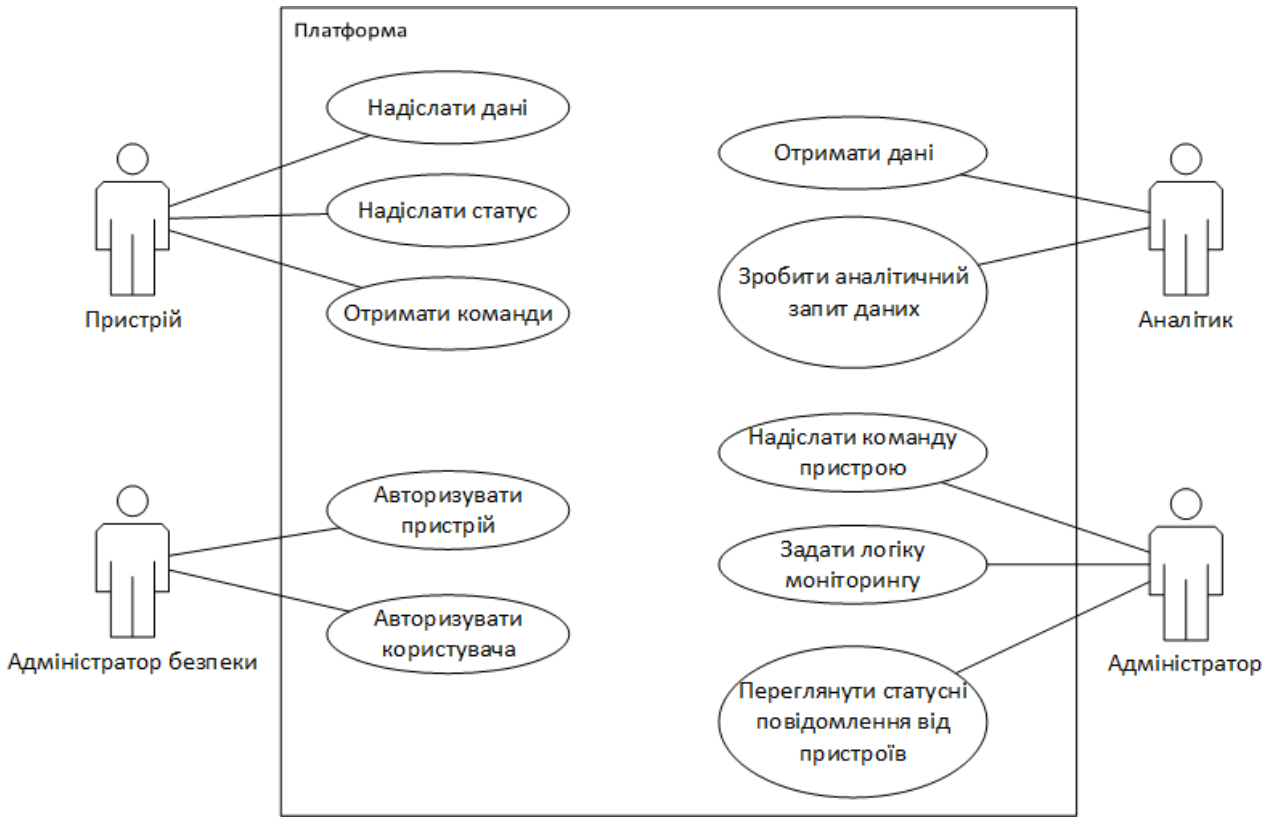

Рисунок 1.1 – Діаграма прецедентів, що відображає взаємодію із загальною функціональністю платформи

Виділено чотири основні групи користувачів системи. Маємо пристрій (або сенсор), який надсилає дані до системи, що обробляються та зберігаються; надсилає статусні повідомлення, які потім отримують адміністратори; приймає відправлені адміністраторами команди. Адміністратори також задають логіку моніторингу.

Адміністратор безпеки вводить автентифікаційні дані для пристроїв та інших користувачів системи. Для адміністраторів і аналітиків задаються їх права доступу, що показують, якими саме пристроями вони можуть керувати чи від яких отримувати дані.

Аналітики можуть запитувати в системи готові дані, що вже зберігаються та пройшли первинну обробку, або сформувати аналітичний запит, який вимагає виконання певних обчислень із наявними даними.

Також у роботі описано можливий варіант використання системи на прикладі мережі метеорологічних станцій; досліджено, які компоненти системи

є специфічними для даного сценарію використання, а які є загальними та можуть застосовуватися до мереж будь-яких пристроїв.

Деталізуємо обсяг роботи. Типова архітектура систем у сфері Інтернету речей зображена на рис. 1.2.

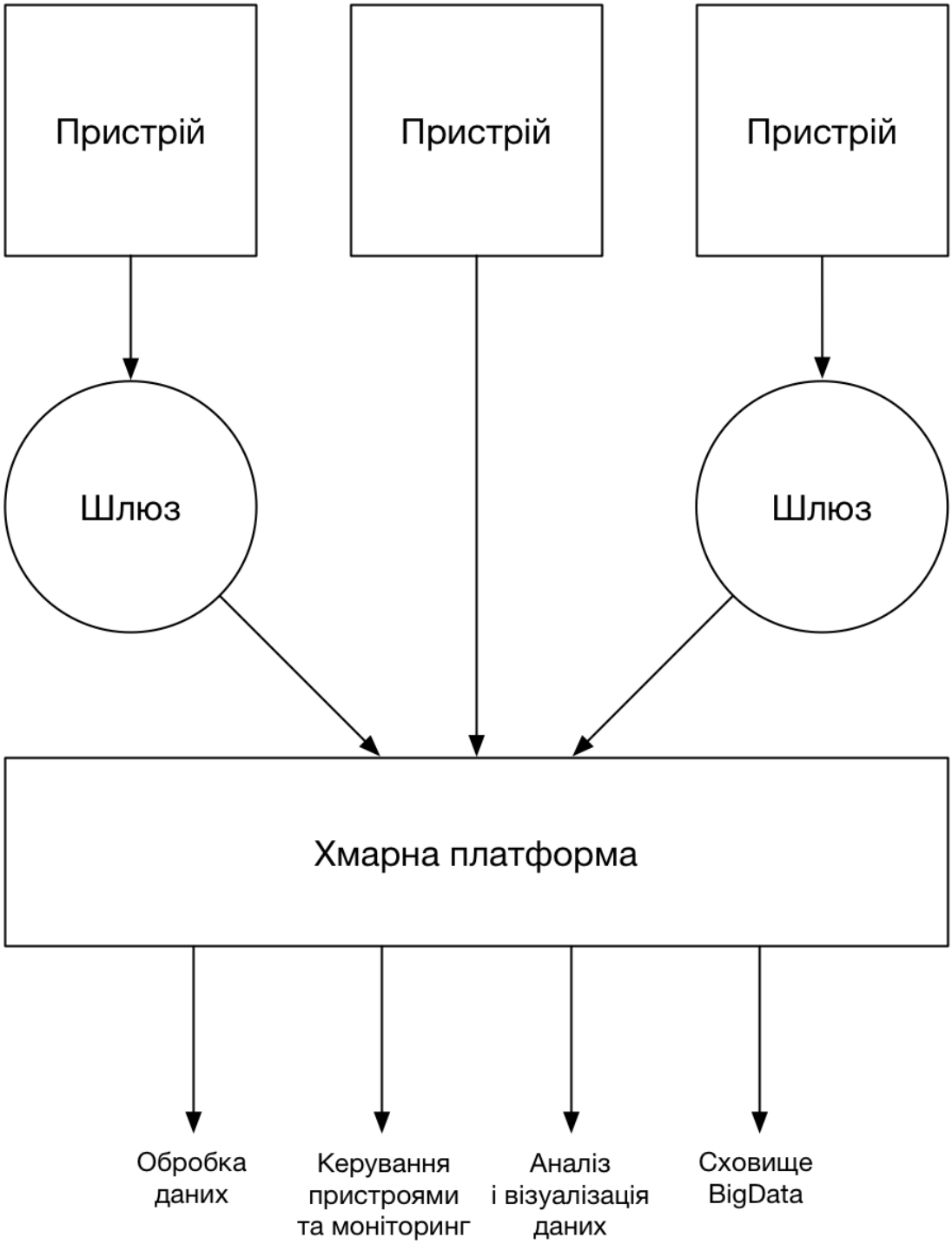

Рисунок 1.2 – Типова архітектура систем інтернету речей

Тобто бачимо декілька пристроїв на різноманітних платформах, які певним чином надсилають дані до серверу. Частина пристроїв мають вбудовані плати,

які дозволяють надсилати дані безпосередньо до сервера. Наприклад, пристрій може мати дротове підключення до Інтернету через Ethernet, або доступ до точки бездротового доступу (Wi-Fi), які забезпечують веб-доступ. Часто, якщо на певній території чи в будівлі на невеликій відстані розташовано декілька пристроїв, то безпосередньо вони з'єднуються зі шлюзом (кількість шлюзів набагато менша за кількість сенсорів), який, в свою чергу, забезпечує передачу даних до сервера через Інтернет. Це робиться з різних причин:

- **економічні:** наявність складного мережевого обладнання збільшує вартість пристроїв;
- **споживання енергії:** спеціалізовані протоколи передачі даних, такі як Bluetooth Low Energy, забепечують більш ефективне використання енергії, що особливо важливо, якщо пристрої працюють на автономному живленні;
- **програмне забезпечення:** пристрої можуть передавати дані до шлюзу в довільному бінарному форматі, який є простим у розробці та мінімізує витрати процесорного часу. Шлюз, в свою чергу, трансформує дані в певний загальноприйнятий (скажімо, JSON чи Protobuf) формат і передає їх серверу. Також мережу від пристроїв до шлюзу можна вважати безпечною та передавати відкриті дані, а на шлюзі їх шифрувати та віддавати в мережу по захищеному протоколу (такому як SSL).

Отже, після того як дані надходять до сервера, вони проходять певну попередню обробку та зберігаються. Потім ці дані доступні для подальшого аналізу.

У даній роботі розглядається саме побудова серверної платформи. Специфічні питання розробки програмного забезпечення для пристроїв і шлюзів (firmware) великою мірою знаходяться поза межами цього проекту: описуються лише базові питання реалізації клієнтської частини платформи.

Важливим також є сформулювати **нефункціональні вимоги** до платформи:

- **Масштабованість.** Збільшення обчислювальних ресурсів має відповідно підвищувати пропускну здатність системи (залежність має бути якомога ближчою до лінійної). До того ж, аби забезпечити вимоги Big Data систем та для хмарного розгортання, масштабованість має бути горизонтальною, тобто такою, яка дає можливість нарощувати продуктивність системи за допомогою збільшення кількості вузлів у хмарному дата-центрі. Горизонтальне масштабування є також набагато більш економічно ефективним за вертикальне.
- **Відмовостійкість.** Потрібно забезпечити таку архітектуру системи, яка має резервні компоненти (реплікація), а також здатна реагувати на відмови та перевантаження в процесі роботи без повної зупинки сервісу. Особливо важливою відмовостійкість є для тих частин системи, які забезпечують прийом і зберігання даних. Під час недоступності серверу пристрої просто нездатні довгий час зберігати свої дані, аби потім передати їх серверу, через обмеженість обсягу їхньої пам’яті.
- **Продуктивність.** Реалізація системи має бути максимально ефективною, аби забезпечити пропускну здатність на рівні сотень тисяч запитів на секунду і більше для підсистеми прийому та обробки інформації. Також необхідно обробляти аналітичні запити за досить короткий час (не довше декількох хвилин).

1.3 Існуючі рішення і шляхи їх покращення

Наразі існує декілька платформ, що працюють в цій галузі. Найбільш відомі з них: Amazon Web Services IoT Platform, Google Cloud IoT Platform, ThingWorx, Oracle IoT, IBM IoT Platform.

Здебільшого існуючі рішення пропонують загальний набір функціональності, який включає зберігання даних, а також досить широкі можливості з їх відображення і аналізу.

Усі перераховані платформи є комерційними із закритим вихідним кодом, що є їх значним недоліком. Така закритість ускладнює їх розширення, адаптацію й оптимізацію під конкретний сценарій використання. Підприємства, які використовують платформи, що працюють за такою моделлю, часто прив'язані до постачальника (vendor lock-in) і не можуть переорієнтуватися на застосування іншої платформи без значних витрат часу та коштів.

Платформ з відкритим вихідним кодом майже немає. Проте цілком можливо розробити функціональну платформу, що використовуватиме open-source компоненти та бібліотеки, оскільки наразі доступно багато рішень із відкритою ліцензією, які можуть стати частинами системи і які вже є стабільними та мають широкі можливості. В даній роботі досліджено, порівняно та використано відповідні компоненти.

Існуючі платформи також лише іноді мають готові рішення по очищенню помилкових даних. Проте загалом подібні платформі можуть пропонувати базові інструменти зі статистики чи машинного навчання, які придатні майже для всіх сценаріїв використання. Однак, варто зауважити, що очищення даних є великою і складною частиною науки про дані (data science). Особливо перспективною галуззю є виявлення аномалій (anomaly detection). Online anomaly detection наразі є однією з найменш досліджених розділів машинного навчання.

У роботі також розглядається організація моніторингу та передачі адміністративних даних: команд від адміністратора до пристрою та статусної інформації від пристрою (чи серверу моніторингу) до адміністратора. Ця функціональність є дуже важливою для ефективного керування великими групами пристроїв, в той час як наявні платформи майже не реалізують подібні можливості.

На відміну від існуючих платформ розроблене рішення прагне до універсальності: замість використання специфічних протоколів передачі даних та надання SDK для пристроїв від кожного окремого постачальника, воно використовує загальноприйняті у Web протоколи, для яких існує багато

бібліотек, що прискорює та здешевлює розробку клієнтського ПЗ (тобто того, яке працює на пристроях і передає дані в мережу).

Також універсальність розробленого рішення полягає у наданні можливості спеціалізації платформи під конкретний сценарій. Наприклад, для всіх типів пристроїв необхідно зберігати дані. Але для різних галузей використання пристроїв обробка цих даних буде відрізнятися, тому доцільно надавати можливість розширювати платформу, задаючи певні ланцюжки операцій обробки даних під конкретні потреби.

Розглянемо для прикладу одну з найбільш функціональних і популярних із існуючих платформ – **AWS IoT Platform** [14]. Її архітектуру зображено на рис. 1.3.

На діаграмі показано основні компоненти системи. Розглянемо їх детальніше:

- SDK для пристрою, за допомогою якого надсилаються повідомлення.
- Вузол автентифікації та авторизації, який використовує SigV4 або сертифікати X.509.
- Шлюз пристроїв, який забезпечує комунікацію за принципом pub/sub по протоколах MQTT, WebSocket, HTTP 1.1.
- Регістр для зберігання інформації про пристрої.
- «Тіні» пристроїв – механізм збереження останнього отриманого стану пристрою для його подальшого запиту через REST API.
- Механізм задання правил, який дозволяє описати інтеграцію з іншими сервісами AWS.

Як бачимо, AWS IoT пропонує вельми довершені інструменти безпеки, але є повністю зав’язаною на сервіси AWS, використовує непоширений протокол MQTT та не надає API для визначення власних ланцюжків обробки даних.

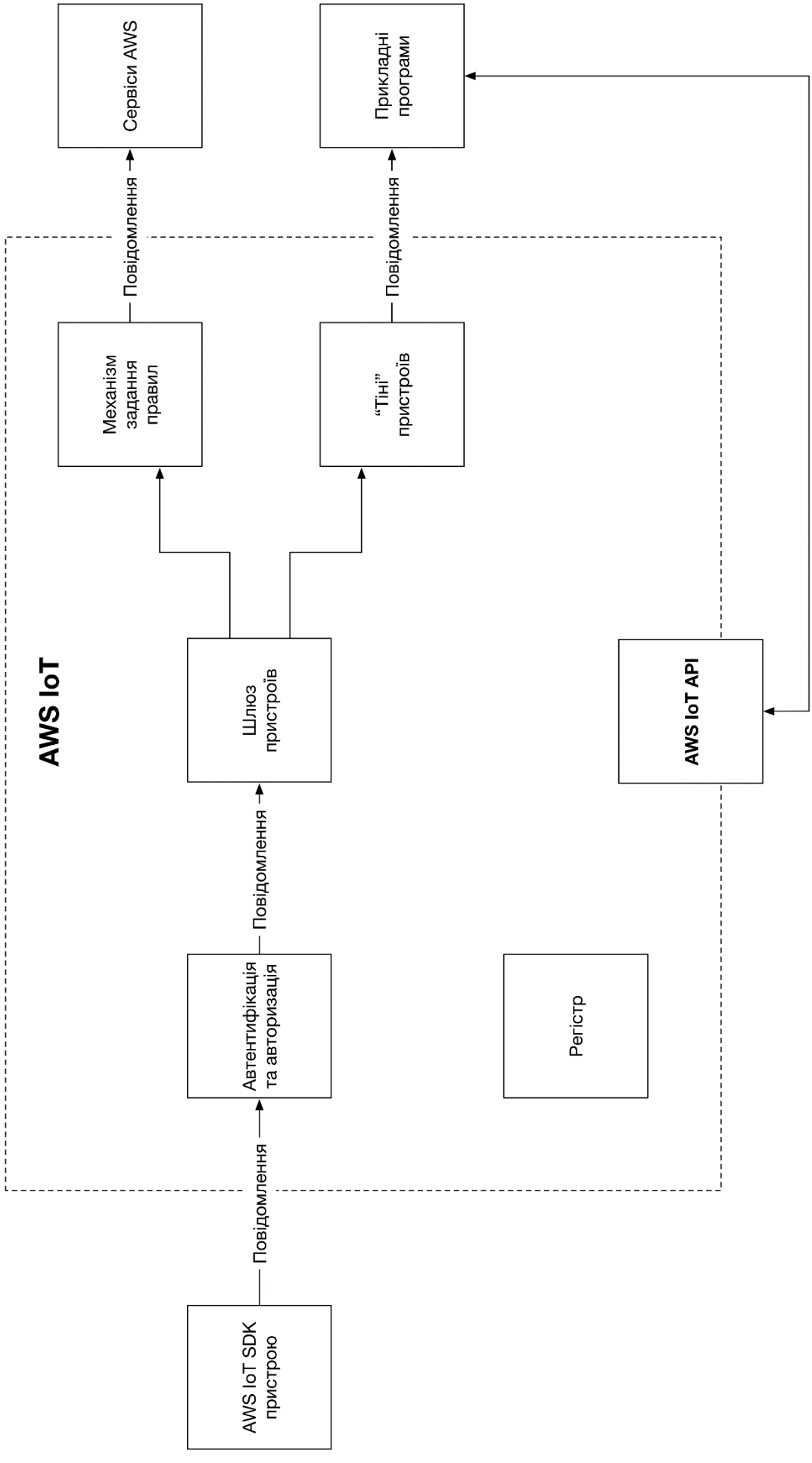

Рисунок 1.3 – Архітектура AWS IoT Platform

Зробимо огляд іншої платформи для Інтернету речей – **IBM IoT Platform**. Розробники розділили її функціональність на чотири основних блоки [15]:

1. **Підключення.** Платформа пропонує різноманітні SDK для пристроїв і шлюзів, а також API для взаємодії із клієнтським програмним забезпеченням. Ці можливості доступні для різних мов програмування, включно з оптимізованими низькорівневими рішеннями на embedded C. Цей блок функціональності призначений для безпечного підключення пристроїв до платформи для передачі даних. Основними протоколами передачі є MQTT і HTTPS.
2. **Керування ризиками.** Ця група функцій відповідає за керування великими групами пристроїв, дозволяючи користувачам переглядати інформацію про їхнє функціонування на панелях управління. Також наявні повідомлення адміністраторів про нештатні ситуації, які дозволяють оперативно ізолювати різноманітні інциденти.
3. **Керування інформацією.** Блок функціональності, що дозволяє управляти збереженням і архівуванням операційної інформації та метаданих. Наявні можливості по парсингу та трансформації даних, генерування звітів. Також можливо вводити дані з різноманітних джерел і платформ.
4. **Аналітика.** Інтеграція з IBM Watson дозволяє користувачам виконувати складний аналіз даних, застосовуючи наявні підсистеми Watson. Наприклад, блок аналізу природньої мови, різноманітні алгоритми машинного навчання, інструменти аналітики відео та зображень, аналіз текстуальних даних.

Отже, основною перевагою IBM IoT Platform є широкі можливості по аналізу даних, які реалізуються системою IBM Watson. Інша функціональність є досить типовою для подібних платформ. Основним недоліком є закритість платформи, а відповідно, неможливість її розширення.

Ще однією функціональною та вартою огляду платформою є **Oracle IoT**. Розглянемо детальніше її можливості, які умовно поділяються на три частини [16]:

1. **Підключення.** Цей блок має таку функціональність:
    - *Віртуалізація пристроїв.* Кожен пристрій представляється як набір ресурсів, аби абстрагувати пов'язані з підключенням пристрою складності та стандартизувати інтеграцію.
    - *Надійна комунікація.* Наявна надійна, двонапрямлена комунікація, яка гарантує доставку повідомлень через ненадійні мережі та до/від пристроїв, які працюють і підключаються до мережі лише час від часу.
    - *Гнучка топологія.* Пристрої можуть підключатися до платформи, використовуючи різні мережеві топології, такі як клієнтська бібліотека з програмним забезпеченням шлюзу, або безпосередньо через REST API. Доступні бібліотеки пристроїв на Java SE, Java ME, POSIX C. Для шлюзів наявна підтримка таких не-IP протоколів, як Bluetooth, Z-Wave, Modbus і OPC.
2. **Аналіз.** Надає такі можливості:
    - *Потокова обробка.* Аналіз потоків даних у режимі реального часу з агрегацією подій, фільтрацією та кореляцією.
    - *Збагачення даних.* Можливість збагачувати необроблені потоки даних контекстуальною інформацією, такою як метадані пристроїв та створювати композитні потоки даних.
    - *Сховище подій.* Проаналізовані потоки даних можуть надсилатися до проінтегрованих сховищ та сервісів, що є частиною Oracle Cloud.
3. **Інтеграція.** Має такі функції:
    - *Підключення підприємств.* Дані IoT можуть бути спрямовані до іншого програмного забезпечення, що керує процесами та з'єднано з Oracle Cloud.

- *REST API.* Дозволяє будувати інтеграцію з іншими системи, як в Oracle Cloud, так і поза ним.
- *Команди пристроям та їх контроль.* Надає можливість надсилати повідомлення пристроям, незалежно від наявності в них у цей момент з’єднання з мережею.

Як бачимо, перевагою Oracle IoT є широкі можливості по передачі даних та керування пристроями, а також підтримкою різних протоколів (особливо для шлюзів). Проте, для використання більшості можливостей сервіси користувача системи мають застосовувати Oracle Cloud.

2. ТЕОРЕТИЧНИЙ ОПИС ПОБУДОВИ ПЛАТФОРМИ ДЛЯ ІНТЕРНЕТУ РЕЧЕЙ

2.1 Архітектура системи

Опишемо основні компоненти системи, потоки даних між ними, а також її входи та виходи. Зобразимо це на діаграмі (рис. 2.1). На діаграмі також одразу ж показано деякі сторонні системи та бібліотеки (наприклад, бази даних), які будуть використовуватися в тому чи іншому компоненті. Їхній вибір буде обґрунтовано в подальших розділах.

На вході системи (блок «Джерела даних») маємо різноманітні пристрої, а саме температурний сенсор, сенсор тиску, сенсор якості повітря чи будь-який інший. Вони надсилають дані до пулу фронтальних серверів (блок «Фронтальні сервери»). Разом із даними пристрої передають автентифікаційну інформацію та інформацію про себе (наприклад, ідентифікатор пристрою, за яким можна визначити його тип).

Фронтальні сервери виконують перевірку автентифікаційної інформації. Якщо вона неправильна (тобто відправник невідомий), дані відхиляються. Відповідно до даних про пристрій, на фронтальних серверах виконується диспетчеризація даних: визначається тип повідомлення та надсилається у відповідну тему (topic) у черзі повідомлень (блок «Apache Kafka»). Також фронтальні сервери перевіряють цілісність даних (виконують валідацію).

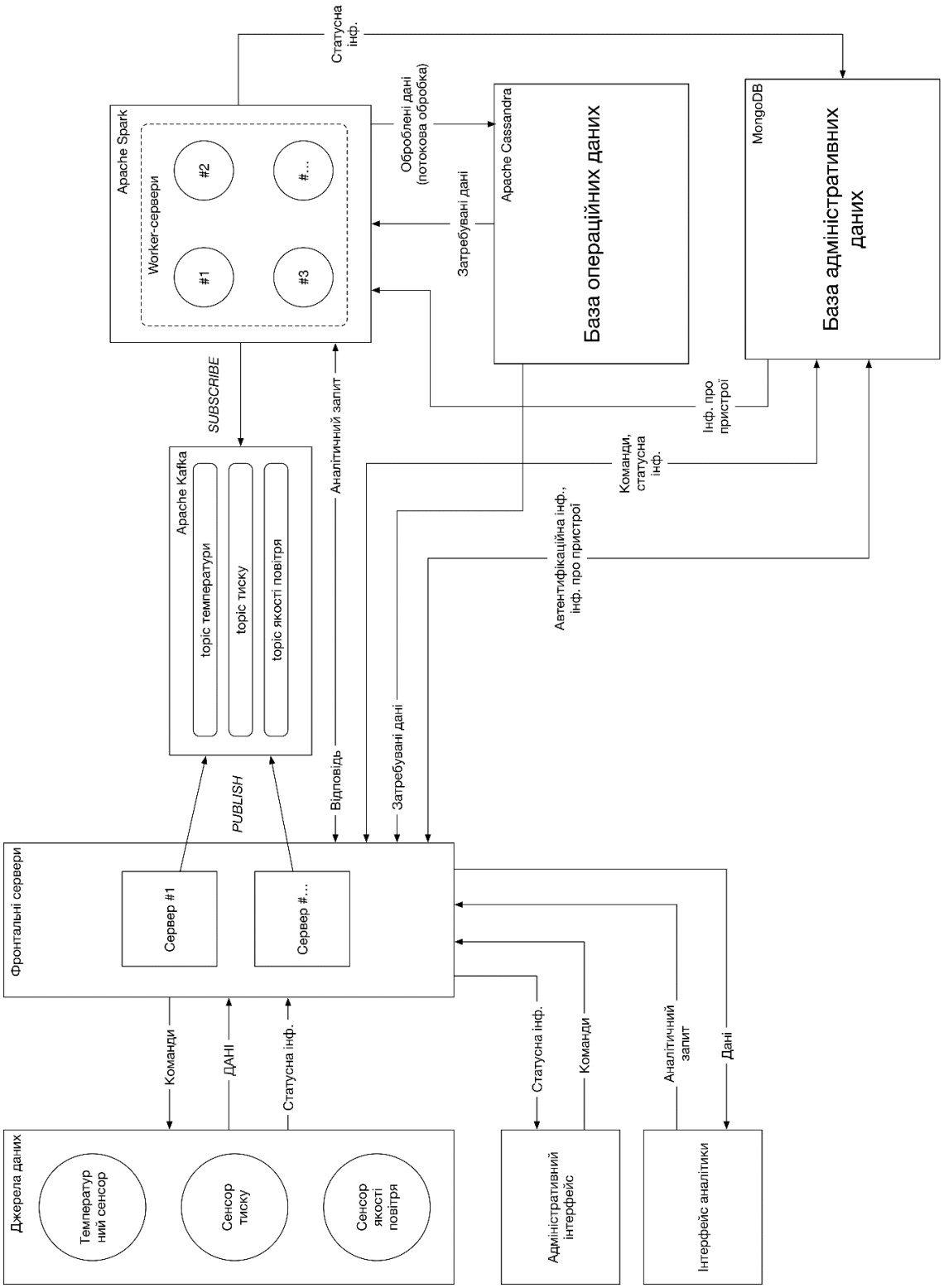

Рисунок 2.1 – Запропонована архітектура платформи для Інтернету речей

Черга повідомлень реалізує шаблон проектування publish-subscribe. Фронтальні сервери публікують (publish) дані у Apache Kafka.

На боці subscribe розташовано систему обробки даних – Apache Spark. Вона виконує первинну обробку даних, які згодом зберігає у базі операційних даних. Обробка даних на цьому етапі може бути суто потоковою або мікропакетною (micro-batch). Мікропакетна обробка може емулювати потокову обробку, тим самим дозволяючи застосовувати інструменти і логіку пакетної обробки. Також мікропакетна обробка природньо відповідає логіці агрегування та віконної (windowing) обробки, коли, наприклад, треба зібрати певну кількість показань за хвилину, знайти їх середнє значення та зберегти його у базі. Також можна застосовувати фільтрацію (або детекцію аномалій) на рівні мікропакету.

База даних, в якій містяться операційні дані, забезпечує постійне зберігання оброблених даних. З точки зору продуктивності, це сховище повинне бути оптимізоване під запис, бо операцій зчитування, очевидно, набагато менше. Проте черга повідомлень слугує буфером на випадок, якщо база операційних даних не здатна (наприклад, під час скачка навантаження) зберігати дані зі швидкістю їх надходження. Принагідно зауважити, що від цієї бази не вимагаються такі значні гарантії, які забезпечують традиційні транзакційні БД.

Наразі буде описано послідовність **проходження інформації від пристроїв до постійного сховища**. Наочно потоки даних в цьому сценарії використання показано на діаграмі (рис. 2.2).

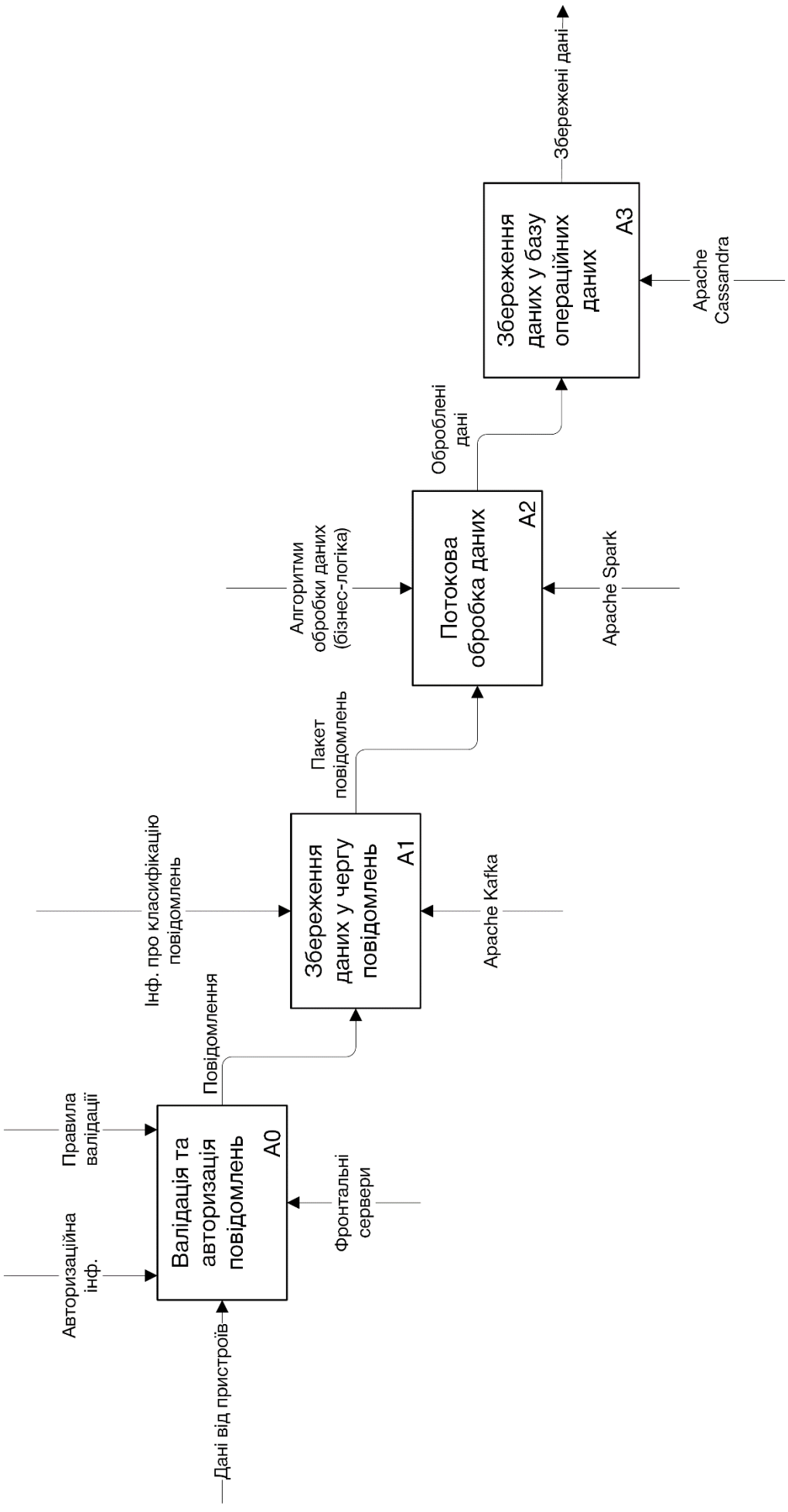

Рисунок 2.2 – Діаграма потоків даних і процесів при отриманні даних від сенсорів

Опишемо **передачу інформації від пристрою до адміністратора**. Такою інформацією може бути певна статусна інформація, скажімо, стан акумуляторів пристрою чи певні попередження про неполадки.

Статусна інформація надходить від пристроїв до фронтальних серверів, що, як і при отриманні операційних даних, валідують її та виконують автентифікацію адресанта запиту. Після цього дані надсилаються до бази адміністративних даних.

В цьому випадку зазвичай потрібні значні гарантії постійного зберігання та узгодженості даних. Тому після того, як фронтальний сервер передасть дані до БД, він очікує підтвердження їх отримання та зберігання перед тим, як надсилати відповідне підтвердження пристрою, який може реалізовувати певну логіку повторної спроби в разі, якщо повідомлення є важливим.

Після збереження цієї інформації адміністратор надсилає запит до фронтальних серверів, які перевіряють його права доступу до цих даних та роблять запит до БД відповідно предикату запиту (наприклад, статусні повідомлення пристрою зі вказаним ідентифікатором).

Потоки даних для цього сценарію використання продемонстровано на діаграмі (рис. 2.3).

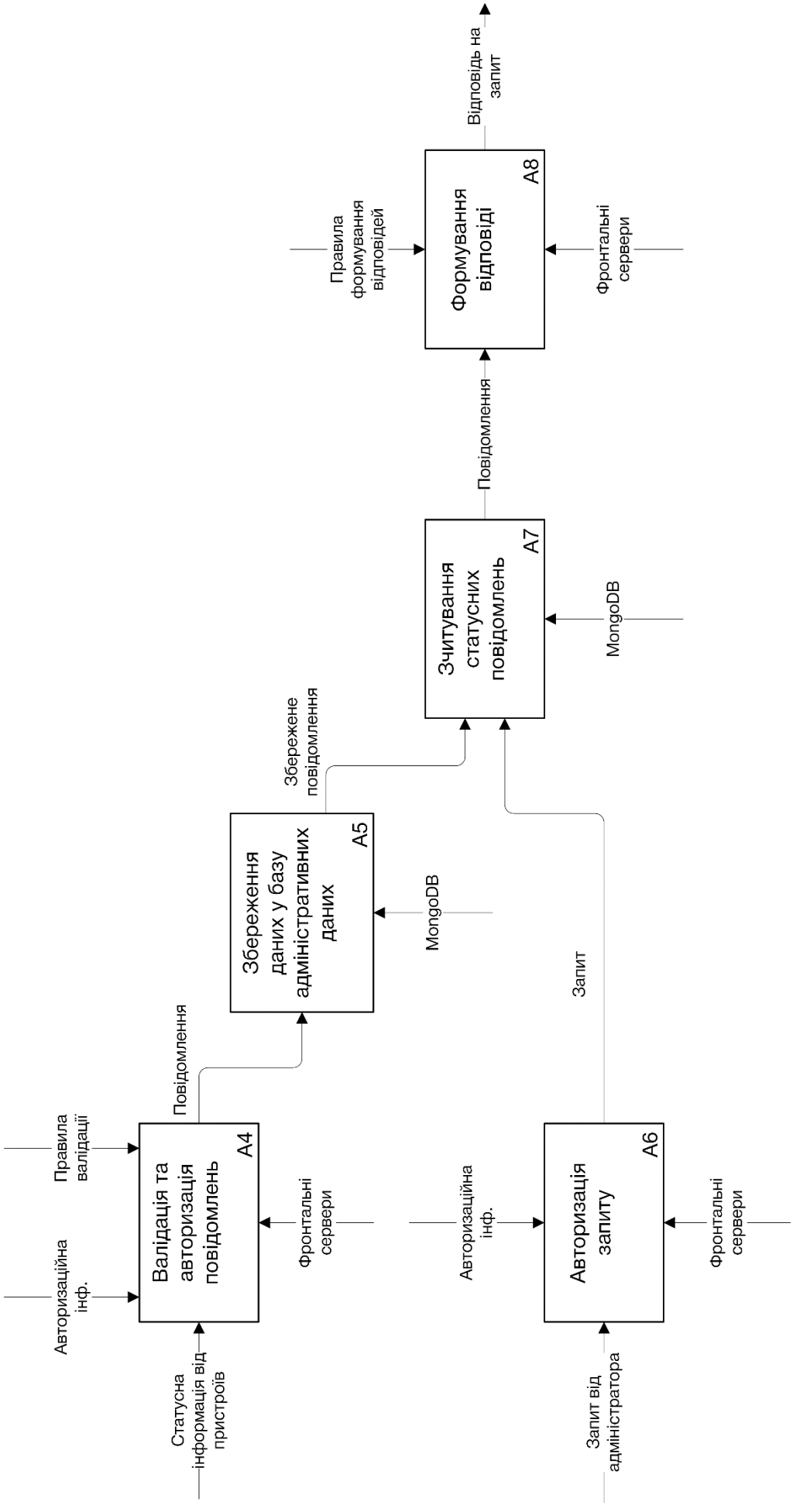

Рисунок 2.3 – Діаграма потоків даних і процесів при передачі статусної інформації від пристрою до адміністратора

Іще одним **джерелом статусної інформації є механізми моніторингу**. Адміністратори у вигляді правил задають певні умови, які мають виконуватися при очікуваній роботі пристрою, та визначають порядок їх перевірки.

Наприклад, адміністратор може визначити правило, що певний пристрій має надсилати не менше деякої кількості повідомлень за одиницю часу. У разі, якщо ця умова не виконується, тобто спрацьовує певний тригер, платформа надсилає статусне повідомлення, аби сповістити адміністратора про можливу неполадку.

Подібні умови можуть перевірятися в різних частинах системи. Фронтальні сервери можуть контролювати статус з’єднання з пристроями. Підсистема обробки даних може періодично (за розкладом) робити запит до сховища операційних даних задля того, аби виявити потенційно некоректні дані, спричинені неправильною роботою пристрою.

Опишемо також **передачу команд від адміністратора до пристрою**. Команда від адміністратора надходить до фронтального сервера, який надійно зберігає її в базі адміністративних даних.

Пристрої періодично опитують фронтальні сервери на предмет появи нових команд. Потоки даних показано на рис. 2.4.

Така архітектура потрібна для того, щоби надійно доправляти дані до пристроїв, бо вони можуть бути тимчасово недоступні (відключені або не мати з’єднання з сервером).

Узагалі, зменшити затримку передачі даних і зекономити ресурси можна було б, доправляючи дані безпосередньо (в разі можливості). Скажімо, якщо фронтальний сервер має відкрите з’єднання з пристроєм (скажімо, по WebSocket), то при надходженні команди від адміністратора можна відправляти її відразу. Але це б зменшило гарантії доставки. Якщо команду записано в базу, то пристрій може отримати її, виконати, і лише після цього направити підтвердження фронтальному серверу, який відмітить команду як прийняту чи видалить її. У разі прямого доправлення команди пристрою, він може підтвердити її отримання, але якщо під час виконання станеться збій, пристрій

не зможе повторно запитати цю команду у фронтального сервера, бо її не буде збережено. Якщо ж не підтверджувати отримання команди до її виконання, то адміністратору доведеться довго очікувати завершення свого запиту, що погіршило б його сприйняття інтерфейсу порівняно з асинхронною схемою, коли він отримує підтвердження відразу після того, як його команда збережена в базі.

Здійснімо тепер опис дій, які відбуваються при отриманні аналітичних запитів. Фронтальні сервери автентифікують запит і визначають його тип. Запит може бути задовільнений наявними даними або вимагати додаткових обчислень.

Наявні дані – це ті, які пройшли первинну обробку та зберігаються в базі операційних даних. При цьому типі запиту достатньо зчитати відповідні дані та надіслати їх.

Якщо ж отримано аналітичний запит, то потрібно сформувати план його виконання, зчитати дані з однієї або з обох БД, а потім обробити їх. Якщо потрібна лише додаткова обробка наявних даних, то на вхід системи обробки подається лише інформація з бази операційних даних. База адміністративних даних може використовуватися, якщо запит вимагає зчитування певної інформації про пристрої. Скажімо, аналітику можуть знадобитися агреговані дані від сенсорів в певній місцевості. У цьому випадку потрібно спочатку пошукати в базі адміністративних даних ідентифікатори пристроїв у досліджуваній місцевості, а потім зчитати операційні дані від відповідних сенсорів за отриманими ідентифікаторами.

З урахуванням вищеописаного, потоки даних у сценарії виконання аналітичних запитів виглядають так, як показано на діаграмі (рис. 2.5).

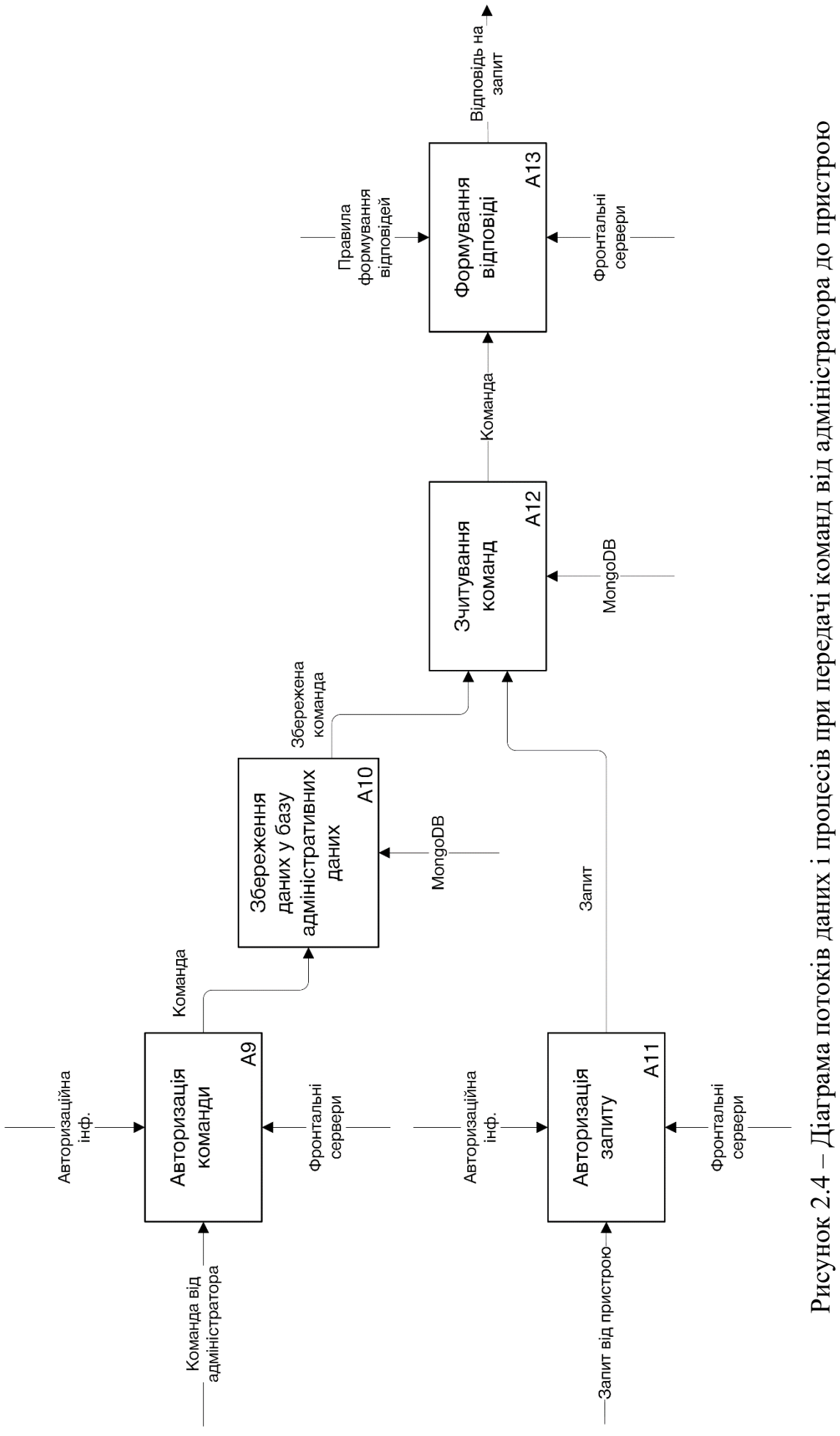

Рисунок 2.4 – Діаграма потоків даних і процесів при передачі команд від адміністратора до пристрою

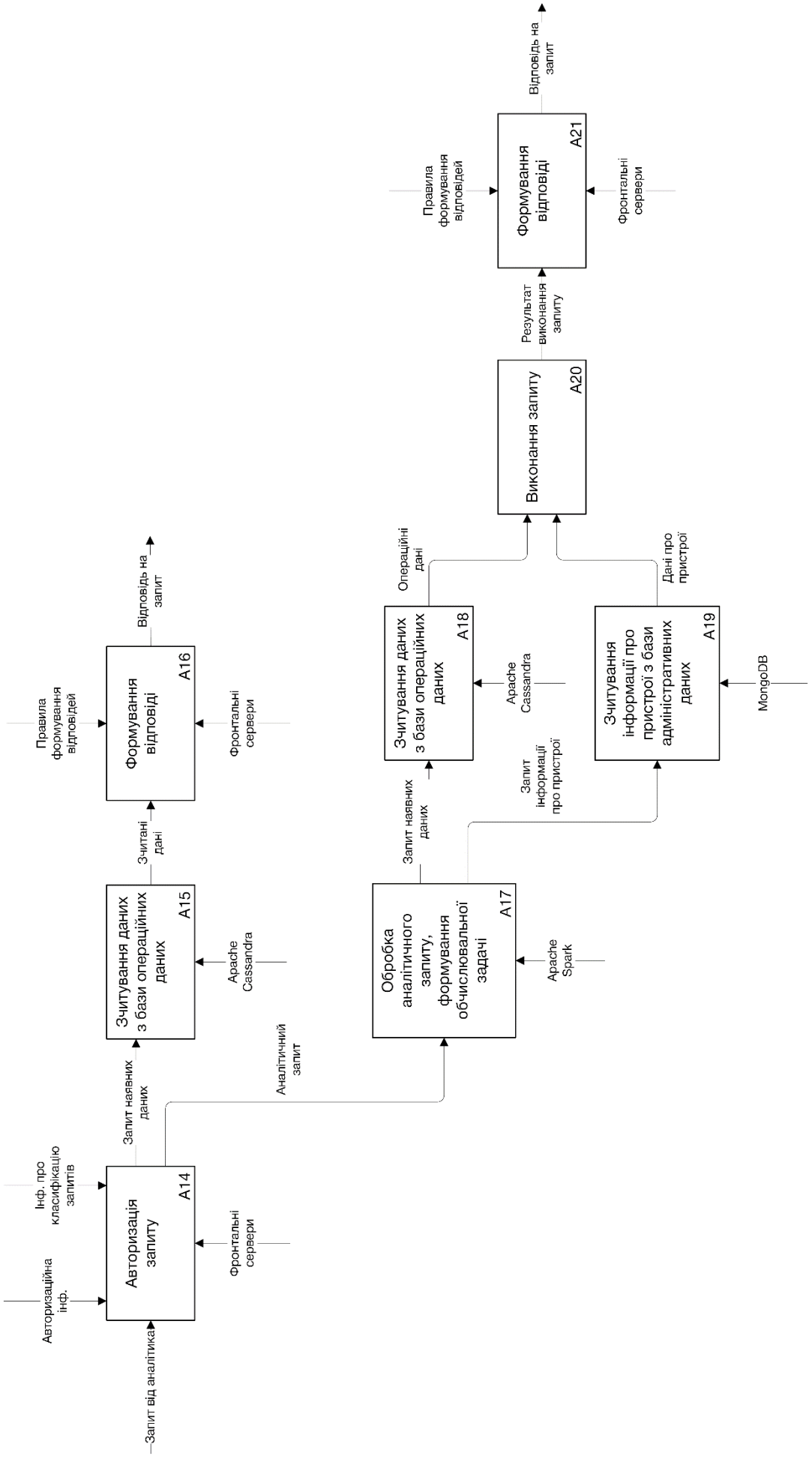

Рисунок 2.5 – Діатрама потоків даних і процесів при обробці запитів від аналітика

2.2 Протоколи і формати обміну даними між компонентами системи

Опишемо протоколи передачі даних між компонентами системи.

Зв’язок між пристроями (або шлюзами) і платформою відбувається по зашифрованому протоколу HTTP/2, який використовує криптографічний протокол TLS 1.2. Шифрування трафіку потрібне, аби уникнути перехоплення даних та інших атак. Протокол HTTP/2, стандартизований [17] у 2015 році, дуже добре підходить для IoT, бо:

- Розроблений як оптимізація HTTP/1.1.
- Реалізує стискання HTTP-заголовків (HPACK) за допомогою коду Хаффмана. Цей метод має високу ефективність, але потребує мало пам’яті, що дуже важливо для малопотужних пристроїв.
- За допомогою таких технік як pipelining та мультиплексування заохочує використання одного TCP-з’єднання, що дозволяє уникати повільного тристороннього рукостискання (three-way handshake), яке, до того ж, потребує додаткових ресурсів. У зашифрованому протоколі перевикористання з’єднання дозволяє уникнути ще більш довгого TLS діалогу (negotiation).
- Передбачає операцію PING на рівні протоколу, що дозволяє використовувати її для перевірки працездатності з’єднання з пристроєм.
- Підтримує високорівневу сумісність із HTTP/1.1: методи, коди стану, URI, поля заголовків.

Багато платформ Інтернету речей використовують MQTT, який є набагато ефективнішим за HTTP/1.1. Проте із появою HTTP/2 за продуктивністю HTTP/2 і MQTT майже зрівнялися.

Головною перевагою використання HTTP є його поширеність і загальноприйнятість у Web, що означає наявність функціональних і стабільних серверних і клієнтських бібліотек.

HTTP/2 реалізує техніку server push, яка дозволяє серверу ініціювати відправку даних клієнту замість традиційної моделі «запит-відповідь». Проте більш природнім і ефективним для повнодуплексного зв’язку є застосування протоколу WebSocket [18]. Пристрої, якщо вони мають можливість установити з’єднання з сервером, підтримують відкритий канал, по якому можуть отримувати команди адміністраторів від сервера за протоколом WebSocket. Використовується захищений варіант протоколу WebSocket – WebSocket Secure (WSS).

У якості основного формату даних, у якому фронтальні сервери отримують інформацію, використовується JSON. Він, як і HTTP, є дуже поширеним; може читатися людиною (тобто не є бінарним); є простим для формування та парсингу. Недоліком порівняно з бінарними форматами є розмір повідомлень, але його можна усунути за рахунок застосування стандартних методів стискання, таких як gzip.

Загалом, на фронтальних серверах реалізується REST API, який використовується пристроями, адміністраторами та аналітиками.

Як уже було зазначено, зв’язок між пристроями та точкою входу в хмарну інфраструктуру платформи відбувається з використанням зашифрованого з’єднання. Дешифрування (SSL termination) відбувається на рівні балансувальника навантаження, і фронтальні сервери отримують уже відкриті дані. Це можна зробити тому, що з’єднання всередині дата-центру можна вважати безпечним: трафік або ізолюється від інших клієнтів хостингу за допомогою інструментів віртуалізації, або інфраструктура платформи може розташовуватися в повністю фізично ізольованому сегменті мережі. До того ж, це дозволяє виконувати дешифрування за допомогою спеціально пристосованих більш продуктивних рішень, а на фронтальних серверах витрачати ресурси лише на основну логіку обробки запитів.

Передача даних між чергою повідомлень, підсистемою обробки даних, базою операційних даних, базою адміністративних даних і фронтальними серверами відбувається у певних бінарних форматах через TCP, які реалізуються

відповідними бібліотеками. Наприклад, підсистема обробки даних може обмінюватися в процесі обчислень між своїми робочими (worker) серверами серіалізованими об'єктами.

Наочно формати даних і протоколи їх передачі між різними компонентами системи показано на рис. 2.6.

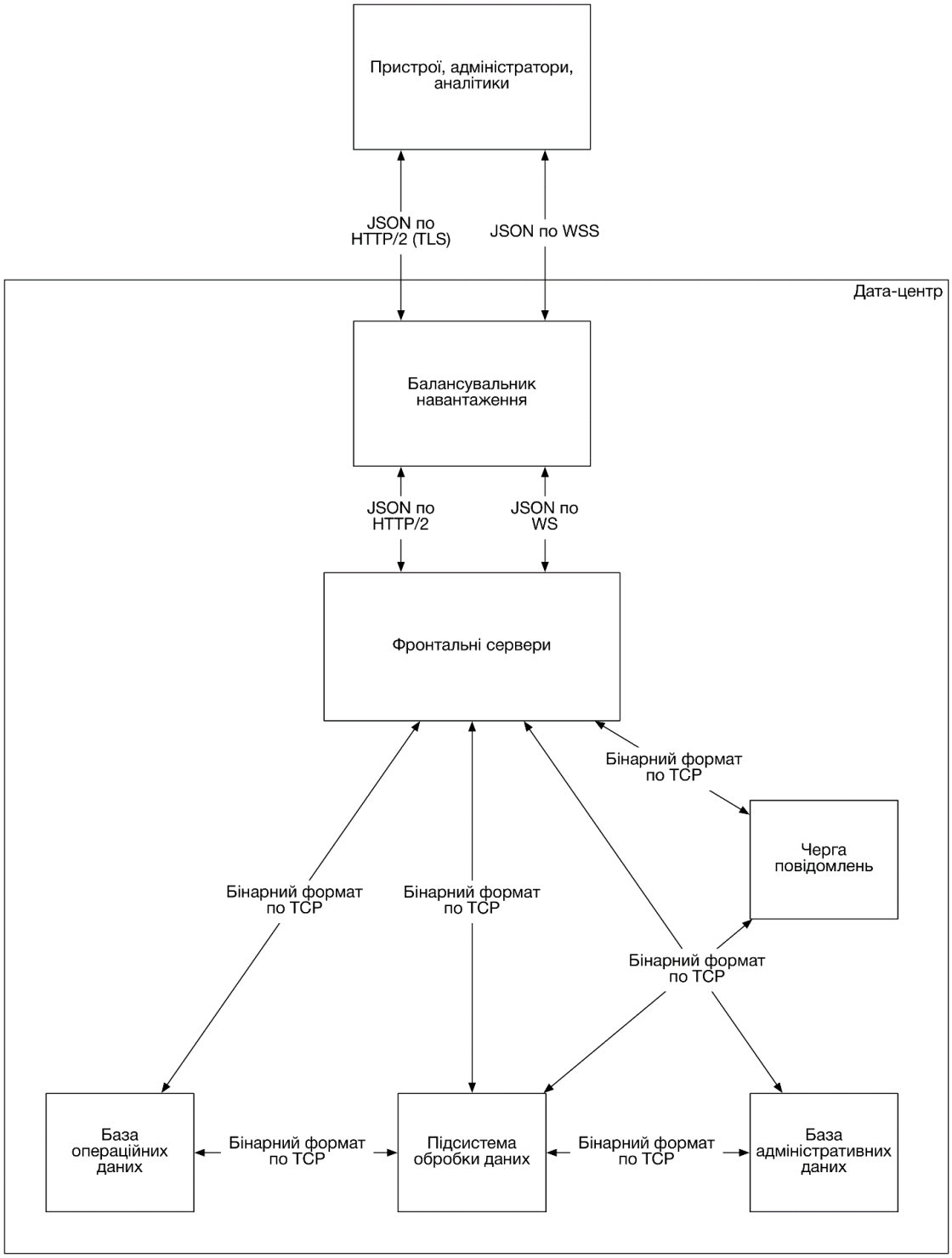

Рисунок 2.6 – Запропоновані формати даних і протоколи їх передачі між компонентами системи

2.3 Мікросервісна архітектура

Мікросервісний стиль архітектури [19] – підхід до розробки цілісної програми як набору маленьких сервісів, кожен із яких працює у власному процесі та з'єднуються з іншими за допомогою легких механізмів, таких як RPC або HTTP. Сервіси будуються відповідно до певної задачі та можуть незалежно розгортатись автоматизованими системами. Наявний деякий мінімум централізованого керування такими сервісами.

Традиційні серверні системи зазвичай будуються як *моноліти* – логічно окремі виконувані програми. І цей підхід є природнім: уся логіка обробки запитів працює в одному процесі, що дозволяє використовувати наявні інструменти мови програмування для поділу програми на класи, функції та простори імен. Моноліт можна масштабувати горизонтально, запустивши багато екземплярів поза балансувальником навантаження.

Монолітне ПЗ є досить успішним, але має свої недоліки. Цикли зміни моноліту зв'язані один з одним, тобто зміна невеликої частини системи вимагає повторного збирання (компіляції) і розгортання. З плином часу стає складно підтримувати гарну модульну структуру, утримуючи зміни, що стосуються конкретного модуля, лише всередині нього. Масштабувати доводиться весь моноліт замість окремих частин, які потребують більшої кількості ресурсів.

Мікросервіси ж можна розгортати та масштабувати незалежно один від одного. Вони також забезпечують чіткі межі між модулями, навіть дозволяючи реалізовувати окремі підсистеми на різних мовах програмування. Таке розмежування допомагає керувати складністю кодової бази, оскільки кожен модуль матиме публічний API, який міститиме лише потрібну функціональність, а все інше буде інкапсульовано та не мати значення для розробки інших сервісів, які залежать від цього.

Основним недоліком мікросервісної архітектури є збільшення складності обробки помилок. На відміну від моноліту, кожен виклик сервісу може

закінчитися невдачею. Тому потрібно розробляти механізми моніторингу стану сервісів, перевіряти різноманітні метрики їх функціонування, а також автоматизовувати відновлення мікросервісу в разі його відмови.

Позитивним моментом є те, що відмови в мікросервісній архітектурі є великою мірою ізольованими. Якщо задоволення запиту вимагає виклику багатьох сервісів і частина з них недоступна, можливо надати менш повну відповідь (graceful degradation).

Складності також можуть бути викликані забезпеченням послідовності розгортання залежних мікросервісів, а також необхідністю керування транзакціями, які взаємодіють із декількома підсистемами.

Як бачимо, запропонована у розділі 2.1 архітектура великою мірою ввібрала ідеї мікросервісної архітектури. Компоненти чітко розділені та мають свій виділений набір функціональності.

Фронтальні сервери забезпечують уніфіковану точку входу до системи. При надходженні запиту, відповідно до практик мікросервісної архітектури, фронтальні сервери роблять необхідну кількість викликів API сервісів, а потім об’єднують отримані результати для формування відповіді.

Як було описано в розділі 2.2, внутрішня комунікація між серверами відбувається у бінарному форматі по TCP з міркувань ефективності, але з точки зору коду, кожен сервіс надає публічний API, який інкапсулює створення повідомлень до відповідного сервісу.

Крім розмежування відповідальності головною перевагою є можливість горизонтального масштабування кожного сервісу окремо. Наприклад, якщо надходження операційних даних значно пришвидшується без збільшення потоку адміністративних даних, можна масштабувати відповідну СКБД окремо.

2.4 Підсистема прийому операційних даних. Організація, вимоги та обґрунтування вибору стороннього рішення

Оскільки дані надходять на високій швидкості та можуть відбуватися сплески трафіку, потрібно мати буфер між базою даних і фронтальними серверами для балансування навантаження. Для цього архітектура платформи передбачає використання черги повідомлень.

Черги повідомлень задають асинхронний комунікаційний протокол. Це означає, що відправник і одержувач повідомлення не повинні взаємодіяти із чергою повідомлень одночасно. Повідомлення, які розміщуються в черзі, зберігаються до їх отримання приймачем.

Важливою рисою черги повідомлень є забезпечення стійкості та надійності, які реалізуються за допомогою різних стратегій зберігання даних. Для підвищення надійності доставки повідомлень є можливість зберігати їх на диску до їхнього отримання одержувачем. Навіть якщо програма-приймач або черга повідомлень припинить роботу через збій, повідомлення будуть в безпеці та будуть доступні одержувачам, щойно система буде знову працездатна.

Використовуючи чергу повідомлень, можна підтримувати набагато більший обсяг повідомлень. Загалом, застосування архітектури з чергою повідомлень – вдала стратегія організації асинхронної обробки великих даних.

Сформулюємо вимоги до черги повідомлень, які треба врахувати при виборі з наявних рішень:

1. **Висока продуктивність.** Потрібно забезпечувати високу швидкість запису повідомлень у чергу.
2. **Горизонтальна масштабованість.** Можливість підвищити пропускну здатність системи за допомогою збільшення кількості серверів.
3. **Можливість зберігати повідомлення на диску.** Потрібно, аби зменшити кількість втрачених даних при відключенні сервера з чергою повідомлень. Конче потрібним це є при читанні даних з черги крупними пакетами з

великими інтервалами – у цьому разі за відсутності постійного зберігання можна втратити велику кількість даних.

4. **Реплікація.** Для певного збільшення гарантій зберігання даних і збереження доступності черги повідомлень для одержувачів у разі відключення частини серверів.
5. **Здатність конфігурувати рівень забезпечення надійності зберігання повідомлень**. Тобто те, на скільки серверів відбувається реплікація та як часто повідомлення зберігаються на диску. Потрібно для того, аби налагодити систему так, щоби надійніше зберігати повідомлення від пристроїв, які мають великий період між операціями надсилання даних. При великому навантаженні повинна бути можливість навпаки послабити такі гарантії.
6. **Можливість розділяти повідомлення на групи за темами.** Таким чином, можна спрямувати одержувачів повідомлень, що реалізують специфічну для певного виду пристрою логіку, на отримання повідомлень з конкретної черги, в яку вміщуються відповідні дані.

Найкраще описані потреби задовольняє Apache Kafka. Обґрунтуємо цей вибір.

2.4.1 Apache Kafka – черга повідомлень для прийому операційних даних

Apache Kafka – це брокер повідомлень із відкритим вихідним кодом, розроблений Apache Software Foundation і написаний на Scala. Kafka – розподілений, партиціонований (розділений), реплікований журнал записів (commit log). Він пропонує функціональність системи повідомлень, але має унікальну архітектуру [20].

Спочатку коротко опишемо термінологію:

- Стрічка повідомлень ділиться на категорії, що називаються темами (topic).
- Процеси, які публікують (publish) повідомлення у Kafka називаються постачальниками (producer).

- Процеси, які підписуються (subscribe) на теми та обробляють стрічку опублікованих повідомлень, називаються споживачами або приймачами (consumer).
- Kafka працює як кластер, що охоплює один або більше серверів, кожен із яких називається брокер (broker).

Kafka виконує реплікацію журналу для партицій кожної теми на конфігуровану кількість серверів (цей параметр може встановлюватися для кожної теми окремо). Це дозволяє робити автоматичне перемикання на репліку в разі збою серверу в кластері, аби повідомлення залишалися доступними навіть за наявності відмов.

Kafka активно використовує файлову систему для зберігання та кешування повідомлень. Хоча диски зазвичай уважаються повільними, правильно розроблена дискова структура може працювати із такою самою швидкістю, як мережа. Лінійні зчитування та записи – найбільш передбачувані схеми використання, тому вони ефективно оптимізуються операційною системою. Сучасні ОС надають техніки read-ahead (читання у сторінковий кеш (pagecache)) і write-behind (запис у сторінковий кеш), які заздалегідь завантажують дані у великі блоки та групують малі логічні операції запису у великі фізичні операції.

Аби компенсувати різницю у продуктивності між оперативною і дисковою пам’яттю, сучасні операційні системи все більш агресивно використовують головну пам’ять для кешування дисків. Майже вся вільна пам’ять може бути використана для кешування диску з мінімальними накладними витратами при її вивільненні. Усі операції читання та запису проходять через цей уніфікований кеш. Така функція не може бути вимкнена простим чином без використання прямого I/O, тому якщо процес підтримує власний кеш, то дані, дуже ймовірно, будуть продубльовані у сторінковому кеші ОС, що означатиме зберігання однієї і тієї ж інформації двічі.

До того ж, оскільки Kafka працює на JVM, то:

- Накладні витрати на зберігання об'єктів є дуже великими, що часто збільшує розмір даних удвічі (або більше).

- Збирання сміття (garbage collection) у JVM значно сповільнюється при збільшення розміру даних, що зберігаються на купі (heap).

Із урахуванням вищеописаних факторів, використання файлової системи із сторінковим кешем є більш ефективним за підтримання в пам’яті кешу або іншої структури тому, що: як мінімум удвічі збільшується доступний кеш через автоматичний доступ до всієї вільної пам’яті, і, швидше за все, іще вдвічі за рахунок зберігання компактної структури байтів замість окремих об’єктів. У результаті розмір кешу може досягати 28-30 Гб на машині з 32 Гб пам’яті, без витрат на збирання сміття. До того ж, цей кеш зберігатиме дані навіть при перезапуску процесу, у той час як кеш у процесі потрібно перебудовувати (що може зайняти до 10 хвилин для 10 Гб кешу), а в іншому випадку при старті з порожнім кешем продуктивність буде дуже низькою. Це також спрощує реалізацію, оскільки підтримка узгодженості між кешем і файловою системою виконується ОС, яка робить це більш ефективно та коректно, порівняно із реалізаціями в процесах. Якщо патерн використання диску показує часті лінійні читання, то read-ahead буде заповнювати кеш потрібними даними при кожній операції зчитування.

Натомість, аби підтримувати якомога більший кеш у пам’яті та оперативно скидати дані на диск, коли закінчується місце, робиться навпаки. Усі дані відразу ж пишуться в файлову систему, але це не обов’язково спричиняє операції фізичного запису. Це означає, що дані переносяться до сторінкового кешу ядра.

Пакетна обробка є запорукою ефективності. Для її забезпечення постачальник даних (producer) намагається накопичити дані в пам'яті та надіслати більші пакети однією операцією. Пакетна обробка може бути сконфігурована, аби накопичувати не більше певного фіксованого числа повідомлень або чекати не більше певної межі затримки (скажімо, 10 мс). Це дозволяє виконувати менше I/O операцій, кожна з яких буде більшою за обсягом. Конфігурована буферизація дає можливість знайти компроміс між додатковою затримкою та більшою пропускною здатністю.

Споживач даних (consumer) працює, надсилаючи “fetch” запити до брокерів, яке керують партиціями, дані з яких йому потрібні. Із кожним запитом він указує зміщення (offset) у журналі, а отримує сегмент логу, який починається із зазначеної позиції. Таким чином, у споживача даних є значний контроль над цією позицією, а отже він здатен зчитати повідомлення повторно, якщо потрібно.

Можна виділити декілька різних видів гарантій доправлення повідомлень:

- *Максимум один раз (at most once)* – повідомлення можуть бути втрачені, але ніколи не доставляються повторно.
- *Як мінімум один раз (at least once)* – повідомлення ніколи не втрачаються, але можуть бути доправлені повторно.
- *Точно один раз (exactly once)* – повідомлення доставляється точно один раз.

Опишемо семантику доставки повідомлень у Kafka. При публікації повідомлення існує поняття запису (commit) повідомлення в журнал. Щойно повідомлення записане, воно не буде втрачене, якщо працездатним залишається як мінімум один брокер, що обслуговує партицію, у яку здійснили запис. Якщо у постачальника даних сталася мережева помилка при публікації повідомлення, він не може визначити, сталася помилка до чи після запису повідомлення. Ця семантика схожа на вставку рядка в таблицю в БД із автоматично згенерованим ключем.

Таким чином, Kafka за замовчанням гарантує «at-least-one» доставку та дозволяє користувачу реалізувати схему «at-most-once» шляхом відключення повторень на стороні постачальника та запису зміщення перед обробкою пакету повідомлень. «Exactly-once» семантика вимагає кооперації із кінцевою системою зберігання даних.

Apache Kafka використовує *Apache Zookeeper* – розподілений сервіс конфігурації та синхронізації. Він застосовується для координації вузлів у кластері, відслідковування статусу брокерів, запису позицій при зчитуванні повідомлень тощо.

2.5 Підсистема обробки даних. Організація, вимоги та обґрунтування вибору стороннього рішення

Описана в підрозділі «Архітектура платформи» схема обробки даних реалізує так звану **лямбда-архітектуру** (lambda architecture). Розглянемо цей концепт детальніше.

Лямбда-архітектура – архітектура обробки даних, створена для роботи з великими обсягами даних, використовуючи одночасно переваги підходів пакетної та потокової обробки даних. Цей підхід до архітектури намагається збалансувати затримку, пропускну спроможність і відмостійкість, застосовуючи пакетну обробку для надання всебічних та точних розрізів (view) пакетів даних, одночасно використовуючи потокову обробку в режимі реального часу для створення розрізів операційних даних. Ці два розрізи даних можуть бути об’єднані перед виводом. Поширення лямбда-архітектури обумовлене збільшенням обсягів даних, потребами аналітики реального часу та намаганням зменшити затримки MapReduce.

Лямбда-архітектура працює з моделлю даних, яка має незмінюване (до якого дані лише дописуються, тобто append-only) джерело. Ця модель призначена для запису та обробки подій із мітками часу, які дописуються до існуючих даних замість перезапису. Стан системи визначається природнім упорядкуванням даних по часу.

Лямбда-архітектуру зображено на діаграмі (рис. 2.7).

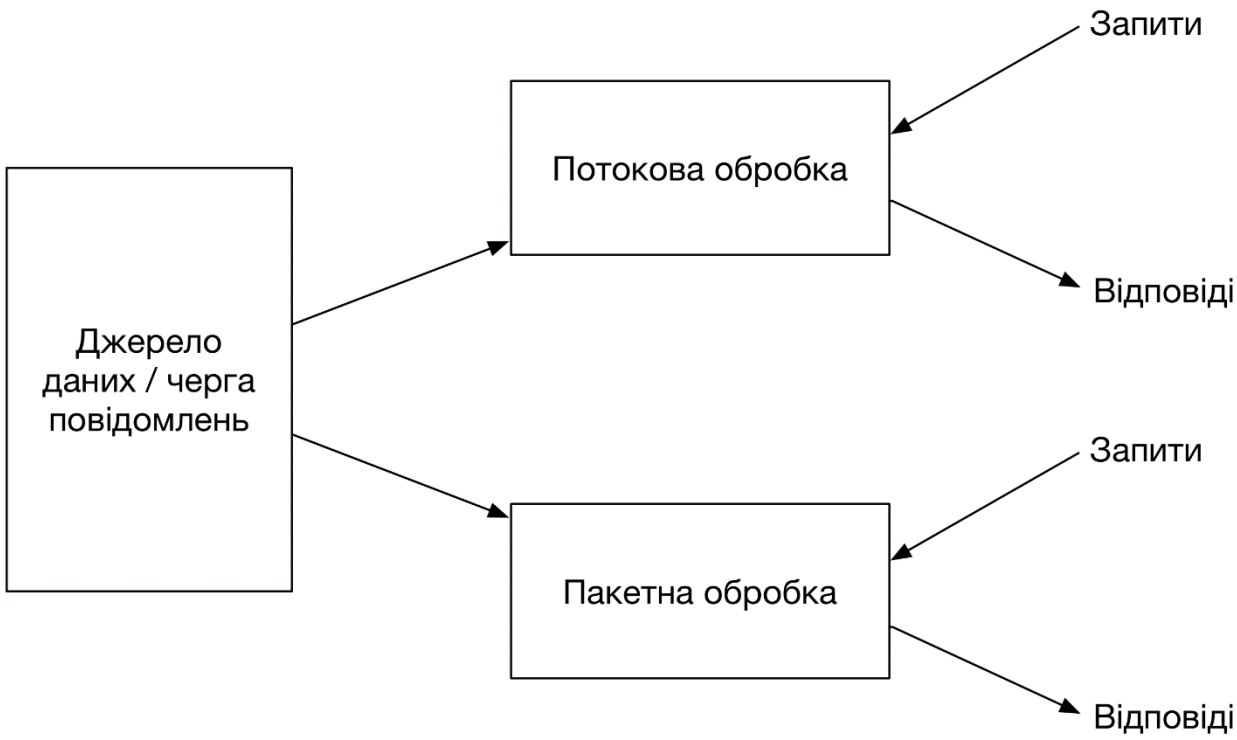

Рисунок 2.7 – Лямбда-архітектура

Опишемо також підхід **MapReduce**, який частково застосовується для обробки даних платформою. Цей підхід призначений для обробки задач, які паралелізуються. Він складається із таких кроків (рис. 2.8):

- *Map:* кожен worker-сервер застосовує функцію map до даних і записує результат у тимчасове сховище. Головний сервер гарантує, що оброблена буде лише одна копія вхідних даних.
- *Shuffle:* worker-сервери розподіляють дані відповідно до вихідних ключів (виходів функції map) так, щоб усі дані, які відповідають одному ключу, розташовувались на одному worker-сервері.
- *Reduce:* worker-сервери обробляють кожну групу даних, відповідно до ключа, паралельно.

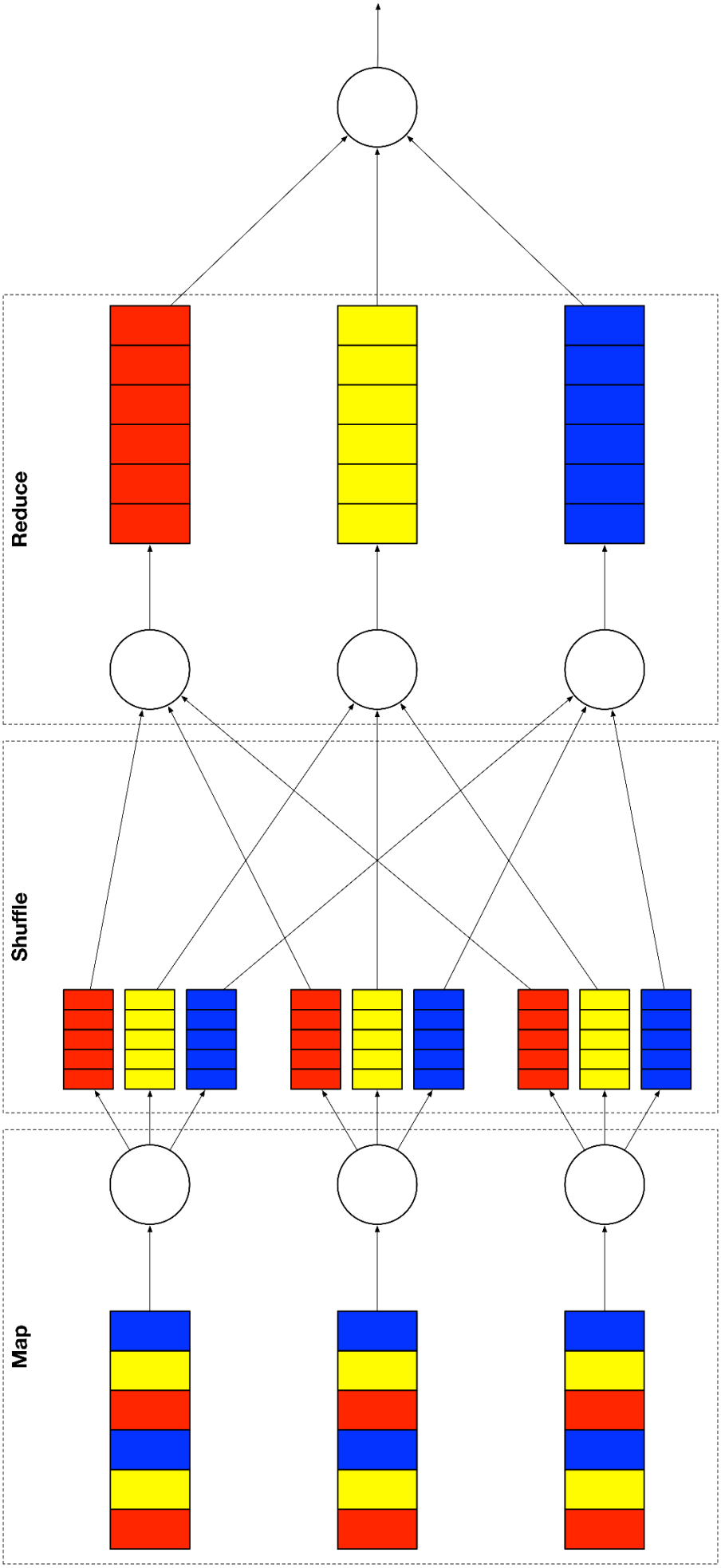

Рисунок 2.8 – Підхід MapReduce

Розглянемо обчислення у парадигмі MapReduce на прикладі обробки значень температури, отриманих від декількох пристроїв (рис. 2.9). Обробка полягатиме в усередненні значень температури для кожного пристрою (це робиться в межах часового вікна деякої довжини).

На вході маємо JSON-записи, що містять ідентифікатор пристрою (усього їх 3) в полі “deviceId” та значення температури в полі “temp”. Початковий етап обробки (map) формує пари “ключ-значення”: ключем є ідентифікатор пристрою, а значенням – температура.

Уважатимемо, що MapReduce виконується на трьох серверах. Крок “shuffle” розподіляє дані так, аби на першому worker-сервері знаходилися дані від першого пристрою, на другому – від другого тощо.

Після “shuffle” на кожному сервері знаходяться записи з однаковими значеннями ключа. Крок “reduce” полягає в обчисленні середнього арифметичного значень температури.

Наприкінці обчислень маємо три пари “ключ-значення”, що містять відповідно id пристрою та середнє значення показань температури, отриманих від нього.

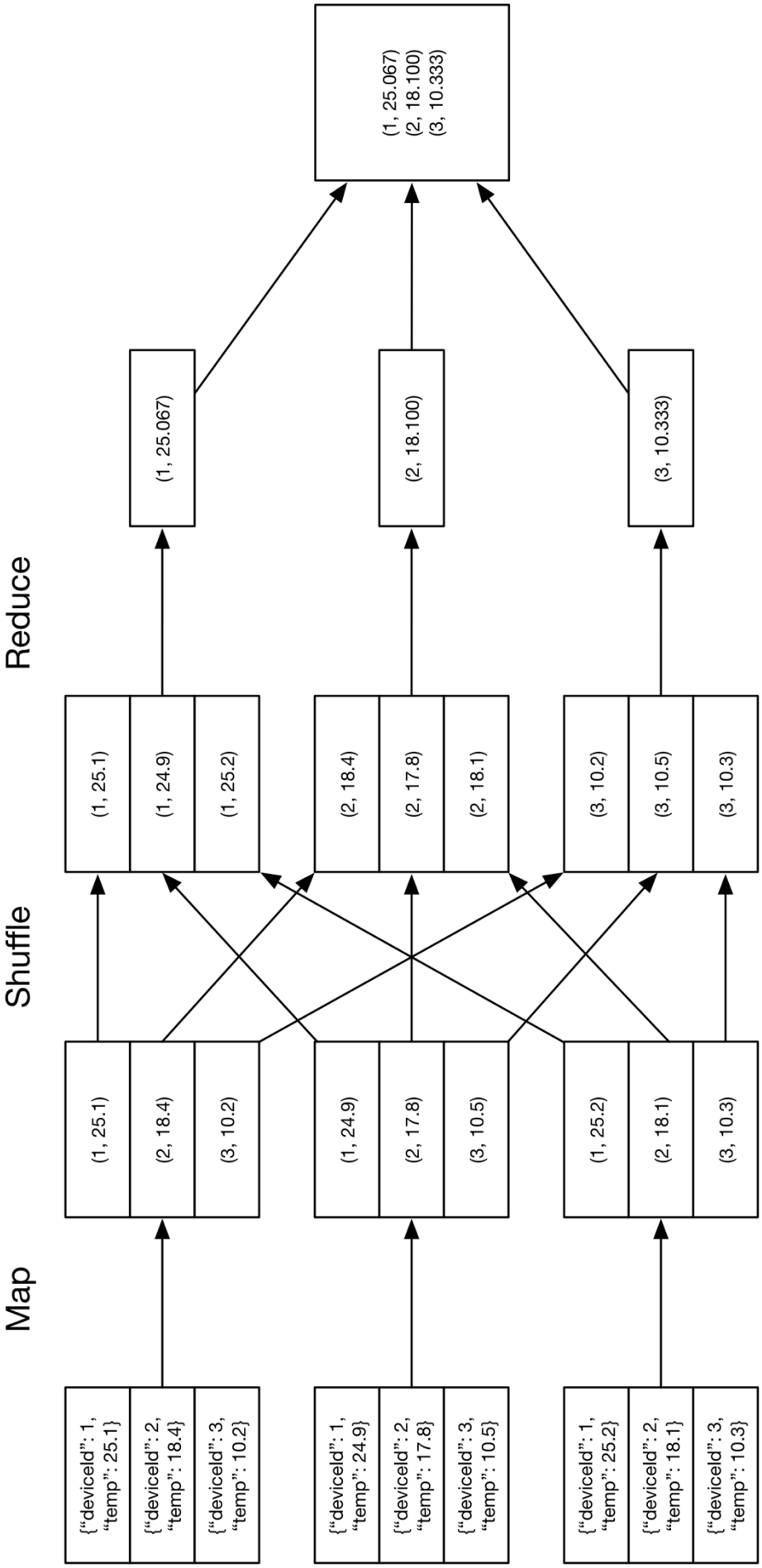

Рисунок 2.9 – Приклад обчислень у парадигмі MapReduce

Сформулюємо також вимоги, яким має відповідати підсистема обробки даних:

- **Висока продуктивність.** Продуктивність потокової обробки не має бути вузьким місцем у системі. Аналітичні запити також потрібно виконувати якомога швидше.
- **Масштабованість.** Можливість підвищити пропускну здатність ланцюжка потокової обробки даних чи прискорити аналітичні запити за допомогою збільшення кількості worker-серверів.
- **Наявність інтерфейсів зв'язку із базами даних і чергами повідомлень.** Підтримка таких API на рівні системи дозволяє спростити розробку чи адаптацію під змінені джерела чи сховища даних.
- **Бібліотека операцій і алгоритмів.** Як мінімум, система повинна підтримувати базові операції статистичного аналізу.
- **Можливість розширення функціональності бібліотеки.** Система має бути розширюваною, аби за необхідності мати можливість додати якийсь специфічний крок обробки даних чи алгоритм.

Для задоволення вищеозначених потреб оберемо систему Apache Spark. Опишемо детальніше її можливості та переваги.

2.5.1 Apache Spark – система обробки даних

Apache Spark – це система обробки великих даних із відкритим вихідним кодом, побудована для забезпечення високої швидкості роботи, простоти використання та складної аналітики [21]. Розроблена у 2009 році у AMPLab (UC Berkeley), публічний реліз як Apache проекту відбувся у 2010 році.

Spark має значні переваги порівняно з іншими Big Data і MapReduce технологіями, такими як Hadoop і Storm.

Перш за все, Spark дає всебічний та уніфікований каркас для роботи з різними за природою (текстові, графові тощо) і джерелом (пакетні чи потокові) наборами даних.

Доведено [22], що Spark працює до 100 разів швидше за Hadoop при виконанні операцій у пам'яті та до 10 разів швидше при використанні диска. Має велику бібліотеку вбудованих даних і алгоритмів. На додачу до операцій Map і Reduce, підтримує SQL запити, потокову обробку, машинне навчання та обробку графових даних. Ці можливості можуть використовуватися окремо або поєднуватися в один ланцюжок обробки даних.

Hadoop і Spark. Hadoop як технологія обробки великих даних була популярною близько 10 років і довела свою ефективність. MapReduce – чудове рішення для однопрохідних обчислень, але не настільки ефективне для задач, які потребують багатопрохідних обчислень і алгоритмів. Кожен крок у ланцюжку обробки даних має по одній фазі Map і Reduce, і розв'язок будь-яких задач необхідно описувати цими операціями.

Вихідні дані кожного кроку потрібно зберігати у розподіленій файловій системі перед початком наступного кроку. Тому цей підхід є досить повільним через реплікацію та дискове сховище. До того ж, рішення, засновані на Hadoop, зазвичай використовують кластери, складні в побудові та керуванні. Вони також потребують інтеграції декількох інструментів для різних сценаріїв використання (як Mahout для машинного навчання та Storm для потокової обробки).

Для складних обчислень потрібно зв’язувати множину завдань MapReduce і виконувати їх послідовно. При цьому кожне завдання має велику затримку та не може стартувати до повного закінчення попереднього.

Spark дозволяє розробляти комплексні багатокрокові ланцюжки обробки даних, застосовуючи патерн орієнтованого ациклічного графу (directed acyclic graph – DAG). Він також підтримує спільне використання даних у пам’яті різними графами, що дозволяє багатьом завданням оперувати одними й тими самими даними.

Spark працює з існуючою інфраструктурою Hadoop Distributed File System (HDFS), розширяючи та додаючи до неї функціональність.

Особливості Spark. Spark покращує MapReduce, надаючи більш швидкий та ефективний крок Shuffle при обробці даних. Продуктивність в декілька разів

вища за інші Big Data технології за рахунок зберігання даних у оперативній пам’яті та обробки, близької до реального часу (near real-time).

Spark підтримує лініве обчислення (lazy evaluation) запитів, що допомагає оптимізувати їх виконання. Високорівневий API спрощує розробку та надає узгоджену архітектурну модель для складних рішень.

Проміжні результати зберігаються в пам’яті, а не на диску, що корисно, якщо над певним набором даних виконується декілька операцій. Узагалі, система обробки працює як у пам’яті, так і на диску. Якщо дані не вміщаються в пам’яті, виконуються зовнішні операції. Spark можна використовувати для обробки наборів даних, обсяг яких перевищує сукупну пам’ять серверів у кластері.

Spark намагається зберігати якомога більше даних у пам’яті. Частина даних може знаходитися на диску. Використання оперативної пам’яті – одна з головних переваг у продуктивності, але потрібно відповідним чином аналізувати вимоги до апаратного забезпечення, щоби нею ефективно користуватися.

Resilient Distributed Dataset [23] (RDD) – основний концепт у Spark. RDD можна порівняти з таблицею у БД. Він може містити будь-який тип даних. Spark зберігає RDD у різних партиціях.

Ця абстракція дозволяє оптимізувати обробку, змінюючи порядок обчислень. Вони також відмовостійкі, бо RDD відомо, як повторно створити чи обчислити набір даних.

RDD незмінювані. Його можна модифікувати за допомогою трансформації, але вона повертає новий RDD, а вихідний RDD залишається незмінним. Підтримуються два типи операцій:

- *Трансформації.* Повертають новий RDD. Виклик функції трансформації RDD повертає новий RDD, до явного запуску обчислень вони не стартують (ліниві обчислення). Прикладами трансформацій є map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, pipe, і coalesce.
- *Дії.* Дія відразу запускається та повертає значення. Приклади: reduce, collect, count, first, take, countByKey, і foreach.

Окрім Spark Core API, є також додаткові бібліотеки, які є частиною екосистеми Spark і надають додаткові можливості у сфері аналітики та машинного навчання. Ці компоненти включають:

- **Spark Streaming.** Може використовуватися для обробки потокових даних у режимі реального часу. Він базується на мікропакетному стилі обчислень і обробки. Використовується абстракція DStream, яка є послідовністю RDD. Основною перевагою є те, що один і той самий код, що описує ланцюжок обробки даних, можна використовувати як для пакетної обробки даних, так і для потокової.
- **Spark SQL.** Надає можливість відкрити доступ до наборів даних Spark через JDBC API та запускати запити, схожі на SQL, використовуючи традиційні інструменти BI (Business Intelligence) та візуалізації. За допомогою цієї бібліотеки можна виконати ETL (Extract, Transform, Load) даних із різних форматів (JSON, Parquet, реляційна БД та інших), здійснити їх перетворення та зробити доступними для ad-hoc запитів.
- **Spark MLlib.** Масштабована бібліотека машинного навчання, що містить традиційні алгоритми навчання та утиліти, які включають класифікацію, регресію, кластеризацію, колаборативну фільтрацію, зменшення розмірності та низькорівневі примітиви оптимізації.

2.6 Підсистема зберігання операційних даних. Організація, вимоги та обґрунтування вибору стороннього рішення

Після первинної обробки дані записуються в базу операційних даних, яка забезпечує їх постійне зберігання. Із цього сховища дані також зчитуються фронтальними серверами для виконання запитів, а також підсистемою обробки даних для подальшого виконання обчислень, необхідних для задоволення аналітичного запиту. Отже, сформулюємо вимоги, яким має відповідати база операційних даних:

1. **Оптимізована для запису.** Зазвичай, майже в усіх подібних сховищах операційних даних в Big Data системах, таких як розглядувана платформа, простежується основна особливість: кількість операцій запису набагато вища за число операцій читання. Цей патерн використання відрізняється від того, під який оптимізовані традиційні реляційні бази даних. Вони найкраще пристосовані до сценарію, коли в БД лише час від часу записуються дані, але часто надходять запити на зчитування. Наприклад, якщо є певна таблиця, у якій містяться новини на певному веб-сайті, то очевидно, що записів в неї здійснюється набагато менше, ніж зчитувань. Якщо ж сховище даних обслуговує систему IoT, то в будь-яку секунду воно повинне виконувати десятки тисяч запитів запису. Зчитування ж треба робити лише по запитах адміністраторів чи аналітиків. Високу продуктивність зчитувань теж було б непогано мати, але вона значно менш важлива за швидкість записів.
2. **Горизонтальна масштабованість.** Збільшення кількості серверів має підвищувати пропускну спроможність для операцій запису, а також обсяг даних, що зберігаються.
3. **Доступність.** За допомогою реплікації повинна забезпечуватися доступність бази (для записів і зчитувань) в разі відключення частини кластеру.
4. **Послаблені гарантії транзакційності.** База операційних даних вимагає значно слабших гарантій транзакційності за традиційний набір ACID, який гарантується реляційними даними. Наприклад, якщо йдеться про узгодженість, то більшість сценаріїв використання задовольнить консистентність в остаточному рахунку (eventual consistency).
5. **Мінімальний набір функціональності запитів.** Оскільки для виконання запитів застосовується підсистема обробки даних, база операційних даних має надавати лише мінімальний набір операцій, таких як запис, читання, видалення тощо. Скажімо, якщо при виконанні

запиту потрібно застосувати агрегаційну функцію (таку як суму, мінімум чи максимум), то це буде робитися підсистемою обробки даних, а базі операційних даних лише потрібно зчитати необхідні дані (відфільтровані по певному ключу, наприклад, дані за конкретну дату).

З урахуванням перелічених вимог зрозуміло, що реляційна база навряд чи підійде для зберігання операційних даних і треба використовувати NoSQL рішення. Реляційні БД записують дані набагато повільніше, аніж зчитують. Реплікація наявна в деяких SQL базах даних, але її функціональність часто обмежена і вона має певні особливості, обумовлені вимогами транзакційності ACID. Горизонтальна масштабованість у реляційних баз (sharding) є дуже складною з тих же причин. Це призводить до того, що коли реляційній базі бракне місця на диску або вона не може обслуговувати потрібну кількість запитів за одиницю часу, то без значних затрат часу та збільшення складності системи можна лише нарощувати потужність одного сервера (вертикальне масштабування). До того ж, необхідно зберігати великі обсяги даних, а продуктивність запитів до таблиць у реляційних базах значно зменшується при збільшенні розміру таблиць.

Із NoSQL баз для зберігання операційних даних оберемо Apache Cassandra. Опишемо, як вона влаштована та чому відповідає поставленим вимогам.

2.6.1 Apache Cassandra – база даних для зберігання операційних даних

Apache Cassandra – це розподілена система керування базами даних із відкритим вихідним кодом, розроблена для роботи з великими обсягами даних, розподіленими по великій кількості неспеціалізованих серверів і надання високої доступності без єдиної точки відмови (single point of failure) [24]. Cassandra є надійною навіть для кластерів, частини яких розташовано в різних дата-центрах. СКБД Cassandra написана на мові Java і об'єднує в собі повністю розподілену hash-систему Dynamo, що забезпечує практично лінійну масштабованість при збільшенні обсягу даних.

Теорема CAP. Для розуміння Cassandra, варто спочатку згадати теорему CAP. Вона твердить, що розподілена система не може надати більше двох із таких гарантій:

- узгодженість даних (усі вузли бачать однакові дані у будь-який момент часу);
- доступність (гарантія того, що кожен запит отримає відповідь про його успішне або помилкове завершення);
- стійкість до розділення (незважаючи на розділення на ізольовані секції або втрати зв'язку з частиною вузлів, система не втрачає стабільність і здатність коректно відповідати на запити).

Cassandra надає користувачам сильні гарантії доступності та стійкості до розділення, тобто з точки зору теореми CAP є AP системою. Узгодженість є конфігурованою, що дає можливість знайти компроміс між узгодженістю та затримкою (latency). При записуванні/зчитуванні даних можна задати рівень узгодженості для кожного запиту.

Модель даних BigTable / Log-structured. У моделі даних BigTable первинний ключ і назви колонок відображаються на відповідні значення, формуючи багатовимірне відображення. Кожна таблиця має багато вимірів. Мітка часу (timestamp) є одним із таких вимірів, який дозволяє таблиці версіонувати дані, а також використовується для внутрішнього збирання сміття (видалених даних). У таблиці 2.1 показано структуру даних; ключ рядка (row key) виконує роль ідентифікатора колонки, яка слідує за ним, а назва колонки та значення зберігаються у суміжних блоках.

Таблиця 2.1 – Структура даних у Cassandra

Ключ рядка	Колонка 1	…	Колонка N
	Значення 1	…	Значення N
	Сімейство колонок		

Розглянемо приклад зберігання інформації для сценарію роботи з метеорологічними даними. У таблиці 2.2 ключем рядка є ідентифікатор пристрою (“deviceId”), а в колонках “temperature” і “pressure” записані відповідно значення температури та тиску.

Таблиця 2.2 – Приклад зберігання метеорологічних даних

deviceId = 1	temperature	pressure
	24.3	738.4
deviceId = 2	temperature	pressure
	16.1	762.9

Варто зауважити, що в кожному рядку зберігаються не тільки значення, а і назви колонок, що дозволяє структурі (schema) бути динамічною.

Сімейства колонок. Колонки згруповані у множини, що називаються сімействами колонок (column families), які можуть бути адресованими ключем рядка. Створення сімейства колонок є обов'язковим для зберігання даних. Зазвичай намагаються, аби кількість різних сімейств колонок у просторі ключів (keyspace) була невеликою, і щоб сімейства лише іноді змінювалися. Проте кількість колонок у сімействі необмежена.

Простір ключів. Простір ключів (keyspace) – це група сімейств колонок. Стратегії реплікації та контроль доступу застосовуються на рівні простору ключів. Якщо порівняти з реляційними БД, то простір ключів – це база даних (schema), а сімейство колонок – таблиця.

SSTable (Sorted String Table). SSTable – це формат зберігання даних у файлах, який використовує Cassandra; це впорядкована незмінювана (immutable) структура рядків колонок (пар «назва-значення»). Є в наявності операції пошуку значення, асоційованого із певних ключем, а також перебору пар «назва колонки – значення» у вказаному діапазоні ключа. Кожна SSTable містить послідовність ключів рядків і множину пар «ключ-значення» колонок. Існує індекс і позиція початку ключа рядка у файлі індексу, який зберігається окремо. Скорочений

індекс завантажується у пам’ять при відкритті SSTable для оптимізації обсягу пам’яті, необхідної для індексу. Пошук рядків може виконуватися за допомогою одного дискового пошуку (disk seek) і послідовного сканування.

Memtable. Memtable – місце в пам’яті, куди записуються дані під час операцій оновлення чи видалення. Це тимчасове розташування, дані з якого будуть записані на диск у формі SSTable, щойно воно заповниться. Узагалі, операція оновлення чи запису у Cassandra – це послідовний запис у журнал записів (commit log) на диску та оновлення пам'яті; отже, записи є такими ж швидкими, як і запис у пам'ять. Це виглядає так, як показано на рис. 2.10.

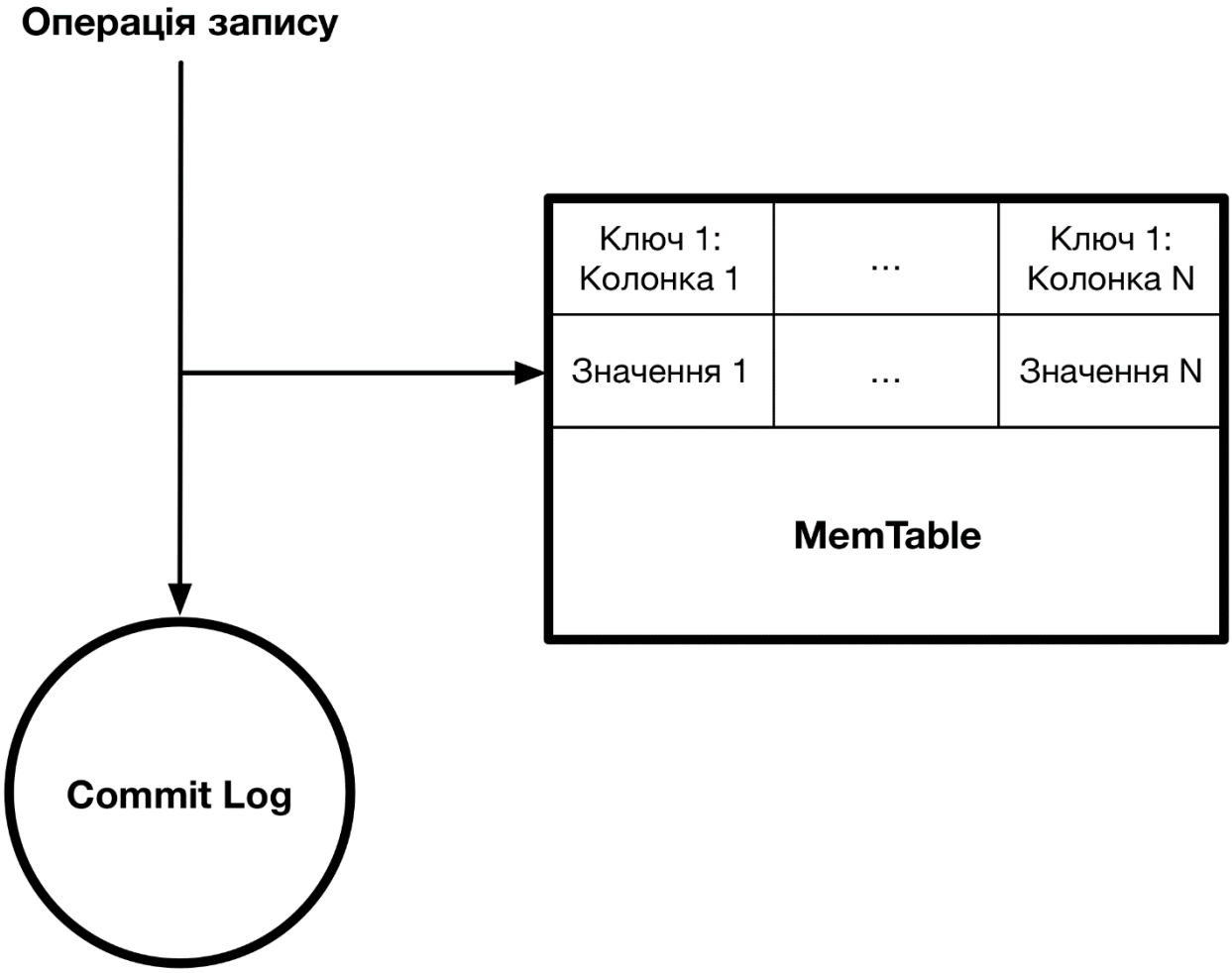

Рисунок 2.10 – Схема запису у Cassandra

Операції читання у Cassandra об’єднують дані з різних SSTable і дані з Memtable. Запити зчитування завжди повинні містити ключ рядка (первинний ключ) за винятком випадку сканування діапазону.

Коли надходить множина запитів на оновлення певної колонки, Cassandra використовує надані клієнтами мітки часу для вирішення конфліктів. Операції видалення працюють по-іншому: оскільки SSTable незмінювані, Cassandra записує так званий tombstone, аби уникнути довільного запису (random write). Tombstone – спеціальне значення, яке записується замість негайного видалення даних та може видалятися під час збирання сміття. Воно може надсилатися вузлам у кластері, які не отримали перший запит на видалення.

Компактифікація. Аби обмежити кількість файлів, які задіяні при зчитуванні даних, та для вивільнення місця, зайнятого непотрібними даними, Cassandra виконує компактифікацію (compaction). Тобто компактифікація об’єднує декілька (ця кількість налаштовується) SSTable в одну велику SSTable. Спочатку SSTable мають такий самий розмір, як і Memtable. Таким чином, розміри SSTable експоненційно зростають у міру їх старіння та надходження даних.

Партиціонування та схема реплікації схожа на ту, яка описана в статті про Dynamo [25].

Gossip protocol. Cassandra – peer-to-peer система без єдиної точки відмови; інформація про топологію кластера передається за допомогою gossip protocol. Gossip protocol схожий на плітки у соціальних комунікаціях (звідки й назва). Згідно цьому протоколу, вузол B передає іншим вузлам у кластері те, що він знає про стан вузла A. Ці вузли передають іншим інформацію про A, і через деякий час усі вузли дізнаються про стан A.

Розподілена хеш-таблиця. Ключовою особливістю Cassandra є можливість інкрементально масштабуватися. Це включає здатність динамічно партиціонувати дані по набору вузлів у кластері. Cassandra партиціонує дані в кластері, застосовуючи консистентне хешування та випадковим чином розподіляє рядки по мережі відповідно до хешу ключа рядка. Коли вузол

включається в кільце, йому присвоюється знак (token), яким позначається місце його розташування в кільці (рис. 2.11).

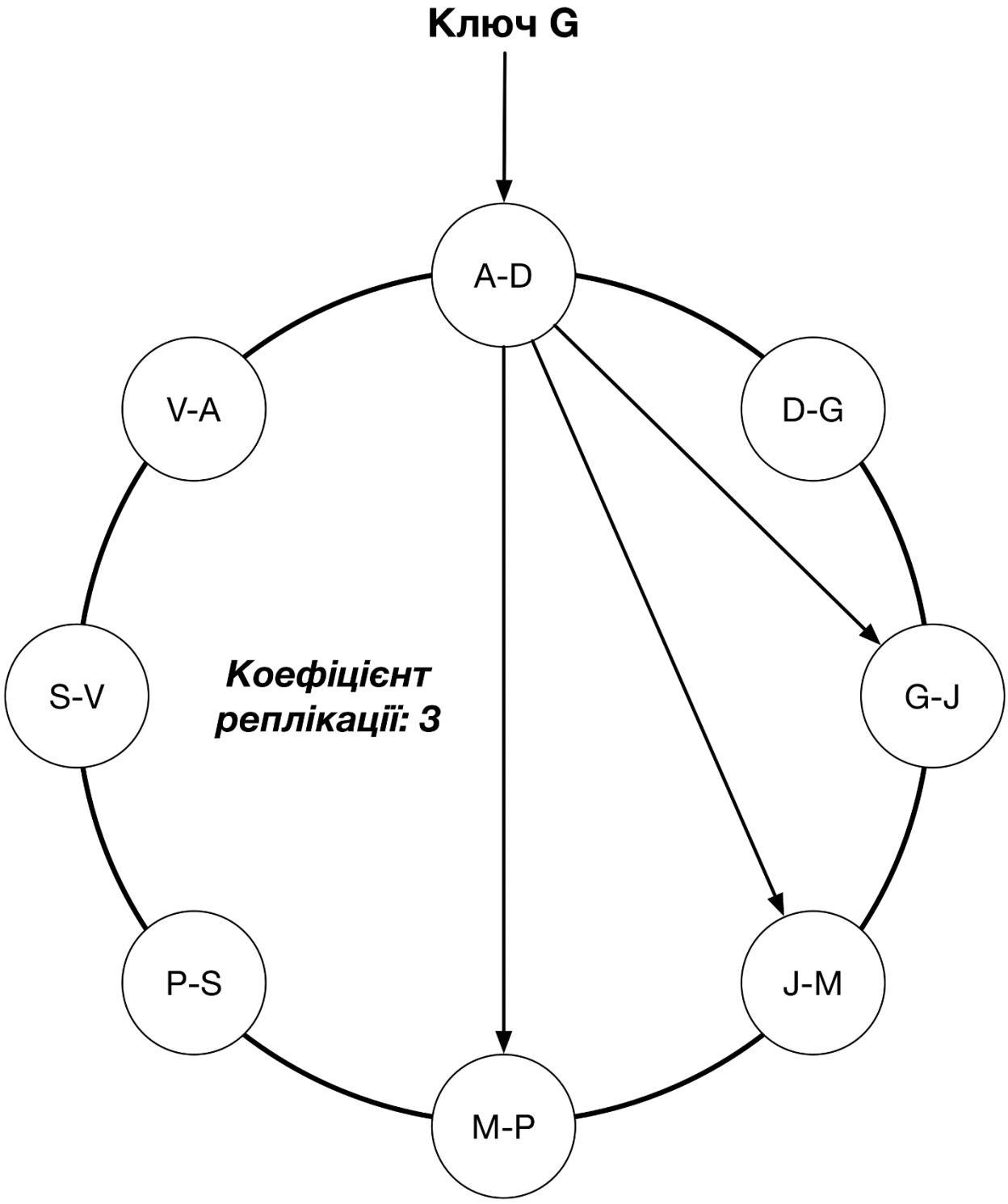

Рисунок 2.11 – Партиціонування даних при записі в Cassandra

Розглянемо випадок, коли коефіцієнт реплікації (replication factor) дорівнює 3; клієнти випадковим чином пишуть або читають із вузла-

координатора (кожен вузол у системі може бути координатором і вузлом із даними) у кластері. Цей вузол обчислює хеш ключа рядка, що дає інформацію щодо того, на якому вузлі у кільці записати дані. Координатор також записує на сусідні вузли відповідно до коефіцієнту реплікації та порядку розташування вузлів у кільці.

Консистентність в остаточному рахунку. За достатній період часу без змін, усі оновлення повинні поширитися системою, узгоджуючи репліки. Cassandra підтримує як сильну узгодженість (strong consistency), так і консистентність в остаточному рахунку (eventual consistency); це може контролюватися клієнтом, що виконує операцію.

Cassandra підтримує різні рівні консистентності при записуванні чи зчитуванні даних. Рівень узгодженості описує число реплік із даними, з якими повинен з’єднатися вузол-координатор перед тим, як підтверджувати клієнту завершення операції. Якщо $W + R > K$, де W – кількість реплік, які повинні підтвердити запис, R – кількість вузлів, які повинні підтвердити читання, K – коефіцієнт реплікації, то забезпечується сильна консистентність. Доступні такі рівні консистентності:

- ONE: R/W як мінімум на один вузол;
- ALL: R/W на всі вузли для цього ключа партиціонування;
- QUORUM: R/W як мінімум на $\lfloor K/2 \rfloor+1$ вузлів, де K – коефіцієнт реплікації.

Коли вузли зупинені для технічного обслуговування, Cassandra збереже інформацію щодо оновлення даних на цих вузлах, яка буде використана, щойно вузли знову стануть доступними. Також, для забезпечення консистентності, Cassandra застосовує такі техніки, як hinted handoffs, read repairs і anti-entropy repairs.

2.7 Підсистема зберігання адміністративних даних. Організація, вимоги та обґрунтування вибору стороннього рішення

Платформа передбачає передачу статусної інформації від пристроїв до адміністраторів та команд від адміністраторів до пристроїв. Також призначені адміністраторам статусні повідомлення можуть генеруватися тригерами моніторингу. Як було описано в розділі «Архітектура платформи», ці дані обов’язково надійно зберігаються перед їх передачею отримувачу.

Також у цій базі зберігається автентифікаційна/авторизаційна інформація (тобто права доступу користувачів і пристроїв) та дані про пристрої (їх тип, розміщення та інша інформація).

Для безпеки паролі зберігаються у захешованому алгоритмом Bcrypt вигляді. **Bcrypt** – функція хешування паролів, розроблена Нільсом Провосом і Девідом Мазіересом [26], яка базується на шифрі Blowfish. Окрім використання солі (salt) для захисту від атак із застосуванням райдужних таблиць (rainbow table), bcrypt є адаптивною функцією: із часом кількість ітерацій може бути збільшена, аби сповільнити її обчислення, так що вона залишається стійкою для атак методом “грубої сили” навіть із збільшення кількості наявних обчислювальних ресурсів.

Опишемо вимоги, які висуваються до бази адміністративних даних:

1. **Надійність зберігання.** Якщо БД підтверджує виконання запиту запису, маємо бути впевнені, що дані збережено в постійне сховище, тобто після збою сервера їх не буде втрачено.
2. **Сильна узгодженість.** Якщо база адміністративних даних розподілена, то після успішного запису певного значення операція зчитування повинна обов’язково повертати те ж саме значення.
3. **Реплікація.** Мінімізувати час недоступності бази у випадку збою сервера за допомогою підтримання репліки основного сервера в режимі hot standby та перемикання на неї (failover). Важливо для зберігання статусної

інформації: пристрої можуть повторювати запити деякий час (якщо вони важливі), але їх можливості щодо зберігання даних сильно обмежені.

4. **Функціональна мова запитів.** Сховище адміністративних даних містить інформацію про пристрої, яка може застосовуватися при виконанні аналітичних запитів. Наприклад, може бути потрібно відфільтрувати пристрої по певному критерію. Також конче необхідною є підтримка геопросторових (geospatial) запитів. Оскільки платформа орієнтована на збір і аналіз даних із великої кількості розподілених у просторі пристроїв, геопросторові запити потрібні для виконання аналітичних запитів, скажімо, пошуку пристроїв у певному радіусі від вказаних координат. Повнотекстовий пошук (full-text search) також стане в нагоді при аналізі даних, що містяться в базі адміністративних даних. Наприклад, може знадобитися пошукати статусні повідомлення, що містять конкретне слово.
5. **Ефективні операції видалення.** Із описаних типів даних, які зберігаються в цій базі, найбільший обсяг мають статусні повідомлення та команди. Але більшість із них можна видаляти після їх отримання. Отже, база має підтримувати ефективні операції видалення, які швидко виконуються та вивільнюють місце на диску.
6. **Динамічна структура даних.** Зручно мати довільну структуру даних, аби спростити адаптацію під змінювані формати, що використовуються пристроями.

Реляційні бази загалом задовольняють вимоги (1)-(3). Вони традиційно забезпечують значні гарантії зберігання даних, а в більш нових версіях (наприклад, PostgreSQL 9.0+) мають підтримку режиму hot standby.

Мова запитів SQL має широкі аналітичні можливості, але стандарт не описує геопросторові запити, тому вони доступні лише як розширення в деяких БД, а їх можливості є дещо обмеженими (наприклад, у MySQL [27]). Можливості повнотекстового пошуку теж не дуже розвинені в реляційних БД, тому для нього знадобиться використовувати окреме рішення.

До того ж, реляційні бази неефективно видаляють дані: такі операції є дуже повільними, а вивільнення простору на диску є ще довшим і складнішим. Наприклад, вивільнення місця у таблиці InnoDB (MySQL) потребує повторного створення таблиці (тобто створення нової таблиці, копіювання даних зі старої в нову та видалення старої) [27].

Більше за інші рішення наші вимоги задовольняє MongoDB. Опишемо, як вона влаштована та чому підходить.

2.7.1 MongoDB – база даних для зберігання адміністративних даних

MongoDB – крос-платформна документо-орієнтована база даних. Класифікована як NoSQL, MongoDB замість традиційних табличних структур реляційних баз використовує схожі на JSON документи з динамічною структурою (формат називається BSON), що робить інтеграцію даних швидшою. Розроблена MongoDB Inc. і опублікована як безкоштовне ПО із відкритим вихідним кодом під комбінацією ліцензій GNU Affero General Public License і Apache License. На липень 2015 року MongoDB є четвертою за популярністю СКБД і найбільш популярним документо-орієнтованим сховищем.

Оглянемо основні риси MongoDB:

Ad-hoc запити. Підтримуються запити полів, діапазонів, пошук з регулярними виразами. Запити можуть зчитувати окремі поля документів та включати задані користувачем JavaScript функції, які виконуються безпосередньо сервером.

Індексування. Проіндексувати можливо будь-яке поле в документі, включно з масивами та вкладеними документами (індекси в MongoDB концептуально схожі на індекси в реляційних СУБД). Доступні первинні та вторинні індекси.

Реплікація. Доступність підтримується за рахунок наборів реплік (replica sets). Набір реплік складається із двох або більше копій даних. Кожен такий набір може виконувати роль первинної або вторинної репліки. Первинна за замовчанням виконує всі записи та читання, тобто забезпечується **сильна**

узгодженість. Вторинні підтримують копію даних первинної репліки. У разі збою первинної репліки, набір реплік автоматично обирає, яка з вторинних реплік стане первинною. Вторинні репліки можуть (опціонально) виконувати операції читання, але дані будуть консистентні в остаточному рахунку.

Балансування навантаження. MongoDB горизонтально масштабується за допомогою шардінгу. Користувач обирає ключ, який визначає, як розподіляються дані в колекції. Вони діляться на діапазони та розподіляються по кількох серверах. Ключ шардінгу також може хешуватися для рівномірного розподілу.

Агрегація. Дозволяє виконувати запити, аналогічні SQL GROUP BY. Оператори агрегування можуть поєднуватися в ланцюжки, як Unix pipes. Також є оператор lookup, який об'єднує документи.

Швидкі операції видалення. MongoDB зберігає дані у двобічно зв'язаному списку, тому для видалення потрібно лише змінити два вказівника. Але, **автоматична компактифікація відсутня**, тому місце на диску не вивільнюється автоматично, а реалізується окремою утилітою.

Обмежені колекції. Підтримують колекції фіксованого розміру, що називаються обмежені колекції (capped collections). Вони підтримують порядок вставки, та при досягненні вказаного розміру поводяться як циклічна черга (circular queue). Можуть бути корисними для статусних повідомлень і команд, наприклад, аби не зберігати більше вказаного числа записів. Використання таких колекцій може бути набагато ефективнішим за регулярне видалення записів.

Відсутність підтримки багатодокументних транзакцій. ACID транзакції підтримуються лише на рівні документа. Але, як буде показано пізніше, структура документів, що буде використовуватися, не вимагає багатодокументних транзакцій.

Динамічна структура. Оскільки значення в документах зберігаються разом із назвами полів, можна динамічно змінювати модель даних.

Повнотекстовий пошук. Підтримуються досить широкі можливості повнотекстового пошуку, вимагає наявності текстового індексу.

Геопросторові запити. За наявності відповідного індексу, можна виконувати різноманітні геопросторові запити, наприклад, шукати точки на заданій відстані від вказаної. При цьому обчислення виконуються з високою точністю, бо відстань, наприклад, відраховується від точки на сфері, тобто враховується кривизна Землі.

2.8 Деякі аспекти досягнення високої ефективності платформи

Як вже було зазначено, платформа Інтернету речей має бути здатна обслуговувати величезну кількість даних, а тому бути високоефективною. Відмітимо деякі аспекти, які дозволяють цього досягти.

По-перше, величезний вплив на продуктивність має **організація мережевого вводу/виводу**. Описана **проблема 10000 з'єднань** – задача оптимізації коду та конфігурації сервера так, аби мати можливість обслуговувати паралельно велику кількість з'єднань.

Традиційний метод, який був популярним деякий час тому, передбачав використання одного потоку операційної системи на з'єднання. Наприклад, якщо надходить запит на зчитування даних із бази, потік надсилає його та блокується до того моменту, як від СКБД надійде відповідь. При надходженні іншого запиту до сервера він запускає новий потік.

Але із такою моделлю кількість одночасно оброблюваних запитів обмежується кількістю потоків ОС, які можна запустити. Ця кількість, зазвичай, значно менша за 10000. Обмежується вона в основному кількістю доступної пам'яті, бо для кожного потоку потрібно виділити певний простір для його стеку.

Ефективним рішенням є використання **асинхронного I/O**, тобто такого, який не блокує інші операції (тобто потік виконання) під час очікування завершення передачі. Наприклад, на Java Virtual Machine можна використовувати можливості Java NIO (New IO), які реалізують асинхронний ввід/вивід за допомогою опитування (polling) джерел даних. При застосуванні цього методу, якщо очікуються дані із мережі, періодично перевіряється стан

мережевого сокету на предмет їх надходження; при цьому потік не блокується, інші обчислення та запити продовжують виконуватися.

Переваги використання асинхронного вводу/виводу не обмежуються збільшенням кількості з'єднань, які сервер здатен обслуговувати паралельно. Приріст продуктивності також забезпечується більш ефективним використанням CPU: у моделі "один потік на з'єднання" наявні дуже великі накладні витрати на перемикання контексту (context switch); із асинхронним I/O опитувати всі сокети можна лише в одному потоці.

Але це не потребує застосування суто однопотокової архітектури. Якщо при надходженні мережевих запитів потрібно виконувати складні обчислення та використовується багатоядерна система, то потік, який опитує джерела даних, може виконувати диспетчеризацію – надсилати задачі пулу потоків. Результати обчислень можуть повертатися головному потоку для відправки в мережу.

Асинхронний ввід/вивід наразі є одним із найефективніших методів розробки високопродуктивних серверів. Хоча, цей підхід має свої недоліки. Код, написаний у асинхронному стилі, набагато складніший для розуміння за послідовний "блокуючий". Також, за умови організації обчислень таким чином, як описано в попередньому абзаці, коли потік-диспетчер зчитує дані та передає задачу в інший потік, а результат повертається в потік I/O, потрібне застосування інструментів синхронізації. Це є складним і значно підвищує шанс появи помилок, пов'язаних із гонками та іншими проблемами, які потрібно вирішувати при організації паралелізму.

Іншим засобом підвищення продуктивності є організація **stateless архітектури**, тобто такої, яка не потребує зберігання на сервері стану між запитами. Наприклад, використання такої моделі авторизації, яка не вимагає перевірки деякого збереженого стану чи звертання до бази даних.

У сучасних розподілених системах часто вузьким місцем є пропускна спроможність мережевого обладнання, а не обчислювальні здатності процесорів. Тому використання таких форматів даних, які вимагають більшої кількості операцій CPU на формування, але виробляють більш компактні повідомлення,

здатне підвищити загальну продуктивність розподіленої системи. Цей підхід застосовується у Apache Spark, який для передачі даних між компонентами використовує стиснений компактний формат передачі серіалізованих об’єктів.

Загалом, оптимізовувати роботу системи варто, таргетуючи найбільш повільні та затратні операції. Скажімо, при повсюдному використанні JVM варто звертати увагу на операції, які створюють велику кількість об’єктів – це створює тиск на garbage collector, вимагаючи більш активного застосування CPU на збирання сміття та довших пауз. Іншим прикладом є традиційна серіалізація, яка використовує Reflection API, тобто інструменти динамічного доступу до полів об’єктів. Саме рефлексія є найвужчим місцем при серіалізації, тому ефективні серіалізатори використовують розумні підходи для того, аби її уникнути, такі як ручна робота із байткодом для динамічної кодогенерації.

2.9 Організація автентифікації. JSON Web Token

Автентифікація потрібна для запитів, що надходять від пристроїв та інших користувачів системи (адміністраторів, аналітиків тощо). Потрібно забезпечити значну захищеність системи, яка, втім, дозволяє підтримувати високу пропускну спроможність обробки запитів.

Оберемо **JSON Web Token (JWT)** – сучасний інструмент, який має багато переваг перед традиційними методами автентифікації. JWT – відкритий стандарт [28], який визначає компактний і самодостатній спосіб безпечної передачі інформації між сторонами у вигляді JSON об’єкту. Ця інформація може бути перевірена, бо вона має цифровий підпис. JWT можуть бути підписані за допомогою секретного слова (алгоритм HMAC) або парою з публічного та приватного ключів (RSA).

Завдяки невеликому розміру JWT можна пересилати у URL, у POST параметрі чи всередині HTTP заголовка. До того ж, компактність збільшує швидкість передачі. Самодостатність означає, що токен містить усю необхідну

інформацію про користувача, що дозволяє не робити запити до бази даних більш ніж один раз.

JWT складаються з трьох частин:

- **Заголовок.** Містить дві частини: тип токена (JWT) і алгоритм хешування, який використовується (наприклад, HMAC SHA256 або RSA). Заголовок кодується за допомогою Base64Url та є першою частиною JWT.
- **Корисне навантаження (payload).** Містить інформацію про користувача та додаткові метадані. Є зарезервовані, публічні та приватні поля. Корисне навантаження також кодується Base64Url та є другою частиною JWT.
- **Підпис.** Для створення підпису беруться закодовані заголовок і корисне навантаження, а також секретне слово, і підписуються за допомогою вказаного в заголовку алгоритму. Підпис використовується, аби впевнитись, що відправник JWT – той, за кого він себе видає, і повідомлення не було змінене на шляху до серверу.

При використанні JWT для автентифікації, коли користувач успішно входить до системи зі своїми автентифікаційними даними, сервер повертає JSON Web Token, який клієнт має зберегти локально. Цей підхід заміняє традиційний, коли на сервері створюється сесія, а клієнту повертається її ідентифікатор.

Коли користувачу потрібно отримати доступ за захищеної адреси чи ресурсу, він надсилає JWT, наприклад, у HTTP-заголовку «Authorization».

JWT – це механізм автентифікації, що не зберігає стану (stateless), бо стан користувача не потрібно зберігати на сервері, – токен містить усю необхідну інформацію.

JWT потрібно передавати через захищене з’єднання, використовуючи, наприклад, HTTPS. Це обумовлено тим, що криптографічний алгоритм забезпечує лише перевірку того, що токен і справді був сформований сервером у такому вигляді, в якому він отриманий. При використанні відкритого протоколу третя сторона досить просто може подивитися вміст повідомлення.

Використовуючи цю схему, для підключення до платформи пристрої надсилають на спеціальну адресу свій ідентифікатор і пароль. Сервер перевіряє

за базою адміністративних даних, чи відомо йому про такий пристрій і чи потрібно йому з ним працювати. У разі позитивного рішення сервер надсилає пристрою JWT, у якому записує додаткову інформацію про пристрій і його права доступу. Також доцільно при формуванні JWT вмістити в його корисне навантаження тип пристрою. Далі, аби надіслати дані серверу, клієнтське ПЗ пристрою додає JSON Web Token до HTTP-заголовку кожного запиту. При отриманні запиту, який містить токен, сервер лише перевіряє його цифровий підпис за допомогою відповідного алгоритму. У токені міститься вся необхідна інформація: *ідентифікатор пристрою*, що позначає, звідки отримані ті чи інші дані, та *тип пристрою*, за допомогою якого фронтальний сервер може виконати диспетчеризацію даних – надіслати повідомлення у відповідну тему в черзі повідомлень. Отже, не потрібно робити жодних додаткових запитів до бази даних, аби обробити повідомлення; це значно підвищує пропускну спроможність системи.

2.10 Схема розгортання системи. Інструменти керування хмарним кластером

Для розгортання платформи можна обрати один із хмарних хостингів, таких як Amazon Web Services або Google Cloud Platform. Але динамічне масштабування сервісів вимагає спеціальних рішень, які забезпечують роботу системи в цілому.

Перш за все, коли потрібно розгорнути більше за один фронтальний сервер, з’являється необхідність застосувати балансувальник навантаження, тобто інструмент розподілення навантаження по кільком серверам.

2.10.1 Балансування навантаження

Для балансування навантаження використаємо HAProxy.

HAProxy – серверне програмне забезпечення для забезпечення високої доступності та балансування навантаження для TCP і HTTP-застосунків шляхом розподілення вхідних запитів на декілька серверів [29]. Програма написана на C.

HAProxy має відкритий вихідний код і розповсюджується згідно з ліцензією GNU General Public License (GNU GPL v2).

Можливості HAProxy:

- Періодична перевірка доступності back-end серверів, на які переадресуються запити користувачів. Дуже корисно, бо ця функція виключає сервер з пулу, по якому розподіляються запити, щойно він перестає бути доступним із якихось причин.
- Підтримка HTTPS (TLS) і HTTP/2. Як уже було описано, захищений (TLS) варіант HTTP/2 – основний протокол комунікації клієнтського ПЗ із фронтальними серверами.
- Підтримка IPv6, HTTP стискання (deflate, gzip, libslz), постійного (persistent) HTTP з'єднання.
- Підтримка WebSocket.

Щодо продуктивності, то із тестів відомо, що HAProxy на серверах, оснащених процесором Xeon E5 (2014 року), здатен забезпечувати потік даних до 40-60 Гбіт/с. Він написаний максимально ефективно та використовує мінімум процесорного часу, тому наразі обмежувальним фактором є продуктивність мережевої карти, а не процесору чи програмного балансувальника навантаження.

При досягненні певної межі продуктивності HAProxy можна застосувати ще більш ефективне рішення, яке масштабується майже необмежено – **балансування на рівні DNS**. HAProxy забезпечує можливість закріплення сесій (session stickiness), яка дозволяє в межах сесії направляти запити конкретного клієнта на відповідний сервер, який обслуговує його сесію. У DNS-балансуванні такої функціональності нема, але платформа її і не потребує: уся архітектура, включно з автентифікацією, побудована так, щоби не зберігати стану між запитами. Отже, можна застосувати найпростіший алгоритм *Round robin DNS*, який пересилає запити по черзі серверам у списку.

2.10.2 Пошук сервісів

Якщо певне клієнтське програмне забезпечення викликає деякий сервер, то очевидно, що для цього йому потрібно знати його IP і порт. У традиційних програмах, що працюють на невеликій кількості власних серверів підприємства, мережеві локації сервісів є досить статичними. Їх можна зчитувати з деякого конфігураційного файлу, який час від часу оновлюють.

У сучасних хмарних системах із мікросервісною архітектурою дізнатися IP і порт потрібного сервіса набагато складніше. Мережеві адреси присвоюються серверам динамічно. До того ж, набір сервісів постійно змінюється через автоматичне масштабування чи відмови. Отож, потрібно мати більш складний механізм пошуку сервісів.

Відмінним рішенням є Consul. **Consul** – це розподілена система конфігурації та пошуку сервісів, яка має такі можливості [32]:

- *Пошук сервісів через DNS чи HTTP.* Одні клієнти Consul надають (реєструють) сервіс, а інші мають змогу зробити запит і знайти його.
- *Перевірка працездатності (health check).* Така інформація може використовуватися для моніторингу стану кластера чи для відвернення трафіка від сервера, у функціонуванні якого спостерігаються проблеми.
- *Сховище «ключ-значення».* Може використовуватися для конфігурування чи конфігурації сервісів.
- *Підтримка багатьох дата-центрів.*

На кожному вузлі, який надає сервіси Consul, працює *агент*. Він відповідальний за перевірку працездатності.

Агенти зв’язуються з одним чи більше *серверів* Consul. На серверах дані зберігаються та реплікуються. Із реплік обирається головна. Consul може функціонувати з одним сервером, але для уникнення сценаріїв відмови серверів, які призводять до втрати даних, рекомендується використовувати від 3 до 5. В кожному дата-центрі варто мати окремий кластер Consul.

Будь-які компоненти інфраструктури, яким потрібно знайти сервіс, можуть зробити запит до будь-якого сервера чи агента. Агенти автоматично перенаправляють запити до серверів.

Для формування відповіді на запити Consul вимагає наявності кворуму реплік для забезпечення сильної консистентності. При наявності розділення мережі, Consul жертвує доступністю, тобто з точки зору теореми CAP є CP системою.

Порівняно з іншими подібними рішеннями, основною перевагою Consul є те, як реалізована перевірка працездатності серверів. Більшість альтернативних систем роблять мережеві запити до сервісів для визначення їх статусу. Такі перевірки дають набагато менше інформації, ніж агенти Consul, які працюють на кожному сервері та мають можливість збирати набагато більш детальну системну інформацію. Традиційні реалізації перевірки працездатності також вимагають кількості роботи, яка лінійно залежить від кількості вузлів у кластері, а отже, погано масштабуються. Клієнти Consul використовують gossip protocol для передачі даних про статус серверів, який дає можливість масштабуватися до кластерів будь-яких розмірів без концентрації основного навантаження на певній групі серверів.

2.10.3 Розгортання і масштабування

Рішення про масштабування можуть приймати налаштовані інструменти хмарних хостингів, такі як **Amazon Auto Scaling** [30]. Amazon Auto Scaling може запускати нові сервери при перевищенні деякої границі навантаження CPU або використання оперативної пам’яті. Так само цей інструмент може зупиняти частину серверів, якщо кластер споживає мало ресурсів.

Після запуску сервера (початкового, в результаті масштабування чи при відновленні після збою) потрібно встановити на цьому сервері необхідне програмне забезпечення та запустити відповідний сервіс, тобто певний виконуваний код. Є два основних підходи до розгортання:

- Push розгортання. Після запуску серверу йому передають необхідний виконуваний код, наприклад, за допомогою rsync (Unix).
- Pull розгортання. Сервер стартує з певною конфігурацією, в якій указано, звідки можна скачати виконуваний код. Сервер його запитує в певного центрального сховища, збирає та запускає.

Push розгортання краще підходить для невеликих кластерів фіксованого розміру. Для динамічного масштабування треба використовувати pull підхід, поміщаючи виконуваний код сервісу в центральне сховище так, щоби при автоматичному запуску сервера він самостійно розгортав сервіс.

Для запуску сервісу необхідно також встановити необхідні залежності, такі як JRE для Java-застосунків. Залежностей може бути багато, тому цей аспект розгортання теж варто автоматизувати.

Зручним і функціональним рішенням для автоматизації розгортання є Docker. **Docker** – утиліта з відкритим вихідним кодом, яка автоматизує розгортання застосунків усередині програмних контейнерів, надаючи додатковий рівень абстракції та автоматизації виртуалізації на рівні ОС у Linux [31].

Docker реалізує високорівневий API для легких контейнерів, у яких ізольовано виконуються процеси. Побудований так, аби використовувати можливості, які надає ядро Linux (інструменти cgroups і namespaces), Docker-контейнер, на відміну від віртуальної машини, не потребує (та не включає) окремої операційної системи. Натомість він застосовує ізоляцію ресурсів (CPU, пам’яті, I/O, мережі тощо) та окремі простори імен, аби ізолювати застосунок в операційній системі. Це дозволяє створити для прикладних програм такий розріз операційної системи, який має власний простір ідентифікаторів процесів, а також структуру файлової системи та мережеві інтерфейси. Різні контейнери користуються одним ядром, але кожен з них може бути обмежений у використанні ресурсів.

Використання Docker для створення та керування контейнерами спрощує створення розподілених систем, дозволяючи програмам працювати автономно на окремих серверах або на кластері.

Екосистема Docker, окрім основного модуля Docker Engine, містить такі компоненти:

- *Docker Compose* – дозволяє простим чином описати та запустити конфігурацію багатоконтейнерної програми з усіма її залежностями;
- *Docker Swarm* – інструмент керування кластером, що дозволяє логічно об’єднати кілька Docker Engine в один;
- *Docker Registry* – утиліта для зберігання та розповсюдження Docker образів. Корисна для організації pull розгортання – із цього регістру сервери будуть скачувати образи.

Покажемо схему розгортання системи. На рис. 2.12 зображено діаграму розгортання (UML Deployment Diagram), яка описує фізичне розташування програмних компонентів платформи на серверах і зв’язки між ними.

Бачимо, що є окремий сервер – балансувальник навантаження, який розподіляє запити по фронтальних серверах. Є також сервер координації, на якому розташовується програмне забезпечення, що керує кластером. Consul забезпечує моніторинг і пошук сервісів для фронтальних серверів, а Zookeeper необхідний для внутрішніх потреб по координації вузлів кластеру Kafka. Робота MongoDB забезпечується одним master-сервером і одним-двома slave-серверами.

Важливим аспектом також є розташування Spark і Cassandra. Оскільки більшість даних для виконання обчислень береться з бази операційних даних, то доцільно на одному фізичному сервері запускати Spark Worker і вузол Cassandra. Spark, а також драйвер для його підключення до Cassandra (Spark Cassandra Connector), мають широкі можливості по організації обчислень таким чином, аби підвищувати локальність даних, тобто максимально зменшувати переміщення даних мережею.

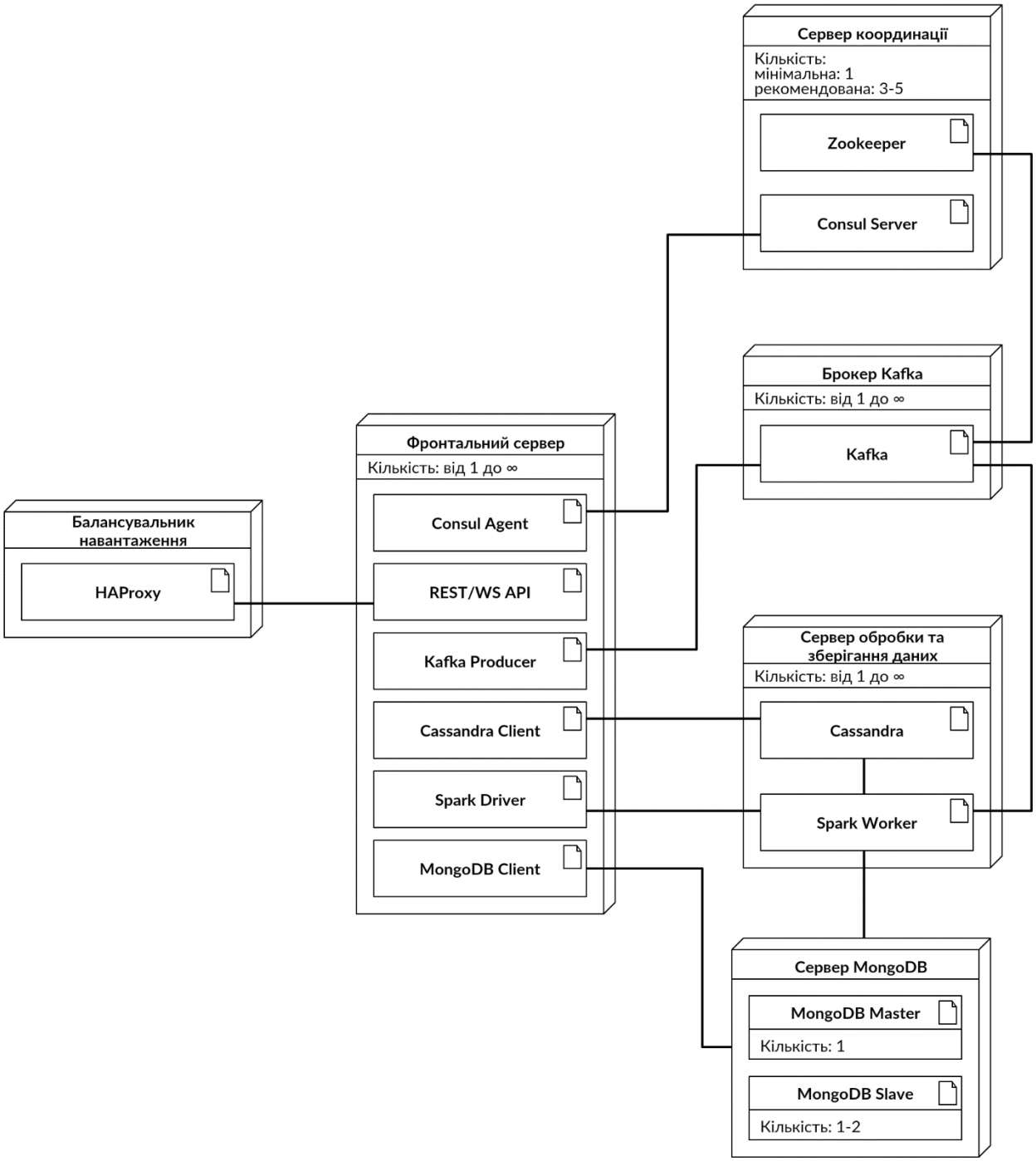

Рисунок 2.12 – Запропонована діаграма розгортання платформи

3. ПРОГРАМНА РЕАЛІЗАЦІЯ ПЛАТФОРМИ. АСПЕКТИ РЕАЛІЗАЦІЇ. ПРИКЛАДИ ВИКОРИСТАННЯ

Практичну реалізацію платформи було виконано мовами Java (версія 8) і Scala (версія 2.11). Програмний код основних компонентів наведено в додатку. Були використані такі бібліотеки:

- **Undertow** – написаний на Java веб-сервер, що є дуже ефективним завдяки застосуванню неблокуючого вводу/виводу. Дозволяє описувати логіку за допомогою окремих обробників, які обслуговують запити на певну адресу. За допомогою цієї бібліотеки було реалізовано фронтальні сервери, а саме HTTP (REST API) і WebSocket інтерфейси.
- **Kafka Clients** – клієнтські інтерфейси для взаємодії із кластером Apache Kafka.
- Набір бібліотек та інтерфейсів для роботи з **Apache Spark**.
- **Spark-Cassandra Connector** – драйвер Apache Cassandra з інтерфейсом, який інтегрований із API Apache Spark.
- **MongoDB Async Driver** – драйвер MongoDB, реалізований з використанням Java NIO для неблокуючих запитів до бази даних.
- Бібліотека для створення та перевірки **JSON Web Token**.
- **Apache Commons Codec** – набір реалізацій кодеків, наприклад, Base64 для формування JSON Web Token.
- **SLF4J** – Java API для логування різноманітних інформаційних і діагностичних повідомлень про хід роботи програми.
- **Logback** – реалізація SLF4J.
- **JUnit** – бібліотека для юніт-тестування.

3.1 REST API

Фронтальні сервери мають REST-інтерфейс із такими кінцевими точками (endpoint), як описано в таблиці 3.1. Указано відносні адреси, кожна з яких має префікс /api.

Таблиця 3.1 – Кінцеві точки REST API

Метод	**Адреса (шаблон адреси)**	**Опис**
GET	/auth/login/{login} /password/{password}	Використовується всіма користувачами системи для отримання токену автентифікації за вказаними даними доступу (параметри {login} і {password}).
POST	/data	Використовується пристроями для збереження даних у черзі повідомлень.
POST	/device	Додає новий пристрій.
GET	/device/{id}	Повертає дані про пристрій з указаним id (розташування, тип тощо).
PUT	/device/{id}	Змінює дані про пристрій з указаним id.
POST	/user	Додає нового користувача системи (адміністратора, аналітика тощо).
GET	/user/{id}	Повертає дані про вказаного користувача (права доступу тощо).
PUT	/user/{id}	Оновлює дані про користувача з відповідним id.
GET	/report/device/{id}	Повертає дані зі вказаного пристрою, оброблені відповідним чином.
POST	/device/{id}/command	Викликається адміністратором для запису команди для пристрою.

Таблиця 3.1 (закінчення) – Кінцеві точки REST API

Метод	**Адреса (шаблон адреси)**	**Опис**
GET	/device/{id}/command	Повертає актуальні команди, призначені вказаному пристрою.
PUT	/device/{id}/command/{id}	Оновлює команду зі вказаним id. Наприклад, пристрій указує результат її виконання.
POST	/device/{id}/status	Записує статусне повідомлення пристрою. Викликається пристроєм або тригером моніторингу.
GET	/device/{id}/status	Викликається адміністратором для перегляду актуальних статусних повідомлень пристрою.
PUT	/device/{id}/status/{id}	Оновлює статусне повідомлення зі вказаним id, наприклад, для запису відмітки про його перегляд адміністратором.

3.2 Структура бази адміністративних даних

База адміністративних даних, побудована на MongoDB, має дві колекції: користувачів і пристроїв.

Колекція користувачів зберігає інформацію про їхні права доступу. Хоча MongoDB підтримує динамічну схему, основний набір полів у документах є незмінним. Наведемо структуру документів у цій колекції:

{

"login": String,

"password": String,

"role": String,

"info": Object

}

Вищенаведена структура описана в ієрархічному форматі, подібному JSON, у якому записано ключі – назви полів у документі, та значення – типи даних у цих полях.

Поля “login” і “password” мають рядковий тип і призначені для автентифікації.

Поле “role” визначає роль користувача, а відповідно і його права доступу. Наприклад, користувач може бути адміністратором, аналітиком тощо. У реалізації на Java доцільніше зберігати роль користувача у вигляді переліченого типу даних (enum), але MongoDB його не підтримує. Можна було б зберігати enum у MongoDB як число, але рядковий тип є більш наочним, а те, що він є менш компактним, не створює проблем через невелику кількість записів у цій таблиці.

У полі “info” можна (опціонально) зберегти додаткову інформацію про користувача у вигляді об’єкту з довільними полями.

Колекція пристроїв містить інформацію про пристрої, яка використовується для їх автентифікації та різноманітних запитів. Структура документів у цій колекції така:

{

"device_id": String,

"password": String,

"type": String,

"status_messages": Array<Object>,

"commands": Array<Object>,

"location": GeoJSON,

"info": Object

}

Поля “device_id” і “password” використовуються для автентифікації. “type” – тип пристрою, enum. “location” містить координати пристрою у форматі

GeoJSON. Поля “status_messages” і “commands” містять відповідно статусні повідомлення і команди пристрою. “info” – довільний об’єкт з додатковою інформацією.

Опис **статусних повідомлень** має такий вигляд:

{

"created": Timestamp,

"by": String,

"type": String,

"payload": Object,

"received": Timestamp

}

“created” і “received” – відповідно часові мітки створення запису та його отримання адміністратором. “by” – відправник повідомлення (підсистема моніторингу чи пристрій). “type” – тип повідомлення (наприклад, інформаційне чи повідомлення про помилку). “payload” – довільний об’єкт із описом умісту повідомлення.

Об’єкти, які описують **команди**, загалом мають таку структуру:

{

"created": Timestamp,

"payload": Object,

"received": Timestamp,

"result": Object

}

Тобто містять дату та час створення (запису в базу) – “created”; часову мітку моменту отримання пристроєм – “received”; об’єкт із довільним набором полів, що описує команду – “payload”; та об’єкт із динамічною структурою, що описує результат виконання команди пристроєм – “result”.

3.3 Приклад використання системи

Опишемо приклад використання платформи з мережею метеорологічних станцій. Загальна архітектура системи в даному прикладі відповідає тій, яка зображена на рис. 2.1.

Для того, аби пристрої могли почати роботу з платформою, потрібно занести дані про них, виконавши запит **POST /device** і помістивши опис пристрою в тіло запиту. Наприклад, він може виглядати так:

```
{
    "device_id": "weather-1",
    "password": "$2a$10$N9qo8uLO",
    "type": "weather",
    "location": {
        "type": "Point",
        "coordinates": [50.417329, 30.526006]
    }
}
```

Потім пристрій робить запит **GET /auth/login/{device_id}/password/{password}** і отримує JSON Web Token, що містить “device_id”. Після цього пристрій може надсилати операційні дані до платформи.

Нехай запит **POST /data** виглядає таким чином:

```
{
    “data”: {
        “temperature”: 19.852,
        “pressure”: 752.1
    }
}
```

Тобто метеорологічна станція повідомляє значення температури та тиску в узгоджених одиницях виміру.

При записі в чергу повідомлення ключем повідомлення є “device_id”, що береться із токену автентифікації. Даними повідомлення є значення поля “data” у вищенаведеному JSON об’єкті.

Нехай логіка потокової обробки даних полягає у віконному обчисленні базових статистичних параметрів даних: кількості записів, середнього значення та стандартного відхилення відповідної величини.

Ланцюжок потокової обробки даних у Apache Spark виглядатиме так:

1. Виконати парсинг JSON і повернути пару (кортеж), що складається з ключа повідомлення та об’єкту з відповідними числами з рухомою комою – значеннями температури та тиску.
2. Застосувати операцію *reduceByKeyAndWindow*, яка в межах указаного вікна та для записів з однаковим ключем редукує множину об’єктів в один, що містить вищезазначені статистичні параметри даних.
3. Привести результат обчислень до вигляду, придатного для збереження в базу.

Для описаної задачі наведемо структуру бази операційних даних. Apache Cassandra дозволяє визначати власні типи даних (UDT – user-defined types). Скористаємося цією можливістю та опишемо тип даних, що містить потрібні статистичні параметри:

```
CREATE TYPE iot.stats (count BIGINT, avg DOUBLE, std_dev DOUBLE)
```

Наведений запит створює тип “stats” у просторі ключів “iot”, що містить кількість об’єктів (64-бітне ціле), середнє значення та стандартне відхилення (64-бітні числа з рухомою комою).

Створимо також таблицю для зберігання даних, що пройшли первинну обробку:

```
CREATE TABLE IF NOT EXISTS iot.weather
(device_id TEXT,
timestamp TIMESTAMP,
temperature_stats FROZEN<stats>,
pressure_stats FROZEN<stats>,
```

PRIMARY KEY (device_id, timestamp))

Маємо відповідно рядковий ідентифікатор пристрою, час запису результату обробки пакету даних та статистичні параметри даних температури та тиску, які мають створений раніше тип (ключове слово FROZEN потрібне для використання UDT). Первинний ключ таблиці є складеним і містить у собі id пристрою та мітку часу запису.

Наприклад, пристрій надсилає значення температури та тиску кожної секунди. Якщо довжина вікна встановлена на одну хвилину, то щохвилинно в базу надходитимуть такі записи, як показано у табл. 3.2.

Таблиця 3.2 – Приклад даних, що пройшли первинну обробку

device_id	**timestamp**	**temperature_stats**	**pressure_stats**
weather-1	2016-05-07 14:39:10.036	{count: 60, avg: 24.89167, std_dev: 0.975773}	{count: 60, avg: 753.21864, std_dev: 5.336478}

Для виконання аналітичного запиту, який вимагає агрегації за більшими інтервалами, наприклад, для обчислення середньої температури за день, виконується операція редукції по ключу, у якій ключем вважається порядковий номер часового інтервалу потрібної довжини, відрахований від 00:00:00 01.01.1970 (Unix epoch).

Для аналізу метеорологічних даних у конкретній місцевості, платформа спочатку зробить геопросторовий запит до бази адміністративних даних для пошуку ідентифікаторів пристроїв, які знаходяться в розглядуваній місцевості, а потім зчитає з бази операційних даних показання пристроїв із відповідними id.

3.4 Приклад взаємодії платформи з Raspberry Pi

Для демонстрації можливостей платформи до неї було підключено мікрокомп’ютер Raspberry Pi 3. Він має операційну систему на базі Linux, а

конкретніше, дистрибутив Ubuntu MATE. Це дає можливість розробити скрипт, що взаємодіє із платформою, мовою Python, оскільки більшість ОС Unix містять інтерпретатор для неї, або його нескладно встановити.

Розглянемо код детальніше. Задамо адресу фронтального сервера, а також логін і пароль пристрою, зареєстрованого в системі:

```
domain = 'http://localhost:8080'
login = 'RaspberryPi'
password = 'pass'
```

Для HTTP-запитів застосуємо бібліотеку **requests**. Спочатку виконаємо запит GET для отримання токену автентифікації:

```
import requests
r = requests.get('{0}/api/auth/login/{1}/password/{2}'.format(domain, login, password))
```

Виконаємо парсинг JSON-відповіді та отримаємо значення поля, що містить токен:

```
token = r.json()['token']
```

Для подальшого використання сформуємо заголовок, який буде застосовуватися для автентифікації при надсиланні даних:

```
auth_header = {'Authorization': 'Bearer {0}'.format(token)}
```

Створимо функцію, яка отримує температурні дані. У якості сенсора використаємо термометр, що вимірює температуру процесора мікрокомп’ютера:

```
def get_cpu_temperature():
    res = os.popen('vcgencmd measure_temp').readline()
    return float(res.replace("temp=", "").replace("'C\n", ""))
```

Визначена функція виконує shell-команду vcgencmd, яка є частиною firmware Raspberry Pi, та виконує парсинг результату.

Тепер реалізуємо функцію, яка надсилає дані платформі:

```
def send_data():
    temperature_data = get_cpu_temperature()
    data = {'data': {'temperature': temperature_data}}
```

print('{0}. Sending data: {1}'.format(datetime.datetime.utcnow(), str(temperature_data)))

requests.post('{0}/api/data'.format(domain), data=json.dumps(data), headers=auth_header)

Означена функція спершу отримує величину температури (число). Потім формується об’єкт з даними у вигляді словника (dictionary), структура якого відповідає JSON. Далі записуємо в консоль інформацію про час відправки повідомлення та надіслане значення. Наприкінці функції виконуємо POST запит на адресу прийому операційних даних, вказавши у тілі запиту JSON представлення повідомлення, а в заголовку – токен автентифікації.

Визначимо функцію, яка надсилає дані з вказаним інтервалом:

```
def send_data_continuously(interval):
    while True:
        send_data()
        time.sleep(interval)
```

Викличемо її:

```
send_interval_seconds = 0.025
send_data_continuously(send_interval_seconds)
```

Тепер можна помістити вказаний код у файл. Тоді для початку роботи пристрою з платформою достатньо буде лише скопіювати його в пам’ять і запустити цей скрипт.

Під час роботи скрипт записує в лог такі повідомлення:

2016-05-17 14:37:48.527367. Sending data: 56.52267634619887

2016-05-17 14:37:48.561703. Sending data: 54.90458634655109

2016-05-17 14:37:48.595569. Sending data: 53.406799712390935

Оброблені (проагреговані) дані отримуються за допомогою GET-запиту на **/report/device/{id}** та зображуються в браузері на графіку:

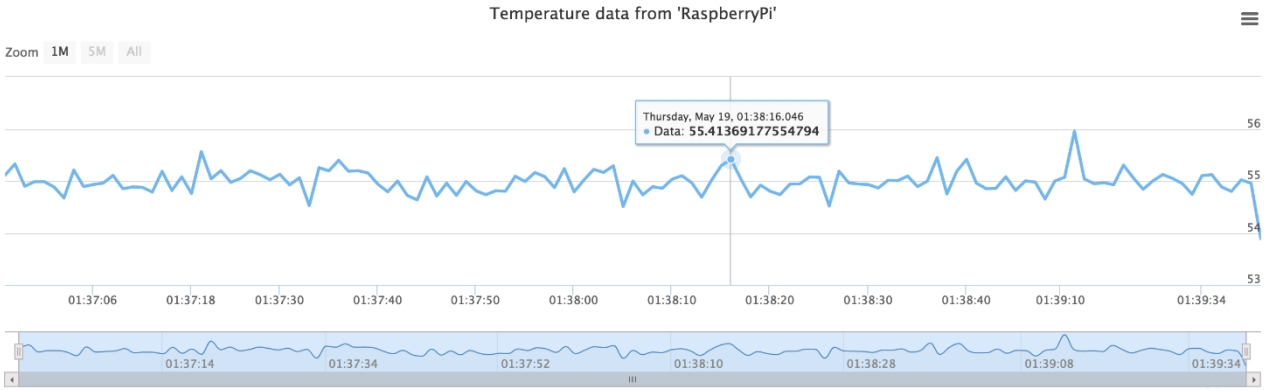

Рисунок 3.1 – Візуалізація оброблених даних

Уся тестова система показана на фотографії (рис. 3.2). В лівому кутку в білій коробці знаходиться Raspberry Pi, який під’єднано до монітору в центрі. На ньому зображується процес роботи клієнтського скрипта. На моніторі праворуч видно графік, який показує проагреговані дані, що надійшли від пристрою.

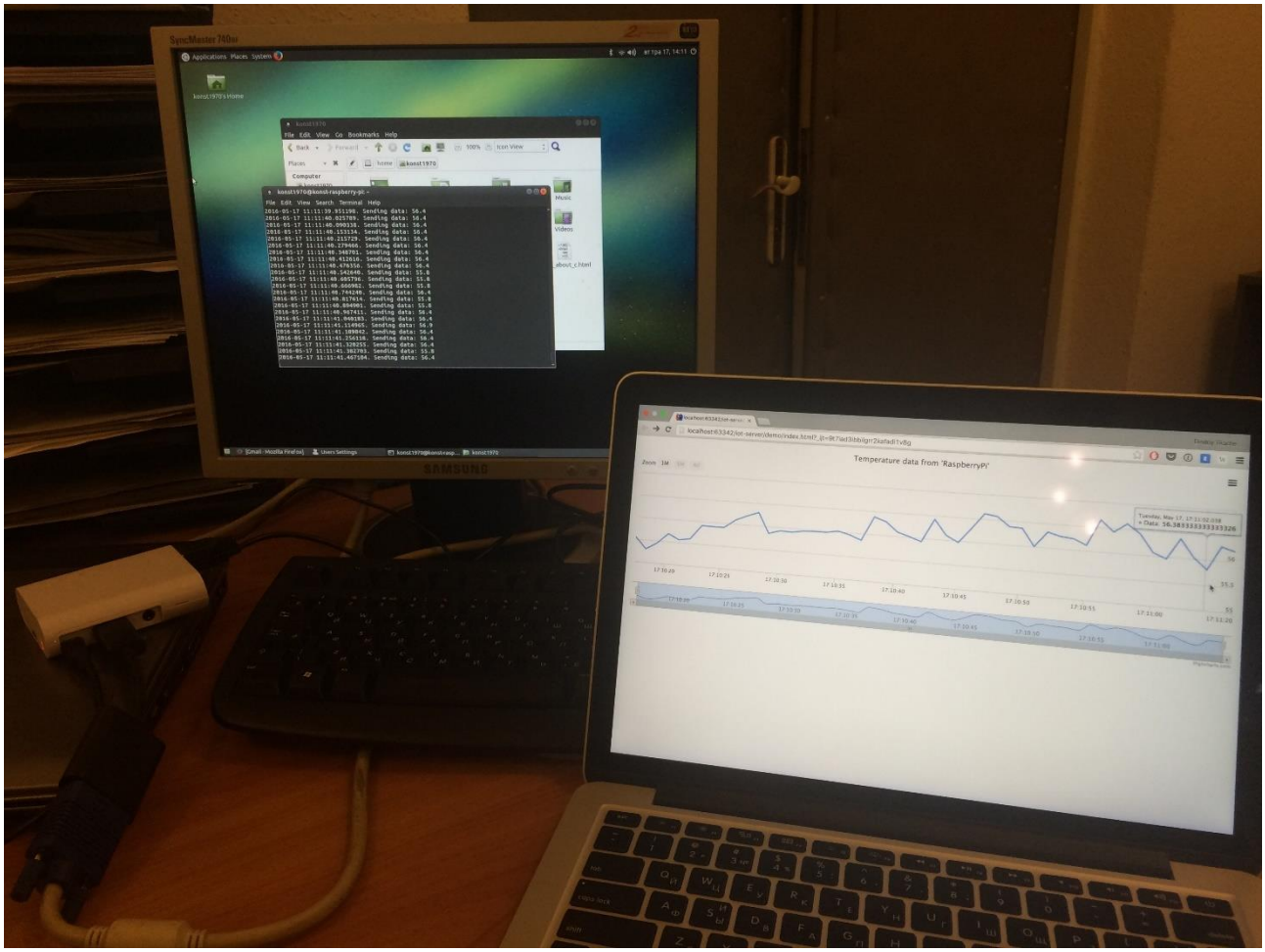

Рисунок 3.2 – Тестова система

4. ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

У даному розділі проводиться оцінка основних характеристик хмарної платформи для Інтернету речей, призначеної для прийому, обробки та зберігання даних, що надходять від різноманітних пристроїв із сенсорами. Програмний продукт був розроблений мовами програмування Java і Scala та працює на платформі Java Virtual Machine (JVM).

Розроблений програмний продукт є крос-платформенним, а отже може працювати на серверах під управлінням операційних систем Linux, Mac, Windows.

Нижче наведено аналіз різних варіантів реалізації модулю 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 на основі 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.

4.1 Постановка задачі

У роботі застосовується метод ФВА для проведення техніко-економічного обґрунтування розробки хмарної платформи для Інтернету речей. Основні проектні рішення визначаються відповідно до вимог до всієї системи, тому її складові підсистеми мають їм задовольняти. Тому фактичний аналіз представляє собою аналіз функцій програмного продукту, призначеного для прийому, обробки та зберігання даних, що надходять від пристроїв мережі Інтернету речей.

Відповідно цьому варто обирати і систему показників якості програмного продукту.

Технічні вимоги до продукту наступні:

- програмний продукт повинен функціонувати на звичайних серверах (commodity hardware) із стандартним набором компонент і поширеними в сучасних дата-центрах характеристиками;
- забезпечувати високу швидкість обробки великих обсягів даних, як у реальному часі (потокова обробка), так для виконання аналітичних запитів;
- забезпечувати зручність і простоту взаємодії з користувачами системи, а також із розробниками програмного забезпечення у випадку, якщо вони

займаються написанням додатків до платформи чи інтегрують її в інші системи певного підприємства;

- передбачати мінімальні витрати на впровадження програмного продукту.

4.1.1 Обґрунтування функцій програмного продукту

Головна функція F_0 – розробка хмарної платформи, яка виконує прийом, обробку та зберігання даних, що надійшли від пристроїв мережі Інтернету речей. Відповідно до цієї мети, можна виділити такі основні функції ПП:

F_1 – вибір мов програмування, що використовуються для реалізації платформи;

F_2 – вибір СКБД, що використовується для зберігання операційних даних;

F_3 – вибір підсистеми обробки даних.

Кожна з основних функцій може мати декілька варіантів реалізації.

Функція F_1:

а) мова програмування Python;

б) мови програмування, що працюють на платформі JVM, – Java та Scala.

Функція F_2:

а) фреймворк Apache Hadoop;

б) фреймворк Apache Spark.

Функція F_3:

а) розподілена нереляційна база даних Apache Cassandra;

б) реляційна база даних PostgreSQL.

4.1.2 Варіанти реалізації основних функцій

Варіанти реалізації основних функцій наведені у морфологічній карті системи (рис. 4.1). На основі цієї карти побудовано позитивно-негативну матрицю варіантів основних функцій (таблиця 4.1).

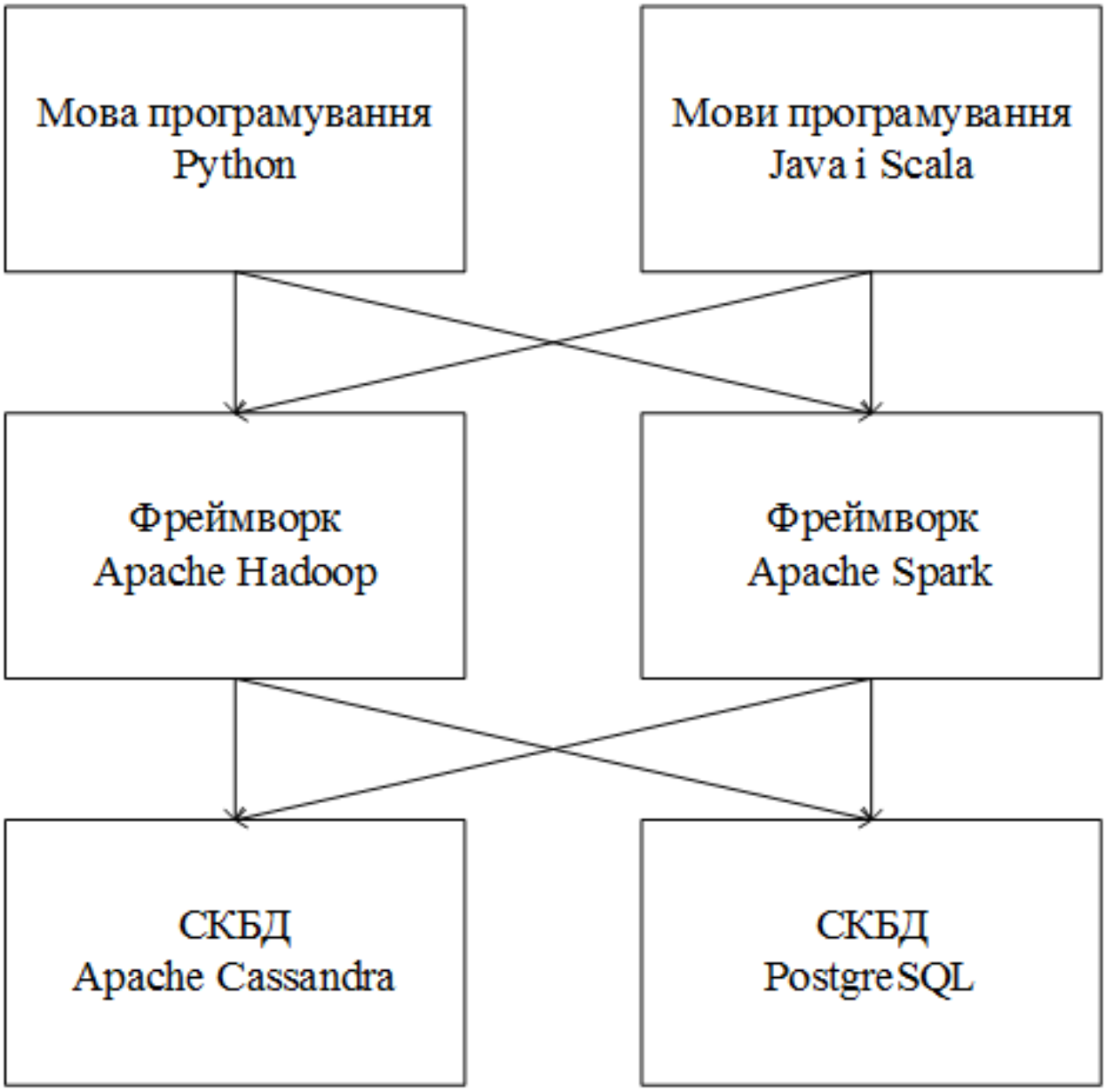

Рисунок 4.1 – Морфологічна карта

Морфологічна карта відображує всі можливі комбінації варіантів реалізації функцій, які складають повну множину варіантів ПП.

Таблиця 4.1 – Позитивно-негативна матриця

Основні функції	**Варіанти реалізації**	**Переваги**	**Недоліки**
F1	*А*	Більш простий код, який легше та швидше писати, а також дешевше підтримувати	Низька швидкість виконання коду; більша ймовірність появи помилок через динамічну типізацію
	Б	Висока швидкість виконання; менше помилок завдяки статичній типізації; багато наявних бібліотек	Дещо вищі витрати часу на розробку коду
F2	*А*	Наявно декілька реалізацій від різних постачальників; висока стабільність і гарна документація; наявність великої кількості готових реалізацій ланцюжків обчислень у парадигмі MapReduce	Повільні обчислення через активне використання диску; дещо обмежені можливості обчислень у парадигмі MapReduce; складність реалізації багатопрохідних алгоритмів
	Б	Дуже висока швидкість виконання обчислень завдяки використанню оперативної пам’яті; ширші можливості побудови ланцюжка обчислень	Дещо менша популярність і стабільність, що підвищує вартість реалізації

Таблиця 4.1 (закінчення) – Позитивно-негативна матриця

Основні функції	**Варіанти реалізації**	**Переваги**	**Недоліки**
F3	*А*	Здатна працювати з величезними обсягами даних завдяки масштабованості та високій ефективності	Несумісна з реляційними БД; вища вартість розробки через меншу стабільність бібліотек і менш популярну (ніж SQL) мову запитів
	Б	Популярна та функціональна мова запитів; стабільні бібліотеки	При збільшенні кількості даних значно зменшується продуктивність; не здатна масштабуватися на декілька серверів

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.

Функція *F1:*

Оскільки продуктивність платформи має велике значення, швидкість виконання коду є критичною, і з цих причин варіант а) має бути відкинутий.

Функція *F2:*

Обидва варіанти загалом підходять, бо хоча варіант а) і менш продуктивний, він має інші переваги, так що його не можна однозначно відкинути.

Функція *F3:*

Оскільки платформа має зберігати гігантські обсяги даних для обслуговування мереж з великою кількістю пристроїв, відсутність можливості горизонтально масштабуватися робить варіант б) непридатним.

Таким чином, будемо розглядати такі варіанти реалізації ПП:

1. F1б – F2а – F3а

2. F1б – F2б – F3а

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

4.2 Обґрунтування системи параметрів ПП

4.2.1 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 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.

4.2.2 Кількісна оцінка параметрів

Гірші, середні і кращі значення параметрів вибираються на основі вимог замовника й умов, що характеризують експлуатацію ПП як показано у табл. 4.2.

Таблиця 4.2 – Основні параметри ПП

Назва параметра	**Умовні позначення**	**Одиниці виміру**	**Значення параметра**		
			гірші	**середні**	**кращі**
Швидкодія мови програмування	X1	оп/мс	2000	11000	19000
Об’єм пам’яті для збереження даних	X2	Мб	32	16	8
Час обробки даних алгоритмом	X3	мс	800	420	60
Потенційний об’єм програмного коду	X4	кількість рядків коду	2000	1500	1000

За даними таблиці 4.2 будуються графічні характеристики параметрів – рис. 4.2 – рис. 4.5.

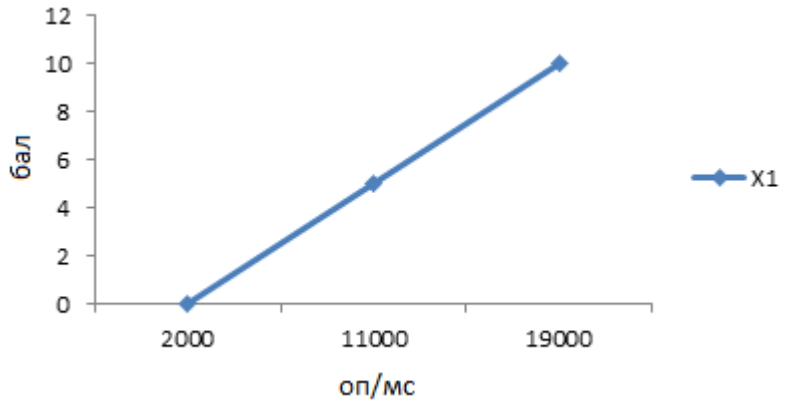

Рисунок 4.2 – X1, швидкодія мови програмування

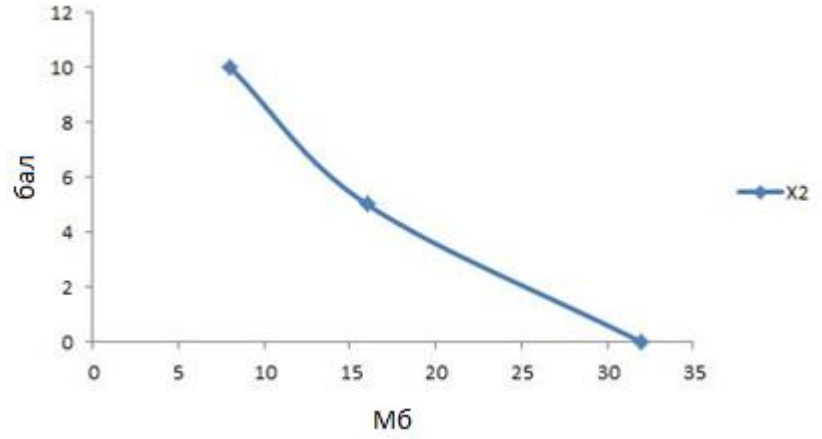

Рисунок 4.3 – X2, об’єм пам’яті для збереження даних

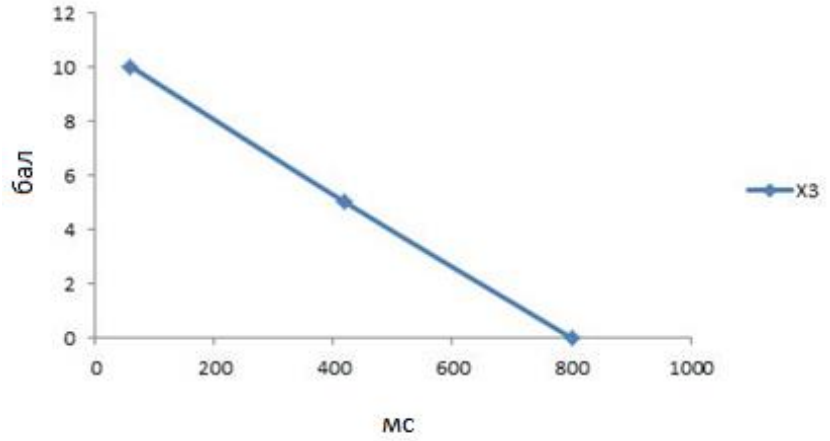

Рисунок 4.4 – X3, час обробки даних алгоритмом

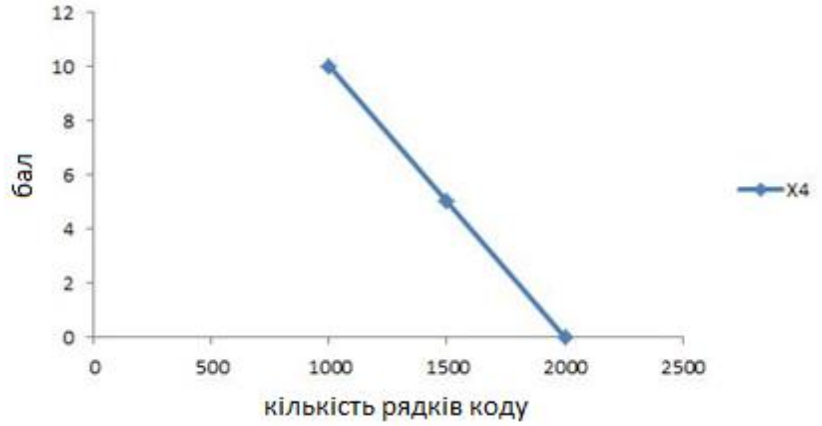

Рисунок 4.5 – X4, потенційний об’єм програмного коду

4.2.3 Аналіз експертного оцінювання параметрів

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка

хмарної платформи, яка реалізує основні підсистеми: прийому, зберігання та обробки даних, які надійшли від пристроїв із сенсорами в мережі Інтернету речей.

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 4.3.

Таблиця 4.3 – 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	4	3	4	4	4	4	4	27	0,75	0,56
X2	Об’єм пам’яті для збереження даних	Мб	4	4	4	3	4	3	3	25	-1,25	1,56
X3	Час обробки даних алгоритмом	мс	2	2	1	2	1	2	2	12	-14,25	203,06
X4	Потенційний об’єм програмного коду	кількість рядків коду	5	6	6	6	6	6	6	41	14,75	217,56
	Разом		15	15	15	15	15	15	15	105	0	420,75

Для перевірки степені достовірності експертних оцінок, визначимо наступні параметри:

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

$$R_i = \sum_{j=1}^{N} r_{ij} R_{ij} = \frac{Nn(n+1)}{2} = 105,$$

де N – число експертів, n – кількість параметрів;

б) середня сума рангів:

$$T = \frac{1}{n} R_{ij} = 26{,}25.$$

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:

$$S = \sum_{i=1}^{N} \Delta_i^2 = 420{,}75.$$

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 420{,}75}{7^2(5^3 - 5)} = 1{,}03 > W_k = 0{,}67$$

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 4.4.

Таблиця 4.4 – Попарне порівняння параметрів

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	=	>	=	<	=	<	<	<	0,5
X1 і X3	<	<	<	<	<	<	<	<	0,5
X1 і X4	>	>	>	>	>	>	>	>	1,5
X2 і X3	<	<	<	<	<	<	<	<	0,5
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 А=$\| a_{ij} \|$.

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 4.5, різниця значень коефіцієнтів вагомості не перевищує 2%, тому більшої кількості ітерацій не потрібно.

Таблиця 4.5 – Розрахунок вагомості параметрів

Параметри x_i	Параметри x_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	0,5	1,5	3,5	0,219	22,25	0,216	100	0,215
X2	1,5	1,0	0,5	1,5	4,5	0,281	27,25	0,282	124,25	0,283
X3	1,5	1,5	1,0	1,5	5,5	0,344	34,25	0,347	156	0,348
X4	0,5	0,5	0,5	1,0	2,5	0,156	14,25	0,155	64,75	0,154
Всього:					16	1	98	1	445	1

4.3 Аналіз рівня якості варіантів реалізації функцій

Визначаємо рівень якості кожного варіанту виконання основних функцій окремо.

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

Варіант а) більш простий для реалізації: при його використанні потрібно 1150 рядків коду (*X4*), у той час як варіант б) вимагає 1700 рядків. Але за умови вибору варіанту а) платформа працювати повільніше: час обробки даних (*X3*) становитиме 600 мс, тоді як варіант б) забезпечує обробку даних за 200 мс.

Коефіцієнт технічного рівня для кожного варіанта реалізації ПП розраховується так (таблиця 4.6):

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

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

Таблиця 4.6 – Розрахунок показників рівня якості варіантів реалізації основних функцій ПП

Основні функції	Варіант реалізації функції	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1(X1)	А	11000	5	0,215	1,075
F2(X3)	А	600	3	0,348	1,044
	Б	200	8		2,784
F2(X4)	А	1150	8,5	0,154	1,309
	Б	1700	3		0,462
F3(X2)	Б	16	5	0,283	1,415

За даними з таблиці 4.6 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} = 1,075 + 1,044 + 1,309 + 1,415 = 4,843

K_{K2} = 1,075 + 2,784 + 0,462 + 1,415 = 5,736

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

4.4 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 реалізації платформи.

При використанні варіанту а) з’являється додаткова задача до завдання 2):

3. Оптимізація використання диску при обробці даних.

Варіант б) вимагає виконання такої підзадачі для виконання завдання 2):

4. Реалізація алгоритму обробки даних.

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 2.

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

$$T_O = T_P \cdot K_П \cdot K_{СК} \cdot K_М \cdot K_{СТ} \cdot K_{СТ.М}, \quad (4.1)$$

де T_P – трудомісткість розробки ПП; $K_П$ – поправочний коефіцієнт; $K_{СК}$ – коефіцієнт на складність вхідної інформації; $K_М$ – 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 чотирьох 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 4.1, 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 другої групи складності, степінь новизни Б), тобто $Т_Р$ = 27 людино-днів, $К_П$ = 0,9, $К_{СК}$ = 1, $К_{СТ}$ = 0,8:

$Т_2 = 27 \cdot 0,9 \cdot 0,8 = 19,44$ людино-днів.

Для третього завдання (використовується алгоритм третьої групи складності, ступінь новизни Г), тобто $Т_Р$ = 8 людино-днів, $К_П$ = 0,3, $К_{СК}$ = 1, $К_{СТ}$ = 0,8:

$Т_3 = 8 \cdot 0,3 \cdot 0,8 = 1,92$ людино-днів.

Для четвертого завдання (використовується алгоритм другої групи складності, степінь новизни В), тобто $Т_Р$ = 19 людино-днів, $К_П$ = 0,6, $К_{СК}$ = 1, $К_{СТ}$ = 0,8:

$Т_4 = 19 \cdot 0,6 \cdot 0,8 = 9,12$ людино-днів.

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

$Т_I = (122,4 + 19,44 + 1,92) \cdot 8 = 1150,08$ людино-годин;

$Т_{II} = (122,4 + 19,44 + 9,12) \cdot 8 = 1207,68$ людино-годин.

Найвищу трудомісткість має варіант II.

В розробці беруть участь два програмісти з окладом 7000 грн., один фінансовий аналітик з окладом 9500 грн. Визначимо зарплату за годину за формулою:

$$С_Ч = \frac{М}{T_m \cdot t} \text{ грн.},$$

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

$$С_Ч = \frac{7000 + 7000 + 9500}{3 \cdot 21 \cdot 8} = 46,62 \text{ грн.}$$

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. $С_{ЗП} = 46,62 \cdot 1150,08 \cdot 1,2 = 64340,08$ грн.

II. $С_{ЗП} = 46,62 \cdot 1207,68 \cdot 1,2 = 67562,45$ грн.

Відрахування на єдиний соціальний внесок становить 22%:

I. $С_{ВІД} = С_{ЗП} \cdot 0,22 = 64340,08 \cdot 0,22 = 14154,82$ грн.

II. $С_{ВІД} = С_{ЗП} \cdot 0,22 = 67562,45 \cdot 0,22 = 14883,54$ грн.

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

Так як одна ЕОМ обслуговує одного програміста з окладом 7000 грн., з коефіцієнтом зайнятості 0,2 то для однієї машини отримаємо:

$С_Г = 12 \cdot М \cdot К_З = 12 \cdot 7000 \cdot 0,2 = 16800$ грн.

З урахуванням додаткової заробітної плати:

$С_{ЗП} = С_Г \cdot (1+ К_З) = 16800 \cdot (1 + 0,2) = 20160$ грн.

Відрахування на єдиний соціальний внесок:

$С_{ВІД} = С_{ЗП} \cdot 0,22 = 20160 \cdot 0,22 = 4435,2$ грн.

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 10000 грн.

$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР} = 1,15 \cdot 0,25 \cdot 10000 = 2875$ 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:

$С_Р = К_{ТМ} \cdot Ц_{ПР} \cdot К_Р = 1,15 \cdot 10000 \cdot 0,05 = 575$ 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:

$С_{ЕЛ} = Т_{ЕФ} \cdot N_С \cdot К_З \cdot Ц_{ЕН} = 1706,4 \cdot 0,156 \cdot 0,2 \cdot 2,0218 = 107,64$ грн.,

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

Накладні витрати розраховуємо за формулою:

$С_Н = Ц_{ПР} \cdot 0,67 = 10000 \cdot 0,67 = 6700$ грн.

Тоді, річні експлуатаційні витрати будуть:

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

$$С_{ЕКС} = 20160 + 4435,2 + 2875 + 575 + 107,64 + 6700 = 34852,84 \text{ грн.}$$

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$С_{М\text{-}Г} = С_{ЕКС} / Т_{ЕФ} = 34852,84 / 1706,4 = 20,42$ 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:

$$С_М = С_{М\text{-}Г} \cdot Т$$

I. $С_М = 20,42 \cdot 1150,08 = 23484,63$ грн.;

II. $С_М = 20,42 \cdot 1207,68 = 24660,83$ грн.

Накладні витрати складають 67% від заробітної плати:

$$С_Н = С_{ЗП} \cdot 0,67$$

I. $С_Н = 64340,08 \cdot 0,67 = 43107,85$ грн.;

II. $С_Н = 67562,45 \cdot 0,67 = 45266,84$ грн.

Отже, вартість розробки ПП за варіантами становить:

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

I. $С_{ПП} = 64340,08 + 14154,82 + 23484,63 + 43107,85 = 145087,38$ грн.;

II. $С_{ПП} = 67562,45 + 14883,54 + 24660,83 + 45266,84 = 152463,66$ грн.

4.5 Вибір кращого варіанта ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$К_{ТЕРj} = К_{Кj} / С_{Фj},$$

$К_{ТЕР1} = 4,843 / 145087,38 = 3,34 \cdot 10^{-5}$;

$К_{ТЕР2} = 5,736 / 152463,66 = 3,76 \cdot 10^{-5}$

Як бачимо, найбільш ефективним є другий варіант реалізації програми з коефіцієнтом техніко-економічного рівня $К_{ТЕР2} = 3,76 \cdot 10^{-5}$.

4.6 Висновки до розділу 4

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 перший варіант реалізації програмного продукту. У нього виявився найкращий показник техніко-економічного рівня якості $К_{ТЕР} = 3,76 \cdot 10^{-5}$.

Цей варіант реалізації програмного продукту має такі параметри:

- мови програмування, що працюють на платформі JVM, – Java та Scala;
- фреймворк Apache Spark;
- розподілена нереляційна база даних Apache Cassandra.

ВИСНОВКИ

У ході виконання даного дипломного проекту було досліджено та побудовано платформу для Інтернету речей.

Зокрема, було виділено основні вимоги, що висуваються до подібних систем; вивчено основні підходи до вирішення задач обробки великих за обсягами даних; досліджено існуючі платформи та виокремлено шляхи їх покращення.

Розроблена платформа задовольняє поставленим функціональним і нефункціональним вимогам та містить реалізацію таких підсистем: приймання, зберігання, обробки операційних даних; автентифікації та безпеки; роботи з адміністративними даними; моніторингу. Це дозволяє застосування її такими виділеними групами користувачів: пристроями, адміністраторами, аналітиками і адміністраторами безпеки.

Було запропоновано та реалізовано таку розподілену сервісно-орієнтовану (мікросервісну) архітектуру платформи, яка забезпечує високу продуктивність і майже нескінченну масштабованість. Це дозволяє нарощувати потужність системи для обслуговування мереж пристроїв будь-яких розмірів і для збільшення можливостей підсистеми аналітики даних. Ефективна реалізація компонентів системи дає змогу оптимізувати витрати на апаратне забезпечення платформи.

Платформа також гарантує значний рівень відмовостійкості, що знижує ймовірність втрати операційних даних і забезпечує високий рівень її доступності. Досліджені та реалізовані аспекти безпеки надають достатній рівень захищеності даних при передачі й убезпечують систему від різноманітних інформаційних атак.

У ході розробки платформи були з належним обґрунтуванням відібрані та застосовані сучасні системи, протоколи, формати даних, бібліотеки та бази

даних. Використанні сторонні рішення мають відкритий вихідний код, що спрощує розширення системи, має економічні та інші переваги.

Було детально описано різноманітні аспекти обраних архітектурних рішень: від обґрунтування високорівневого поділу на функціональні модулі до особливостей ефективної реалізації вводу/виводу та впливу обраної схеми автентифікації на масштабованість архітектури платформи.

На відміну від існуючих платформ, розроблене рішення є добре розширюваним у тому сенсі, що його можливо легко адаптувати під аналітику даних, які надходять від різноманітних пристроїв і сенсорів, задаючи ланцюжки обробки даних. Протоколи та формати даних було обрано не лише з огляду на їх ефективність, а ще й враховуючи зручність і простоту розширення платформи.

Іншими ключовими особливостями платформи є моніторинг і функціональність надійної передачі службової інформації між пристроями та адміністраторами. Ці можливості дозволяють ефективно керувати мережею пристроїв, а також гнучко налаштовувати інструменти попередження про можливі неполадки та оперативно на них реагувати.

Запропоновані та застосовані в роботі архітектурні та технічні рішення є достатньо універсальними, що дозволяє використовувати їх для побудови різноманітних систем обробки великих даних та інших платформ для Інтернету речей, спеціалізованих для більш вузьких областей.

Роботу платформи було показано за допомогою демонстраційного прикладу, в якому моделюється робота системи з мережею метеорологічних станцій. Також до платформи було підключено мікрокомп’ютер Raspberry Pi 3 та продемонстровано обробку даних із його сенсора. Ці приклади охоплюють більшість можливостей системи та показують її застосування всіма групами користувачів. Описано реалізацію на боці клієнта процесів реєстрації та автентифікації пристрою, а також організацію відправки даних до сервера.

ПЕРЕЛІК ПОСИЛАНЬ

1. Доповідь компанії Gartner. – Режим доступу: http://www.gartner.com/newsroom/id/3165317. – Дата доступу: 23.05.2016.
2. Доповідь компанії Cisco. – Режим доступу: http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoE_Economy.pdf. – Дата доступу: 23.05.2016.
3. Nathan Marz. Big Data: Principles and best practices of scalable realtime data systems / Nathan Marz, James Warren. – Manning Publications, 2015. – 328 p.
4. Arvind Sathi. Big Data Analytics: Disruptive Technologies for Changing the Game / Arvind Sathi. – Mc Press, 2012. – 96 p.
5. Vijay K. Garg. Elements of Distributed Computing / Vijay K. Garg. – Wiley-IEEE Press, 2002. – 448 p.
6. Gerard Tel. Introduction to Distributed Algorithms / Gerard Tel. – Cambridge University Press, 2000. – 612 p.
7. M. Tamer Özsu. Principles of Distributed Database Systems / M. Tamer Özsu, Patrick Valduriez. – Springer, 2011. – 846 p.
8. Refactoring: Improving the Design of Existing Code / [Martin Fowler, Kent Beck, John Brant et al.]. – Addison-Wesley Professional, 1999. – 464 p.
9. Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship / Robert C. Martin. – Prentice Hall, 2008. – 464 p.
10. Design Patterns: Elements of Reusable Object-Oriented Software / [Erich Gamma, Richard Helm, Ralph Johnson et al.]. – Addison-Wesley Professional, 1994. – 395 p.
11. Martin Fowler. Patterns of Enterprise Application Architecture / Martin Fowler. – Addison-Wesley Professional, 2002. – 560 p.
12. Len Bass. Software Architecture in Practice / Len Bass, Paul Clements, Rick Kazman. – Addison-Wesley Professional, 2012. – 640 p.

13. Luke Hohmann. Beyond Software Architecture: Creating and Sustaining Winning Solutions / Luke Hohmann. – Addison-Wesley Professional, 2003. – 352 p.
14. Інформація про платформу AWS IoT. – Режим доступу: https://aws.amazon.com/iot/how-it-works/. – Дата доступу: 24.05.2016.
15. Інформація про платформу IBM IoT Platform. – Режим доступу: http://www.ibm.com/internet-of-things/iot-platform.html. – Дата доступу: 25.05.2016.
16. Інформація про платформу Oracle IoT. – Режим доступу: https://cloud.oracle.com/iot. – Дата доступу: 25.05.2016.
17. Специфікація HTTP/2 (Internet Engineering Task Force. Request for Comments 7540). – Режим доступу: https://tools.ietf.org/html/rfc7540. – Дата доступу: 26.05.2016.
18. Специфікація WebSocket (Internet Engineering Task Force. Request for Comments 6455). – Режим доступу: https://tools.ietf.org/html/rfc6455. – Дата доступу: 27.05.2016.
19. Sam Newman. Building Microservices: Designing Fine-Grained Systems / Sam Newman. – O'Reilly Media, 2015. – 280 p.
20. Офіційна документація Apache Kafka. – Режим доступу: http://kafka.apache.org/documentation.html. – Дата доступу: 29.05.2016.
21. Офіційна документація Apache Spark. – Режим доступу: http://spark.apache.org/docs/latest/. – Дата доступу: 31.05.2016.
22. Інформація про Sort Benchmark і його результати. – Режим доступу: http://sortbenchmark.org/. – Дата доступу: 31.05.2016.
23. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing / [Matei Zaharia, Mosharaf Chowdhury, Tathagata Das et al.]. – Режим доступу: https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf. – Дата доступу: 31.05.2016.

24. Офіційна документація Apache Cassandra. – Режим доступу: https://wiki.apache.org/cassandra/. – Дата доступу: 02.06.2016.
25. Dynamo: Amazon's Highly Available Key-value Store / [Giuseppe DeCandia, Deniz Hastorun, Madan Jampani et al.]. – Режим доступу: http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf. – Дата доступу: 02.06.2016.
26. Niels Provos. A Future-Adaptable Password Scheme / Niels Provos, David Mazières // Proceedings of 1999 USENIX Annual Technical Conference. – 1999. – P. 81-92.
27. Офіційна документація MySQL. – Режим доступу: http://dev.mysql.com/doc/refman/5.7/en/. – Дата доступу: 02.06.2016.
28. Специфікація JSON Web Token (Internet Engineering Task Force. Request for Comments 7519). – Режим доступу: https://tools.ietf.org/html/rfc7519. – Дата доступу: 03.06.2016.
29. Офіційна документація HAProxy. – Режим доступу: http://www.haproxy.org/. – Дата доступу: 04.06.2016.
30. Офіційна документація Amazon Web Services. – Режим доступу: https://aws.amazon.com/documentation/. – Дата доступу: 04.06.2016.
31. Офіційна документація Docker. – Режим доступу: https://www.docker.com/. – Дата доступу: 04.06.2016.
32. Офіційна документація Consul. – Режим доступу: https://www.consul.io/. – Дата доступу: 04.06.2016.

ДОДАТОК А

Лістинг програми

MessageQueueSender.java

```
package com.iot.mq;

import java.util.concurrent.CompletionStage;

public interface MessageQueueSender {
	CompletionStage<Void> send(String topic, String key, String data);
}
```

KafkaSender.java

```
package com.iot.mq;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;

import java.util.Properties;
import java.util.concurrent.CompletableFuture;

public class KafkaSender implements MessageQueueSender {
	private final Producer<String, String> producer;

	public KafkaSender() {
		Properties properties = new Properties();
		properties.put("bootstrap.servers", "localhost:9092");
		properties.put("acks", "1");
		properties.put("key.serializer", StringSerializer.class.getName());
		properties.put("value.serializer", StringSerializer.class.getName());
		producer = new KafkaProducer<>(properties);
	}

	public CompletableFuture<Void> send(String topic, String key, String data) {
		ProducerRecord<String, String> record = new ProducerRecord<>(topic,
key, data);
		CompletableFuture<Void> future = new CompletableFuture<>();
		producer.send(record, (metadata, exception) -> {
			if (exception != null) {
				future.completeExceptionally(exception);
			} else {
				future.complete(null);
			}
		});
		return future;
	}
}
```

AuthenticationService.java

```
package com.iot.auth;

import java.util.concurrent.CompletionStage;

public interface AuthenticationService {
        CompletionStage<Boolean> authenticate(String login, String password);
}
```

MongoAuthenticationService.java

```
package com.iot.auth;

import com.mongodb.async.client.MongoClient;
import com.mongodb.async.client.MongoClients;
import com.mongodb.async.client.MongoCollection;
import com.mongodb.async.client.MongoDatabase;
import org.bson.Document;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

import static com.mongodb.client.model.Filters.eq;

public class MongoAuthenticationService implements AuthenticationService {
        public static final String DB_NAME = "iot";
        public static final String COLLECTION_NAME = "device_info";

        public static final String DEVICE_ID = "device_id";
        public static final String PASSWORD = "password";

        private final MongoCollection<Document> deviceInfoCollection;

        public MongoAuthenticationService() {
                MongoClient mongoClient = MongoClients.create();
                MongoDatabase db = mongoClient.getDatabase(DB_NAME);
                deviceInfoCollection = db.getCollection(COLLECTION_NAME);
        }

        @Override
        public CompletionStage<Boolean> authenticate(String login, String password) {
                CompletableFuture<Boolean> future = new CompletableFuture<>();
                deviceInfoCollection.find(eq(DEVICE_ID, login)).first((result,
exception) -> {
                        if (exception != null) {
                                future.completeExceptionally(exception);
                                return;
                        }

                        if (result == null) {
                                future.complete(false);
                                return;
                        }

                        Object pwd = result.get(PASSWORD);
                        if (pwd instanceof String && pwd.equals(password))
                                future.complete(true);
```

```
                        else
                                future.complete(false);
                });
                return future;
        }
}
```

JwtTokenService.java

```
package com.iot.token;

import com.auth0.jwt.JWTSigner;
import com.auth0.jwt.JWTVerifier;

import java.util.Map;
import java.util.regex.Pattern;

public class JwtTokenService {
        private static final String SECRET = "feRtfVqoyDJWVVoP4X2m";

        private static final Pattern AUTH_HEADER_PATTERN = Pattern.compile("^Bearer$",
Pattern.CASE_INSENSITIVE);

        private final JWTVerifier verifier = new JWTVerifier(SECRET);
        private final JWTSigner signer = new JWTSigner(SECRET);

        public String createToken(Map<String, Object> claims) {
                return signer.sign(claims);
        }

        public Map<String, Object> verifyToken(String authorizationHeader) throws
TokenParseException,
                        TokenVerificationException {
                String token = extractToken(authorizationHeader);
                try {
                        return verifier.verify(token);
                } catch (Exception e) {
                        throw new TokenVerificationException(e);
                }
        }

        private static String extractToken(String authorizationHeader) throws
TokenParseException {
                if (authorizationHeader == null)
                        throw new TokenParseException("Authorization header value is
null");

                String[] parts = authorizationHeader.split(" ");
                if (parts.length != 2) {
                        throw new TokenParseException("Incorrect format: '" +
authorizationHeader +
                                        "'. Format is Authorization: Bearer
[token]");
                }

                String scheme = parts[0];
                String credentials = parts[1];
```

```
                if (AUTH_HEADER_PATTERN.matcher(scheme).matches()) {
                        return credentials;
                } else {
                        throw new TokenParseException("Incorrect scheme: " + scheme);
                }
        }
}
```

HttpUtils.java

```
package com.iot.http;

import io.undertow.server.HttpServerExchange;

import java.util.Deque;
import java.util.Map;

import static io.undertow.util.StatusCodes.BAD_REQUEST;
import static io.undertow.util.StatusCodes.INTERNAL_SERVER_ERROR;

public class HttpUtils {
        public static Void sendRequestError(HttpServerExchange exchange) {
                return sendError(exchange, BAD_REQUEST);
        }

        public static Void sendServerError(HttpServerExchange exchange) {
                return sendError(exchange, INTERNAL_SERVER_ERROR);
        }

        public static Void sendError(HttpServerExchange exchange, int code) {
                return sendError(exchange, "", code);
        }

        public static Void sendError(HttpServerExchange exchange, String body, int
code) {
                exchange.setStatusCode(code);
                exchange.getResponseSender().send(body);
                return null;
        }

        public static String extractQueryParameter(String parameter, Map<String,
Deque<String>> queryParameters) {
                Deque<String> values = queryParameters.get(parameter);

                if (values == null || values.isEmpty())
                        return null;

                return values.getFirst();
        }
}
```

AuthenticationHttpHandler.java

```
package com.iot.http;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.iot.auth.AuthenticationService;
import com.iot.token.JwtTokenService;
import io.undertow.server.HttpHandler;
import io.undertow.server.HttpServerExchange;
import io.undertow.util.Headers;
import io.undertow.util.Methods;
import io.undertow.util.StatusCodes;

import java.util.Deque;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.CompletionStage;

import static com.iot.http.HttpUtils.extractQueryParameter;
import static com.iot.http.HttpUtils.sendError;
import static com.iot.http.HttpUtils.sendServerError;
import static io.undertow.util.StatusCodes.UNAUTHORIZED;

public class AuthenticationHttpHandler implements HttpHandler {
        public static final String DEVICE_ID = "deviceId";
        public static final String LOGIN = "login";
        public static final String PASSWORD = "password";

        private final AuthenticationService authenticationService;
        private final JwtTokenService tokenService;
        private final ObjectMapper objectMapper;

        public AuthenticationHttpHandler(AuthenticationService authenticationService,
JwtTokenService tokenService,
                                         ObjectMapper objectMapper) {
                this.authenticationService = authenticationService;
                this.tokenService = tokenService;
                this.objectMapper = objectMapper;
        }

        @Override
        public void handleRequest(HttpServerExchange exchange) throws Exception {
                Map<String, Deque<String>> queryParameters =
exchange.getQueryParameters();
                String login = extractQueryParameter(LOGIN, queryParameters);
                String password = extractQueryParameter(PASSWORD, queryParameters);

                if (login == null || password == null) {
                        sendError(exchange, "Login credentials are not present",
UNAUTHORIZED);
                        return;
                }

                exchange.dispatch();
                CompletionStage<Boolean> authenticationFuture =
authenticationService.authenticate(login, password);
                authenticationFuture
                                .thenAccept((authenticated) -> {
                                        if (authenticated) {
                                                createAndSendToken(login, exchange);
                                                return;
```

```
                                        }

                                        sendError(exchange, "Authentication
credentials are invalid", UNAUTHORIZED);
                                })
                                .exceptionally(throwable ->
sendServerError(exchange));
        }

        private void createAndSendToken(String login, HttpServerExchange exchange) {
                Map<String, Object> tokenClaims = new HashMap<>();
                tokenClaims.put(DEVICE_ID, login);
                String token = tokenService.createToken(tokenClaims);
                String json = objectMapper.createObjectNode().put("token",
token).toString();
                exchange.getResponseSender().send(json);
        }
}
```

DataHttpHandler.java

```
package com.iot.http;

import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.iot.mq.MessageQueueSender;
import com.iot.token.JwtTokenService;
import com.iot.token.TokenParseException;
import com.iot.token.TokenVerificationException;
import io.undertow.server.HttpHandler;
import io.undertow.server.HttpServerExchange;
import io.undertow.util.HeaderValues;
import io.undertow.util.Headers;
import io.undertow.util.Methods;
import io.undertow.util.StatusCodes;

import java.io.IOException;
import java.util.Map;

import static com.iot.http.AuthenticationHttpHandler.DEVICE_ID;
import static com.iot.http.HttpUtils.*;
import static com.iot.mq.KafkaTopics.DATA_TOPIC;
import static io.undertow.util.StatusCodes.UNAUTHORIZED;

public class DataHttpHandler implements HttpHandler {
        public static final String DATA = "data";

        private final MessageQueueSender sender;
        private final JwtTokenService tokenService;
        private final ObjectMapper objectMapper;

        public DataHttpHandler(MessageQueueSender sender, JwtTokenService tokenService,
ObjectMapper objectMapper) {
                this.sender = sender;
                this.tokenService = tokenService;
                this.objectMapper = objectMapper;
        }
```

```
        @Override
        public void handleRequest(HttpServerExchange exchange) throws Exception {
                exchange.getRequestReceiver().receiveFullBytes(this::processMessage,
(exch, e) -> sendServerError(exchange));
        }

        private void processMessage(HttpServerExchange exchange, byte[] message) {
                String authorizationHeader = getAuthorizationHeader(exchange);
                if (authorizationHeader == null) {
                        sendError(exchange, "No 'Authorization' header",
UNAUTHORIZED);
                        return;
                }

                Map<String, Object> tokenPayload;
                try {
                        tokenPayload = tokenService.verifyToken(authorizationHeader);
                } catch (TokenParseException e) {
                        sendError(exchange, "Token could not be parsed",
UNAUTHORIZED);
                        return;
                } catch (TokenVerificationException e) {
                        sendError(exchange, "Token could not be verified",
UNAUTHORIZED);
                        return;
                }

                JsonNode json;
                try {
                        json = objectMapper.readTree(message);
                } catch (IOException e) {
                        sendRequestError(exchange);
                        return;
                }

                JsonNode data = json.get(DATA);
                String deviceId = getDeviceId(tokenPayload);
                if (deviceId == null || data == null) {
                        sendRequestError(exchange);
                        return;
                }

                exchange.dispatch(); // do not end exchange when this method returns,
because saving to kafka is async
                sender.send(DATA_TOPIC, deviceId, data.toString())
                                .thenRun(() -> exchange.getResponseSender().close())
                                .exceptionally((throwable ->
sendServerError(exchange)));
        }

        private static String getAuthorizationHeader(HttpServerExchange exchange) {
                HeaderValues authorizationHeaderValues =
exchange.getRequestHeaders().get(Headers.AUTHORIZATION);
                return authorizationHeaderValues == null ? null :
authorizationHeaderValues.getFirst();
        }

        private static String getDeviceId(Map<String, Object> tokenPayload) {
                if (tokenPayload == null)
                        return null;
```

```
                Object o = tokenPayload.get(DEVICE_ID);

                if (o instanceof String) {
                        String deviceId = (String) o;
                        return deviceId.isEmpty() ? null : deviceId;
                }

                return null;
        }
}
```

HttpServer.java

```
package com.iot.http;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.iot.auth.AuthenticationService;
import com.iot.auth.MongoAuthenticationService;
import com.iot.mq.KafkaSender;
import com.iot.token.JwtTokenService;
import com.iot.weather.http.ReportHttpHandler;
import io.undertow.Undertow;

import static io.undertow.Handlers.routing;

public class HttpServer {
        public static void main(final String[] args) {
                JwtTokenService tokenService = new JwtTokenService();
                ObjectMapper objectMapper = new ObjectMapper();
                AuthenticationService authenticationService = new
MongoAuthenticationService();
                DataHttpHandler dataHttpHandler = new DataHttpHandler(new
KafkaSender(), tokenService, objectMapper);
                AuthenticationHttpHandler authenticationHttpHandler =
                                new AuthenticationHttpHandler(authenticationService,
tokenService, objectMapper);
                ReportHttpHandler reportHttpHandler = new ReportHttpHandler();

                Undertow server = Undertow.builder()
                                .addHttpListener(8080, "localhost")
                                .setHandler(routing()
                                                .get("/api/report/device/{id}",
reportHttpHandler)
                                                .post("/api/data", dataHttpHandler)

        .get("/api/auth/login/{login}/password/{password}", authenticationHttpHandler))
                                .build();

                server.start();
        }
}
```

DoubleStatsCounter.scala

```
package com.iot.util

import scala.math.sqrt

case class DoubleStats(count: Long, avg: Double, stdDev: Double)

case class DoubleStatsCounter(var n: Long = 0, var sum: Double = 0.0, var sumOfSquares:
Double = 0.0) {
  def add(number: Double): DoubleStatsCounter = {
    n += 1
    sum += number
    sumOfSquares += number * number
    this
  }

  def merge(anotherCounter: DoubleStatsCounter): DoubleStatsCounter = {
    n += anotherCounter.n
    sum += anotherCounter.sum
    sumOfSquares += anotherCounter.sumOfSquares
    this
  }

  def avg = sum / n

  def stdDev = sqrt(sumOfSquares / n - avg * avg)

  def stats = DoubleStats(n, avg, stdDev)
}

object DoubleStatsCounter {
  def apply(number: Double): DoubleStatsCounter = new DoubleStatsCounter().add(number)
}
```

WeatherDomain.scala

```
package com.iot.weather

import java.util.Date

import com.iot.util.{DoubleStats, DoubleStatsCounter}

object WeatherDomain {
  // db
  val (keyspace, table) = ("iot", "weather")

  // columns
  val (deviceId, timestamp, temperatureStats, pressureStats) = ("device_id",
"timestamp", "temperature_stats", "pressure_stats")
  val (count, avg, stdDev) = ("count", "avg", "std_dev")

  case class AggregateWeatherDataRecord(temperatureStats: DoubleStatsCounter,
                                        pressureStats: DoubleStatsCounter) {
    def merge(anotherRecord: AggregateWeatherDataRecord) = {
      temperatureStats.merge(anotherRecord.temperatureStats)
      pressureStats.merge(anotherRecord.pressureStats)
      this
```

```
    }
  }

  case class WeatherDatabaseRecord(deviceId: String, timestamp: Date,
                                   temperatureStats: DoubleStats, pressureStats:
DoubleStats)

  case class WeatherDataRecord(temperature: Double, pressure: Double)
}
```

ReportHttpHandler.scala

```
package com.iot.weather.http

import com.datastax.spark.connector._
import com.iot.http.HttpUtils.{extractQueryParameter, sendServerError}
import com.iot.weather.WeatherDomain._
import io.undertow.server.{HttpHandler, HttpServerExchange}
import io.undertow.util.HttpString
import org.apache.spark.{SparkConf, SparkContext}
import org.json4s.NoTypeHints
import org.json4s.jackson.Serialization
import org.json4s.jackson.Serialization._

import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.{Failure, Success}

class ReportHttpHandler extends HttpHandler {
  val conf = new SparkConf()
    .setMaster("local[*]")
    .setAppName(getClass.getSimpleName)
    .set("spark.cassandra.connection.host", "127.0.0.1")

  val sc = new SparkContext(conf)

  implicit val formats = Serialization.formats(NoTypeHints)

  val corsHeader = new HttpString("Access-Control-Allow-Origin")

  override def handleRequest(exchange: HttpServerExchange): Unit = {
    var weatherTable = sc.cassandraTable[WeatherDatabaseRecord](keyspace, table)
    Option(extractQueryParameter("id", exchange.getQueryParameters))
      .foreach(id => weatherTable = weatherTable.where(s"$deviceId = ?", id))
    exchange.dispatch
    val rowsFuture = weatherTable.collectAsync()
    exchange.getResponseHeaders.add(corsHeader, "*")
    rowsFuture.onComplete {
      case Success(rows) => exchange.getResponseSender.send(write(rows))
      case Failure(e) => sendServerError(exchange)
    }
  }
}
```

WeatherStreamingProcessor.scala

```
package com.iot.weather.stream

import java.util.Date

import com.datastax.spark.connector.cql.CassandraConnector
import com.datastax.spark.connector.streaming._
import com.iot.mq.KafkaTopics.DATA_TOPIC
import com.iot.util.DoubleStatsCounter
import com.iot.weather.WeatherDomain._
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.kafka.KafkaUtils
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.{SparkConf, SparkContext}
import org.json4s._
import org.json4s.jackson.JsonMethods._

object WeatherStreamingProcessor extends App {
  implicit val formats = DefaultFormats

  val batchDuration = Seconds(1)

  val conf = new SparkConf()
    .setMaster("local[*]")
    .setAppName(getClass.getSimpleName)
    .set("spark.cassandra.connection.host", "127.0.0.1")

  val sc = new SparkContext(conf)
  val ssc = new StreamingContext(sc, batchDuration)

  type Key = String
  val (zkQuorum, groupId) = ("localhost:2181", "weather-consumer")

  val stats = "stats" // type

  // json fields
  val (temperature, pressure) = ("temperature", "pressure")

  CassandraConnector(conf).withSessionDo { session =>
    session.execute(s"DROP KEYSPACE IF EXISTS $keyspace")
    session.execute(s"CREATE KEYSPACE IF NOT EXISTS $keyspace " +
      s"WITH REPLICATION = {'class': 'SimpleStrategy', 'replication_factor': 1 }")
    session.execute(s"CREATE TYPE $keyspace.$stats ($count BIGINT, $avg DOUBLE, $stdDev
DOUBLE)")
    session.execute(
      s"""CREATE TABLE IF NOT EXISTS $keyspace.$table
      ($deviceId TEXT, $timestamp TIMESTAMP,
      $temperatureStats FROZEN<stats>,
      $pressureStats FROZEN<stats>,
      PRIMARY KEY ($deviceId, $timestamp))""")
    session.execute(s"TRUNCATE $keyspace.$table")
  }

  val stream = KafkaUtils.createStream(ssc, zkQuorum, groupId, Map(DATA_TOPIC -> 1),
StorageLevel.MEMORY_ONLY)

  stream
    .map(parseKafkaRecord)
    .map(toAggregateRecord)
```

```
    .reduceByKeyAndWindow((r1: AggregateWeatherDataRecord, r2:
AggregateWeatherDataRecord) => r1.merge(r2),
      batchDuration, batchDuration)
    .map(toDatabaseRecord)
    .saveToCassandra(keyspace, table)

  ssc.start()

  def parseKafkaRecord(t: (String, String)): (Key, WeatherDataRecord) = {
    val json = parse(t._2)
    (t._1, WeatherDataRecord(
      (json \ temperature).extractOrElse(0.0),
      (json \ pressure).extractOrElse(0.0)))
  }

  def toAggregateRecord(t: (Key, WeatherDataRecord)): (Key, AggregateWeatherDataRecord)
=
    (t._1, AggregateWeatherDataRecord(DoubleStatsCounter(t._2.temperature),
DoubleStatsCounter(t._2.pressure)))

  def toDatabaseRecord(t: (Key, AggregateWeatherDataRecord)) =
    WeatherDatabaseRecord(t._1, new Date(), t._2.temperatureStats.stats,
t._2.pressureStats.stats)
}
```