Міністерство освіти і науки України
Державний університет «Одеська політехніка»

Інститут штучного інтелекту та робототехніки
**Кафедра «Комп'ютерні системи»**

**Колесніченко Дмитро Віталійович,**
студент групи УК-161

# КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ЗБЕРІГАННЯ І ОБРОБКИ ВЕЛИКИХ ДАНИХ В УМОВАХ ОБМЕЖЕНИХ РЕСУРСІВ

Спеціальність: 123 – "Комп'ютерна інженерія"
Спеціалізація: Спеціалізовані комп'ютерні системи

**Керівник:**
Ситніков Валерій Степанович,
доктор техн. наук, професор

Одеса — 2021

# ЗМІСТ

ВСТУП ..... 5

1. АНАЛІТИЧНИЙ ОГЛЯД ..... 6

1.1. ОГЛЯД БАЗ ДАНИХ ..... 6

1.2. ВИДИ БАЗ ДАНИХ ..... 7

1.3. ДОПОМІЖНІ СЕРВІСИ ДЛЯ БАЗ ДАНИХ ..... 15

1.4. АНАЛІЗ ТЕХНІЧНОГО ЗАВДАННЯ ..... 19

1.5. ВИСНОВКИ ..... 23

2. ДОСЛІДЖЕННЯ КОМПЛЕКСНОГО МЕТОДУ ЗБЕРІГАННЯ І ОБРОБКИ ДАНИХ
**ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.**

2.1 ДОСЛІДЖЕННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ ЗБОРУ ТА ОБРОБКИ ВЕЛИКИХ ДАНИХ
**ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.**

2.2 ДОСЛІДЖЕННЯ ІСНУЮЧИХ МЕТОДІВ ЗБЕРІГАННЯ ВЕЛИКИХ ДАНИХ ..... 27

2.3 ДОСЛІДЖЕННЯ ІСНУЮЧИХ МЕТОДІВ ОБРОБКИ ДАНИХ З СЕРВЕРІВ ..... 29

2.4 СТВОРЕННЯ КОМПЛЕКСНОГО МЕТОДУ ЗБЕРІГАННЯ ВЕЛИКИХ ДАНИХ ..... 29

2.5 ВИСНОВКИ ..... 31

3. СТВОРЕННЯ КОМПЛЕКСНОГО МЕТОДУ ЗБЕРІГАННЯ І ОБРОБКИ ДАНИХ ..... 32

3.1. ВИМОГИ ДО РОБОТИ УДОСКОНАЛЕННОЇ СИСТЕМИ ТА ВИБІР ЇЇ КОМПОНЕНТІВ ..... 32

3.2. ВИБІР КОМПОНЕНТІВ ..... 32

3.3. КЕШУВАННЯ ДАНИХ ..... 41

3.4. ВИБІР SSD ТА ПІДКЛЮЧЕННЯ ..... 45

3.5. РЕАЛІЗАЦІЯ КОМПЛЕКСНОГО МЕТОДУ ЗБЕРІГАННЯ ТА ОБРОБКИ ДАНИХ ..... 47

4. ЕКСПЕРЕМЕНТУВАННЯ РОБОТИ СИСТЕМИ ..... 64

4.1 ЕКСПЕРЕМЕНТУВАННЯ РОБОТИ MYSQL ..... 64

4.2 ЕКСПЕРЕМЕНТУВАННЯ РОБОТИ CLICKHOUSE З KAFKA ..... 66

ВИСНОВКИ ..... 68

СПИСОК ЛІТЕРАТУРИ ..... 69

ДОДАТКИ А-Б ..... 72

# ВСТУП

Історія баз даних розпочинається разом із створення перших компьютерів. Спочатку це були просто кімнати у яких зберігали якусь інформацію на перфокартах. З часом, зі створенням перших дискет та жорстких дисків з’явилася змога зберігати дані у вигляді файлів. Але перша революція сталася у історії баз даних починається з одного з найбільших інженерних подвигів минулого століття: польоту на Місяць.

Американська компанія Rockwell склала контракт разом із урядом США на участь у космічному проекті Apollo. Так як побудова космічного корабля включає у себе збирання мільйонів деталей та зв’язок між ними, то була створена система зберігання та управління файлами, яка мала змогу відстежувати інформацію про всі деталі. Однак у ході наступних перевірок виявилася велика проблема. З'ясувалося, що велика частина даних повторюються у декількох файлах і тому даних стало занадто багато.

Зіткнувшись із завданням зберігання інформації про велику кількість деталей, компанія Rockwell разом із IBM у 1968 році розробила автоматизовану систему замовлень. Названа IMS (Information Management System – система управління інформацією), це було основою концепції СУБД.

Мета дослідження ж цієї дипломної роботи – підвищити ефективність зберігання та обробки великих даних у поточній системі, використовуючи існуючі ресурси чи з мінімальним підвищенням бюджету. Така необхідність виникла через збільшення обсягів даних необхідних для зберігання, надійність їх зберігання та підвищення швидкості доступу до них, так як поточні методи є застарілими та повільними, а кількість даних збільшилася у рази. До того ж потрібно ще й зробити систему легкою до масштабування.

Об’єктом дослідження цієї роботи є процес зберігання і обробки великих даних, взаємодії бази даних з іншими компонентами програмної системи. Предметом дослідження є методи зберігання та пристрої обробки великих даних у умовах обмежених обчислювальних ресурсів.

Інноваційність роботи полягає у тому, що наразі дуже важко знайти комплексні методи, які допомагають невеликим компаніям зберігати великі дані, не масштабуючи систему та використовуючі існуючі ресурси. Більшість невеликих та середніх продуктових компаній використовують застарілі методи, а зі збільшенням даних – збільшують обчислювальні ресурси та дискові масиви. Даний метод допомагає розвернути швидку та надійну систему, яку буде легко масштабувати та змінювати під свої потреби, а окрім цього вона буде дуже економною. Мову програмування також можна використовувати будь яку.

За матеріалами досліджень підготовлена публікація у науковий журнал.

# 1. АНАЛІТИЧНИЙ ОГЛЯД

## 1.1. Огляд баз даних

База даних (БД) - це впорядкований набір логічно взаємопов'язаних даних, які існують для того, щоб зберігати та використовувати необхідну користувачам інформацією. [1] Такою інформацією можуть бути:

- Дані за замовленнями у інтернет магазині
- Дані користувачів соціальної мережі
- Досягнення у онлайн грі
- Наукові дослідження та інше.

Головне завдання БД - це гарантоване зберігання великих обсягів інформації (так звані записи даних) та надання швидкого доступу до неї коричтувачам, пристрою чи програмі. Типова база даних зображена на рис. 1.1.

Таким чином, БД складається з двох частин:

- Збереження інформації
- Системи керування нею

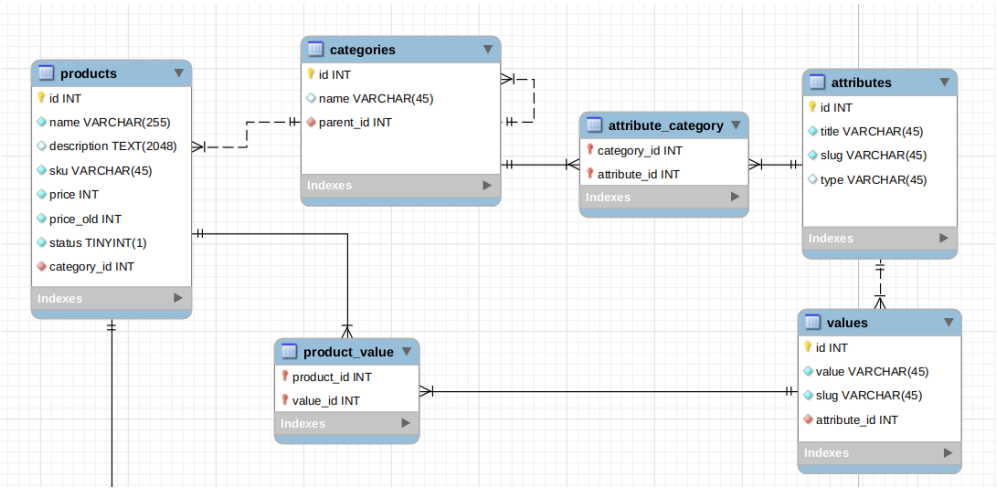



Рисунок 1.1 – Типова база даних інтернет магазину

## 1.2. Види баз даних

Серед найпоширеніших баз даних виділяють наступні (від найперших до найновіших):

- Прості структури даних
- Реляційні бази даних чи SQL бази даних
- NoSQL бази даних
- NewSQL бази даних

Хоча і серед программістів часто можно почути думку, що документо-оріентовані бази (NoSQL) використовують частіше та постійно заміщують реляційні бази даних (SQL) – це є не зовсім так. [2]

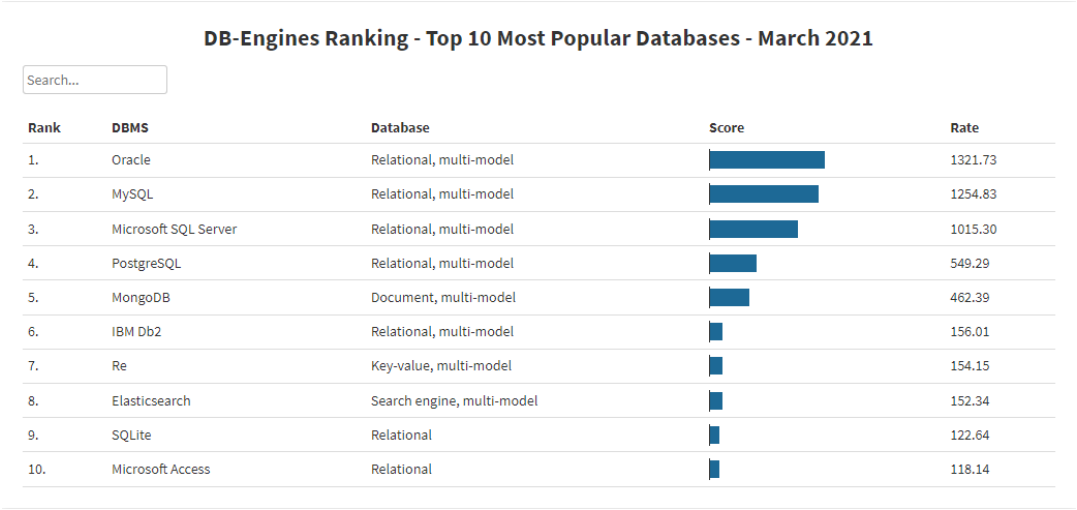
Насамперед причиною таких думок є те, що на сьогодняшній день більшість программістів працюють на аутсорс компанії, тобто ті компанії, що роблять швидко нові додатки. У такому випадку достатньо використати MongoDB, чи просто використати потужності Amazon, Google Cloud або Microsoft Azure. Але у будь якого швидкого рішення є свої мінуси. У випадку з MongoDB – низька швидкість при зберіганні великої кількості даних та їх великий розмір при зберіганні. У випадку ж з Amazon, Google Cloud та Microsoft Azure – це дороге обслуговування та потреба платити за кожний використаний гігабайт місця.

Тож якщо мова йде про будь яку продуктову компанію, то її керівництво зацікавлено у економії бюджету на зберіганні великої кількості данних та у швидкому доступі до них. Найкращим вибором у такому випадку є використання реляційних баз даних та особистих чи арендованих серверів.

Серед найпоширеніших таких баз даних є Oracle, MySQL, Microsoft SQL Server та PostgreSQL. [3]

На рисунку 1.2 показано як станом на березень 2021 року виглядає запит на бази даних, їх релевантність та рейтинг, за даними DB-Engines Ranking. І це

не є дивно, так як все-таки банки, великі продуктові компанії, та інші все таки використовують саме реляційні бази даних.



DB-Engines Ranking - Top 10 Most Popular Databases - March 2021

Search...

| Rank | DBMS | Database | Score | Rate |
|---|---|---|---|---|
| 1. | Oracle | Relational, multi-model | | 1321.73 |
| 2. | MySQL | Relational, multi-model | | 1254.83 |
| 3. | Microsoft SQL Server | Relational, multi-model | | 1015.30 |
| 4. | PostgreSQL | Relational, multi-model | | 549.29 |
| 5. | MongoDB | Document, multi-model | | 462.39 |
| 6. | IBM Db2 | Relational, multi-model | | 156.01 |
| 7. | Re | Key-value, multi-model | | 154.15 |
| 8. | Elasticsearch | Search engine, multi-model | | 152.34 |
| 9. | SQLite | Relational | | 122.64 |
| 10. | Microsoft Access | Relational | | 118.14 |

Рисунок 1.2 – Рейтинг баз даних станом на березень 2021 року

### 1.2.1. Прості структури даних

До напростіших баз даних, які використовуються майже всюди, хоч і ми можемо не здогадуватися про них, відносяться прості структури даних. У такому вигляді зберігаються, наприклад, паролі чи права на файли у Unix-подібних системах. Приклад такої схеми зображено на рисунку 1.3

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
syslog:x:102:106::/home/syslog:/usr/sbin/nologin
bob:x:1000:1000:Bob Smith,,,:/home/bob:/bin/bash
```

Рисунок 1.3 – Приклад простої структури даних

У такій базі даних, зазвичай, дані зберігаються через двокрапку (:) чи через кому (,) чи через вертикальну лінію чи декілька (|).

Плюси:

- Можна зберігати дані у будь-якому вигляді та будь якого типу
- Не потрібно встановлювати ніякий софт
- Швидко сконфіговувати

Мінуси:

- Важко робити пошук
- Погана агрегація
- Відкрите зберігання, а з тим проблеми з конфідеційністю та безпекою
- Неможливість зберігати великі дані

### 1.2.2. Реляційні БД (SQL)

Реляційні БД є найросповсюдженішими. Окрім швидкого пошуку, та стабільності вони пропонують велику кількість можливостей для конфігурації. [4] Такі бази даних зберігаються у вигляді таблиць, у яких кожна колонка відповідає за деякий тип даних та його довжину. До найросповсюдженіших реляційних баз даних відносяться Oracle, MySQL та PostgreSQL. [5] Серед них є як і рядкові реляційні бази даних, на кшталт MySQL, так і стовпцові, наприклад ClickHouse. [6] Різниця полягає у тому, що стовпцеві бази швидше дістають усю колонку, ніж це роблять рядкові, які у свою чергу швидше дістають деякі записи за деякими значеннями у базі. На рисунку 1.4 зображено структуру бази.

Таким чином ClickHouse на відміну від MySQL краще агрегує великі масиви даних, в той час як остання СУБД краще дістає деякі рядки.

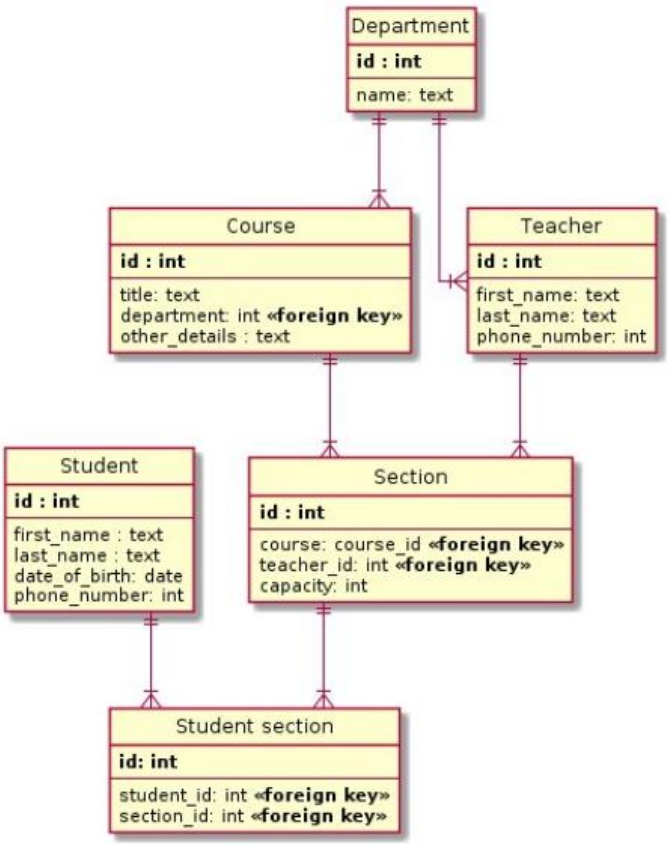



Рисунок 1.4 – Приклад структури реляційної бази

Плюси:

- Можливість конфігурувати як хочеться
- Швидкий пошук
- Ефективне зберігання
- Велика спільнота підтримки

Мінуси:

- Великий порог входження
- Можливість випадково зробити недостатньо хорошу безпеку
- Важка конфігурація та необхідність планування бази даних

### 1.2.3. NoSQL бази даних

Серед баз, що користуються великим попитом серед маленьких аутсорсінгових компаній та звичайних користувачів, нерідко проскакують такі бази, як MongoDB, Aerospike чи Redis. І це не є дивно, так як такі бази, які у свою чергу називаються документоорієнтованими, дуже швидко конфігуруються та покривають більшу частину запитів. Вони виглядають як JSON об’єкт, у якому в рамках однієї бази чи документи можуть бути дані різних типів, різного виду та з різною кількості полей.[7]

Це є найкращим вибором, якщо стан продукту може змінюватися у ході розробки, а необхідність підтрімувати стабільність та швидкість розробки більша, аніж час та гроші на наймання спеціаліста по розробці баз даних.

До того ж такі бази краще захищені. Приклад структури такої бази можно побачити на рисунку 1.5.

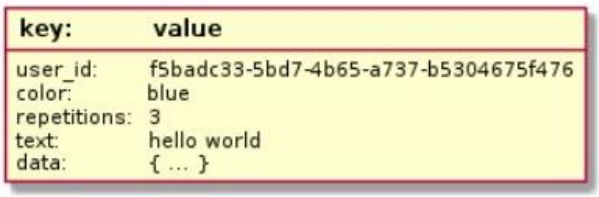



Рисунок 1.5 – Приклад структури документоорієнтованної бази даних

Плюси:

- Швидкість розробки
- Можливість налаштувати швидкий пошук
- Можливість змінювати структуру бази чи документа у ході розробки

Мінуси:

- Неефективне зберігання та непостійна структура
- Значно менша швидкість на відмінно від реляційних баз даних

### 1.2.4. NewSQL та багатомодельні бази даних

NewSQL і багатомодельні БД є різними типами баз даних, але вирішують єдину групу проблем, викликаних односторонніми підходами реляційних або документоорієнтованих баз даних. Чому б не об'єднати переваги обох груп? Так робить NewSQL бази даних. Вони успадковують реляційну структуру і семантику, але побудовані з використанням сучасних та масштабованих конструкцій. Основною метою було забезпечення кращої масштабованості, ніж реляційні БД, та більш високі гарантії узгодженості, аніж у NoSQL. [8] Таким чином компроміс між узгодженістю і доступністю є фундаментальною проблемою розподілених баз даних, описаних теоремою CAP.

Це дає змогу використовувати такі бази даних у більшості випадків, коли не має змоги жертвувати швидкістю чи стабільністю. Серед найвідоміших таких баз даних є MemSQL, VoltDB та CockroachDB. Така структура зображена на рисунку 1.6.

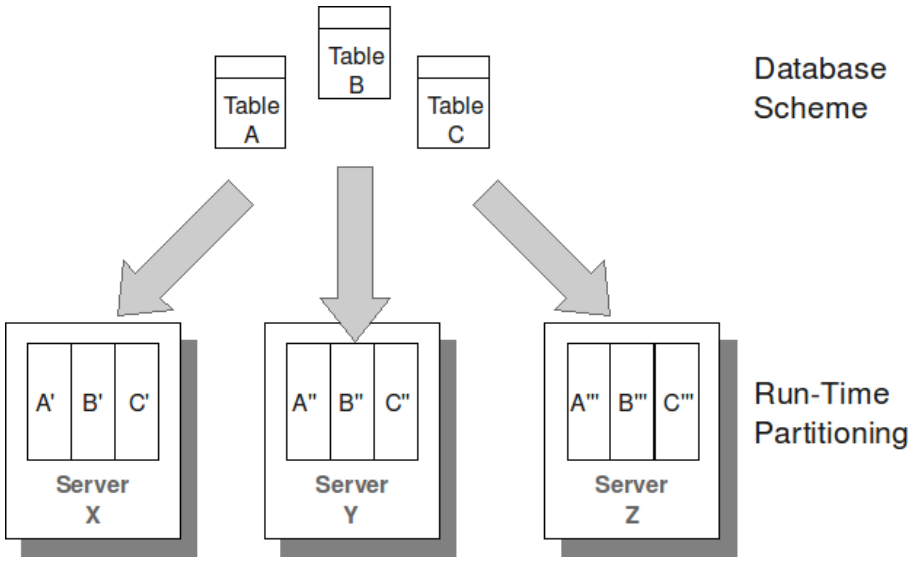



Рисунок 1.6 – Приклад структури багатомодельної бази даних

Єдиними мінусами є погана підтримка користувачів та недостатня довіра. Тож поки що єдиним вибіром є традиційні бази даних.

### 1.3. Допоміжні сервіси для баз даних

Окрім самих баз даних ми можемо звернутися за допомогою до деяких сервісів, взаємодія СУБД з якими робить швидшим пошук чи зручнішим зберігання даних. До таких сервісів відносяться:

- Брокери повідомлень
- Системи кешування даних
- Пошукові сервіси

Єдиний недолік таких сервісів - це необхідність знати їх всі, як з ними працювати та у якому випадку вони можуть знадобитися.

#### 1.3.1. Брокери повідомлень

Першими серед допоміжних сервісів ми розглянемо брокери повідомлень. Основна задача таких сервісів є швидка передача даних між серверами, мовами чи різними базами даних. [9] Працюють зазвичай вони за принципом, що є producer та consumer, який ще іноді називають subscriber.

Producer відправляє дані до серверу з брокером повідомлень, у той час, як Consumer очікує на дані та збирає їх з серверу як тільки вони з’являються.

Серед найпоширеніших брокерів повідомлень можна виділити Kafka та RabbitMQ. Структура брокера повідомлень показана на рисунку 1.7.

Ми будемо використовувати Kafka, так як у нього є змога ефективно працювати у парі з Clickhouse Summing Merge Tree.

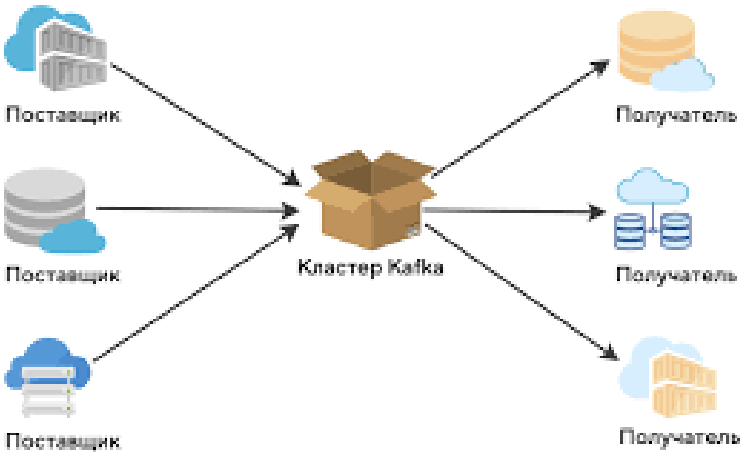



Рисунок 1.7 – Структура брокеру повідомлень Kafka

### 1.3.2. Системи кешування даних

Також для швидкого тимчасового зберігання ми можемо використовувати системи кешування даних. Вони дають змогу зберігати дані за схемою “хеш – значення”, а саме зберігання роблять у оперативній пам’яті. [10] Таким чином зберігання є дуже швидким, але для невеликої кількості даних, які потрібно постійно збирати іншими базами даних.

Таким є сервіс Memcached. Його дуже легко підключити майже у будь-яку кодову базу і на даний час він є найшвидшим серед таких сервісів. Приклад використання Memcached на рисунку 1.8. Він є частим вибором у HighLoad.

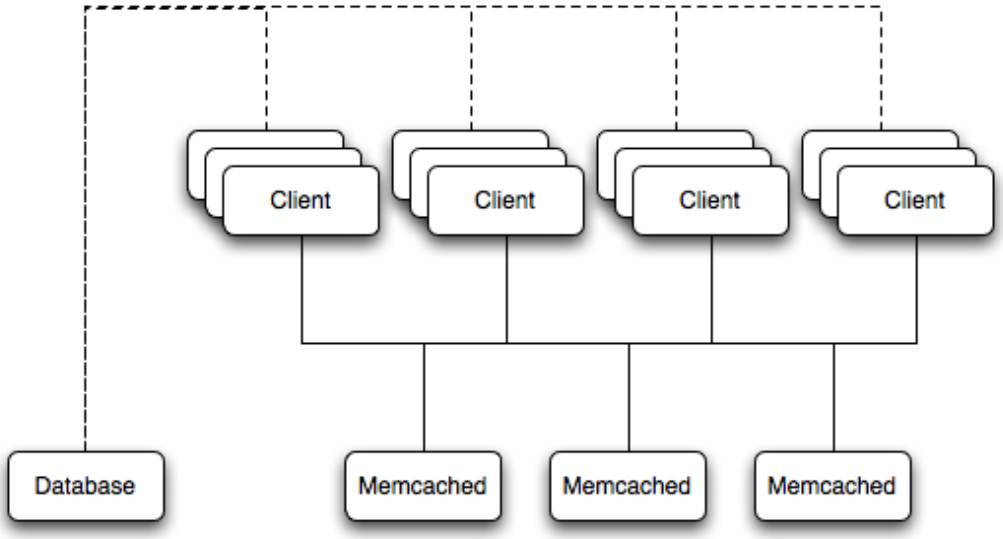



Рисунок 1.8 – Приклад використання сервісу кешування Memcached

### 1.3.3. Пошукові сервіси

Пошукові сервіси використовуються разом з СУБД, щоб покращити швидкість виконання специфічних запитів. Вони потребують вміння ними користуватися та великого часу на конфігурування.

Найвідомішим серед таких є Elasticsearch - пошуковий двигун з json rest api, що використовує Lucene і написаний на Java. Опис всіх переваг цього двигуна доступно на офіційному сайті.

Подібні двигуни застосовуються при складному пошуку за основою документів. Наприклад, пошук з урахуванням морфології мови чи пошук за geo координатами. [11] Структура Elasticsearch зображена на рисунку 1.9.

Я б хотів мати змогу використовувати його у дипломній роботі, але на його використання та вивчення потрібно потратити велику кількість часу.

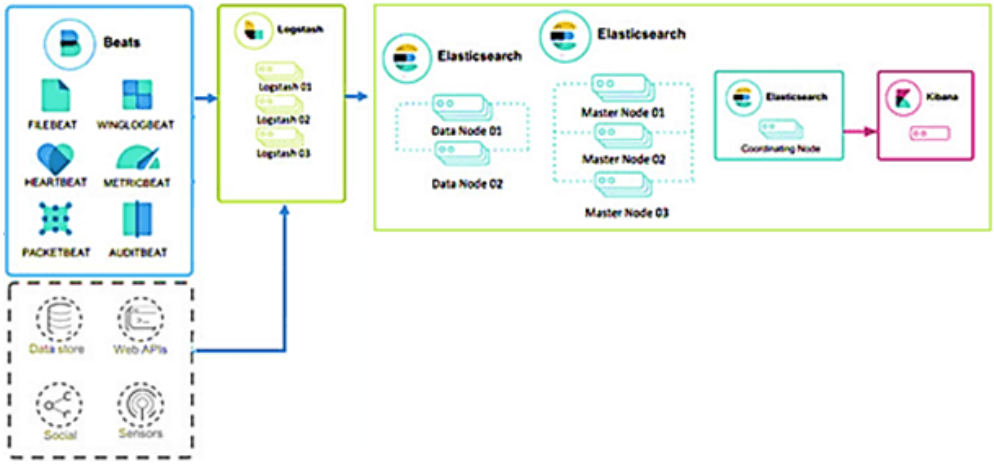



Рисунок 1.9 – Структура Elasticsearch

## 1.4. Аналіз технічного завдання

Основне завдання полягає в тому, щоб збільшити на проекті ефективність зберігання великої кількості даних в умовах обмежених ресурсів, підвищити швидкість їх обробки, вставку нових даних та об’єм даних, що займають місце у сховищі.

Даний продукт займається рекламою, а саме аукціонами по куплі місць на сайті та продажу реклами від рекламодавців. Тобто коли від власнику сайту приходить запит на рекламу на якесь місце на сайті, потім ми робимо запит у рекламодавців на такий вид, тип та розмір реклами. Серед усіх відповідей ми знаходимо найкращу та відправляємо її власнику сайту.

Так як даний продукт потребує зберігання великої кількості даних: переможців аукціонів, інтереси користувачів, запити з сайтів та додатків, то потрібно їх ефективно зберігати. Доступ до них має бути швидкий, стабільний і економити місце на зберігаючому пристрої.

Для того, щоб бути впевненим у порядку дій, розпочнемо з аналізу бази MySQL за допомогою таблиць, які пропонує нам використовувати документація. Також для того, щоб мати уявлення про кількість запитів до

серверів, бази даних та обсяг даних будемо використовувати Grafana, який э одним із інструментів Prometheus, підключений раніше до проекту. [12] Це дає нам змогу у реальному часі у браузері бачити зміни, які трапляются після зміни будь чого у проекті.

Для того, щоб взяти з бази даних MySQL обсяг даних використаемо наступний запит, який виводить дані по всім таблицям у БД:

```
SELECT table, round(sum(bytes) / 1024/1024/1024, 2) as size_gb from system.parts WHERE active GROUP BY table;
```

Результатом будуть дані у GB зображені на рисунку 1.10 . Ця БД є базою, яка зберігає основні дані за користувачами, покази які трапилися, їх приклади та конфігурацію всіх точок входу та виходу. Так як більшість таблиць є варіаціями однієї і тієї самої, у формат CSV я виніс тільки  найновіші на момент розробки.

| 1 | table_name | size_in_GB |
|---|---|---|
| 2 | impressionsTable_07_2021 | 40.92 |
| 3 | impressionsTable_06_2021 | 38.99 |
| 4 | impressionsTable_02_2021 | 35.78 |
| 5 | impressionsTable_03_2021 | 34.96 |
| 6 | impressionsTable_04_2021 | 33.63 |
| 7 | impressionsTable_03_2021 | 32.96 |
| 8 | impressionsTable_02_2021 | 32.68 |
| 9 | impressionsTable_06_2021 | 30.99 |
| 10 | impressionsTable_02_2021 | 28.78 |
| 11 | impressionsTable_05_2021 | 28.34 |
| 12 | impressionsTable_05_2021 | 25.34 |
| 13 | impressionsTable_08_2021 | 24.84 |
| 14 | impressionsTable_07_2021 | 19.92 |
| 15 | impressionsTable_04_2021 | 15.63 |
| 16 | adm | 15.32 |
| 17 | domains | 5.32 |
| 18 | impessionsExamples | 4.61 |
| 19 | dspSettings | 0.53 |
| 20 | sspSettings | 0.42 |

Рисунок 1.10 – БД MySQL з основною інформацією та кількість використовуємого простору диску.

Отож ми бачимо, що таблиці зберігаються щомісяця. Це необхідно, так как партнери можуть зробити запит на дані, які були минулого року та й взагалі на протязі усього існування контракту. А так як MySQL э не найшвидшою базою даних [13], якщо пошук буде за якоїсь конкретної години чи дня у повній таблиці, то раніше було прийняте рішення розділяти за місяцями.

Далі у Grafana було виділено скільки використовує трафіку БД (рисунок 1.11), скільки підключень до бази (рисунок 1.12) та скільки було table locks (рисунок 1.13). Останнє – це величина, яка показує частоту необроблених запитів через велику частину підключень, та середній час затримки через це.

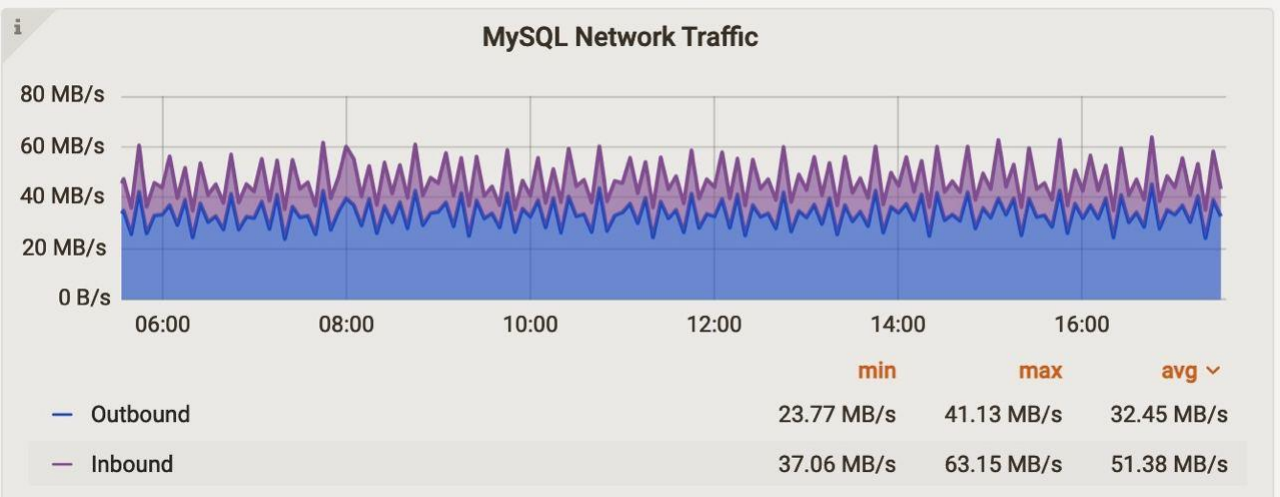



Рисунок 1.11 – Використання інтернет трафіку БД

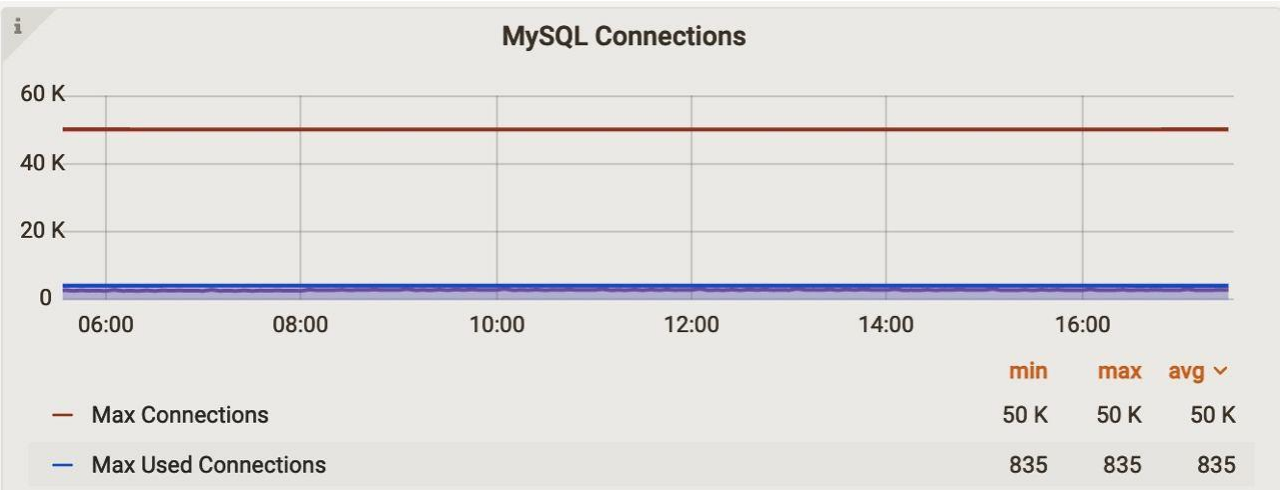



Рисунок 1.12 – Кількість підключень до БД

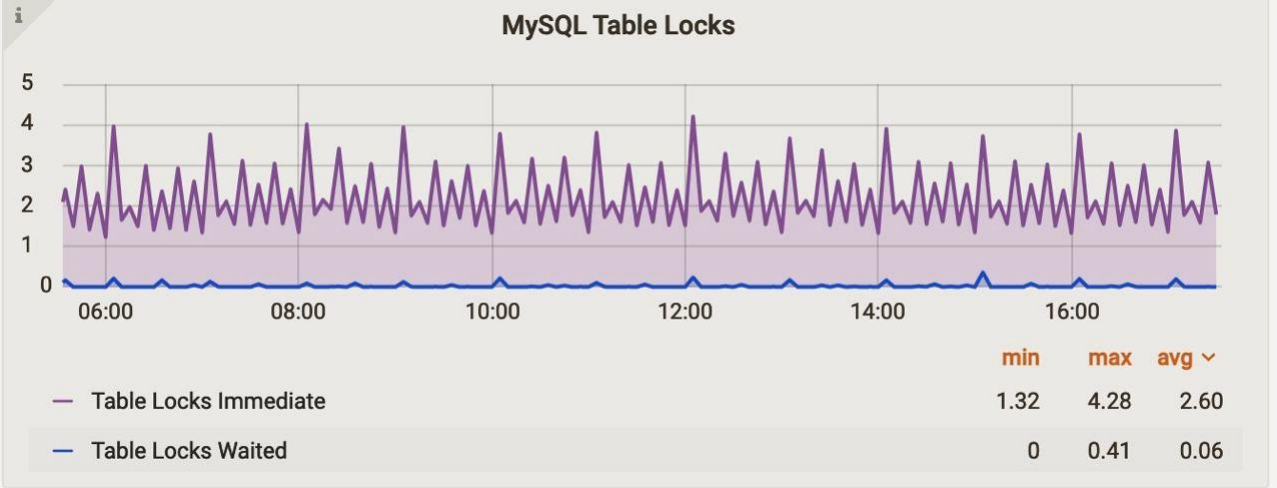



Рисунок 1.13 – Кількість Table Locks БД

Як ми можемо побачити, використовуеться дуже велика кількість інтернет трафіку, а інтернет трафік у датацентрах зазвичай виходить дорожче, за ренту серверів. Це пов’язано насамперед з великою кількістю запитів, які можна зменшити у рази. Також, через велику кількість запитів можуть з’являтися table lock, які не дають змогу зробити звернення до бази та додати запис чи достати з неї щось. І хоча спочатку MySQL база для таких даних підходила майже ідеально через хорошу архітектуру – з часом стало зрозуміло, що треба з цим щось робити.

Основні параметри, вплив на які ми досліджуємо, це:

- Кількість займаємого місця даними, не втарачаючи детальність
- Швидкість доступу до даних
- Швидкість зберігання та пошуку даних
- Кількість використовуємого інтернет-трафіку
- Кількість одночасних звертань до даних
- Кількість невиконаних звертань

## 1.5. Висновки

У цьому розділі я дослідив історію виникнення СУБД, їх види, сфери використання та актуальність. Було зроблено короткий екскурс по сучасним базам даних, та як їх зазвичай використовують. Також дослідив які проблеми ми маємо наразі.

Обмеженими ресурсами ж є сервер, з 12-ядерним процессором та 32 гігабайтами оперативної пам’яті з жорстким диском на 16 терабайтів.

Тепер можна переходити до більш детального дослідження комплексного методу зберігання великих даних, проблем які треба вирішити та як зберігати великі дані не використовуючи додаткові потужності.

# 2. ДОСЛІДЖЕННЯ МЕТОДІВ ЗБЕРІГАННЯ ВЕЛИКИХ ДАНИХ

## 2.1 Дослідження інформаційної системи збору та обробки великих даних

Так як наше завдання полягає в тому, щоб підвищити єфективність зберігання даних, зменшити їх вагу, підвищити швидкість доступу до них та мати можливість зберігати більш детальну інформацію, то нам треба для початку дослідити яка схема роботи на даний час використовується у системі.

На рисунку 2.1 можно побачити, що наразі ми використовуємо тільки MySQL та Nodejs, що є не найкращим варіантом коли мова заходить за великі дані. [15] На кожний запит, що приходить, ми відповідаємо і зберігаємо його у оперативній пам’яті за допомогою NodeJS. Окрім цього дані теж видаляються силами NodeJS що дає ще більше навантаження. Після того, як ми отримали показ ми одразу його записуємо до MySQL і теж силами NodeJS. Важливо знати, що NodeJS э однопотоковою мовою, хоч і можливо увімкнути процеси на окремих ядрах. Це є дуже великим ударом по продуктивності. [16]

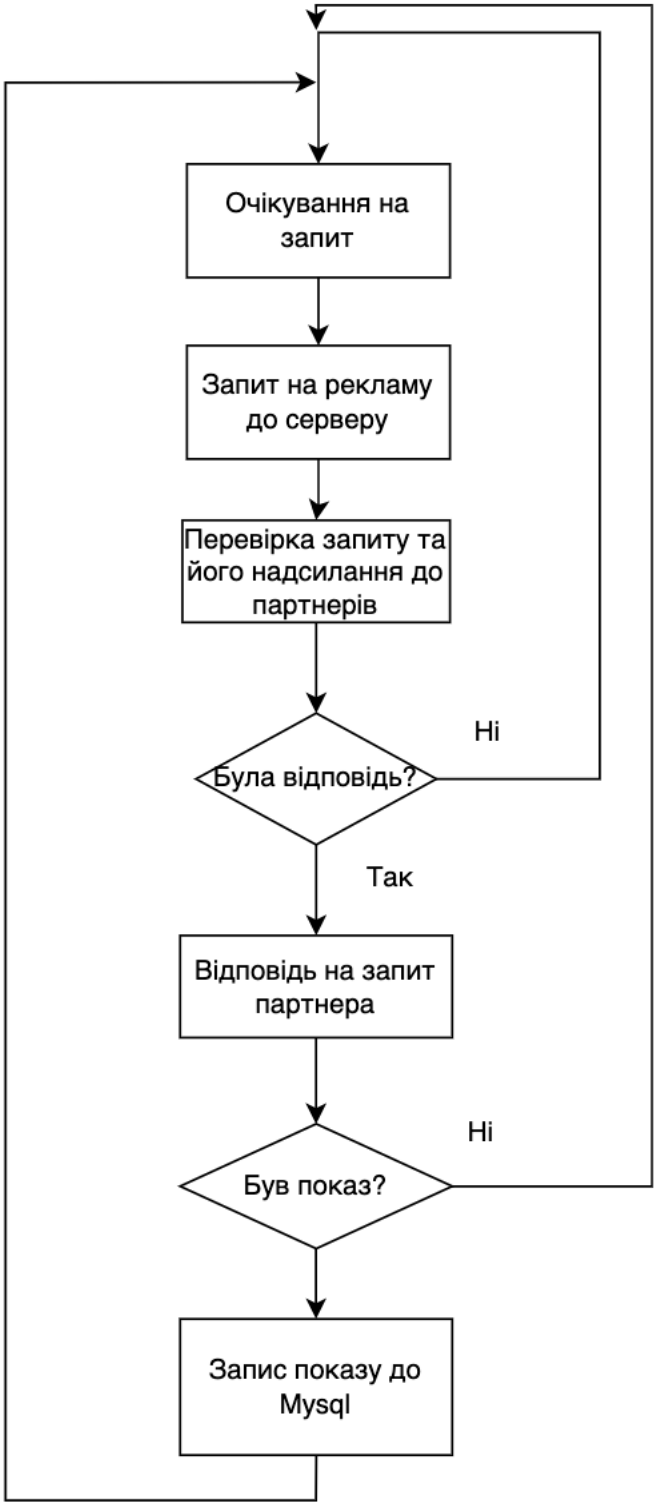



Рисунок 2.1 – Поточна система зберігання показів та обробки запитів

Так як ми зберігаємо відповіді за допомогою NodeJS то ми можемо їх втратити якщо ми просто перезавантажимо процес, що є втратою точності, бо як тільки з’являться нові покази, в нас вже не буде даних про них, а тому ми не зможемо зберігати їх та відповідати партнерам. Щоб не було втрат, перед кожною зміною коду треба вимикати отримання та відправлення запитів на 20 хвилин, щоб отримати усі покази і тільки після цього перезавантажувати процеси, щоб код став актуальним.

Окрім цього, MySQL сама по собі не є найкращим варіантом для зберігання великих даних, так як вона є строковою базою даних. [17] А це значить, що якщо ми будемо отримувати схожі дані від серверу, то ми не зможемо їх агрегувати та швидко діставати потрібні нам колонки, якщо нам, наприклад, для статистики потрібні тільки деякі поля.

Через те, що ми не можемо агрегувати дані, ми втрачаємо багато місця лише на зберігання однотипних массивів інформації. А щоб їх дістати, треба зробити великий запит, який буде довго виконуватися, збирати необхідні дані і після цього силами мови, якою це буде виконуватися – агрегувати.

До того ж таких викликів буде не один на п’ять хвилин, а може бути й двадцять чи більше, так як запит на інформацію робиться від декількох людин одночасно. Це дуже сповільнює роботу фронтенду та робить доступ до інформації дуже довгим.

Окрім цього треба не забувати, що за трафік у датацентрі теж треба платити, тож якщо робиться велика кількість запитів, то це є дуже потужним навантаженням на мережу.

Щоб вирішити всі ці питання ми можемо перейти до дослідження сучасних методів зберігання великих даних. [18] До того ж у нас вже є потужні сервери, ресурси яких використовуються абсолютно неправильно. Для початку ми знайдемо заміну MySQL, чи дослідимо у парі з чим йому буде краще працювати, так як сама по собі MySQL якщо її правильно використовувати – дуже швидка. [19]

## 2.2 Дослідження існуючих методів зберігання великих даних

У першу чергу треба розглянути що взагалі таке великі дані. Різні джерела та книги тлумачать це поняття по різному і це не випадково. Щоб це зрозуміти нам потрібно спам’ятати таку річ, як закон Мура. [20]

Ще давно Мур вивів гіпотезу, за якою ми маємо наступне:

- Тактова частота процессорів буде зростати у два рази кожні півтора року
- Теоретично можлива кількість транзисторів на один кристал буде збільшуватись двічі щороку
- Продуктивність пристроїх буде збільшуватися у два рази кожні вісімнадцять місяців
- Ціна вироблення чипів буде зменшуватися у два раз кожні півтора року

Звісно з часом ця гіпотеза десь давала розуміння, що не все так просто та вона має плавучий характер. Але у основному такі поштовхування у сфері процессорів збільшують кількість комп’ютерів, а з ними кількість оброблюваних даних, а з тим і кількість необхідного простору для зберігання даних. Тож якщо раніше великими даними могли вважаться гігабайти даних, потім терабайти, а наразі й петабайти даних. Але останне стосується зазвичай тільки великих компаній, як Google чи Microsoft. [21] Тож у нашому випадку великими даними є терабайти даних. Тепер, зрозумівши що таке великі дані переходимо до методів їх зберігання.

Спочатку ми повинні розвантажити MySQL та додати до роботи бази даних та софт, який зроблено спеціально для зберігання великих даних. Серед такого софту можна використовувати Elasticsearch та Kafka, а для зберігання даних PostgreSQL, ClickHouse чи MariaDB. [22]

Серед усього вищезазначенного накращим вибором було б вивчення Elasticsearch та зберігання даних у PostgreSQL. Elasticsearch пропонує нам

потужні інструменти для пошуку великих масивів даних, а PostgreSQL широкий вибір типів даних для зберігання, потужній двигун та великі можливості для конфігурування.

Все це було б дуже гарною можливістю, але на вивчення такого методу наразі не має багато часу. MariaDB майже по тієй самій причині відпадає, так як для неї треба ElasticSearch для кращої роботи. Тож цей варіант не є нашим.

Можна згадати за MongoDB та NewSQL бази даних, але так як наш проект є довготривалим, то звертатися до новітніх СУБД буде поганою практикою, адже по ним майже немає ніякої документації. А ось MongoDB можна використати через свою гнучкість на сервері, тільки не як заміну MySQL, адже таким чином ми все ще не зможемо агрегувати дані, що для нас є найважливішим, коли мова заходить за великі дані. [24]

Тож ми зупиняємося на варіанті з ClickHouse, якому буде допомагати Kafka. Також, ми залишаємо MySQL лише як базу для зберігання конфігурації користувачів.

Clickhouse ми обираємо через те, що він дає нам змогу агрегувати дані за якимось схожими полями, таким чином зберігаючи менше інформації ми зберігаємо її більш детально, не використовуючи багато простору. [25] Робить він це за допомогою своїх Engine, до яких є добра документація. Також, таким чином ми знімаємо необхідність їх агрегувати фронтенду, зменшуємо навантаження на мережу та можемо використовувати простоючі потужності для інших цілей.

Kafka ми обираємо через легку інтеграцію з ClickHouse та можливість швидко відправляти дані пачками з серверів, а не по одному, й підключати легко нові сервери. [26] Важливо пам’ятати що Kafka є лише брокером повідомлень, та було б доречно підключити ще й ElasticSearch для пошуку, але у нас все ще не так багато можливостей та часу.

Окрім цього ми додамо словники до MySQL та ClickHouse як найкращу можливість для зберігання однакових даних та дещо змінимо запити до БД. [28]

## 2.3 Дослідження існуючих методів обробки даних з серверів

Тепер, коли ми розібралися з базами даних та їх заміною - можна перейти до змін на самих серверах та методів діставання даних з них. Для цього ми можемо замінити мову программування, додати ще скрипти, які будуть виконуватися за допомогою Crontab, не зберігати дані на сервері, а зберігати їх у вигляді JWT токену, який будемо відправляти у відповіді на партнерів, додати брокери повідомлень чи якісь динамічні бази даних. [29]

Замінювати мову программування немає можливості, так як вивчати нову мову програмування може зайняти велику кількість часу, а пошук нових програмістів – ще й кошти. До того ж більш типізовані та швидкі мови програмування не дають змогу швидко змінювати код та випускати його у роботу, так як зазвичай ще треба проходити этап компіляції. Тож треба продовжувати працювати з NodeJS та боротися з його недоліками. [30]

Першою проблемою є зберігання відповідей у оперативній пам’яті за допомогою процессу NodeJS та важкість внесення нових змін до коду через неможливість без проблем зупинити роботу процессів.

Найкращим та швидким варіантом буде зберігання даних у сервісі кешування, який не буде перезавантажуватися разом з процесами NodeJS. Звісно, можна було б використовувати якісь БД, але через це ми би втрачали швидкість. Тут, наразі, серед вибору є Redis, Aerospike та Memcached.

Redis та Aerospike – це дуже потужні інструменти, які можна застосовувати для таких цілей. Але іноді краще – ворог доброго. Так, вони дають нам змогу створювати будь яку кількість баз, методи зберігання інформації, інкрементування значень, та все ж нам достатньо лише зберігання тимчасово відповіді. [31]

Тож ми можемо зупинити свій вибір на Memcached, через свою легкість у використанні та швидкість. У нього немає дуже великої кількості функціоналу, а зберігає він усього лише пару ключ-значення. Ключ треба повертати у відповіді, з якої, при звертанні, ми отримаємо необхідні нам дані з Memcached.

Так, ми б могли зберігати дані у вигляді JWT токену та відправляти його до партнера, але у такому випадку посилання на показ було б дуже великим, так як даних багато. До того ж нам все одно треба десь зберігати ключі до нього, але ми пам’ятаємо що у такому разі ми не зможемо легко вносити зміни до коду через неможливість перезавантажувати процес NodeJS.

Тож ми приходимо до висновку, що Memcached нам точно потрібен. Дані з нього будут видалятися тількі при зупинці його сервісу чи перевключенні серверу. До того ж тут є змога використовувати time to live (TTL), який буде автоматично видаляти відповіді, якщо не прийшло показу за 20 хвилин.

Щоб зберігати дані для подальшого їх використання і не відправляти їх на кожному показі до бази даних ми будемо використовувати MongoDB.

Такий вибір було зроблено через велике комьюніті цієї БД, її гнучкість та легкість і безпеку конфігурування. [32] Таким чином ми маємо змогу вносити зміни до коду легко, перезавантажувати процеси NodeJS, а дані все одно будуть зберігатися. До того ж у MongoDB дані будуть зберігатися навіть після перезавантаження серверу, що є ще однією перевагою у безпеці даних.

Залишилося лише обрати як ми будемо збирати дані з MongoDB, адже сам він не може відправляти дані на інші сервери. Для цього ми можемо використати майже будь який інструмент, який буде швидко та легко збирати дані з усіх серверів.

Нам буде достатньо для цього написати будь який скрипт на сервері з БД, які будуть заходити на всі сервери, збирати дані з MongoDB та складати їх у базу даних. Наразі я це зроблю на PHP, так як для написання скриптів це дуже легка мова. Викликати скрипт можна за допомогою Crontab.

Crontab – це таблиця у Unix-подібних системах яка дозволяє за допомогою спеціальних записів викликати необхідні команди. [33] Таким чином, записавши у Crontab нашу команду для виклику PHP скрипту кожні 10 хвилин ми будемо збирати дані з усіх серверів.

## 2.4 Створення комплексного методу зберігання великих даних

Після дослідження існуючих методів було виведено наступний комплексний метод для зберігання великих даних, використовуючи існуючі ресурси, з заміною деяких компонентів та программного забезпечення.

Розгортаємо Clickhouse на окремому від MySQL сервері та створюємо таблиці, розгортаємо на окремому сервері Kafka, конфігуруємо їх. Підключуємо до кожного з серверів на Node.js MongoDB, Memcached, Kafka. Пишемо код для зв'язку між усіма компонентами. Розгортаємо на сервері з MySQL скрипт для витягування даних з серверів з MongoDB. Створюємо образи таблиць у MongoDB, конфігуруємо її для безпеки. Замінюємо жорсткі диски на твердотільні на сервері з ClickHouse.

Після усього цього буде працювати наступний алгоритм. Ми запускаємо у роботу усі сервіси та сам сервер на NodeJS. Коли приходить запит на рекламу, то ми відправляємо його до партнерів, які нам можуть відповісти рекламою а можуть не відповісти. У цей час ми зберігаємо приклад та детальну інформацію про запит до Kafka broker, який підключається до інстансу Nodejs та який зберігає дані у собі на протязі 5 хвилин, після чого відправляє всі дані на сервер з Kafka. Clickhouse маючи створені таблиці у якості Consumer очікує на дані на сервері Kafka, та збирає їх як тільки побачить, сам агрегує та вставляє у основну таблицю.

Коли запит повертається від партнерів, у разі позитивної відповіді, ми оброблюємо її, зберігаємо у Memcached, зберігаємо приклад до Kafka Broker та відправляємо назад до партнеру, який зробив запит на рекламу, з деякою ціною.

Якщо наша відповідь перемогла у аукціоні та трапився показ, нам приходить запит на рекламу, яку треба показати. У цей час ми повертаємо рекламу, а самі дістаємо показ по хешу, який відправили у відповіді до партнера з Memcached, зберігаємо його основні частини у MongoDB та детально у Kafka Broker, у іншу чергу, не в ту саму що й пишуться приклади. Якщо показу не було на протязі 20 хвилин, то Memcached сама видаляє запис.

Кожні 10 хвилин на сервері з скриптом на Node.js спрацьовує Crontab, витягує дані з серверів, робить агрегацію та зберігає їх у MySQL. Окрім цього на цьому сервері з MySQL ще й розвернуто API, до якого сервери звертаються кожні 20 хвилин, а API повертає необхідні конфігурації для партнерів, які зберігаються у MySQL.

Схематично все це зображено на рисунку 2.1.

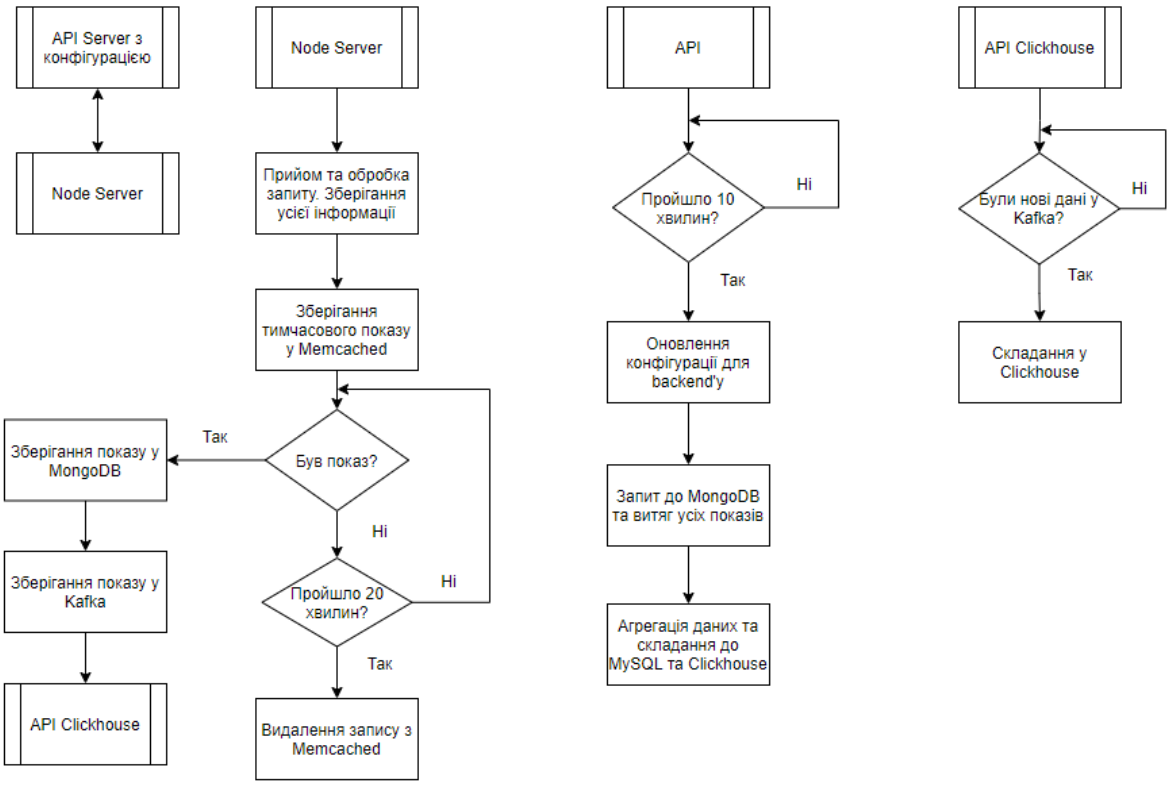



Рисунок 2.1 - Загальний алгоритм роботи системи

## 2.5 Висновки

Після вивчення усієї інформації було вибрано наступні рішення для реалізації комплексного методу зберігання великих даних в умовах обмежених ресурсів:

- Використовувати ClickHouse у деяких випадках замість MySQL, так як у нього є змога використовувати свої Engine для агрегації та пришвидшення пошуку по базі, зменшення зберігаємих об’ємів інформації
- Зробити запис до бази проміжками, а ні щоразу, як з’являється новий показ, використовуючі написані скрипти та Crontab
- Використовувати сервіс Kafka, як брокер повідомлень, через велику кількість серверів
- Замінити жорсткі диски на твердотільні
- Оптимізувати роботу MySQL разом із оптимізацією викликів
- Використання Memcached як сервіс кешування.
- Використання MongoDB як гнучкої бази даних для зберігання показів перед тим, як вони запишуться до бази.

# 3. СТВОРЕННЯ КОМПЛЕКСНОГО МЕТОДУ ЗБЕРІГАННЯ І ОБРОБКИ ДАНИХ

## 3.1. Вимоги до роботи удосконаленної системи та вибір її компонентів

Вищезазначений план було дещо змінено під час реалізації, тож для початку ми розглянемо те, що ми будемо використовувати та що замінювати:

- Треба прибрати з MySQL запис детальної інформації по показам та перенесення її до Clickhouse.
- Для роботи Clickhouse буде використовуватися брокер повідомлень Kafka, який буде об’єднувати роботу серверу на Node.JS та Clickhouse.
- До MySQL будуть робитися запити тільки за загальною інформацією, яку треба часто оновлювати. Для цього підключимо MongoDB та Memcached.
- Опит серверів замість звертання щоразу до БД.
- Підключення словників та оптимізація звертань до БД.

### 3.1.1. Вибір заміни MySQL та конфігурування

Для початку, я зроблю заміну запису майже всієї інформації з MySQL до Clickhouse. Такою інформацією будуть усі таблиці, у яких є спільні дані у колонках, по яким можна агрегувати дані засобами Clickhouse та не використовувати додатково ресурси серверу на агрегацію поза ним та доставленням до нього. Щоб пояснити це рішення – перейдемо до технічного опису та порівняння.

Тож Clickhouse – це SQL-подібна база даних, тобто реляційна. Її створила та підтримує компанія Yandex. Вона є дуже легкою, потужною, а виклики до неї пишутся майже на тих самих викликах, що й у звичайній MySQL базі даних. Ще до найважливіших переваг можна віднести те, що вона є колонковою базою даних, на відміну від MySQL, яка є строковою, має Merge Engine та легко інтегрується з Kafka.

Колонкові бази є кращим рішенням, коли з'являється необхідність діставати постійно якусь колонку, а ні увесь рядок цілком. Це подрібно, коли нам потрібні дані для статистики тільки з деяких колонок, адже зазвичай нам для статистики не потрібні усі поля, а тільки ім'я чи країна. Як виглядають колонкові та строкові таблиці у записах можна побачити на рисунках 3.1 та 3.2 відповідно.

| Row: | #0 | #1 | #2 | #N |
|---|---|---|---|---|
| WatchID: | 89354350662 | 90329509958 | 89953706054 | ... |
| JavaEnable: | 1 | 0 | 1 | ... |
| Title: | Investor Relations | Contact us | Mission | ... |
| GoodEvent: | 1 | 1 | 1 | ... |
| EventTime: | 2016-05-18 05:19:20 | 2016-05-18 08:10:20 | 2016-05-18 07:38:00 | ... |

Рисунок 3.1 – Запис даних у колонковій реляційній базі даних

| Row | WatchID | JavaEnable | Title | GoodEvent | EventTime |
|---|---|---|---|---|---|
| #0 | 89354350662 | 1 | Investor Relations | 1 | 2016-05-18 05:19:20 |
| #1 | 90329509958 | 0 | Contact us | 1 | 2016-05-18 08:10:20 |
| #2 | 89953706054 | 1 | Mission | 1 | 2016-05-18 07:38:00 |
| #N | ... | ... | ... | ... | ... |

Рисунок 3.2 – Запис даних у строковій реляційній базі даних

Також, ще одною перевагою є можливість сконфігурувати базу так, як заманеться. Можна самому робити обмеження на кількість колонок, розмір

строк, максимальний розмір таблиць, максимальний розмір запиту та відповіді до нього і інше. На рисунку 3.3 показано приклад конфігураційного файлу до ClickHouse.

```
<?xml version="1.0"?>
<!--
  NOTE: User and query level settings are set up in "users.xml" file.
-->
<yandex>
    <logger>
        <!-- Possible levels: https://github.com/pocoproject/poco/blob/poco-1.9.4-release/Foundation/include/Poco/
Logger.h#L105 -->
        <level>trace</level>
        <log>/var/log/clickhouse-server/clickhouse-server.log</log>
        <errorlog>/var/log/clickhouse-server/clickhouse-server.err.log</errorlog>
        <size>1000M</size>
        <count>10</count>
        <!-- <console>1</console> --> <!-- Default behavior is autodetection (log to console if not daemon mode an
d is tty) -->
    </logger>
    <!--display_name>production</display_name--> <!-- It is the name that will be shown in the client -->
    <http_port>8123</http_port>
    <tcp_port>9000</tcp_port>
    <mysql_port>9004</mysql_port>
    <!-- For HTTPS and SSL over native protocol. -->
    <!--
```

Рисунок 3.3 – Приклад конфігураційного файлу Clickhouse

Що до Merge Engine – це дуже потужний інструмент, який використовує Clickhouse для агрегації будь якої інформації за ключами, чи записом тільки необхідних даних серед усіх, що приходять. Так, це теж можна конфігурувати за його допомогою.

Так SummingMergeTree Engine дозволює нам створити таблицю, яка буде агрегувати дані за деякими полями, складаючи ключ. І при тому, після складання, буде аггрегувати та підсумовувати дані у інших колонках. Приклад коду для створення такої бази на рисунку 3.4. На цьому прикладі всі поля крім totalRequest та updatedAt є ключем, за яким інкременується значення totalRequest, а updatedAt є датою, яка є основним ключем агрегації.

```
create table main.requestStatsTotal
(
    ssp           Int64,
    pub           String,
    source       String,
    country      String,
    totalRequest Int64,
    updatedAt    Date
)
    engine = SummingMergeTree(updatedAt, (updatedAt, ssp, pub, source, country), 8192);
```

Рисунок 3.4 – Приклад створення таблиці з використанням SummingMergeTree

Далі ці таблиці, за допомогою Merge Engine можна скласти в одну, для легшого доступу до них. Це необхідно, так як створюватися таблиці будуть щомісяця, а окрім цього, щоб не робити виклики до декількох таблиць та шукати необхідні дані, ми зробимо таблицю, яка буде збирати всі дані за ключами у цих таблицях та зберігати тільки посилання на них, тож не буде займати місце. Приклад створення такої таблиці на рисунку 3.5

```
create table sspRequestStats
(
    ssp               String,
    dsp               String,
    source                String,
    pub           String,
    country               String,
    width                 UInt32,
    height                UInt32,
    device                String,
    total          Int32,
    requestSent      Int32,
    bid            Int32,
    bidSent          Int32,
    impression            Int32,
    spend              Float32
)
    engine = Merge(main, 'sspRequestStats_');
```

Рисунок 3.5 – Приклад створення таблиці з використанням Merge Engine

Ще одним важливим Engine э ReplacingMergeTree. За допомогою нього ми можемо  записувати до таблиці тільки особливі дані. Тобто ті дані, у яких хоча б одне поле відрізняється від інших. Також ми можемо задати час, який будуть існувати дані у таблиці. Таким чином ми легко перенесемо запис прикладів та зменшимо обсяг займаємих даних у рази. Приклад створення такої таблиці на рисунку 3.6.

```
create table main.exampleLogs
(
    ts           DateTime,
    request  String,
    bid_request  String,
    response String,
    ev     Int16,
    dsp       Int16,
    source       String,
    user      UInt32,
    placement UInt32
)
    engine = ReplacingMergeTree(ts)
        PARTITION BY toYYYYMMDD(ts)
        PRIMARY KEY (ev, dsp, source, user, placement)
        ORDER BY (ev, dsp, source, user, placement)
        TTL ts + toIntervalDay(1)
        SETTINGS index_granularity = 8192;
```

Рисунок 3.6 – Приклад створення таблиці з використанням ReplacingMergeTree

Окрім усього вищезазначеного можна ще зазначити легку інтеграцію з Kafka, за допомогою Kafka Engine у ClickHouse. Він дає змогу не просто зчитувати дані з серверу, а ще й використовувати будь які Engine з вищезазначених, а тому у нас відпадає необхідність у створенні будь-яких інстансів, а просто відправляти дані одразу до ClickHouse. Приклад реалізації можна побачити на рисунку 3.7, як його використовуэ CloudFlare разом із Clickhouse. Kafka є брокером повідомлень, тож про це у наступному розділі.

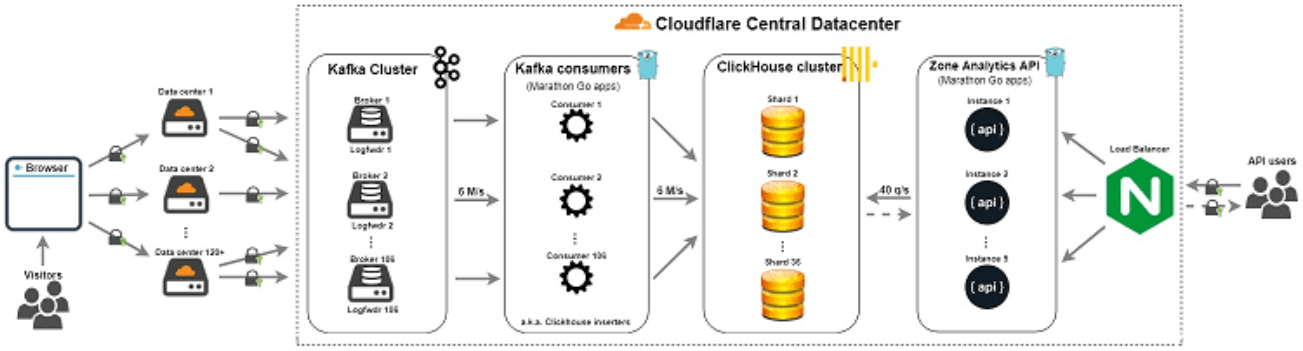



Рисунок 3.7 – Основні компоненти Apache Kafka

### 3.1.2. Використання брокеру повідомлень та інтеграція з Clickhouse

Наступним єтапом буде заміна усіх інстансів NodeJS та PHP які використовувалися раніше для витягнення даних з серверів та їх агрегації на брокер повідомлень. Тут є багато рішень, таких як RabbitMQ чи ActiveMQ, але ми будемо використовувати Kafka.

Особливістю брокерів повідомлень є їх пристосованість до передачі даних з однієї середи до іншої. Такими середами можуть бути мови программування, бази даних та інше.

Ми обираємо ж брокер повідомлень Kafka, так як він легко інтегрується до ClickHouse.

Apache Kafka розробили Apache Software Foundation и LinkedIn, написаний на Java та Scala. Остання є найшвидшою мовою программування на сьогодняшній день. Kafka спроектована як розподілена, горизонтально масштабована система, яка нам забезпечує збільшення пропускної здатності при підвищенні навантаження та за збільшенням систем-підписників. Вона підтримує також тимчасове зберігання даних для подальшої обробки, захист від неправильних чи пошкоджених даних та стабільність при використанні.

У основі роботи Kafka закладені Producer, Server та Consumer (рисунок 3.8).

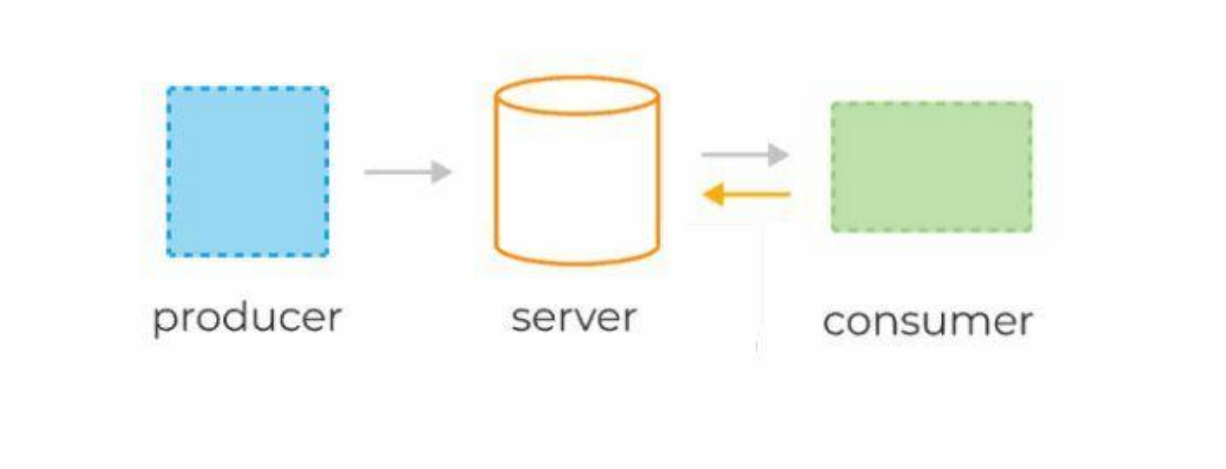



Рисунок 3.8 – Основні компоненти Apache Kafka

Producer – це деякий сервіс, який генерує дані та зберігає їх у сервері, на якому знаходиться Kafka. Такими даними будуть запити, їх приклади та покази, які далі будуть складатися у Consumer. У нашій реалызації це сервери з викликами, написані на Node.JS.

Consumer – це сервіс, який забирає дані з серверу з Kafka та складає їх у себе. У нашій реалізації це буде Clickhouse.

Clickhouse дає змогу нам це зробити за допомогою Kafka Engine. На рисунку 3.9 показано приклад як це зробити.

```
create table main.sspRequests
(
    ssp             Int32,
    dsp             Int32,
    pub       String,
    source            String,
    country           String,
    total     Int64,
    request Int64,
    bid       Int64,
    bidSent     Int64,
    spend         Float64,
    updatedAt         Date
)
    engine = Kafka('192.254.51.32:9092', 'requests_table', 'main', 'TSV')
SETTINGS kafka_num_consumers = 1, kafka_row_delimiter = '\n',
kafka_skip_broken_messages = 10000;
```

Рисунок 3.9 – Приклад створення проміжної таблиці на Kafka Engine у Clickhouse

Ми можемо побачити, що окрім звичайних полей є ще й конфігурація. У ній вказано ip-адресу серверу з Kafka, кількість консьюмерів, з якої таблиці брати дані, у якому форматі вони там складені (‘TSW’), як розділені рядки та через скільки неправильних даних Engine ігнорує дані.

Формат TSW показує, що дані складені через табуляцію. Можна вказувати майже будь яким варіантом, але практика показала, що так найкраще. Також, на етапі тестування можна виявити, якщо дані складуються неправильно та швидко це вирішити.

Окрім проміжної таблиці ми ще маємо створити Materialize View, яка буде використовуватися у ролі Consumer’а на сервері з ClickHouse, тобто брати дані з проміжної та основну таблицю, куди будуть агрегуватися і складатися дані. Основна таблиця створюється як і звичайна таблиця з будь яким Merge Engine. У нашому випадку це SummingMergeTree (рисунок 3.10).

```
create table main.sspRequestsMain
(
    ssp            Int32,
    dsp            Int32,
    pub      String,
    source           String,
    country          String,
    total     Int64,
    request Int64,
    bid       Int64,
    bidSent     Int64,
    spend         Float64,
    updatedAt        Date
)
    engine = SummingMergeTree(updatedAt, (updatedAt, ssp, dsp, pub, source, country), 8192);
```

Рисунок 3.10 – Приклад створення основної таблиці для Kafka у Clickhouse

Далі залишається лише створити Materialize View. Materialize View – це таблиця, яка реагує на будь яку зміну даних у таблиці, до якої вона відноситься і перекладає до основної. Така таблиця зберігає дані, а не тільки робить посилання на них, тож їх краще використовувати як альтернативу звичайним таблицям, якщо потрібно оперативно реагувати на нові дані. Але для нашої задачі це необхідно. До того ж після того, як наша Materialize View дістає дані, то вона їх одразу вставляє до основної таблиці та видаляє у себе. Приклад такої таблиці на рисунку 3.11.

```
CREATE MATERIALIZED VIEW main.sspRequestsView TO main.sspRequestsMain  AS
SELECT
updatedAt,
       ssp,
       dsp,
       pub,
       source,
       country,
       sum(total)
AS totalRequest,
       sum(request) AS request,
       sum(bid)       AS bid,
       sum(bidSent)     AS bidSent,
       sum(spend)         AS spend
FROM main.sspRequests
GROUP BY ssp, dsp, pub, source, country, updatedAt;
```

Ще однією перевагою Kafka є можливість підключати скільки завгодно продюсерів та консюмерів. Окрім цього можна робити репліки даних на інші сервери з Kafka, утворюючи Kafka Cluster з декількох серверів, для впевненості, що вони не загубляться.

Приклад такого кластеру можна побачити на рисунку 3.12.

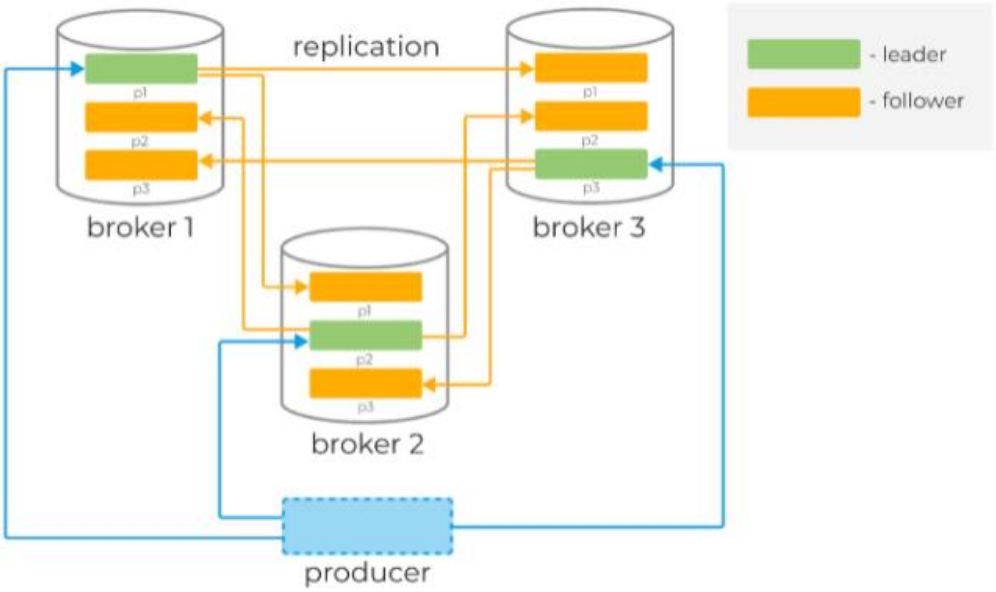



Рисунок 3.12 – Взаємодія Producer та Kafka Cluster

## 3.2. Кешування даних

Тепер можна перейти до реалізації кешування. Це є один з найважливіших етапів, так як попередня система дуже повільна та сильно перегружує всю систему та особливо БД.

Я запропонував підключити до Node серверу, на якому приймаються виклики, MongoDB та Memcached, так як вони дуже чудово будуть працювати у парі та зберігати необхідні нам дані.

MongoDB – э об’єктно-оріентованою базою даних, а це значить, нереляційною. Значною перевагою такої бази є можливість дуже швидко її сконфігувати, змінювати якщо потрібно навіть на протязі роботи, та працює вона відносно швидко. Звісно, не так швидко, як працює MySQL, але було б дуже проблематично розвертати на кожному сервері MySQL, а потім ще й змінювати у кожному поля, якщо це потрібно. А з ходом часу це майже завжди буде потрібно.

Приклад об’єкту, який ми можемо покласти до MongoDB, показано на рисунку 3.13. Таким чином можна побачити, що ми можемо складати не тільки дані на одному рівні вкладення, а ще й на декількох глибше.

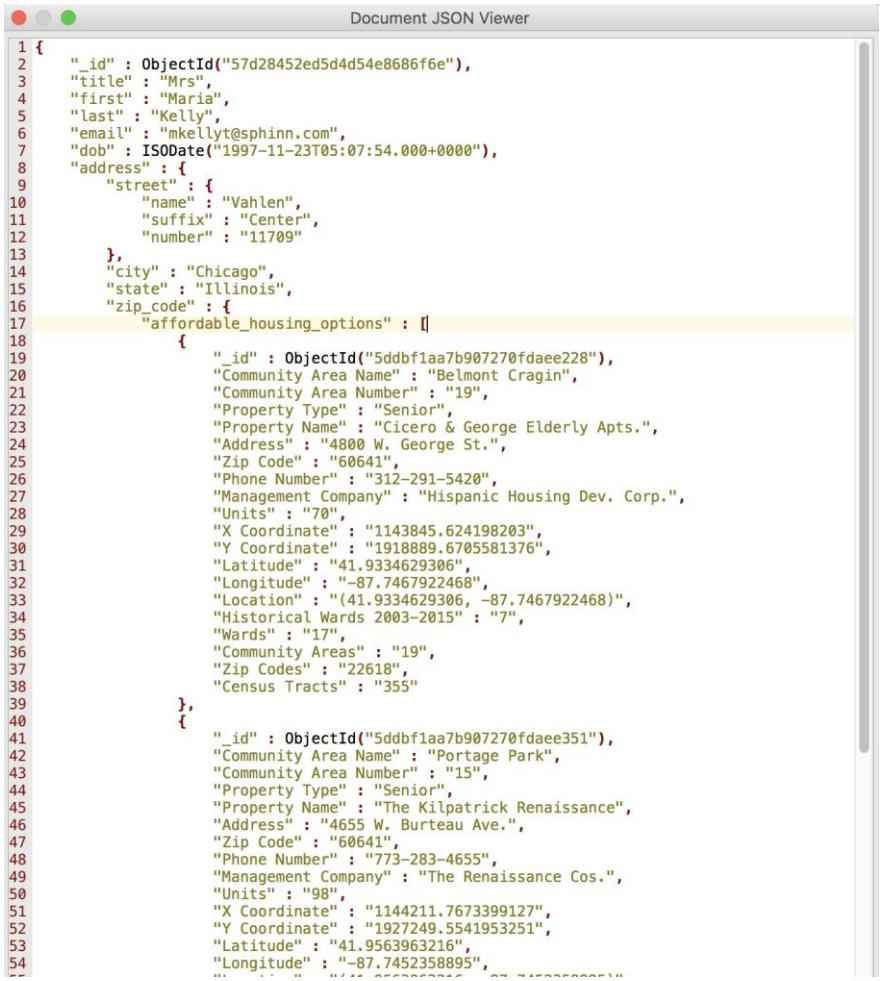



Рисунок 3.13 – Приклад ймовірного об’єкту у MongoDB

Тепер у нас залишається ще одна проблема – де зберігати покази, які ще не сталися, але потрібно до них мати доступ. Раніше з цим справлялася MySQL база даних, але можна поступити елегантніше та підключити сервіс кешування. Таким є Memcached.

Memcached – сервіс кешування, який зберігає інформацію у форматі ключ-значення. Він зберігає дані у оперативній пам’яті, якої у нас вдосталь, а тому працює неймовірно швидко. Єдиною проблемою залишається тільки те, що після виключення сервісу Memcached на сервері чи самого серверу всі збережені дані видаляться. Але так як сервери знаходяться у датацентрі – це не є великою проблемою, так як хоч и обладнання може вийти з ладу, за цим все одно дивляться и попереджують, а якщо зникне світло, то буде час зберегти дані, поки сервери працюють від блоків бесперебойного живлення.

Також можна вивести у плюси легку реалізацію майже з будь якою мовою програмування, а у нашому випадку з Node.js.

У нашій реалізації Memcached буде зберігати покази на протязі 20 хвилин, і видаляти записи, якщо за цей час не було показу. Якщо показ трапився, то Node.js збереже його у MongoDB та видалить запис. Таким чином, ми завжди будемо тримати зайняту оперативну пам’ять приблизно на одному рівні, та не зайдемо за її ліміти.

Схематичний приклад такої реалізації зображено на рисунку 3.14, на якому видно взаємозв’язок нашого сервера, який виступає у ролі додатку, сервісу Memcached та постійної бази данних.

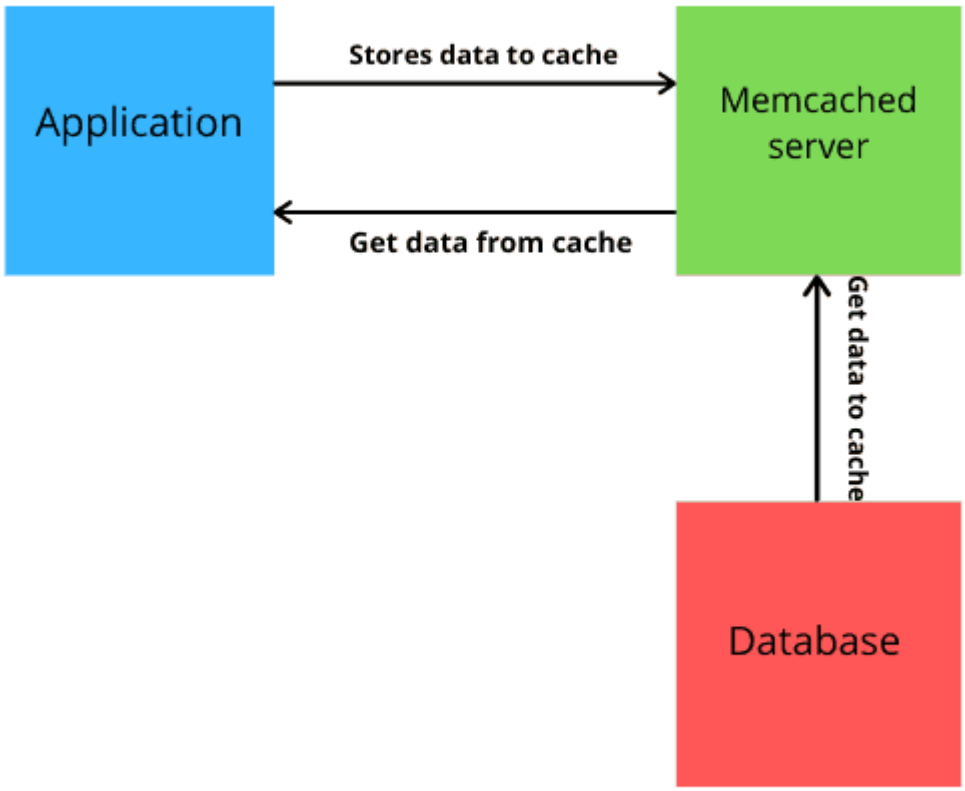



Рисунок 3.14 – Реалізація роботи сервісу кешування Memcached з MongoDB

Наступне, що нам потрібно зробити, це розвернути один інстанс Node на сервері з MySQL. Раз у 10 хвилин за допомогою Crontab буде виконуватися скрипт, який буде збирати інформацію до MySQL з усіх серверів та одразу її агрегувати. Таким чином ми будемо складати до MySQL лише малу та необхідну частину, яка необхідна frontEnd розробникам, а усі основні дані буде продовжувати збирати Kafka до ClickHouse. Після цього ми переходимо до підключення SSD на сервері з Clickhouse, замість HDD.

### 3.3. Вибір SSD та підключення

Тепер нам залишається лише обрати SSD замість HDD. Так як до цього ми використовували рейд з жорстких дисків на 16 ТВ, наразі там буде достатньо рейду з двох чи одного твердотільного диску, загальним об’ємом на 2 ТВ. Такий об’єм не вийде значно дорожче, так як ми просто арендуємо те, що є у датацентрі, а швидкість нам тепер необхідна більше за загальний об’єм дискових массивів.

Звісно можна обійтися й 1 ТВ, але у цілях реалізації майбутнього масштабування будемо використовувати 2 ТВ.

Нам запропонували на вибір Samsung PM983 Enterprise 960GB PCIe Gen3 x4 чи Kingston DC500M 960 GB. Після цього було зроблено порівняння їх між собою на таблицях 3.1 та 3.2.

Таблиця 3.1 - Технічні характеристики твердотільного диску Kingston DC500M 960 GB

| Параметр | Характеристики |
|---|---|
| Виробник | Kingston |
| Тип | SSD накопичувач |
| Лінійка | DC500M |
| Обсяг, ГБ | 960 |
| Інтерфейс | SATA rev. 3.0 |
| Форм-фактор | 2,5 |
| Тип флеш-пам'яті NAND | 3D TLC |
| Максимальна швидкість читання, МБ/с | 555 |

| | |
|---|---|
| Максимальна швидкість запису, МБ/с | 520 |
| Середній час безвідмовної роботи (MTBF), млн. годин | 2 |

Таблиця 3.2 - Технічні характеристики твердотільного диску Samsung PM983 Enterprise 960GB PCIe Gen3 x4

| Параметр | Характеристики |
|---|---|
| Виробник | Samsung |
| Тип | SSD накопичувач |
| Лінійка | PM983 |
| Обсяг, ГБ | 960 |
| Інтерфейс | U.2 |
| Форм-фактор | 2,5 |
| Тип флеш-пам'яті NAND | 3D TLC |
| Максимальна швидкість читання, МБ/с | 3200 |
| Максимальна швидкість запису, МБ/с | 1100 |
| Середній час безвідмовної роботи (MTBF), млн. годин | 2 |

Так як обидва диска нам підходять, але від Samsun виявився значно швидше – обираємо рейд з двух таких дисків, загальним об’ємом 2 ТВ.

## 3.4. Реалізація комплексного методу зберігання та обробки даних

### 3.4.1. Розгортання та конфігурація ClickHouse

У першу чергу треба розгорнути Clickhouse на окремому сервері та сконфігувати його так, щоб був високий рівень захисту й легкий доступ до нього. Розгортати ми будемо на Ubuntu 18, тож інструкція наступна:

1) Підключаємо ключі, за якими знайдемо ClickHouse у репозиторії Ubuntu

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv E0C56BD4
```

2) Підключаємо репозиторій

```
echo "deb http://repo.yandex.ru/clickhouse/deb/stable/ main/" | sudo tee
/etc/apt/sources.list.d/clickhouse.list
```

3) Оновлюємо усі репозиторії

```
sudo apt-get update
```

4) Робимо інсталяцію

```
sudo apt-get install -y clickhouse-server clickhouse-client
```

5) Запускаємо сервіси та дивимося чи запрацювали вони

```
sudo service clickhouse-server start
sudo service clickhouse-server status
```

Ми маємо побачити приблизно те саме, що й на рисунку 3.15.

```
● clickhouse-server.service - ClickHouse Server (analytic DBMS for big data)
   Loaded: loaded (/etc/systemd/system/clickhouse-server.service; enabled; vendor preset: enable
   Active: active (running) since Sat 2018-12-22 07:23:20 UTC; 1h 9min ago
 Main PID: 27101 (ClickHouse-serv)
    Tasks: 34 (limit: 1152)
   CGroup: /system.slice/ClickHouse-server.service
           └─27101 /usr/bin/ClickHouse-server --config=/etc/ClickHouse-server/config.xml
```

Рисунок 3.15 – Результат правильної інсталляції ClickHouse

Тепер переходимо до його конфігурування. Файл конфігурації знаходиться у /etc/clickhouse-server/users.xml. Він виглядає як звичайний xml файл зі звичайним синтаксисом і зберігає дані про усіх користувачів.

Важливо, при створенні нового користувача хешувати його пароль, так як інакше він буде лежати тут у відкритому вигляді, та будь-яка людина, що зайде на сервер, сможе зайти під його профілем. У цьому файлі ми маємо створити профиль та додати користувачів з якимось привілегіями.

На рисунку 3.16 показано, як я створив користувача stats та профіль для нього.

```
<?xml version="1.0"?>
<yandex>
    <!-- Profiles of settings. -->
    <profiles>
        <stats>
            <max_insert_block_size>10485760</max_insert_block_size>
            <stream_flush_interval_ms>60000</stream_flush_interval_ms>
            <distributed_aggregation_memory_efficient>1</distributed_aggregation_memory_efficient
            <max_memory_usage>25000000000</max_memory_usage>
            <max_bytes_before_external_group_by>15000000000</max_bytes_before_external_group_by>

        </stats>
    </profiles>

    <!-- Users and ACL. -->
    <users>
        <stats>
            <password_sha256_hex>password_sha256_hex</password_sha256_hex>
            <networks>
            <ip>::/0</ip>
            </networks>
            <profile>stats</profile>
            <quota>default</quota>
        </stats>
    </users>
    <!-- Quotas. -->
    <quotas>
        <!-- Name of quota. -->
        <default>
            <!-- Limits for time interval. You could specify many intervals with different limits
            <interval>
                <!-- Length of interval. -->
                <duration>3600</duration>

                <!-- No limits. Just calculate resource usage for time interval. -->
                <queries>0</queries>
                <errors>0</errors>
                <result_rows>0</result_rows>
                <read_rows>0</read_rows>
                <execution_time>0</execution_time>
            </interval>
        </default>
    </quotas>
</yandex>
```

Рисунок 3.16 – Конфігураційний файл для Clickhouse

Після цього треба зробити наступні дві команди та перевірити, щоб після останньої ми бачили те саме, що й на рисунку 3.15.

```
sudo service clickhouse-server restart
sudo service clickhouse-server status
```

Тепер треба створити усі необхідні нам таблиці. Для цього треба потрапити до консолі, за допомогою clickhouse-client, імені та паролю що ми щойно створили та виконати наступні команди щоб створити необхідні таблиці:

```
CREATE TABLE name_consumer (
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…
) Engine = MergeTree
PARTITION BY toYYYYMM(time)
ORDER BY (readings_id, time);
```

```
CREATE TABLE name_main (
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…
)
ENGINE = Kafka
SETTINGS kafka_broker_list = 'serverip:9092',
       kafka_topic_list = 'name',
       kafka_group_name = 'readings_consumer_group1',
       kafka_format = 'TSV',
       kafka_max_block_size = 1048576;
       kafka_skip_broken_messages = 10000;
```

Тепер створюємо таблицю Materialize View. Важливо знати, що після її створення Clickhouse почне роботу у якості Consumer та буде діставати дані з вказаного серверу.

```
CREATE MATERIALIZED VIEW name_view TO name_main AS
SELECT
    ColumnName
    ColumnName
    ColumnName
    ColumnName
FROM name_consumer;
```

Для прикладів ми будемо так само створювати нові таблиці, тільки вже з ReplacingMergeTree, щоб старі приклади змінювалися. Приклад такої таблиці нижче.

```
Create Table examples_name
(
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…
)
    Engine = ReplacingMergeTree(ts)
        PARTITION BY toYYYYMMDD(ts)
        PRIMARY KEY (ColumnName, ColumnName, ColumnName, ColumnName)
        ORDER BY (ColumnName, ColumnName, ColumnName, ColumnName)
        TTL ts + toIntervalDay(1)
        SETTINGS index_granularity = 8192;
```

Тож у загальній сумі у нас має бути наразі 6 таблиць. У них буде зберігатися інформація по запитам які приходять. Окрім цього ми додамо таблиці, у яких будуть зберігатися покази помісячно та створимо merge таблицю для них. Для цього зробимо невеликий скрипт, котрий буде діставати дані з MongoDB та складати їх. До того ж цей скрипт буде складати дані на сусідній сервер до MySQL. Напишемо його на PHP для швидкості та підключимо до Crontab.

```
<?php


define('HEALTHCHECK_KEY', 'key');
ini_set('display_errors', 1);
ini_set('display_startup_errors', 1);
error_reporting(E_ALL);
//error_reporting(0);

//Settings for script
ini_set('memory_limit', '30720M'); //memory 20480M
set_time_limit(0); //limit time
date_default_timezone_set('UTC'); //timezone
////////////////////////////////////////////////////////////////////////////////
////////////////////////////

////////////////////////////////////// Settings DB
////////////////////////////////////////////////////////////////
//MySQL
$MYSQL_SETTINGS = [
    'HOST' => 'HOST',
    'PORT' => PORT,
    'USER' => 'USER',
    'PASSWORD' => 'PASSWORD',
    'DATABASE' => 'DATABASE',
    'TABLE' => 'TABLE'
];

//ClickHouse
$CLICK_HOUSE_SETTINGS = [
    'HOST' => 'HOST',
    'PORT' => PORT,
    'USER' => 'USER',
    'PASSWORD' => 'PASSWORD',
    'RESPONSE_TIMEOUT_SEK' => RESPONSE_TIMEOUT_SEK,
    'DATABASE' => 'DATABASE',
    'TABLE' => 'TABLE',
    'TABLE_BIDPRICE' => 'TABLE_BIDPRICE'
];

$INHOUSE_CH_SETTINGS = [
    'HOST' => 'HOST',
    'PORT' => PORT,
    'USER' => 'USER',
    'PASSWORD' => 'PASSWORD',
    'RESPONSE_TIMEOUT_SEK' => RESPONSE_TIMEOUT_SEK,
    'DATABASE' => 'DATABASE',
    'TABLE' => 'TABLE'
];
////////////////////////////////////////////////////////////////////////////////
////////////////////////////
function error_DB($connect, $line = '')
{
    sendAlert(true, 'MySQL Error (' . $connect->errno . ') ' . $connect->error . ' File
Line: ' . $line . "\n", HEALTHCHECK_KEY);
    echo('MySQL Error (' . $connect->errno . ') ' . $connect->error . ' File Line: ' .
$line . "\n");
    return false;
}
```

```
$mysqli_dsp = new mysqli($MYSQL_SETTINGS_DSP['HOST'], $MYSQL_SETTINGS_DSP['USER'],
$MYSQL_SETTINGS_DSP['PASSWORD'], $MYSQL_SETTINGS_DSP['DATABASE']);
if ($mysqli_dsp->connect_errno) {
    sendAlert(true, "Failed connect to MySQL server: {$mysqli_dsp->connect_error}\n",
HEALTHCHECK_KEY);
    throw new Exception("Failed connect to MySQL server: {$mysqli_dsp-
>connect_error}\n");
}

$todayBidsSQL = "select campaignId, sum(bids) as bids from stats
where YMD = toDate(now())
group by pub
FORMAT JSON";

function getDataFromCH($CLICK_HOUSE_SETTINGS, $sql)
{

    $CLICK_HOUSE_URL = 'http://' . $CLICK_HOUSE_SETTINGS['HOST'] . ':' .
$CLICK_HOUSE_SETTINGS['PORT'] . '/?user=' . $CLICK_HOUSE_SETTINGS['USER'] .
'&password=' . $CLICK_HOUSE_SETTINGS['PASSWORD'];

    //curl POST on ClicHouse
    if ($curl = curl_init()) {
        curl_setopt($curl, CURLOPT_URL, $CLICK_HOUSE_URL);
        curl_setopt($curl, CURLOPT_TIMEOUT,
$CLICK_HOUSE_SETTINGS['RESPONSE_TIMEOUT_SEK']); //sek
        curl_setopt($curl, CURLOPT_RETURNTRANSFER, true);
        curl_setopt($curl, CURLOPT_POST, true);
        curl_setopt($curl, CURLOPT_POSTFIELDS, $sql);
        curl_setopt($curl, CURLOPT_HEADER, true);
        curl_setopt($curl, CURLOPT_HTTPHEADER, [
            //'Content-length: ' . $sql_size_len,
            'Content-type: text/plain; charset=utf-8',
            'Connection: Close'
        ]);
        $resultResponse = curl_exec($curl);

        //check response
        $pattern = '/HTTP\/1.1 200 OK/i';
        if (preg_match($pattern, $resultResponse) !== 1) {
            return ("Error response from {$CLICK_HOUSE_SETTINGS['HOST']}:
{$resultResponse} \n");
        }


        $header_size = curl_getinfo($curl, CURLINFO_HEADER_SIZE);
        $header = substr($resultResponse, 0, $header_size);
        $body = substr($resultResponse, $header_size);

        curl_close($curl);
    } else {
        var_dump("Error request curl_init() for clickHouse \n");
        return ("Error request curl_init() for clickHouse \n");
    }

    return json_decode($body)->data;
}
```

```
$todayBids = getDataFromCH($INHOUSE_CH_SETTINGS, $todayBidsSQL);
$sql = "insert into imps (pub, spend) values  ";

foreach ($todayspend as $spend ) {
    $sql.="( {$spend->pub}, {$todayspend->spend}), ";

}
$sql = mb_substr($sql, 0, -2);
$sql .= " ON DUPLICATE KEY UPDATE `todayspend` = VALUES(`spend`)";

$mysqli_dsp = new mysqli($MYSQL_SETTINGS_DSP['HOST'], $MYSQL_SETTINGS_DSP['USER'],
$MYSQL_SETTINGS_DSP['PASSWORD'], $MYSQL_SETTINGS_DSP['DATABASE']);
if ($mysqli_dsp->connect_errno) {
    sendAlert(true, "Failed connect to MySQL server: {$mysqli_dsp->connect_error}\n",
HEALTHCHECK_KEY);
    throw new Exception("Failed connect to MySQL server: {$mysqli_dsp-
>connect_error}\n");
}

$mysqli_dsp->query($sql) or error_DB($mysqli_dsp, __LINE__);
```

Також, треба не забути створити таблиці для цього. Для початку створимо таблиці для запису щомісяця. Далі їх буде створювати скрипт.

```
Create Table name_imps_2021_10_09
(
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…

)
    engine = SummingMergeTree(updatedAt,
             (ColumnName, ColumnName, ColumnName, ColumnName, ColumnName, ColumnName,
ColumnName, ColumnName, ColumnName, ColumnName,
              ColumnName), 8192);
```

Тепер створимо таблицю, у якій дані будуть збиратися у одну за посиланнями за допомогою Merge Engine.

```
Create Table name_main_imps
(
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…
    ColumnName Type…

)
    engine = Merge(main, 'name_imps_');
```

Тепер можна переходити до конфігурування Kafka.

### 3.4.2. Розгортання та конфігурація Kafka

Тепер можна перейти до конфігурації Kafka. Для цього заходимо на окремий сервер, який арендували спеціально для Kafka на Ubuntu 20. Для початку робимо підготовку наступними командами:

```
sudo apt update && sudo apt upgrade
sudo apt install default-jre wget git unzip -y
sudo apt install default-jdk -y
```

Наступним кроком ми робимо скачування Kafka, створюємо для нього папку та розпаковуємо.

```
cd ~
wget https://downloads.apache.org/kafka/2.6.0/kafka_2.13-2.6.0.tgz
sudo mkdir /usr/local/kafka-server && cd /usr/local/kafka-server
```

```
sudo tar -xvzf ~/kafka_2.13-2.6.0.tgz --strip 1
```

Створюємо сервіс, який буде необхідний для запуску Kafka. Цей сервіс називається Zookeeper.

```
sudo vim /etc/systemd/system/zookeeper.service

[Unit]
Description=Apache Zookeeper Server
Requires=network.target remote-fs.target
After=network.target remote-fs.target

[Service]
Type=simple
ExecStart=/usr/local/kafka-server/bin/zookeeper-server-start.sh /usr/local/kafka-
server/config/zookeeper.properties
ExecStop=/usr/local/kafka-server/bin/zookeeper-server-stop.sh
Restart=on-abnormal

[Install]
WantedBy=multi-user.target
```

Створюємо сам сервіс Kafka.

```
sudo vim /etc/systemd/system/kafka.service

[Unit]
Description=Apache Kafka Server
Documentation=http://kafka.apache.org/documentation.html
Requires=zookeeper.service
After=zookeeper.service

[Service]
Type=simple
Environment="JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64"
ExecStart=/usr/local/kafka-server/bin/kafka-server-start.sh /usr/local/kafka-
server/config/server.properties
ExecStop=/usr/local/kafka-server/bin/kafka-server-stop.sh
Restart=on-abnormal

[Install]
WantedBy=multi-user.target
```

Запускаємо щойно створені сервіси.

```
sudo systemctl daemon-reload
sudo systemctl enable --now zookeeper
sudo systemctl enable --now kafka
sudo systemctl status kafka zookeeper
```

Тепер залишається лише інсталювати Cluster Manager та сконфігурувати його.

```
cd ~
git clone https://github.com/yahoo/CMAK.git
vim ~/CMAK/conf/application.conf
cmak.zkhosts="localhost:2181
```

Після конфігурації треба виконати наступні команди.

```
cd ~/CMAK/
./sbt clean dist
cd ~/CMAK/target/universal
unzip cmak-3.0.0.5.zip
cd cmak-3.0.0.5
sudo ufw allow 9000
```

Тепер ми можемо конфігурувати Kafka Cluster у Web IDE (рисунок 3.17). Для цього лише треба зайти у будь-який браузер та вписати ip серверу разом із портом 9000

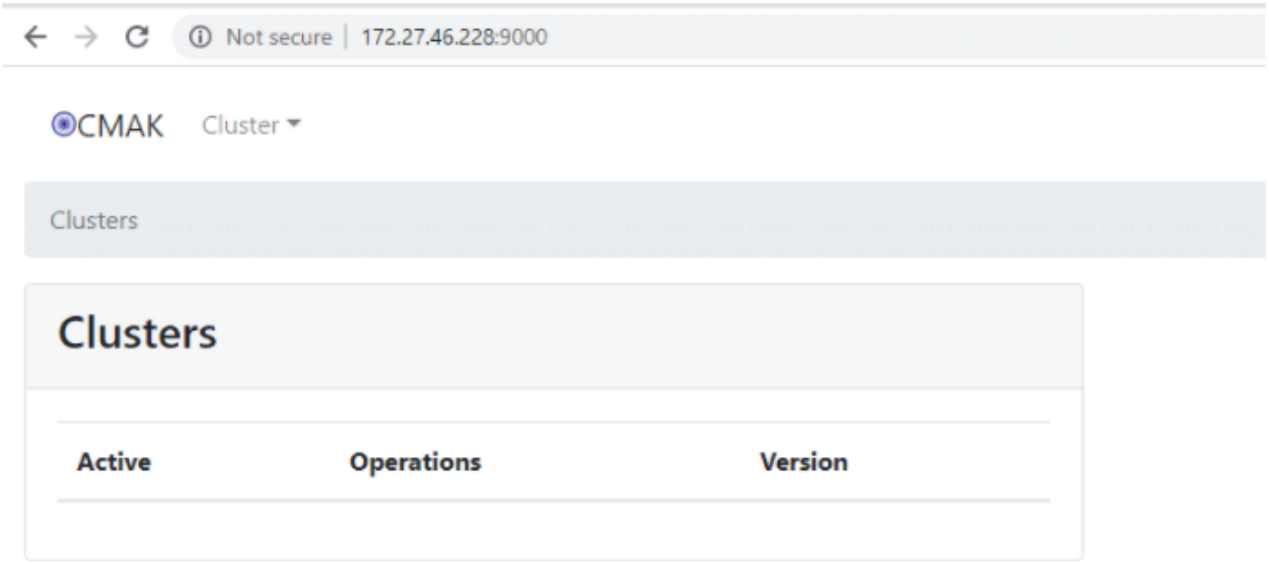



Рисунок 3.17 – Web IDE для конфігурації кластерів Kafka

Приклад створення кластеру можна побачити на рисунках 3.18 та 3.19.

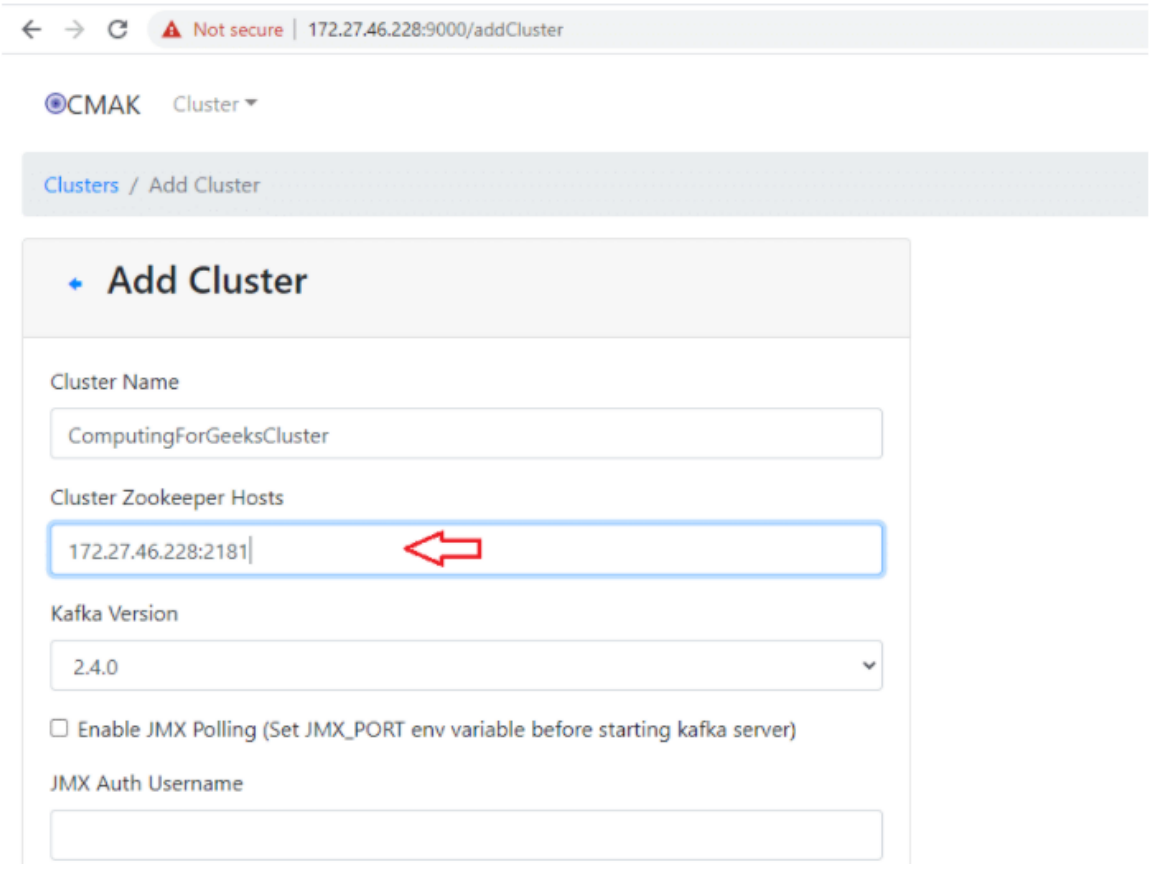



Рисунок 3.18 – Перша частина конфігурування кластеру

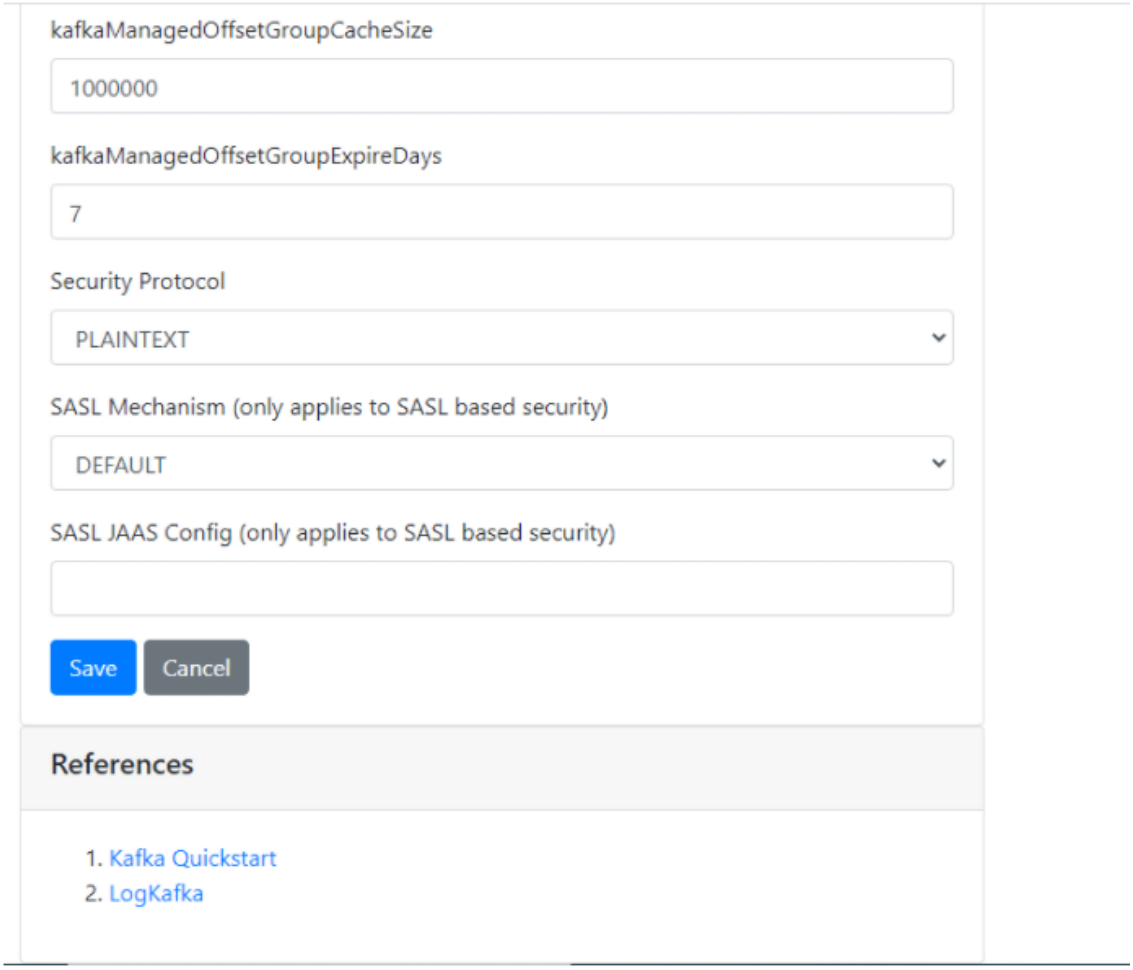



Рисунок 3.19 – Друга частина конфігурування кластеру

Окрім цього ми можемо створювати Topics, у які будуть складатися дані, але у цьому немає сенсу, так як вони можуть створюватися динамічно, як тільки ми почнемо писати щось у новий Topic. На цьому конфігурація Kafka закінчена. Далі залишаються лише сервіси MongoDB та Memcached, а також підключити диски.

### 3.4.3. Підключення MongoDB та Memcached

Для підключення MongoDB та Memcached ми робимо інсталяцію до npm, який використовує NodeJS як менеджер бібліотек. Так потрібно зробити на кожному сервері, але я це автоматизував через Ansible, який дає змогу робити одні й ті самі команди на багатьох серверах.

На кожному сервері треба зробити наступні команди для інсталяції MongoDB сервісу:

```
curl -fsSL https://www.mongodb.org/static/pgp/server-4.4.asc | sudo apt-key add
-
echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu
focal/mongodb-org/4.4 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-
org-4.4.list
sudo apt update
sudo apt install mongodb-org
sudo systemctl start mongod.service
sudo systemctl enable mongod
npm install mongodb
```

Для інсталяції Memcached треба зробити лише одну команду:

```
npm install memcached
```

Тепер усе готово для імплементації у код:

```
curl -fsSL https://www.mongodb.org/static/pgp/server-4.4.asc | sudo apt-key add
-
echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu
focal/mongodb-org/4.4 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-
org-4.4.list
sudo apt update
sudo apt install mongodb-org
sudo systemctl start mongod.service
sudo systemctl enable mongod
```

У коді ми будемо використовувати лише один метод з MongoDB, який виконує вставку:

```
const { MongoClient } = require('mongodb');
const url = 'mongodb://localhost:27017';
const client = new MongoClient(url);
const dbName = 'myProject';
        client.insertOne(insertObject, (err, res) => {
                        if (err) {
                            console.error('Error insert to mongodb (detailed
statistic):', err);
                            process.kill(process.pid, 'SIGTERM');
                        }
 });
```

Для Memcached будемо використовувати методи для вставки, діставання та видалення:

```
let Memcached = require('memcached');
let memcached = new Memcached('localhost:11211',
{retries:10,retry:10000,remove:true,failOverServers:['192.168.0.103:11211']})

memcached.set(`pxcache_${hash}`, pixalateCode, memTtl, empf);

memcached.get(`check_${hash}`, (err, data) => {
        if (!data) return;

        memcached.del(`check_${hash}`, (err) => {
            if (err) console.error(`memcached delete - ${err}`);
        });
}
```

### 3.4.4. Загрози і заходи захисту в цій системі

Так як цим займаються адміністратори і не дозволяють робити це програмістам, я майже не брав участі у відворенні засобів захисту. Але серед основного все одно можно винести такі пункти:

- Зробити доступ тільки по VPN підключенням до серверів
- Зробити доступ до серверів тільки по локальній мережі
- Захистити паролі та конфігурації користувачів від змін. Також паролі захистити за допомогою хешування
- Закрити вихід сервісів до світу, щоб не було можливості до них підключитися, якщо користувач знаходиться не на сервері.
- Робити бекапи даних на серверах щодня
- Оновлювати засоби безпеки Linux
- Видалення усіх користувачів за умовчанням з серверу
- Сконфігурувати IpTables
- Забезпечити бесперебойну роботу серверів у датацентрі

# 4. ЕКСПЕРЕМЕНТУВАННЯ РОБОТИ СИСТЕМИ

У цьому розділі ми тестуемо систему та перевіряємо її роботу з тим, що заплановано, та як вона виконується. Нас у першу чергу цікавить як змінилася швидкість запису, кількість даних, які тепер зберігаються, та швидкість доступу до них.

Тестування буде проводитися за допомогою Grafana, який входить до пакету Prometheus, а також за допомогою системних таблиць Clickhouse.

## 4.1. Експеременування роботи MySQL

Тепер ми можемо розглянути як змінилася кількість запитів, швидкість роботи та кількість займаємого простору у MySQL. Для цього зайдемо до Grafana. На рисунках 4.1, 4.2, 4.3 та 4.4 ми можемо впевнетися по графікам, у скільки разів змінилися дані.

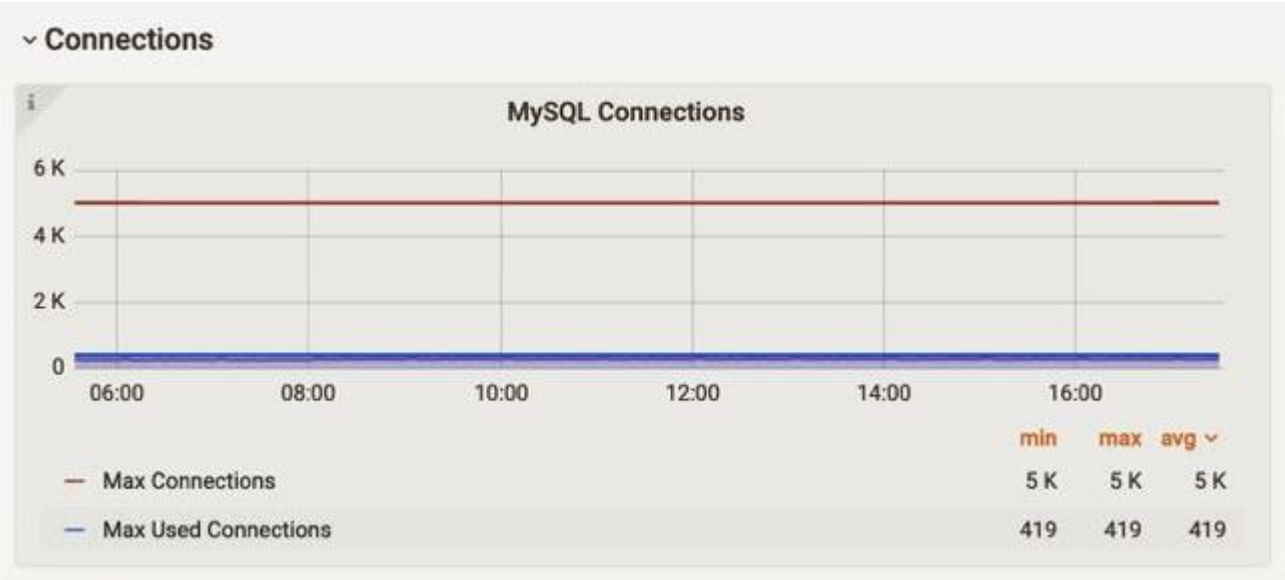



Рисунок 4.1 – Графік Grafana з кількістю підключень

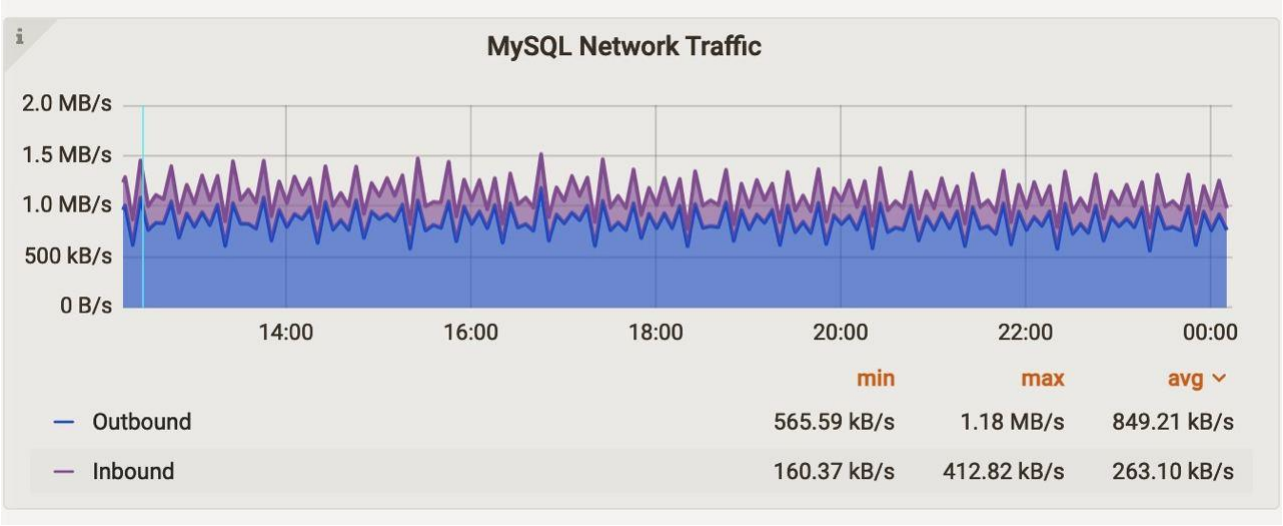



Рисунок 4.2 – Графік Grafana з використанням мережі

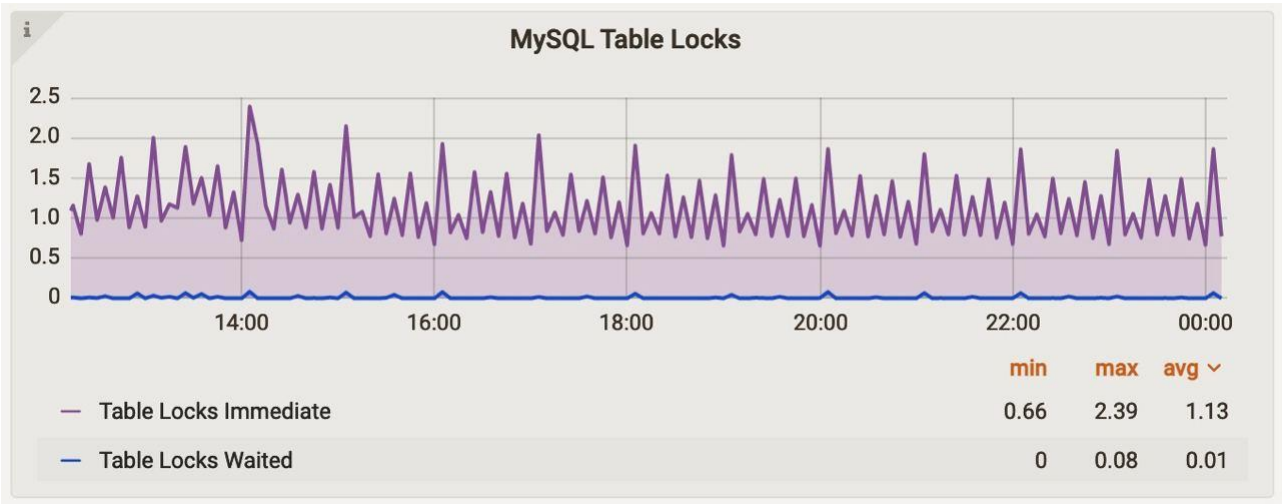



Рисунок 4.3 – Графік Grafana з Table Locks

Таким чином ми бачимо, що кількість Table Locks приблизилася до нуля, кількість підключень зменшилася з середніх 50 тисяч до 5 тисяч, а кількість використовуємого трафіку знизилася зі вхідних 51Mb/s до 263Kb/s у середньому, та вихідного з 32.5Mb/s до 849kB/s.

Займаємий простір також значно знизився, так як ми видалили дані з MySQL і перенесли їх до ClickHouse. На рисунку 4.4 показано частину таблиць, з тих, що залишилися, та скільки вони займають. Також, ми для створили

словники, які зберігаються на самому сервері и майже не займають місця. Вони використовують слова, які найчастіше повторюються, тим самим ще більше зберігаючи простір.

| | column 1 | column 2 ↓1 |
|---|---|---|
| 1 | winimpression_2020 | 4.42 |
| 2 | impression | 3.15 |
| 3 | winimpression_2021 | 3.13 |
| 4 | adm | 2.32 |
| 5 | ads | 0.38 |
| 6 | SSPSettings | 0.03 |
| 7 | win | 0.02 |
| 8 | DSPSettings | 0.02 |

Рисунок 4.4 – Займаєме місце MySQL у Gb

## 4.2. Експерементування роботи ClickHouse з Kafka

Після підключення Kafka та ClickHouse основною проблемою була їх взаємодія. Постійно попадали у чергу дані, які були неправильними. Через велику частину часу було знайдено помилки у коді та вирішено їх. Так як сервери з Clickhouse та Kafka спілкуються по локальній мережі між собою, то за трафік ми не платимо. Робота серверів значно пришвидшилася, так як їм тепер не потрібно постійно звертатися до MySQL.

Також значно зменшився простір, який займають таблиці у Clickhouse. Як можна побачити на рисунку 4.5 – в середньому у 10-15 разів менше. Розробники на FrontEnd стороні та аккаунти підтвердили, що робота стала у рази швидше, та більше немає ситуацій, коли дані не повертаються через Table Locks.

| | column 1 | column 2 |
|---|---|---|
| 1 | impressionsTable_05_2021 | 1.78 |
| 2 | examples_03_2021 | 0.54 |
| 3 | examples_02_2021 | 0.39 |
| 4 | impressionsTable_03_2021 | 1.86 |
| 5 | domains | 0.52 |
| 6 | impressionsTable_04_2021 | 1.96 |
| 7 | impressionsTable_04_2021 | 1.88 |
| 8 | impressionsTable_05_2021 | 1.77 |
| 9 | adm | 1.94 |
| 10 | impressionsTable_03_2021 | 1.89 |
| 11 | impressionsTable_06_2021 | 1.81 |
| 12 | examples_06_2021 | 0.22 |
| 13 | impressionsTable_07_2021 | 1.33 |
| 14 | impressionsTable_07_2021 | 1.94 |
| 15 | impressionsTable_02_2021 | 2.46 |
| 16 | impessionsExamples | 0.78 |
| 17 | impressionsTable_08_2021 | 1.73 |
| 18 | impressionsTable_02_2021 | 3.68 |
| 19 | impressionsTable_06_2021 | 1.91 |
| 20 | impressionsTable_02_2021 | 1.79 |

Рисунок 4.5 – Займаєме місце Clickhouse у Gb

# ВИСНОВКИ

У даній кваліфікаційній роботі було розглянено бази даних, методи зв’язку між ними, їх історію, типи та можливості використання у різних ситуаціях. Я дослідив роботу різних баз даних, порівняв їх між собою, та зробив основне завдання по розробці комплексного методу зберігання та обробки даних з серверів. Було змінено запити, замінено більшість сервісів, підключення сервісу хешування та брокера повідомлень. Таким чином трафік значно знизився, як і кількість займаємого місця. До того ж ми можемо тепер швидко діставати дані з деяких колонок, так як користуємося ClickHouse. Брокер повідомлень Kafka дає нам змогу підключати скільки завгодно Consumer та Producer, а тому все це легко масштабується. MongoDB дав нам змогу динамічно змінювати дані, які ми зберігаємо, а переключившись на ClickHouse та зменшивши займаємий простір ми змогли майже без додаткових трат замінити жорсткі диски на твердотільні.

Основним досядженням у цьому дослідженні є наступні зміни у окремому випадку:

- Зменшення займаємого обсягу у 15 разів
- Збільшення швидкості у середньому у 10 разів та у окремих компонентах до 20 разів
- Великі можливості до легкого масштабування
- Легкість у відстежуванні усіх змін та помилок

Ця робота має бути актуальною для підприємств та компаній, які користуються великими даними і їм потрібно знайти можливості для зберігання цих та подальших даних без збільшення потужностей, швидко і надовго.

# СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ


1. Базы данных. Инжиниринг надежности [Текст]/ Кэмпбелл Л., Мейджорс Ч. - O``Reilly, 2020. - 304 с
2. Технологии проектирования баз данных [Текст]/ Осипов Д. Л. - ДМК Пресс, 2019. – 498 с
3. MySQL 8 для больших данных [Текст]/ Ш. Чаллавала, Д. Лакхатария, Ч. Мехта, К. Патель. - ДМК Пресс, 2018. - 311 с
4. MySQL. Оптимизация производительности, 2-е издание [Текст]/ Д. Заводны, Б. Шварц, П. Зайцев, В. Ткаченко, А. Ленц. - Символ-Плюс, O``Reilly, 2010. - 824 с
5. ClickHouse: очень быстро и очень удобно : веб-сайт. URL: https://habr.com/ru/post/322724/
6. Книга NoSQL. Новая методология разработки нереляционных баз данных [Текст]/ Мартин Ф., Прамодкумар Дж. Садаладж, Диалектика, 2020. - 192 с
7. Брокеры сообщений : веб-сайт. URL: https://www.ibm.com/ru-ru/cloud/learn/message-brokers
8. Підвищення ефективності кешування SQL запитів [Текст]/ Р С. Клейменов, Т. А. Ліхоузова, Саміздат, 2018. - 125 с
9. Релевантный поиск с использованием Elasticsearch и Solr [Текст]/ Тарнбулл Д. - ДМК Пресс, 2018. – 408 с
10. Реляционные базы данных в примерах [Текст]/ Куликов В. - EPAM, 2021. – 424 с
11. SQL The Complete Reference, 3rd Edition [Текст]/ James R. Groff, Paul N. Weinberg, Andy Oppel - McGraw Hill Professional, 2008. – 912 с
12. Введение в системы баз данных Edition [Текст]/ Кристофер Д.- Addison–Wesley, 2003. – 1328 с
13. ClickHouse: как устроен MergeTree : веб-сайт. URL: https://habr.com/ru/post/539538/

14. Practical PostgreSQL [Текст]/ Джошуа Д. Дрейк, John C. Worsley. - O``Reilly, 2002. - 824 с
15. Технологии проектирования баз данных [Текст]/ Осипов Д. Л. - ДМК Пресс, 2019. – 498 с
16. MySQL 8 для больших данных [Текст]/ Ш. Чаллавала, Д. Лакхатария, Ч. Мехта, К. Патель. - ДМК Пресс, 2018. - 311 с
17. MySQL. Оптимизация производительности, 2-е издание [Текст]/ Д. Заводны, Б. Шварц, П. Зайцев, В. Ткаченко, А. Ленц. - Символ-Плюс, O``Reilly, 2010. - 824 с
18. ClickHouse: очень быстро и очень удобно: веб-сайт. URL: https://habr.com/ru/post/322724/
19. Книга NoSQL. Новая методология разработки нереляционных баз данных [Текст]/ Мартин Ф., Прамодкумар Дж. Садаладж, Диалектика, 2020. - 192 с
20. Брокеры сообщений : веб-сайт. URL: https://www.ibm.com/ru-ru/cloud/learn/message-brokers
21. Підвищення ефективності кешування SQL запитів [Текст]/ Р С. Клейменов, Т. А. Ліхоузова, Саміздат, 2018. - 125 с
22. Релевантный поиск с использованием Elasticsearch и Solr [Текст]/ Тарнбулл Д. - ДМК Пресс, 2018. – 408 с
23. Реляционные базы данных в примерах [Текст]/ Куликов В. - EPAM, 2021. – 424 с
24. SQL The Complete Reference, 3rd Edition [Текст]/ James R. Groff, Paul N. Weinberg, Andy Oppel - McGraw Hill Professional, 2008. – 912 с
25. Введение в системы баз данных Edition [Текст]/ Кристофер Д.- Addison–Wesley, 2003. – 1328 с
26. HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера, 3 издание [Текст]/ Прохоренок Н.- Саміздат, 2012. – 900 с
27. Transact-SQL [Текст]/ Фленов М.- 1С, 2006. – 257 с
28. Підвищення ефективності кешування SQL запитів [Текст]/ Р С. Клейменов, Т. А. Ліхоузова, Саміздат, 2018. - 125 с

29. Релевантный поиск с использованием Elasticsearch и Solr [Текст]/ Тарнбулл Д. - ДМК Пресс, 2018. – 408 с

30. Семь баз данных за семь недель. Введение в современные базы данных и идеологию NoSQL [Текст]/ Джим Уилсон, Эрик Редмонд. - ДМК Пресс, 2018. – 424 с

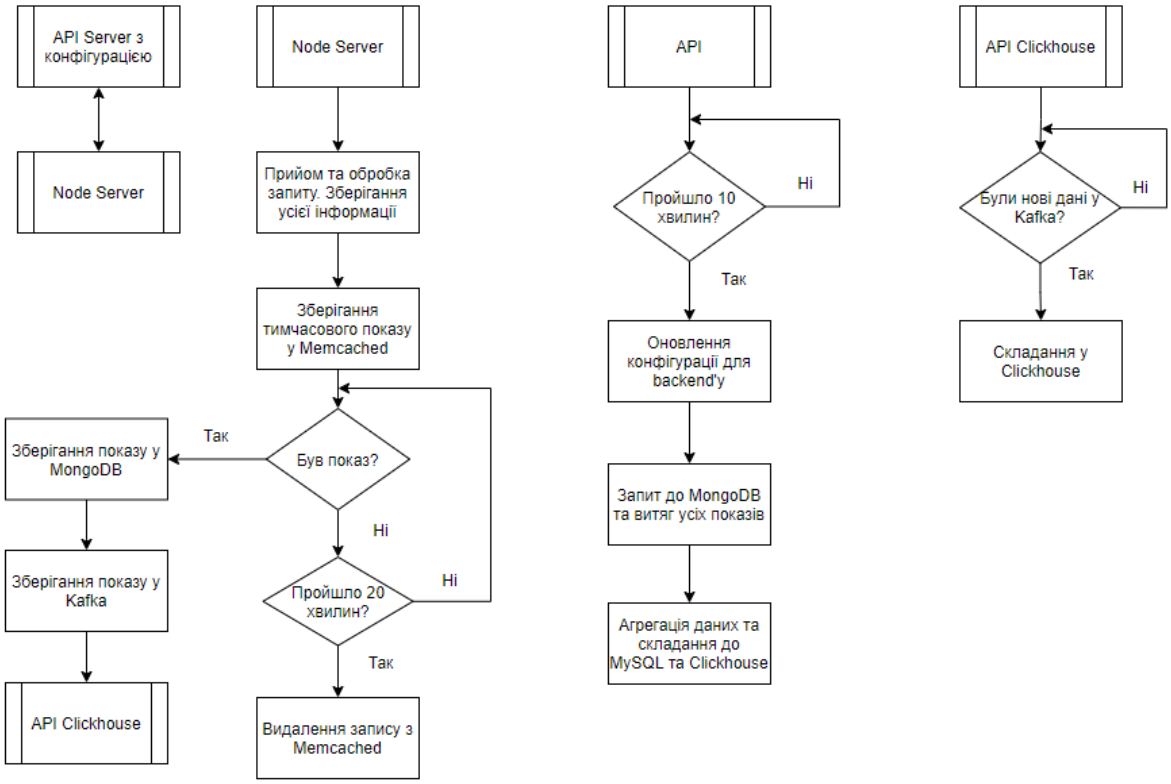
31. SQL The Complete Reference, 3rd Edition [Текст]/ James R. Groff, Paul N. Weinberg, Andy Oppel - McGraw Hill Professional, 2008. – 912 с

32. MongoDB в действии [Текст]/ Бенкер К.- ДМК Пресс, 2011. – 394 с

33. MongoDB: Official Documentation : веб-сайт. URL: https://docs.mongodb.com

**Додаток А**

# Алгоритм взаємодії компонентів між собою



**Додаток Б**

## Алгоритм детальної взаємодії між Kafka та Clickhouse