ДОСЛІДЖЕННЯ МЕТОДІВ ЗБОРУ ТА ОБРОБКИ ДАНИХ В РОЗПОДІЛЕНИХ ІНФОРМАЦІЙНИХ СИСТЕМАХ

С. Журавель, С. Думич, О. Шпур

Національний університет “Львівська політехніка”, вул. С. Бандери, 12, Львів, 79013, Україна

Відповідальний за рукопис: Станіслав Журавель (e-mail: stanislav.zhuravel.mntr.2019@lpnu.ua).

(Подано 30 травня 2021)

У статті розглянуто проблему обробки та аналізу великих об'ємів даних. Сьогодні, у зв'язку з появою джерел інформації, які можуть продукувати великі обсяги даних, оброблення даних стає все складнішим. Для вирішення поставленої проблеми розроблено велику кількість технологій специфічного призначення, проте більшість з них можна зарахувати до двох парадигм обробки даних, а саме потокової та пакетної. В цій статті розглянуто вищенаведені парадигми та технології, які застосовують для прийняття рішень щодо обробки даних, та спробуємо окреслити характеристики, які потрібно брати до уваги під час побудови систем обробки даних. На основі найперспективніших рішень наведемо також реалізацію системи збору та аналізу даних з пристроїв IoT.

Ключові слова: розподілені інформаційні системи; обробка даних; великі об'єми даних.

УДК: 621.126

1. Вступ

Сьогодні багато програм потребують можливості працювати з великими обсягами даних. На відміну від необхідності в обчислювальних можливостях, сира потужність процесора нечасто є обмежувальним фактором для цих систем – більшими проблемами, як правило, є обсяг даних, їх складність та швидкість, з якою вони змінюються.

Під час побудови розподілених систем виникає багато запитань. Як гарантувати, що дані залишаються правильними та повними, навіть коли виникають системні збої всередині системи? Як забезпечити стабільно високий доступ до системи, навіть коли частина вашої системи не працює? Як масштабувати систему, щоб справлятися зі збільшенням навантаження?

2. Аналіз та постановка задачі

За останні роки з'явилося багато нових інструментів для зберігання та обробки даних. Вони оптимізовані для різноманітних випадків використання і більше не входять у традиційні категорії. Наприклад, існують сховища даних, які також використовуються як черги повідомлень (Redis), а є черги повідомлень із гарантіями зберігання, подібними до бази даних (Apache Kafka). Межі між категоріями стають розмитими, а дані несистематизованими. Одночасно розвиваються напрями, які продукують великі об'єми даних, які вимагають від систем можливості обробляти, зберігати та аналізувати значні обсяги даних за короткий проміжок часу.

Існує багато систем баз даних із різними характеристиками, оскільки у різних систем різні вимоги. Є й різні підходи до зберігання, кілька способів побудови пошукових індексів, аналізу тощо. Розробляючи систему, потрібно з'ясувати, які інструменти та підходи є найвідповіднішими для поставленого завдання. І може бути важко поєднати інструменти, коли потрібно зробити щось, чого окремий інструмент не може зробити сам.

Сьогодні широкий перелік вимог до обробки та зберігання даних неможливо задовольнити одним інструментом. Натомість системи часто розділені на підсистеми, поєднані за допомогою програмних рішень.

Отож, постає проблема оброблення даних, які надходять у великих кількостях від різних джерел інформації, водночас стрімко розвиваються багато технологій спеціального застосування, що зумовлює проблему поєднання різних компонентів.

Проблеми розподілених систем детально розглянув Мартін Клепман [1], описавши можливі проблеми та способи їх вирішення для забезпечення системами масштабованості, надійності, ефективності тощо. В своїй роботі Мартін Клепман висвітлює широке коло проблем, на жаль, розглядаючи методи оброблення даних лише в декількох розділах та не наводячи практичних рішень систем, інші ж праці є вузькоспрямованими. Підкреслимо також, що вітчизняної літератури з цих питань мало.

Зважаючи на це, у статті розглянемо та систематизуємо дані про парадигми обробки та можливі методи їх реалізації, а також наведемо приклад реалізації системи для збирання та аналізування даних від пристроїв IoT.

3. Аналіз методів обробки та можливих методів імплементації

Система пакетної обробки приймає велику кількість вхідних даних, виконує роботу з їх оброблення та видає деякі вихідні дані. Робота часто займає деякий час (від кількох хвилин до кількох днів), тому зазвичай користувач не чекає її закінчення. Натомість пакетні завдання часто планують виконувати періодично (наприклад, раз на день). Основним показником продуктивності пакетного завдання, як правило, є пропускна здатність (час, необхідний для обробки вхідного набору даних певного розміру).

Потокові системи (іноді їх називають обробкою майже в реальному часі, або її наближенням), обробляють дані в міру надходження. Це означає, що результати можна отримати майже миттєво. Як і система пакетної обробки, поточковий процесор споживає вхідні дані та видає вихідні дані (а не відповідає на запити, як, наприклад, онлайн-системи, які, отримавши інструкції від клієнта, намагаються якнайшвидше обробити запит та повернути відповідь). Однак потокове завдання працює над подіями незабаром після того, як вони трапляються, тоді як пакетне завдання працює із фіксованим набором вхідних даних. Завдяки цій відмінності у систем обробки потоків менша затримка, ніж в еквівалентних пакетних систем.

Потокову обробку можна представити як програмну взаємодію із брокером повідомлень. Для цього події розміщують у журналі брокера повідомлень (наприклад, ActiveMQ, RabbitMQ або Kafka), потім реалізують частину коду, який відповідатиме за отримання подій (вони і стануть потоком), з журналу брокера та їх безпосереднє оброблення, після оброблення результат повертається назад і публікується у журналі брокера. Процес, який виконує код обробки, називають актором.

Відзначимо велику роль брокера повідомлень у реалізації пакетної обробки, оскільки для потокової обробки він має забезпечувати порядок та однозначне зберігання подій у системі.

Можна провести паралелі між брокерами повідомлень та базами даних. Хоч їх традиційно вважали окремими категоріями інструментів, видно, що брокери повідомлень на основі журналів успішно беруть ідеї із баз даних і застосовують їх до обміну повідомленнями. Також можна зробити навпаки: взяти ідеї з обміну повідомленнями та потоків і застосувати їх до баз даних.

Подія – це запис того, що сталося в певний момент часу. Те, що сталося, може бути дією користувача (наприклад, введення пошукового запиту) або зчитуванням датчика, але це може бути також запис у базу даних. Той факт, що щось записано в базу даних, – це подія, яку можна захопити, зберегти та обробити. Це спостереження свідчить про те, що зв'язок між базами даних та потоками реалізується глибше, ніж просто фізичне зберігання журналів на диску, – це доволі принципово.

Звернемо увагу на журнал реплікацій в розподіленій базі даних, – це потік записів у базу даних, який створює лідер під час оброблення транзакцій. Послідовники застосовують цей потік записів до власної копії бази даних і, отже, отримують точну копію тих самих даних. Події в журналі реплікації описують зміни даних, що відбулися [1].

Також існує принцип “Тотального порядку” (Total order broadcast) в розподілених інформаційних системах, який стверджує: якщо кожна подія являє собою запис у базу даних і кожна репліка обробляє ті самі події в тій самій послідовності, то всі репліки опиняться в тому самому кінцевому стані. Тобто це просто черговий випадок потоків подій!

Отже, для імплементації потокової обробки не обов'язково використовувати брокер повідомлень, сам концепт існує окремо, брокери – лише добрі реалізації цього концепту, наприклад, базу даних Redis можуть використовувати для побудови черги повідомлень. Однак імплементація черги повідомлень в розподіленій системі має багато підводних каменів. Так, для забезпечення “тотального порядку” навіть брокери часто можуть використовувати додаткові системи, такі як Zookeeper, який використовують як сховок метаданих кластера для підтримання його безперебійної роботи. Тож використання брокерів істотно спрощує реалізацію потокової обробки, але не є обов'язковим атрибутом.

Принцип тотального порядку особливо важливий для побудови потокової обробки, оскільки недотримання цього принципу призводить до різних результатів обробки того самого набору даних, а це недопустимо.

Сьогодні брокери можуть представляти один з класів:

1. Брокер повідомлень у стилі AMQP/JMS. Такий брокер призначає індивідуальні повідомлення споживачам, а споживачі підтверджують окремі повідомлення, якщо їх успішно оброблено. Повідомлення видаляються з брокера після їх підтвердження. Цей підхід доречний як асинхронна форма RPC, наприклад, у черзі завдань, де не важлива точна послідовність обробки повідомлень і де немає необхідності повертатися назад і читати старі повідомлення знову після їх оброблення.

2. Брокер повідомлень на основі журналу. Брокер призначає всі повідомлення в розділі одному і тому ж споживчому вузлу і завжди доставляє повідомлення в однаковій послідовності. Паралельність досягається секціонуванням (partitioning), і споживачі відстежують їхній прогрес, встановлюючи контрольний зсув (off-set) останнього обробленого ними повідомлення. Брокер зберігає повідомлення на диску, тому за необхідності можна перейти назад і перечитати старі повідомлення.

Оскільки тільки брокери на основі журналу дають змогу забезпечити послідовність подій, яка є важливою для потокової обробки, саме такий тип брокера можна використати для побудови поточкових систем.

Однак замість того, щоб реалізовувати вищезазначений сценарій з самого початку, можна скористатися фреймворком оброблення потоків, щоб заощадити час. Такий фреймворк дає змогу писати логіку для кожного актора, підключати і відключати їх від джерела даних. Можна надсилати події безпосередньо поточковому процесору або надсилати їх через посередника.

Процесор обробки подій фреймворка виконає важку роботу, збираючи дані, передаючи їх кожному актору, переконуючись, що вони працюють у правильній послідовності, збираючи результати, масштабуючи, якщо навантаження велике, та обробляючи збої. Серед прикладів – Storm, Flink та Samza.

Своєю чергою, ці фреймворки поділено на дві великі групи “фреймворки складної обробки подій” та “фреймворки потокової обробки”, у яких є певні відмінності.

Варто зазначити, що ці відмінності не походять від визначень потокової обробки. Швидше, вони є побічним продуктом історії та випадків використання, з якими їх доводилося розглядати. Це є причиною плутанини між цими двома групами [2]

Отже, зупинимось на відмінностях між двома типами двигунів. Нижче наведені ключові відмінності між двигунами “складної обробки подій” та “потокової обробки”:

1. Двигуни потокової обробки проєктовані для паралельної обробки. Вони підтримують великі обсяги обчислень, на відміну від двигунів складної обробки подій, які мають централизовану архітектуру, зазвичай із двома або кількома вузлами.

2. Механізми обробки потоків змушують писати код, і часто вони не мають операторів вищого рівня, таких як вікна та часові шаблони. На відміну від них, двигуни складної обробки подій надають мови високого рівня. Ця відмінність подібна до взаємозв'язку між сценаріями MapReduce та HIVE SQL.

3. Завдяки своїй історії, що ґрунтується на фондовому ринку, двигуни складної обробки подій налаштовані на низьку затримку. Часто затримка становить декілька мілісекунд. На відміну від цього, більшість механізмів обробки потоку потребує близько секунди для отримання результатів.

4. Механізми потокової обробки наголошують на надійному обробленні повідомлень, часто споживаючи дані з черги, такі як Kafka. На відміну від них, двигуни складної обробки подій часто отримують і обробляють дані в пам'яті, а коли трапляється збій, часто вирішують викинути невдалі події та продовжити. Однак ця поведінка вже змінилася. Більшість механізмів складної обробки подій підтримують надійну обробку даних із черги, такої як Kafka [3].

Тож зазначимо, що для великих об'ємів даних почали ширше застосовуватися двигуни потокової обробки.

Пакетну обробку можна представити за допомогою SMP (симетричної багатопроцесорної обробки), MPP (масивної паралельної обробки) або комбінації схожих на MapReduce систем та розподілених файлових систем.

У системах SMP є апаратне забезпечення, в якому низка фізичних процесорів (або ядер) виконує інструкції з використанням спільної пам'яті та дисків. Така побудова системи передбачає певні обмеження під час роботи з великими об'ємами даних. До них можна зарахувати:

1. Важкий процес масштабування. Хоча можливо додати процесори, пам'ять або додаткові диски, насправді масштабованість обмежена, і, ймовірно, доведеться повністю замінювати машину навіть за порівняно невеликого підвищення навантаження.

2. Неефективне масштабування, додавання додаткових або швидших процесорів не часто збільшує продуктивність у лінійному масштабі. Наприклад, малоімовірно, що додавання на 100 % швидших процесорів подвоїть продуктивність, якщо система повністю не пов'язана з процесором. Здебільшого вузьке місце переходить на інший компонент, і лінійно збільшити потужність неможливо.

3. Стійкість та доступність, оскільки системи SMP використовують єдину платформу, систему потрібно дублювати, щоб забезпечити стійкість у разі відмови компонента або цілого центра оброблення даних. Це часто робить цей варіант дорогим, хоча (теоретично) його можна розгорнути на недорогих серверах. Насправді підхід частіше застосовується на апаратному забезпеченні корпоративної якості з подвійними резервними дисками, мережевими підключеннями та джерелами живлення. Хоча це забезпечує стійкість до несправності, рішення також потребуватиме окремої системи очікування, щоб гарантувати високу доступність [4].

Масивна паралельна обробка – це підхід розподілених обчислень, коли всі окремі вузли розподіленої системи беруть участь у скоординованих обчисленнях. СУБД MPP – це системи управління базами даних, побудовані на основі цього підходу. У цих системах кожен запит

обробляють та виконують вузли системи MPP паралельно, розділяючи обчислення, вони працюють у рази швидше, ніж системи SMP. Однією із додаткових переваг, яку ця архітектура забезпечує, є масштабованість, оскільки можна легко масштабувати систему, просто додаючи до неї нові вузли. Щоб мати можливість обробляти величезні обсяги даних, дані в цих системах розділяють між вузлами, оскільки кожен вузол обробляє лише свої локальні дані. Це ще більше пришвидшує оброблення даних, оскільки використання спільного сховища для такого дизайну було б величезним надлишком – складнішим, дорожчим, менш масштабованим, із меншим паралелізмом. Ось чому більшість рішень СУБД MPP є загальнодоступними та працюють над сховищем DAS (безпосередньо прикріплене сховище) або набором блоків пам'яті, спільним для невеликих груп серверів.

Така побудова зумовила певні переваги порівняно із системами SMP:

1. Масштабованість та паралельність. На відміну від рішень SMP, системи, основані на MPP, мають можливість поступово додавати обчислювальні ресурси та ресурси зберігання, а пропускна здатність покращується переважно із лінійною залежністю. Додавання додаткового вузла такого ж розміру збільшує здатність системи обробляти додаткові запити без значного зниження продуктивності.

2. Вартість та висока доступність, деякі рішення для зберігання даних на основі MPP передбачені для роботи на недорогих апаратних засобах без необхідності використання подвійних резервних компонентів. Ці рішення зазвичай застосовують автоматичну реплікацію даних для підвищення стійкості системи та гарантування високої доступності. Це дає змогу набагато ефективніше використовувати ресурси й уникнути витрат на побудову систем очікування, що використовується багатьма рішеннями на основі SMP.

3. Пропускна здатність читання та запису. Оскільки дані розподіляються по системі, це рішення може досягти надзвичайного рівня пропускної здатності, оскільки як операції читання, так і запису можуть виконуватися паралельно на незалежних вузлах кластера.

4. Бази даних MPP дуже добре справляються із найпоширенішими аналітичними робочими навантаженнями, що, як правило, характеризуються запитом щодо підмножини стовпців із агрегуванням у широкому діапазоні рядків. Це пов'язано з їх моделлю, орієнтованою на стовпці, яка дає змогу доступатися лише до полів, необхідних для запиту (на відміну від транзакційних баз даних, які повинні обробити повністю рядок у таблиці незалежно від полів, потрібних для отримання результату).

Така модель, орієнтована на стовпці, також надає базам даних MPP додаткові функції, корисні для аналітичних навантажень. Вони залежать від бази даних, але часто передбачають можливість стискати дані в стовпці, ефективно індексувати дуже великі таблиці та обробляти широкі денормалізовані таблиці.

Hadoop – це не окремий продукт, а екосистема із відкритим кодом. На перший погляд, архітектура Hadoop/HDFS схожа на архітектуру MPP. Однак найбільша відмінність полягає у тому, що тоді як платформа MPP розподіляє окремі записи (рядки в базі даних) по кластеру, Hadoop просто розбиває дані на довільні блоки, розмір яких 128 Мб (як рекомендує Cloudera) і які потім копіюються щонайменше на два інші вузли для стійкості у разі відмови вузла [5].

Це важливо, оскільки означає, що невеликі файли (із розміром менше ніж 128 Мб) повністю зберігаються на одному вузлі, й навіть файл розміром гігабайт буде розподілений лише на восьми вузлах (плюс репліки). Це важливо, оскільки Hadoop розроблено для роботи з дуже великими наборами даних та великими кластерами. Однак, оскільки невеликі таблиці розподіляються на меншій кількості серверів, це не дуже добре для даних розміром менше ніж 50–100 Гб [2].

Обробка малих наборів даних є проблемою для Hadoop, оскільки в гіршому випадку обробка даних на одному вузлі працює повністю послідовно і паралельно нічого не виконується. Справді, оскільки багато кластерів Hadoop, як правило, використовують велику кількість порівняно повільних та недорогих серверів, продуктивність на малих наборах даних може бути дуже низькою.

До переваг Hadoop можна зарахувати:

1. Продуктивність пакетної обробки. Hadoop може бути чудовим варіантом для досягнення високої пропускної здатності під час оброблення дуже великих наборів даних.
2. Масштабованість. Подібно до систем MPP, додаткові вузли можуть бути додані для розширення кластера Hadoop, і кластери часто можуть перевищувати кластери MPP за кількістю вузлів.
3. Доступність і стійкість. Оскільки дані автоматично копіюються (дублюються) на кілька серверів, стійкість і висока доступність є вбудованими функціями. Це означає, що, наприклад, можливо вивести вузол з кластера для обслуговування, не перериваючи роботи кластера як цілого [6].
4. Вартість (обладнання та ліцензії). Оскільки Hadoop, як правило, розгортається на недорогих серверах, що працюють із програмним забезпеченням з відкритим кодом, вартість обладнання та ліцензій може бути значно нижчою, ніж традиційних пристроїв для зберігання даних корпоративного класу, менші пов'язані з цим ліцензійні витрати.

Незважаючи на всі переваги Hadoop, в нього також є недоліки:

1. Складність управління. Як описано вище, Hadoop – це не окремий продукт, а величезна екосистема програмного забезпечення, і для розгортання часто потрібні кваліфіковані знання багатьох інструментів, зокрема HDFS, Spark, Impala, Hive, Flume, Zookeeper та Kafka. Навіть широкомасштабне рішення MPP може бути значно менш складним у розгортанні та обслуговуванні, ніж Hadoop.
2. Обмеження інструментів запиту. Реляційні системи управління базами даних ураховують десятиліття досвіду автоматизованого налаштування запитів для ефективного виконання складних запитів SQL. Однак більшість інструментів SQL, основаних на Hadoop, не досягають необхідного рівня. Це призводить до вкрай неефективного використання машинних ресурсів, що може спричинити подорожчання системи.
3. Низька продуктивність запитів із малою затримкою. Хоча рішення кешування даних можуть допомогти, Hadoop/HDFS є дуже поганим рішенням для запитів із низькою затримкою, наприклад, для подання даних на інформаційну панель. Знову ж таки, це пов'язано з тим, що архітектура розроблена насамперед для обслуговування дуже великих наборів даних із використанням грубої сили пакетної обробки [7].

4. Реалізація системи збору та аналізу даних

Виокремивши в попередньому підрозділі методи, найсприятливіші для аналізу даних, доцільно підтвердити їх дієвість на практиці. Як визначено вище, найдоцільнішими методами сьогодні виявились потокова та пакетна обробка. Підкреслено, що ці методи мають певну специфіку використання, а отже, можуть співіснувати разом, доповнюючи систему важливими функціями.

Тому поставимо мету – підтвердити дієвість та можливість співіснування підходів, побудувавши систему, яка використовує їх одночасно. Зазначимо, що сфери використання аналізу даних на практиці дуже широкі. Оскільки сьогодні одним із джерел великих об'ємів даних є IoT пристрої, розробимо систему, метою якої буде збирання та аналізування даних, як у реальному часі, такі дані, зібраних з часом у результаті роботи системи. Джерелом даних будуть датчики, які вимірюють температуру, вологість, наявність диму тощо.

Для аналізу майже реального часу скористаємося потоковою обробкою, а саме брокером повідомлень зі збереженням історії (поняття збереження історії розглянемо нижче), а саме Apache Kafka, який заслужено завоював передові позиції у практичному використанні, оскільки має гнучку архітектуру та дає змогу організовувати й інші цікаві рішення, зокрема повне відновлення даних після збою системи (наприклад, підхід Change Data Capture, розглянутий вище тощо).

Для аналізу не в реальному часі скористаємось пакетною обробкою, за допомогою якої застосуємо до зібраних даних алгоритм машинного навчання, щоб відповісти на питання статистичного аналізу та підтвердити перевагу використання MapReduce порівняно з аналітикою, одержаною на базах даних MPP.

Оскільки джерелом інформації в системі стануть IoT пристрої, що можуть генерувати великий обсяг даних за короткий період часу, доцільно забезпечити необхідний рівень пропускну здатності та можливість масштабувати систему.

Як показано на рис. 1, підсистема із кластера IoT пристроїв надсилає дані до системи в json форматі, дані приймає аплікація, написана мовою Java, яка виконує роль посередника та, своєю чергою, уніфікує доступ до системи, одразу надсилаючи відповідь про успішне прийняття даних (якщо воно відбулося). В разі вдалого прийняття даних аплікація надсилає дані до відповідного журналу повідомлень. Брокер, своєю чергою, зберігає повідомлення та ініціює процес аналізу отриманих даних, порівнюючи минулі події тощо. В результаті аналізу може бути створена одна з подій, а саме: “FIRE_WARNING”, “FIRE_STARTED”, “FIRE_STOPPED”, які потрапляють до свого журналу. Інші підсистеми можуть підписуватися на ці події та виконувати свої дії, відповідно реагуючи на зміни стану системи, наприклад, надсилати сповіщення тощо. Хоч брокер повідомлень і може зберігати повідомлення на постійній основі, його роллю все ж залишається забезпечення необхідного рівня пропускну здатності системи та аналізу вхідних даних. Тому всі події зберігаються у сховище даних, де можуть бути створені необхідні пошукові індекси для забезпечення швидкого доступу до вибіркового даних. Постійне сховище також може слугувати джерелом даних для подальшої пакетної обробки, не впливаючи на роботу всієї іншої системи.

Рис. 1. Схема розробленої потокової підсистеми для аналізу великих об'ємів даних

Зазначимо, що аналітика в потоковій обробці, як правило, орієнтована на агрегування та статистичні показники для великої кількості подій. Наприклад:

1. Вимірювання швидкості певного виду подій (як часто це відбувається за інтервал часу).
2. Обчислення ковзного середнього значення за певний проміжок часу.

3. Порівняння поточної статистики із попередніми інтервалами часу (наприклад, для виявлення тенденцій або попередження про показники, які є надзвичайно високими або низькими порівняно з тим самим часом минулого тижня).

Таку статистику зазвичай обчислюють через фіксовані часові інтервали – наприклад, іноді необхідно знати середню кількість запитів за секунду за останні 5 хв та час відповіді на них (99-го “процентиля”) за цей період. Усереднення за кілька хвилин згладжує неактуальні коливання від однієї секунди до іншої, даючи водночас своєчасне уявлення про будь-які зміни в схемі. Інтервал часу, протягом якого здійснюється агрегація, відомий як вікно.

Системи потокової аналітики іноді використовують імовірнісні алгоритми, такі як фільтри Bloom, для встановлення належності до групи. Імовірнісні алгоритми дають приблизні результати, але мають ту перевагу, що вимагають значно менше пам’яті в потоковому процесорі, ніж точні алгоритми. Таке використання алгоритмів апроксимації іноді змушує людей думати, що системи обробки потоків завжди є втратними і неточними, але це неправильно: в обробленні потоку немає нічого наближеного, а ймовірнісні алгоритми – це просто оптимізація.

Тож оскільки, як зазначено вище, аналітика в потоковій обробці, як правило, орієнтована на агрегування та статистичні показники для великої кількості подій, наша система, продукуючи відповідні повідомлення (“FIRE_WARNING”, “FIRE_STARTED”, “FIRE_STOPPED”), може бути зарахована до класу так званих систем “складної обробки повідомлень” (complex event processing), оскільки не обчислює статистичні дані, а вимагає пошуку певних моделей подій у великій їх кількості.

Метою пакетної обробки в системі стане проведення аналізу з метою здійснення описової аналітики, яка відповідає на запитання “що сталося?”. Зазвичай пакетна обробка має на меті виконання одного з трьох видів аналізу, а саме згаданої вище описової аналітики, прогностичної аналітики, яка відповідає на запитання “що відбудеться?”, та аналітики “припису”, яка може відповісти на запитання “що робити, щоб досягти виконання поставленого завдання”. Цікаво, що у роботі з великими обсягами даних особливо корисною може стати прогностична аналітика, оскільки в контексті прогностичної аналітики чим більше даних для проведення аналізу, тим точніший буде результат. У нашому ж випадку доцільне здійснення описової аналітики, оскільки її використовують найчастіше, а ми зацікавлені в самій пакетній обробці як факті здійснення аналітики на великому об’ємі даних, систему ж завжди можна доповнити набором відповідних програм для інших видів аналітики.

Описова аналітика, як правило, передбачає побудову звітів за повний період. Вона бере дані та перетворює їх на те, що дослідники можуть візуалізувати, зрозуміти та інтерпретувати. Аналітика надає інформацію про історичні показники та відповідає на запитання про те, що сталося. Описові аналітичні звіти призначені для регулярного запуску та перегляду. Приклади містять звіти про клієнтів, операції, продажі тощо.

В нашому випадку згрупуємо, тобто класифікуємо отримані дані про температуру за їх значенням. Метою такого аналізу може слугувати, наприклад, попередня обробка для виявлення хибних вимірів (якщо датчик зафіксував від’ємну або занадто високу температуру для певного типу місцевості), або ж виділення групи датчиків, які постійно показують завищену або занижену температуру, завдяки чому можна виявити одразу цілі групи несправного обладнання, пошук та перевірка якого могла б зайняти дуже багато часу. Іншим прикладом може бути виявлення завищених або занижених вимірів за певний період конкретним типом обладнання тощо. Здійснення такої аналітики значно пришвидшить адаптацію системи до навколишнього середовища та зменшить вплив несправного обладнання і час, витрачений на виявлення такого обладнання тощо.

Для реалізації пакетної обробки вибрано Apache Spark. Одразу зазначимо що подальший аналіз статистичної аналітики із застосуванням алгоритмів машинного навчання неможливий у класах баз даних, відомих як MPP (massively parallel processing – масова паралельна обробка), які дають змогу виконувати аналітичні агрегації на великих об’ємах даних. Найістотніша відмінність полягає у тому, що бази даних MPP зосереджені на паралельному виконанні аналітичних запитів SQL на кластері машин, тоді як поєднання MapReduce та розподіленої файлової системи забезпечує щось набагато більше схоже на операційну систему загального призначення, яка може запускати довільні програми.

Такою програмою стане реалізація алгоритму кластеризації К-середнє (K-means). Результатом роботи потокової підсистеми буде набір даних, який зберігатиметься у базі даних DynemoDB. Цей набір і слугуватиме набором вхідних даних для пакетної обробки. Приклад даних зображено на рис. 2, де подано частину файла у форматі CSV (перша колонка – час вимірювання температури, далі відповідно uuid датчика, який здійснив вимірювання та відповідно саме вимірювання).

```

createdAt,createdBy,value
2020-10-17 07:32:21,be67faae-1f79-45c7-b218-dbeffe1c8ab2,38
2020-12-19 20:41:03,be67faae-1f79-45c7-b218-dbeffe1c8ab2,32
2020-08-23 21:54:46,e36f14bf-f469-44ef-ada5-5f26887b3e92,40
2020-12-23 20:39:49,61d565e9-64f9-4bf5-b8ef-e6660331dfbc,34
2020-12-13 08:30:35,e9859165-1ce2-40bc-85fe-f503a182ec9b,38
2021-01-14 16:37:48,93548b94-5888-4f45-9177-16243e755ee9,18
2021-03-05 06:46:59,f74cc5c9-e91e-4a66-b46a-b1ba776019ba,41
2020-10-18 03:52:37,d983c7ad-569c-4fad-8615-60d746fd4e75,30
2021-01-26 06:47:03,eb97a620-44d5-4e9e-8df8-e9e56aee78f1,15
2020-04-28 20:32:43,e9834a76-ada7-469e-a2be-43db255d47d4,17

```

Рис. 2. Приклад вмісту набору вхідних даних для пакетної обробки

Зазначимо, що цей файл може містити мільярди рядків, а отже, не може поміститися в пам'ять однієї машини. Місцем зберігання такого файла може стати одна із розподілених файлових систем, наприклад, HDFS Hadoop Distributed File System або Amazon S3 (одразу зазначимо, що Amazon S3 не є розподіленою файловою системою).

Тож наша система буде розширена пакетною підсистемою аналізу даних, яку можна побачити на рис. 3.

Рис. 3. Схема розробленої пакетної підсистеми для аналізу великих об'ємів даних

Проаналізуємо роботу системи за певний час, а саме за рік, для цього експортуємо файл у форматі CSV (приклад якого наведено вище). Файл містить дані про вимірювання, час та яким саме датчиком його здійснено. Накладенням алгоритму К-середнє спробуємо визначити непрацездатні датчики вимірювання температури в кластері IoT пристроїв.

Як видно з рис. 4, виміри розподілено на чотири кластери з відповідними їм центрами. Як бачимо, результат -100 градусів явно виходить за межі розумного, а отже, в кластері є несправні датчики.

Рис. 4. Розподіл вимірів температури за їх центрами та кількістю вимірювань у кластерах

Щоб визначити час несправної роботи, згрупуємо виміри за часом вимірювання (а саме за місяцем). Як видно з рис. 5, неправильні виміри отримано в період з квітня до серпня.

Рис. 5. Розподіл вимірів температури за значенням та часом

Згрупувавши виміри за їх кількістю та часом вимірювання (графік розподілу наведено на рис. 6), бачимо, що кількість вимірювань за цей період перевищує норму. Це можна пояснити введенням у цей період нового обладнання, яке певний час давало хибні показники, після чого його вивели з експлуатації.

Рис. 6. Розподіл кількості вимірювань температури за часом

Відкинувши кластер під номером один (рис. 7), бачимо, що температура тепер у допустимих межах, а кількість вимірювань без різких змін, що свідчить про відсутність вимірів датчиків, введених з квітня до серпня.

Рис. 7. Графіки розподілу вимірів у кластерах та кількість вимірювань, здійснених впродовж року

У цей момент можна сказати, що дані не містять хибних вимірів, а отже, в кластері немає несправних датчиків. Але, згрупувавши дані за ідентифікатором (рис. 8), одразу помічаємо, що два датчики впродовж року показують ту саму температуру, що малоймовірно, інші ж виміри підпорядковані нормальному закону. Отже, в кластері є ще два несправні датчики.

Рис. 8. Графіки розподілу вимірів температури за ідентифікатором датчика

Звернемо увагу, що останній графік побудовано за великої кількості датчиків. Таким способом навряд чи можливо визначити ідентифікатори несправних датчиків, оскільки довелось би переглядати виміри кожного датчика окремо на графіку (схожий запит можна було б виконати за допомогою баз даних MPP, не вдаючись до алгоритмів кластеризації). Якщо ж розглянути такий самий графік без відкидання кластера під номером один (рис. 9), бачимо, що ми відкинули одразу цілу групу, не аналізуючи показників окремих датчиків.

Рис. 9. Графіки розподілу вимірів температури за ідентифікатором датчика без відкидання кластера під номером один

Це підтверджує той факт, що аналітика із застосуванням MapReduce та розподіленої файлової системи, яка дає змогу виконувати довільні програми (в нашому випадку реалізація алгоритму К-середнє), має переваги над аналітикою із використанням баз даних MPP (Massive Parallel Processing), які зосереджені на паралельному виконанні аналітичних запитів SQL на кластері машин.

5. Порівняльний аналіз щодо систем, які існують

Для оцінювання роботи системи порівнюємо її з аналогами систем, код яких є у відкритому доступі (Open source), а саме ThingsBoard та SiteWhere. Для оцінки використаємо два показники:

1. Масштабованість. Платформа IoT повинна підтримувати оброблення тисяч і мільйонів пристроїв із мінімальною затримкою для підтримання критично важливих за часом програм. Для оцінки масштабованості систем порівнюємо пропускну здатність та час відгуку з іншими системами, оскільки велике значення пропускну здатності свідчить про високу продуктивність, важливий також фактор лінійної масштабованості, коли кількість пристроїв збільшується. Пропускную здатність обчислюють як кількість повідомлень, отриманих платформою за секунду. Низький час відгуку свідчить про швидкі реакцію та обробку [7].

2. Стабільність. Стабільність системи вимірюють із використанням кількох факторів, таких як завантаження процесора та активна пам'ять. Використання ЦП визначається з ядра як середнє використання всіх ЦП у системі. Активна пам'ять міститиме інструкції програми в мегабайтах (МБ), що зберігаються в пам'яті під час тесту.

Масштабованість

На рис. 10 та 11 подано результати пропускну здатності та середнього часу відгуку у випадку REST API. Для оцінки використаємо різну швидкість запитів – 100 і 200 запитів за секунду відповідно. Як показано на рис. 10, 11, ThingsBoard має вищу продуктивність у випадку, коли REST досягає максимуму на рівні 1010,7 і 696,3 запитів/с за обох швидкостей у 1000 та 800 підключених пристроїв. У SiteWhere пропускна здатність зростає до 359,2 запитів/с за 100 запитів та 320 запитів/с за 200 запитів, якщо підключено 500 пристроїв, потім вона починає зменшуватися, коли

кількість підключених пристроїв зростає до 800 та 1000. SiteWhere потребує більше часу, ніж Thingsboard, щоб обробити запити, а саме 422 та 1237 мс, проти 93 та 342 мс у Thingsboard за 1000 підключених пристроїв. Подвоєння кількості запитів призводить до збільшення кількості повідомлень, які потрібно обробляти на платформі, тож зі збільшенням часу оброблення зменшується пропускна здатність [7].

Рис. 10. Пропускна здатність REST API ThingsBoard та SiteWhere зі швидкістю 100 і 200 запитів за секунду

Рис. 11. Середній час відгуку REST API ThingsBoard та SiteWhere із швидкістю 100 і 200 запитів за секунду

На рис. 12 наведено залежність пропускної здатності від кількості підключених пристроїв для розробленої системи за 100 та 200 запитів за секунду відповідно. Як видно з рис. 12, пік пропускної здатності припадає на 400 та 200 пристроїв і становить 8250 та 8400 запитів за секунду відповідно, після чого пропускна здатність починає зменшуватись і рухається до позначки 7800 запитів за секунду в обох випадках. Незважаючи на невеликі піки, залежності залишаються лінійними в межах 7600 та 8400 запитів за секунду. Підвищення пропускної здатності за 200 зап/с та 800 підключених пристроїв можна пояснити збиранням сміття віртуальною машиною Java. Середній час відгуку системи (подано на рис. 13) також є лінійною залежністю і наближається до 120 та 115 мілісекунд, коли підключено 1000 пристроїв.

Рис. 12. Пропускна здатність REST API розробленої системи зі швидкістю запитів 100 і 200 запитів за секунду

Рис. 13. Середній час відгуку REST API розробленої системи зі швидкістю 100 і 200 запитів за секунду

На рис. 14 наведено залежність пропускну здатності у випадку використання протоколу MQTT. MQTT – це протокол обміну повідомленнями із застосуванням брокера повідомлень та підходу підписки–публікації, який використовується для пристроїв із обмеженою потужністю. MQTT має три рівні якості обслуговування (QoS). У нашому випадку використано рівень якості QoS-0. Він просто відправляє повідомлення один раз, не гарантуючи його доставки, отже, зв'язок дуже швидкий і має високу пропускну здатність, але він не є надійним, оскільки може відкинути деякі повідомлення. SiteWhere використовує зовнішнього посередника (брокера) для підтримки MQTT, для тесту продуктивності вибрано Eclipse Mosquitto. ThingsBoard має власного посередника (брокера) MQTT [27].

Пропускна здатність Sitewhere краща, ніж Thingsboard. Максимальна пропускна здатність, зафіксована на SiteWhere MQTT, становить 2928,3 та 4451,8 пов/с, якщо підключено 1000 пристроїв. Thingsboard приймає 994 та 1928 пов/с, якщо підключено 1000. Середній час відгуку MQTT є дуже малим наближенням до нуля, оскільки відгук дуже швидкий [7].

Зазначимо, що розроблена система не використовує MQTT для приймання даних, оскільки є лише прототипом системи, а відтак не потребує брокера MQTT, тому що MQTT дає змогу зменшити середній час відгуку системи, і може бути доцільним розширення системи для підтримки цього протоколу. Зазначимо, що використання брокера MQTT не означає відмови від використання брокера Kafka, оскільки Kafka зосереджується на зберіганні та читанні даних, націлюючись на сценарії потокової обробки даних з високою продуктивністю в реальному часі, тоді як брокер MQTT – на спілкуванні між клієнтом та сервером (в нашому випадку приймання повідомлень від пристроїв IoT).

Рис. 14. Пропускна здатність MQTT API ThingsBoard та SiteWhere зі швидкістю запитів 100 і 200 запитів за секунду

Зазначимо, що пропускна здатність розробленої системи, у випадку використання REST API, істотно перевищує пропускну здатність аналогів та навіть протокол MQTT. Це пояснюється використанням асинхронної відповіді. Отримавши запит, основна аплікація лише надсилає повідомлення брокеру та не чекає його оброблення, саме це дає змогу підвищити пропускну здатність. Але оскільки у реальної системи (не прототипу) пропускна здатність може деградувати із введенням додаткового функціоналу під час удосконалення системи та розширення кількості полів у запиті, іноді доцільне введення додаткового протоколу MQTT що також дасть змогу зменшити середній час відгуку системи, оскільки з використанням протоколу MQTT час відгуку може наближатися до нуля.

Стабільність

На рис. 15 показано використання центрального процесора під час надсилання платформі повідомлень зі швидкістю 100 пов/с із застосуванням REST та MQTT. У разі використання REST обидві платформи потребують більше циклів процесора для забезпечення багатопотоковості. ThingSboard споживає більше циклів процесора для обробки запитів і досягає 77,1 % за 1000 підключених пристроїв. З іншого боку, SiteWhere досягає максимального використання центрального процесора, якщо їх підключено 500. У разі застосування MQTT ситуація зворотна, тобто SiteWhere вимагає більше циклів процесора, щоб досягти 67,1 % за 1000 пристроїв [7].

Із застосуванням REST API розробленої системи для 10 пристроїв використання процесора не перевищує 3 %, що менше, ніж в аналогів. Але, як ми вже зазначали, ситуація може змінитись із введенням додаткових функцій. Якщо підключених пристроїв 100, використання процесора

наближається до 58 %, що співмірно з аналогом SiteWhere, але більше, ніж у ThingBord, яка за такої самої кількості пристроїв дорівнює 45 %. За всіх інших значень, навіть 1000 підключених пристроїв, використання процесора не перевищує позначку 70 %, що краще, ніж в аналогів.

Рис. 15. Використання процесора із застосуванням REST та MQTT API ThingsBoard та SiteWhere зі швидкістю публікації за 100 пов/с

Рис. 16. Використання процесора із застосуванням REST API розробленої системи зі швидкістю публікації 100 пов/с

Рис. 17 показує середні значення активної пам'яті в обох випадках REST та MQTT з фіксованою швидкістю 100 пов./с. З рис. 17 видно, що REST API споживає більше пам'яті, ніж MQTT на обох платформах. SiteWhere споживає більше пам'яті, ніж ThingsBoard, у разі використання обох протоколів. Максимальне споживання пам'яті SiteWhere становить 2548 МБ $\approx$ 2,4 ГБ за 1000 пристроїв із використанням REST та 730 МБ за 800 пристроїв у разі MQTT. З іншого боку, максимальне споживання пам'яті ThingsBoard становить 1900 МБ $\approx$ 1,85 ГБ, якщо підключено 800 пристроїв, із використанням REST, та 408,27 МБ, коли 500 пристроїв для MQTT [7].

Рис. 17. Середнє значення активної пам'яті для REST та MQTT API ThingsBoard та SiteWhere зі швидкістю надсилання повідомлень 100 пов/с, Мб

Як видно з рис. 18, розроблена система споживає значно менше пам'яті порівняно з аналогами. Навіть якщо підключено 1000 одиниць, використання пам'яті не перевищує 500 Мб. Проте з рис. 18 видно, що використання протоколу MQTT потребує менше пам'яті, тож у майбутньому може бути доцільним розширення системи протоколом MQTT, адже за кількості 1000 пристроїв SiteWhere та ThingsBoard використання пам'яті не перевищує 400 Мб.

Рис. 18. Середнє значення активної пам'яті для REST API розробленої системи зі швидкістю надсилання повідомлень 100 пов/с, Мб

Також доцільно порівняти систему із платними хмарними рішеннями. Відмінності вказано в таблиці [8].

Порівняння характеристик розробленої системи із відомими хмарними рішеннями

Платформа	Microsoft Azure	AWS	IBM Watson IoT	thethings.io	Розроблена система
1	2	3	4	5	6
Протоколи	HTTP, AMQP, MQTT	HTTP, MQTT, WebSockets	HTTP, MQTT	HTTP, MQTT, CoAP, Websockets	HTTP, можливість розширення іншими протоколами за необхідності

Продовження таблиці

1	2	3	4	5	6
Апаратне забезпечення	Intel, Raspberry Pi2, Freescale, Texas Instruments	Broadcom, Marvell, Renesas, Texas Instruments, Mcrochip, Intel	ARM, Texas Instruments, Raspberry Pi, Arduino Uno	Будь-яке	Будь-яке
Мова	.Net, Java, C, NodeJS	Java,C, NodeJS	C#, C Python, Java, NodeJS	Python, Node.js	Java
Аналітика	Конфігурація середньої складності	Проста конфігурація	Складна конфігурація	Проста конфігурація	Проста конфігурація, можливість аналітики як у реальному часі, так і на збереженому об'ємі даних
Ціноутворення	Оплата пов'язана із кількістю пристроїв та повідомлень за день	Оплата пов'язана з кількістю повідомлень	Оплата пов'язана з кількістю пристроїв, трафіком даних та зберіганням даних	Оплата пов'язана з кількістю пристроїв та обслуговуванням платформи	Оплата пов'язана лише з розгортанням системи

На основі переваг побудованої платформи можна стверджувати, що вона не поступається AWS, IBM або Azure, функціонально всі вони схожі. Однак із певних причин неможливо виконати порівняння один до одного, зокрема тому що:

AWS, Google і Azure – це в основному хмарні сервіси – IoT, як і інші, тоді як у нас спеціально створено ядро платформи IoT.

У нашому випадку ядро IoT є центральною функцією, а хмарні обчислення – функцією підтримки.

Інші платформи, по суті, побудовані як загальний каркас для всіх типів систем. У нашому випадку є платформа IoT, призначена для використання як окремого продукту після розширення.

Інші платформи здебільшого мають різні функції та доповнення, які “добре мати” на платформі IoT, але які можуть не знадобитися для всіх продуктів чи послуг IoT. З іншого боку, пропонується платформа містить саме те, що потрібно як мінімум для кожного продукту та послуги IoT.

Всі інші платформи є керованими, тоді як створене рішення дає змогу керувати власним. З погляду ціни, якщо порівняти, всі інші платформи приблизно однакові.

Такі пропозиції, як автоматичне масштабування, безпека та загальне управління, однакові, незалежно від того, чи потрібно створити власний продукт, чи купити хмарне рішення.

Наша пропозиція має перевагу цілеспрямованої побудови, вона може споживати мало ресурсів і добре налаштовуватися під задачі. Рішення, за суттю, не є багатоцільовим, і не має надмірностей, як хмарні служби [8].

Тож, порівнявши системи, можна стверджувати, що розроблена система має кращі показники за пропускну здатністю (у випадку використання REST із можливістю розширення до MQTT),

середнім часом відгуку, використанням процесора, вона повністю незалежна від апаратного забезпечення, не містить надмірної кількості функцій, як хмарні рішення, надає широкі можливості для аналітики, а оплата пов'язана лише із розгортанням системи.

Список використаних джерел

- [1] Kleppmann M. (2017), *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*, Springfield Missouri United States, O'Reilly Media Inc., pp. 412–425.
- [2] Nathan M., James W. (2001), *Big Data: Principles and best practices of scalable real time data systems*, New York, United States, pp. 320–343.
- [3] Anand R., Jeffrey D. (2014), *Mining of Massive Datasets*, Stanford University, United States California, pp. 422–427.
- [4] Nathan M., James W. (2014), *Big Data: Principles and best practices of scalable real time data systems*, New York, United States, pp. 312–226.
- [5] Marr B. (2015), “Big Data: Using SMART Big Data, Analytics and Metrics to Make Better Decisions and Improve Performance”, *QUIS5 Quality in Services Conference*, Wiley, Karlstad, pp. 222–226.
- [6] Srivatsa H., Jagadeesh M. “Big Data Imperatives: Enterprise “Big Data” Warehouse, “BI” Implementations and Analytics, Apress, New York, New York, United States, June 24, 2013, pp. 311–333.
- [7] Ahmed Alaa, Haitham Hamza, Amira Kotb, “Performance Evaluation of Open Source IoT Platforms”, available at: https://www.researchgate.net/publication/330582133_Performance_Evaluation_of_Open_Source_IoT_Platforms (Accessed 27 April 2021).
- [8] Daniel Toran “IoT Platforms Comparisons”, available at: <https://upcommons.upc.edu/bitstream/handle/2117/114211/tfg-report-daniel-toran.pdf?sequence=1&isAllowed=y> (Accessed 27 April 2021).

RESEARCH OF DATA COLLECTION AND PROCESSING METHODS IN DISTRIBUTED INFORMATION SYSTEMS

S. Zhuravel, S. Dumych, O. Shpur

Lviv Polytechnic National University, 12, S. Bandery Str., Lviv, 79013, Ukraine

The article is devoted to the problem of processing and analysis of large amounts of data. Today, with the advent of sources of information that can produce large amounts of data, data collection is becoming an increasingly complex institution. To solve this problem, a wide range of technologies for specific purposes has been developed. However, most of them can be attributed to two paradigms of data processing, namely streaming and batch. In this article, we will consider the above paradigms and technologies used to build data processing solutions and try to outline several characteristics that need to be considered when building data processing systems. Also, based on the most promising solutions, the implementation of the system of data collection and analysis from IoT devices is given.

Key words: distributed information systems; data processing; large amounts of data.