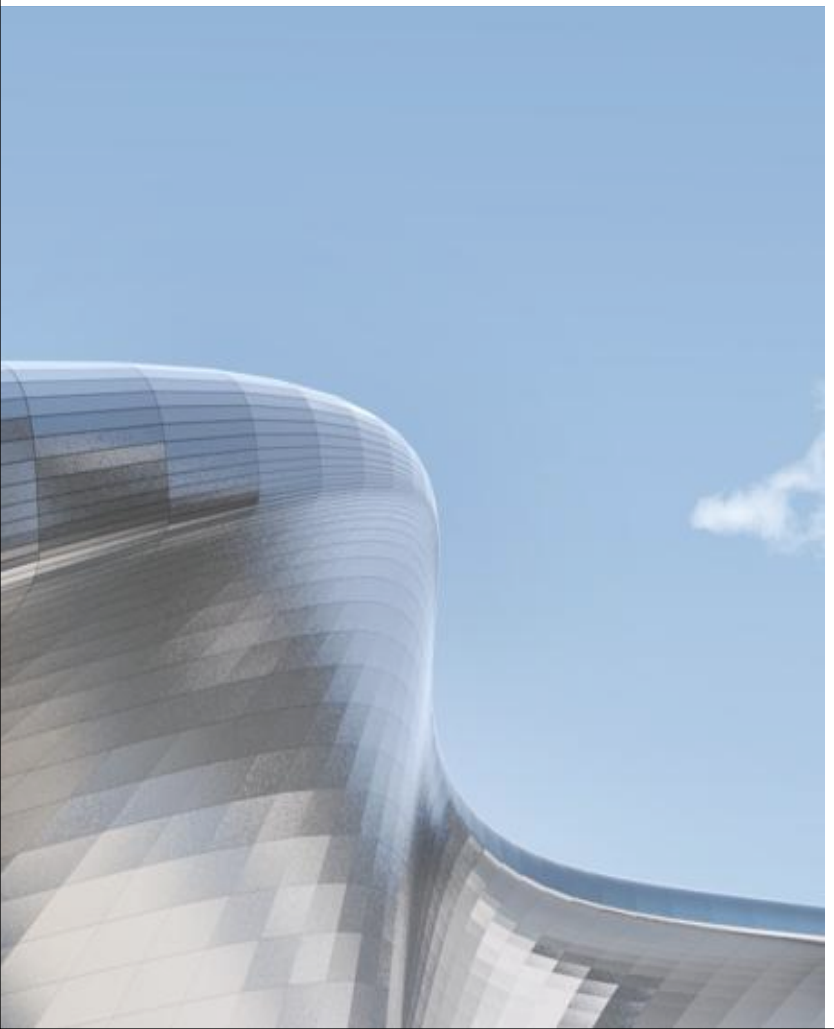
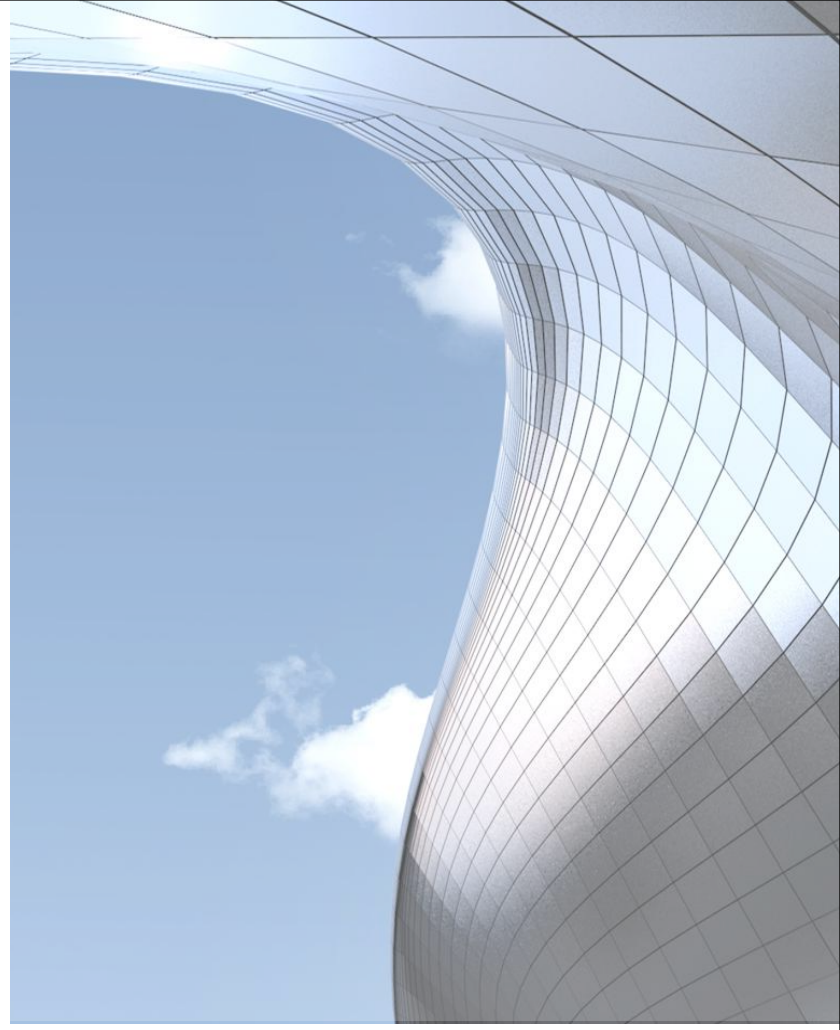




Нереляційні СУБД

Лектор: доцент каф. програмної інженерії ЗНУ
к.т.н., доцент Лимаренко Ю.О.

2025

Зміст курсу

ВСТУП

1 Документоорієнтовані СУБД

2 СУБД типу “Ключ-значення”

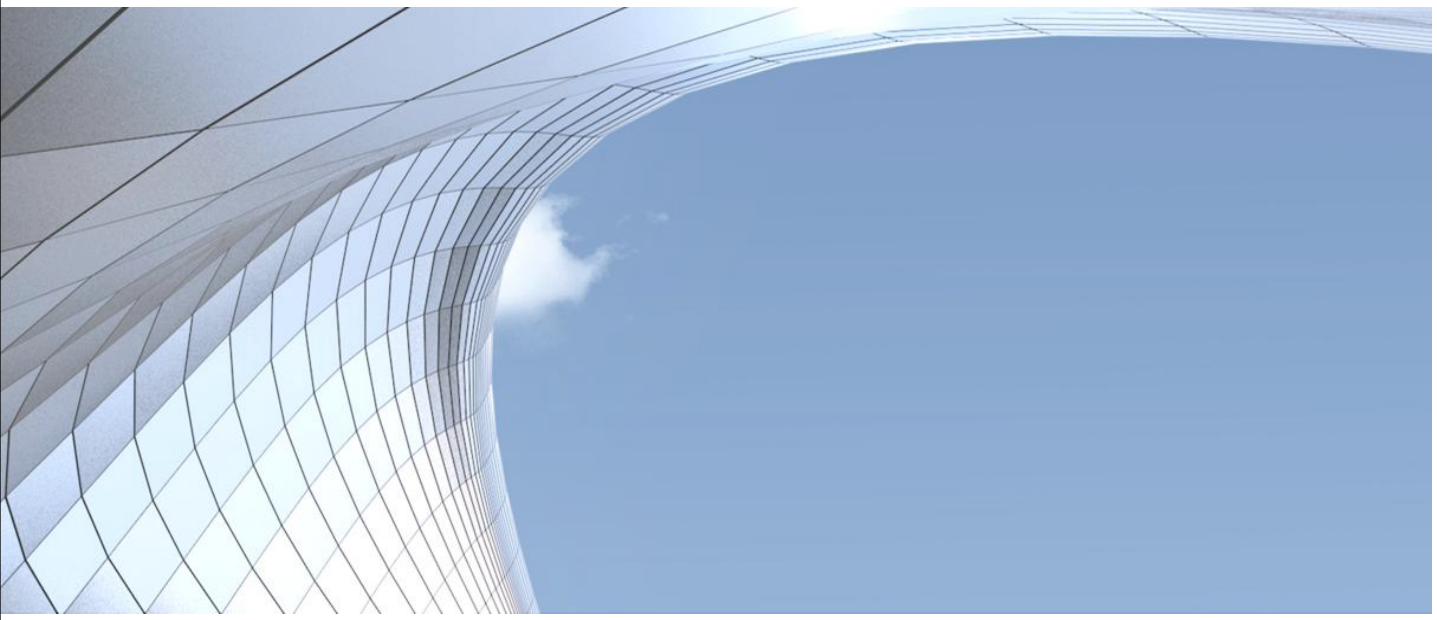
3 Графові СУБД

4 Пошукові СУБД

5 СУБД часових рядів

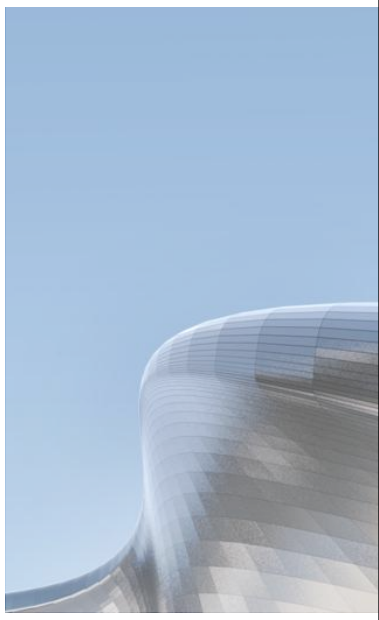
6 Data Analytics

7 Колонкові СУБД



ВСТУП

Причини виникнення
NoSQL рішень



Причини появи нереляційних БД

Упродовж багатьох років **реляційні бази даних** залишались **основним інструментом** для зберігання та обробки інформації.

Вони:

- надавали **потужний механізм запитів**,
- підтримували **транзакційність**,
- забезпечували **цілісність даних**,
- використовували **фіксовану схему даних** (у вигляді таблиць).

Такий підхід був ефективним у традиційних бізнес-системах із чітко структурованими даними.

Проте з початку 2000-х років ситуація почала змінюватися. З'явилися **нові типи застосунків** — динамічні веб-сервіси, соціальні мережі, мобільні додатки, системи збирання телеметрії, IoT-пристрої — які стали генерувати величезні обсяги неструктурованих та напівструктурованих даних.

У таких умовах **реляційні СУБД почали виявляти обмеження**.

Обмеження реляційних СУБД

Негнучкість структури даних — Реляційна модель вимагає жорстко заданої схеми. У швидко змінюваних проєктах це уповільнює розробку.

Проблеми масштабованості — Традиційні СУБД підтримують вертикальне масштабування (заміну сервера на більш потужний), але **мають певні обмеження щодо горизонтального масштабування** (розподіл даних/навантаження по декілької серверах). Проблеми горизонтального масштабування пов’язані з вимогою забезпечення **ACID-властивостей транзакцій** (атомарність, узгодженість, ізольованість, довговічність).

Продуктивність і швидкодія — Ті ж самі ACID-властивості транзакцій певним чином **обмежують пропускну здатність** реляційних баз даних. Це в першу чергу стосується високонавантажених та розподілених систем (реальний час, big data, логування).

Історичний розвиток NoSQL: ключові етапи

- **1998–2000** — Компанії Google і Amazon першими стикаються з масштабними проблемами SQL. Вони створюють власні рішення: Bigtable (Google) і Dynamo (Amazon).
- **2006** — Публікуються наукові статті про Bigtable і Dynamo. Це стає основою для створення відкритих аналогів.
- **2007–2009** — З'являються перші відомі NoSQL СУБД:
  - Cassandra (на основі Dynamo, розроблена Facebook),
  - MongoDB (гнучке зберігання JSON-подібних документів),
  - CouchDB (орієнтована на веб).
- **2009** — Відбувається конференція "NoSQL meetup" у Сан-Франциско. Вперше термін NoSQL починає використовуватись широко. Згодом його трактують як "Not Only SQL", тобто доповнення, а не альтернатива SQL.
- **2010**–далі — Широке впровадження NoSQL у хмарних платформах, Big Data-аналітиці, потоковій обробці, мікросервісах, мобільних застосунках.

NoSQL

Виникнення NoSQL баз даних стало логічною відповіддю на нові виклики в IT-сфері. Вони **не призначені для повної заміни реляційних СУБД**, а використовуються там, де традиційні рішення виявляються надто повільними, негнучкими або немасштабованими.

NoSQL СУБД надають:

- **Гнучкість у моделюванні даних**,
- **Масштабованість** для великих систем,
- **Продуктивність** у сучасних високонавантажених сценаріях.

Деякі NoSQL СУБД мають механізм підтримки ACID-властивостей транзакцій.

ACID vs CAP vs BASE

В контексті СУБД досить часто згадуються абревіатури ACID, CAP, BASE. Який між ними зв’язок і як вони співвідносяться з SQL/NoSQL СУБД?

Обмеження РСУБД виходять на перший план, перш за все, у *розподілених системах*.

Розподіленими системами називають такі, в яких дані зберігаються не на одному сервері, а на кількох вузлах або навіть у різних датацентрах. Необхідність у розподілі виникає, коли:

- **Обсяг даних** настільки великий, що не поміщається на одному сервері.
- Потрібна **надійність**: у разі збою одного вузла дані не повинні зникнути.
- Потрібна **висока доступність**: користувачі з різних регіонів повинні мати швидкий доступ до даних.
- Виникає необхідність масштабування — розподіл навантаження між кількома серверами.

Однак із розподіленістю з’являються і **технічні обмеження**, які випливають з **ACID-властивостей транзакцій**. Одне з ключових обмежень розподілених систем сформульовано у вигляді CAP-теореми.

CAP-теорема

CAP-теорема: для **розподілених систем** можливо вибрати лише **дві** з наступних трьох властивостей:

- **Consistency** – інформація на різних вузлах узгодженаю.
- **Availability** – система відповідає на запити.
- **Partition tolerance** – зв'язки між вузлами можуть обриватися.

Можливі комбінації:

- **CP**-системи гарантують узгодженість і стійкість до розділення, але можуть бути тимчасово недоступні.
- **AP**-системи гарантують доступність і толерантність до розділення, але допускають тимчасову неузгодженість.
- **CA**-системи не витримують мережевих збоїв, тому практично не зустрічаються в розподіленому середовищі.

Для класичних **реляційних** СУБД типовим є прагнення до **CA** за рахунок P. Тоді як різні категорії NoSQL СУБД можуть в своїх реалізаціях наближатися до різних комбінацій: CA, CP, AP.

Але більшість NoSQL СУБД свідомо жертвують повною узгодженістю (Consistency), щоб досягти високої доступності (Availability) та стійкості до мережевих збоїв (Partition tolerance), тобто реалізують **AP**-властивості.

BASE-архітектура

З часом характеристики CAP-теореми привели до так званої *BASE-архітектури*.

BASE-архітектура:

- **Basically Available** – дані доступні завжди, коли до них відбувається звернення.
- **Soft state** – дані можуть знаходитись у неузгодженому стані у якийсь період часу.
- **Eventually consistent** – зрештою дані у сховищі перейдуть в узгоджений стан.

⚠ І CAP-теорема і BASE-архітектура визначають характеристики саме розподілених систем. Якщо система за своєю архітектурою **не є розподіленою** (наприклад, одновузлові інстанси MongoDB чи Redis), то, як правило, в таких системах можна реалізувати повноцінні ACID-властивості транзакцій.

Класи нереляційних СУБД

Документоорієнтовані СУБД

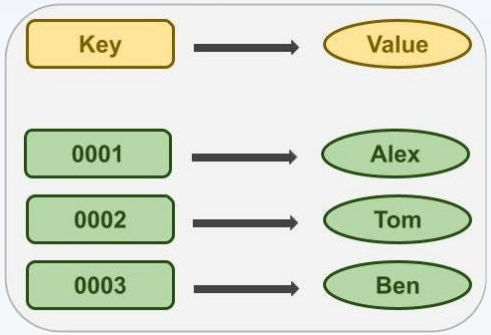
Document 1

Id: “001”,
Name: “Alex Bill”,
Phone: “+001 234 5341”,
Department: “Finance”

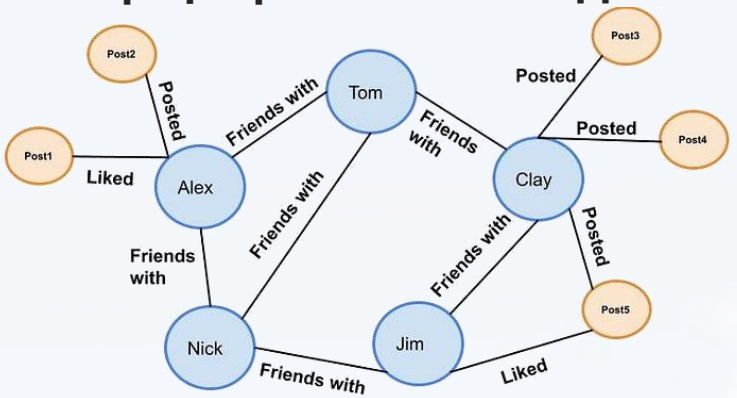
Document 2

Id: “003”,
Name: “Alex Bill”,
Phone: {
Home: “+001 234 5341”,
Office: “+001 111 2759”
}
Department: “Finance”

СУБД типу Ключ-Значення



Граф-орієнтовані СУБД

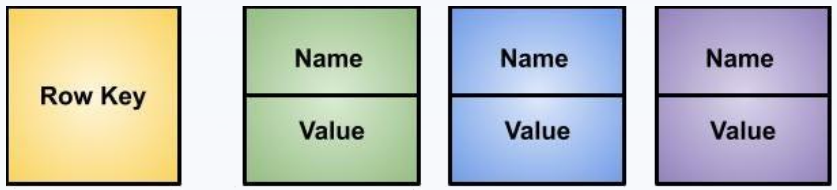


NoSQL

Пошукові СУБД

GEO/GIS СУБД

Колонкові СУБД

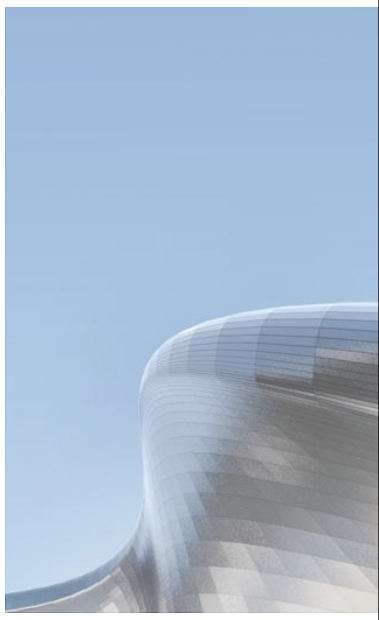


СУБД часових рядів

Векторні СУБД

PART 01

Документо-орієнтовані СУБД



Загальна характеристика документоорієнтованих СУБД

Модель даних подібних сховищ дозволяє **об'єднувати множину пар "ключ-значення"** в абстракцію, яка називається **«документ»**.

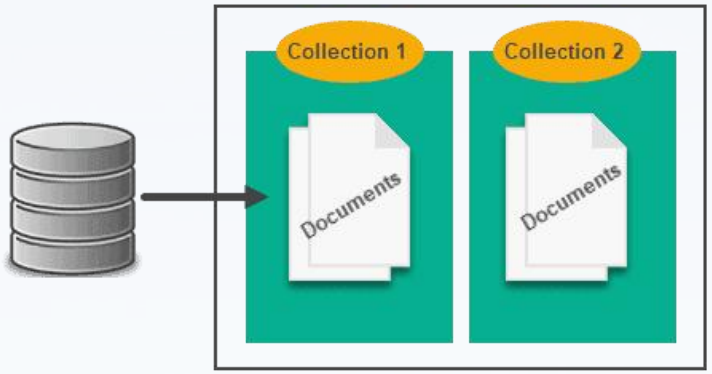
Документи можуть мати вкладену структуру та об'єднуватись у колекції, однак це скоріше зручний спосіб логічного об'єднання, оскільки ніякої **жорсткої схеми документи не мають** і множини пар ключ-значення, навіть у рамках однієї колекції, можуть бути абсолютно довільними.

Робота з документами виконується за ключем, проте існують рішення, що дозволяють здійснювати запити за значеннями атрибутів.

Документоорієнтовані СУБД в основному **використовується для** CMS, платформ блогів, аналітики в реальному часі та систем електронної комерції.

Їх **не слід використовувати** для складних транзакцій, які вимагають кількох операцій або запитів щодо різних агрегатних структур.

Rank			DBMS	Database Model	Score		
Jul 2025	Jun 2025	Jul 2024			Jul 2025	Jun 2025	Jul 2024
1.	1.	1.	MongoDB	Document, Multi-model	403.83	+0.99	-25.99
2.	2.	2.	Databricks	Multi-model	108.04	+3.36	+24.74
3.	3.	3.	Amazon DynamoDB	Multi-model	84.15	+0.81	+13.20
4.	4.	4.	Microsoft Azure Cosmos DB	Multi-model	21.68	-0.75	-5.44
5.	5.	↑ 6.	Firebase Realtime Database	Document	14.35	+0.12	+0.47



Knowledge Check

До переваг документоорієнтованих СУБД відносяться:

A. Гнучка схема зберігання даних
B. Можливість ефективної реалізації складної аналітики на структурованих даних
C. Легке масштабування по горизонталі (sharding)
D. Суворе дотримання ACID-властивостей у транзакціях
E. Обов’язкова попередня нормалізація даних
F. Висока швидкість читання та запису при роботі з неструктурованими даними

Knowledge Check

До переваг документоорієнтованих СУБД відносяться:

A. **Гнучка схема зберігання даних**

B. Можливість ефективної реалізації складної аналітики на структурованих даних

C. **Легке масштабування по горизонталі (sharding)**

D. Суворе дотримання ACID-властивостей у транзакціях

E. Обов’язкова попередня нормалізація даних

F. **Висока швидкість читання та запису при роботі з неструктурованими даними**

MongoDB є на сьогодні **найбільш популярною документоорієнтованою СУБД**, яка не вимагає опису схеми таблиць.

Впевнено входить до десятки найбільш використовуваних баз даних у світі.

- Розробник: американська компанія MongoDB Inc. (попередня назва “10gen”)
- MongoDB написано мовою C++.
- Перший випуск: 2009 р.
- Остання версія: 8.0.10 (28 травня 2025 року)
- Ліцензія: Server Side Public License (SSPL), яка є похідною від GNU.
- MongoDB використовується Netflix, eBay, PayPal, Cisco, BOSCH, Forbs,The New York Times, ...

MongoDB

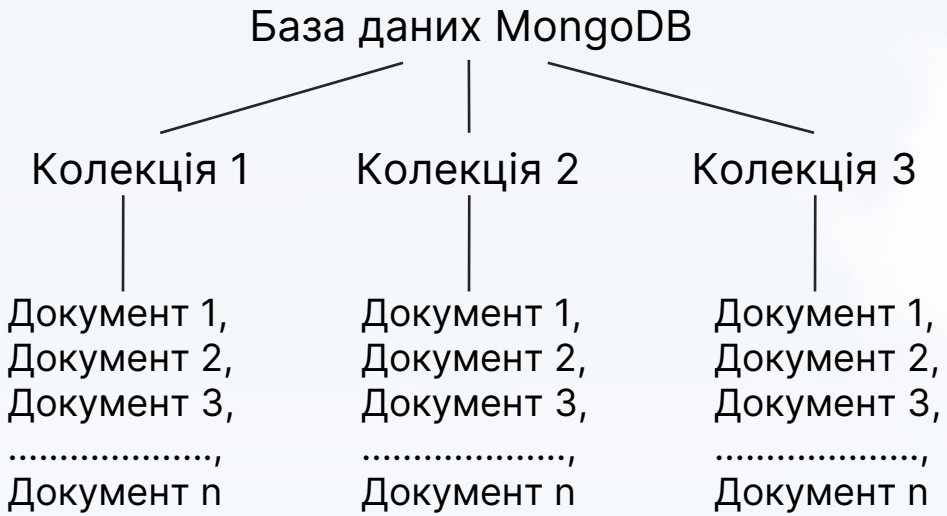
Одиниця зберігання даних у Mongo – **документ**, який представляє собою **множину пар ключів та значень** і зберігається у форматі BSON (Binary JavaScript Object Notation).

Документи можуть містити складну за структурою інформацію.

Набори тематично пов'язаних документів об'єднуються у **колекції**.

```
> db.towns.insert
   ({
      name: "New York",
      population: 22200000,
      last_census: ISODate("2009-07-31"),
      famous_for: ["statue of liberty", "food" ],
      mayor : {
          name : "Michael Bloomberg",
          party : "I"
      }
   })
```

колекція



Документи в одній і тій самій колекції можуть мати зовсім різну структуру. Однак з практичної точки зору це не зовсім зручно.

MongoDB

Є поняття **первинного ключа**, яке описує поле, що має унікальні значення. Це поле _id, яке певною мірою є штучним. Якщо явно не вказати його значення, то MongoDB автоматично його згенерує, гарантуючи унікальність в межах колекції.

```
> db.towns.find()
  ({
      "_id" : ObjectId("4d0ad975bb30773266f39fe3"),
      name: "New York",
      population: 22200000,
      last_census: ISODate("2009-07-31"),
      famous_for: ["statue of liberty", "food" ],
      mayor : {
          name : "Michael Bloomberg",
          party : "I"
      }
  })
```

Таблиця відповідності термінів РСУБД та MongoDB:

SQL	MongoDB
Відношення/Таблиця	Колекція
Кортеж/Рядок	Документ
Атрибут/Стовпчик	Поле
Первинний ключ	ObjectId
Індекс	Індекс
Представлення	Представлення

CRUD-операції. Додавання даних

CRUD - скорочене найменування 4-х базових функцій, що використовуються при роботі зі сховищами даних:

- створення (Create);
- читання (Read);
- редагування (Update);
- видалення (Delete).

Для **додавання** до колекції можуть використовуватись два методи:

- **insertOne()**: додає один документ,
- **insertMany()**: додає кілька документів.

В MongoDB запити є **регістрозалежними** і мають **строгу типізацію**. Тобто наступні два документи не будуть ідентичними:

```
{"age" : "28"}
{"age" : 28}
```

Назви ключів можуть використовуватися в лапках, а можуть і без лапок.

CRUD-операції. Додавання даних

Приклад додавання документу до колекції users бази даних employee:

db.users.**insertOne**({"name": "Tom", "age": 28, "languages": ["english", "spanish"]})

Документ представляє собою набір пар *ключ-значення*. У цьому випадку документ, що додається, має три ключі: name, age, languages, і кожному з них співставляє певне значення. Наприклад, ключу languages в якості значення співставляється масив.

Значення для поля "_id" (тобто унікального ідентифікатора документа), в цьому випадку згенерується автоматично.

Унікальний ідентифікатор можна встановити самостійно при додаванні даних:

db.users.**insertOne**({"_id": 123457, "name": "Tom", "age": 28, "languages": ["english", "spanish"]})

або використовувати для індетифікатора тип ObjectId, який повинен містити рядок з 24 символів:

db.users.**insertOne**({"_id": ObjectId("62e27b1b06adfcddf4619fc6"), "name": "Tom", "age": 28, "languages": ["english", "spanish"]})

CRUD-операції. Читання даних

Для вибірки даних з колекції використовується функція **find()**.

Для фільтрації даних вказуються конкретні значення ключів:

```
db.users.find({name: "Tom", age: 32})
```

Функції **skip()** та **limit()** дозволяють відповідно пропустити вказану кількість документів та обмежити вибірку вказаною кількістю документів. Функція **sort()** використовується для сортування, у якості аргумента використовуючи пару: ключ, за значенням якого буде відбуватись сортування, та значення, що вказує на напрямок сортування: 1 — за збільшенням, -1 — за зменшенням.

Всі ці функції можна поєднувати в одному ланцюжку, використовуючи оператор ".", нариклад:

```
db.users.find().sort({name: 1}).skip(3).limit(3)
```

За замовчуванням вибірка містить усі поля документа, однак, в тому випадку, якщо потрібно вибрати лише конкретні поля, методу find() можна передавати другий аргумент у вигляді JSON-структури, з ключами, що збігаються з назвами стовпців і значеннями 1, якщо поле має потрапляти у вибірку та 0, якщо його необхідно виключити з вибірки.

Наприклад,

```
db.users.find({name: "Tom"}, {_id: 0, name: 1, age: 1})
```

CRUD-операції. Читання даних

У якості селектора можуть виступати не тільки рядки, але і **регулярні вирази**, наприклад, знайти всі документи, в яких значення ключа name починається з літери T:

```
db.users.find({name:/T\w+/i})
```

У таблиці нижче представлені логічні оператори SQL і логічні оператори MongoDB:

Оператор SQL	MongoDB	Опис
<	$lt	менше
<=	$lte	менше або дорівнює
>	$gt	більше
>=	$gte	більше або дорівнює
<>	$ne	не дорівнює
NOT	$not	заперечення
EXISTS	$exists	перевірка існування поля
OR	$or	або
NOT OR	$nor	не або
RLIKE, REGEXP	$regex	відповідність регулярному виразу
LIKE	$elemMatch	відповідність всіх полів вкладеного документу
-	$size	відповідність розміру масива
-	$type	відповідність, якщо поле має вказаний тип
IN	$in	входить у список
NOT IN	$nin	не входить у список
ALL	$all	одночасне співпадіння набору елементів

CRUD-операції. Читання даних

Наприклад, вибірка всіх користувачів віком від 30 до 50 років може бути отримана за допомогою запиту:

```
db.users.find ({age: {$gt: 30, $lt: 50}})
```

Вибірка документів зі значеннями ключа з певного набору матиме вигляд:

```
db.users.find ({languages: {$in: ["english", "french"]}})
```

Якщо ж потібні записи, в яких одночасно присутні всі значення з переліку, використовується $all:

```
db.users.find ({languages: {$all: ["english", "french"]}})
```

Якщо в документах, що зберігаються в колекції, містяться **вкладені об'єкти**, і фільтрація відбувається за параметрами вкладених об'єктів, то використовується оператор "**.**".

Наприклад, якщо ми додамо до колекції користувачів, для яких буде зазначена інформація про компанії, в яких вони працюють:

```
db.users.insertOne({"name": "Alex", "age": 28, company: {"name": "microsoft", "country": "USA"}})
```

то вибірка тих корисувачів, які працюють в компанії "microsoft", буде мати вигляд:

```
db.users.find({"company.name": "microsoft"})
```

CRUD-операції. Читання агрегованих даних

За допомогою функції **countDocuments()** можна отримати кількість елементів у колекції:

```
db.users.countDocuments()
```

Агрегацію можна поєднувати з фільтрацією даних:

```
db.users.find({age: {$gt: 25, $lt: 60}}).countDocuments()
```

Більше можливостей для агрегації даних дає метод **aggregate().** Використання цього методу аналогічно застосуванню виразу GROUP BY в SQL.

Розглянемо дію методу на прикладі:

```
db.users.aggregate({"$group": {_id: "$company", count: {$sum: 1}}})
```

В наведеному прикладі:

$group: агрегатор, який поверне новий документ,

_id: вказує на ключ, за яким потрібно проводити групування ($+назва поля),

$sum: оператор для обчислення.

Зокрема, параметр _id вказує, що групування буде проводитися за ключем company: _id: "$company". Значення параметра $sum дозволяє задати значення, яке буде враховуватись при обчисленні суми. Таким чином, буде розраховано кількість працівників в кожній компанії.

CRUD-операції. Читання агрегованих даних

За допомогою запиту

```
db.users.aggregate({"$group": {_id: "$company", avg_salary:{$avg: $salary}}})
```

можна обчислити середню зарплатню по кожній команії.

Якщо попередньо треба відфільтрувати документи за віком, тобто, окремо порахувати середню зарплатню по компаніях для категорії співробітників віком до 30 років, то запит набуде вигляду:

```
db.users.aggregate([{match: {age:{$lt:30}}}, {"$group": {_id:"$company", avg_salary: {$avg:$salary}}}])
```

В даному випадку match дозволяє попередньо відфільтрувати за віком.

CRUD-операції. Редагування даних

СУБД MongoDB надає кілька можливостей для оновлення даних.

1. Використання методу **save()**. У якості фактичного параметру метод приймає документ. У цьому документі ключем можна передати параметр _id. Якщо метод знаходить документ із таким значенням _id, документ оновлюється. Якщо ж з подібним _id немає документів, документ додається.

   Наприклад,

   db.users.save({name: "Eugene", age : 29, languages: ["english", "german", "spanish"]})

2. Метод **update()** дозволяє більш детально налаштувати оновлення.

   Приймає три параметри:

   - query: приймає запит на вибір документа, який потрібно оновити;
   - objNew: представляє документ з новою інформацією, який замістить старий під час оновлення;
   - options: визначає додаткові параметри при оновленні документів і може приймати два аргументи: upsert і multi.

   Якщо параметр **upsert** має значення true, то mongodb буде оновлювати документ, якщо він знайдений, і створювати новий, якщо такого документа немає. Якщо ж він має значення false, то mongodb не буде створювати новий документ, якщо запит на вибірку не знайде жодного документа.

   Параметр multi вказує, чи повинен оновлюватися перший елемент у вибірці (використовується за замовчуванням, якщо цей параметр не вказано) або повинні оновлюватися всі документи у вибірці.

CRUD-операції. Редагування даних

Наприклад,

```
db.users.update({name: "Tom"}, {name: "Tom", age : 25, married: false}, {upsert: true})
```

Тепер документ, знайдений запитом {name : "Tom"}, буде перезаписаний документом {"name": "Tom", "age" :"25", "married" : false}.

Для оновлення лише одного з ключів використовується оператор **$set**. Якщо документ не містить оновлюване поле, воно створюється:

```
db.users.update({name: "Eugene", age: 29}, {$set: {age: 30}}
```

У цьому випадку оновлювався лише один документ, перший у вибірці. Вказавши значення multi: true, можна оновити всі документи вибірки:

```
db.users.update({name: "Tom"}, {$set: {name: "Tom", age: 25, married: false}}, {multi: true})
```

Для видалення окремого ключа використовується оператор **$unset**:

```
db.users.update({name : "Tom"}, {$unset: {salary: 1}})
```

CRUD-операції. Видалення даних

Для видалення документів у MongoDB передбачений метод **remove()**.

Наприклад,

```
db.users.remove({name : "Tom"})
db.users.remove({age: {$lt : 30}})
db.users.remove({})                    // будуть видалені всі документи з колекції
```

Для видалення бази даних та колекції використовуються методи dropDatabase() та drop() відповідно.

Наприклад,

```
db.users.drop()
```

CRUD-операції. Посилання

MongoDB **не виконує операції з'єднання**. Через розподілену природу системи з'єднання виявилося б вкрай неефективною операцією.

Для забезпечення можливості зберігання в БД логічно пов'язаної інформації у MongoDB існує 2 підходи:

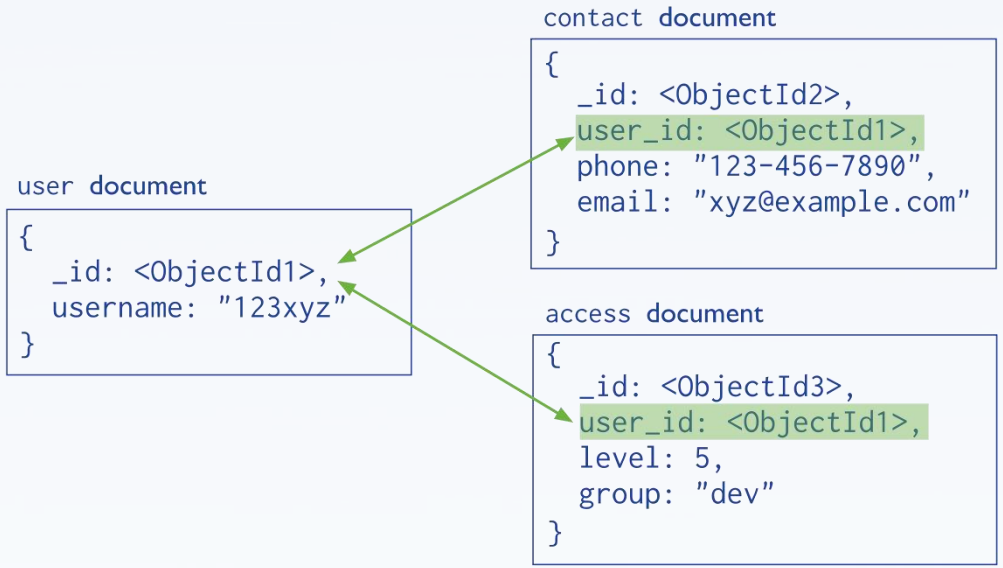
1) **вкладення документів**:

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
             phone: "123-456-7890",
             email: "xyz@example.com"
           },
  access: {
             level: 5,
             group: "dev"
           }
}
```

Embedded sub-document

Embedded sub-document

2) **використання посилань**:



CRUD-операції. Посилання

Існує 2 варіанти організації зв'язку між документами різних колекцій (або однієї колекції):

- **ручна установка посилань**,
- **автоматичне зв'язування**.

Ручна установка посилань зводиться до призначення значення поля _id одного документа полю іншого документа.

Нехай є колекції companies і persons, що представляють компанії та працівників, які працюють у цих компаніях. У колекції companies є документ, що представляє компанію:

```
db.companies.insertOne({"_id" : "microsoft", year: 1974})
```

Тепер додамо до колекції users документ, який представляє працівника. У цьому документі буде поле company, що представляє компанію, де працює працівник. Важливо, що значенням для цього поля ми встановлюємо значення ключа _id доданого вище документа:

```
db.users.insertOne({name: "Tom", age: 28, company: "microsoft"})
```

Тепер отримаємо цей документ з колекції users:

```
user = db.users.findOne({name: "Tom"})
```

Тепер знайдемо посилання на компанію цього користувача в колекції companies:

```
db.companies.findOne({_id: user.company})
```

CRUD-операції. Посилання

Автоматичне зв'язування відбувається за допомогою функціональності dbRef, дію якого розглянемо на наступному прикладі.

Спочатку додамо новий документ до колекції company:

```
google = db.companies.insertOne({name: "google", year: 1998})
```

Тепер створимо новий документ для колекції users, у якого ключ company зв'яжемо з щойно доданим документом google:

```
db.users.insertOne({name: "Sam", age: 25, company: {"$ref": "companies", "$id": google.insertedId}})
```

Тепер протестуємо, зробивши запит на вибірку даних про користувачів разом з компаніями:

```
db.users.aggregate([
    { $lookup: {                        // аналог LEFT OUTER JOIN
        from: "companies",              //  ім'я колекції, з якою потрібно виконати об'єднання
        localField: "company.$id",
        foreignField: "_id",
        as: "companies" }  },
    { $unwind: "$companies"},   // розгортання масиву для вихідного документу
    { $project:                         // вибір конкретних полів, які будуть включені до вихідних документів
      { _id: 1,                         // 1 - включити, 0 - не включати
        name: 1,
        "companies.name": 1,
        "companies.year": 1 } }
])
```

MongoDB: Стратегії моделювання даних

Денормалізація: Прискорює читання та спрощує запити за рахунок включення пов'язаних даних безпосередньо до документа.

Ручні посилання: Забезпечують зв'язок між документами за допомогою масивів ObjectID, пропонуючи таким чином гнучкий підхід до управління схемами.

Денормалізація проти посилань: У цьому питанні завжди є баланс. Зв’язки "один до багатьох" у більшості випадків краще реалізовувати з використанням вбудованих документів, тоді як зв’язки "один до мільйонів" вимагають посилань.

Операції зміни: MongoDB делегує каскадні оновлення та видалення на рівень програми, що є особливо важливим у рамках інтенсивних операцій запису.

Цілісність даних: Через відсутність вбудованих механізмів обмежень у MongoDB **відповідальність за підтримку цілісності даних лежить на застосунку**.

MongoDB: Індекси

Індекси використовуються для **підвищення продуктивності запитів**, дозволяючи MongoDB ефективно знаходити та отримувати дані без необхідності сканування всієї колекції.

Створення інексу:

`db.collection.createIndex(keys, options)` // Наприклад, `db.users.createIndex({ name: 1 })`

`keys`: Об'єкт, що визначає поля для індексування та тип індексу (наприклад, {field: 1} для зростаючого індексу, {field: -1} для спадаючого, {field: "hashed"} для хешованого, {field: "2dsphere"} для геопросторового, {field: "text"} для повнотекстового індексу тощо). Для створення складеного індексу вказується кілька полів: {field1: 1, field2: -1}.

`options`: Необов'язковий об'єкт, який може включати різні параметри, такі як:

unique: true: Створює унікальний індекс, що забороняє дублювання значень в індексованих полях.

sparse: true: Створює розріджений індекс, який індексує лише документи, що містять індексоване поле.

name: "indexName": Дозволяє задати ім'я для індексу.

background: true: Дозволяє створювати індекс у фоновому режимі, не блокуючи інші операції з базою даних.

expireAfterSeconds: <число>: Створює TTL (Time-To-Live) індекс, який автоматично видаляє документи через вказану кількість секунд після значення в індексованому полі (поле повинно бути типу Date).

partialFilterExpression: { <поле>: <умова> }: Створює частковий індекс, який індексує лише документи, що відповідають вказаному фільтру.

Перевірка наявних індексів:

`db.users.getIndexes()`

MongoDB: Індекси

Індекси використовуються для **підвищення продуктивності запитів**, дозволяючи MongoDB ефективно знаходити та отримувати дані без необхідності сканування всієї колекції.

Створення інексу:

`db.collection.createIndex(keys, options)` // Наприклад, `db.users.createIndex({ name: 1 })`

`keys`: Об'єкт, що визначає поля для індексування та тип індексу (наприклад, {field: 1} для зростаючого індексу, {field: -1} для спадаючого, {field: "hashed"} для хешованого, {field: "2dsphere"} для геопросторового, {field: "text"} для повнотекстового індексу тощо). Для створення складеного індексу вказується кілька полів: {field1: 1, field2: -1}.

`options`: Необов'язковий об'єкт, який може включати різні параметри, такі як:

unique: true: Створює унікальний індекс, що забороняє дублювання значень в індексованих полях.

sparse: true: Створює розріджений індекс, який індексує лише документи, що містять індексоване поле.

name: "indexName": Дозволяє задати ім'я для індексу.

background: true: Дозволяє створювати індекс у фоновому режимі, не блокуючи інші операції з базою даних.

expireAfterSeconds: <число>: Створює TTL (Time-To-Live) індекс, який автоматично видаляє документи через вказану кількість секунд після значення в індексованому полі (поле повинно бути типу Date).

partialFilterExpression: { <поле>: <умова> }: Створює частковий індекс, який індексує лише документи, що відповідають вказаному фільтру.

collation: { locale: <string>, strength: <int> }: Дозволяє задати параметри порівняння рядків для індексу (наприклад, для регістронезалежного пошуку).

MongoDB: Реплікації

Реплікація – це процес синхронізації даних на кількох серверах. Цей механізм збільшує доступність даних, копії яких зберігаються на різних серверах. Реплікація захищає базу даних від втрати єдиного сервера та дозволяє зберегти дані у разі технічної несправності одного з серверів.

Переваги реплікації:

- захищає дані,
- підвищена доступність даних,
- резервування даних,
- можливість вилучення одного сервера для технічного обслуговування,
- не впливає на роботу самого додатка.

У MongoDB реплікація досягається шляхом використання **набору копій** (replica set). Це група екземплярів **mongod**, які зберігають **однакові набори даних**.

Серед копій один вузол – це **первинний вузол**, який отримує всі операції **запису**. Всі інші вузли – **вторинні.** Усі дані **копіюються** з первинного вузла у вторинні. Набір копій може мати лише **один первинний вузол**.

Під час відключення первинного вузла (обслуговування, поломка тощо) встановлюється **новий первинний вузол.**

Після повернення первинного вузла в роботу він стає вторинним.

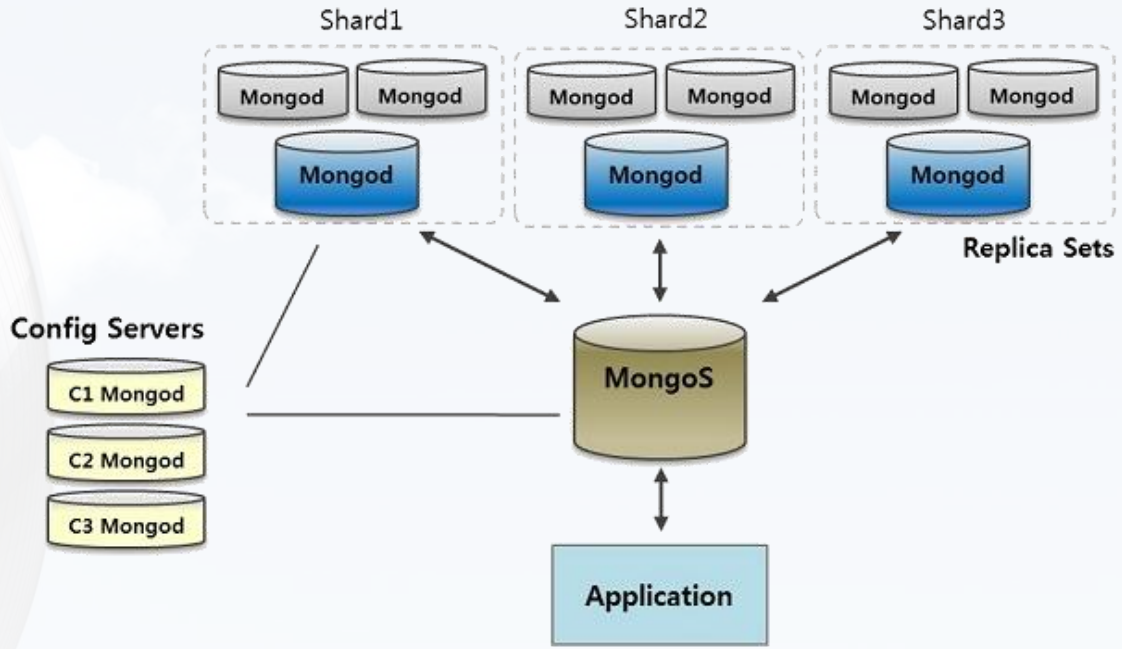
MongoDB: Шардинг

Шардинг - це процес зберігання документів на декількох серверах і це спосіб, яким MongoDB справляється з великими даними.

Зі збільшенням кількості даних, один сервер не може зберігати всі дані, ані записувати їх, ані давати доступ до них.

Шардинг вирішує проблему шляхом горизонтального масштабування. Завдяки цьому механізму ми можемо підключати додаткові сервери для зберігання, запису та читання даних.

Загальна схема шардинга в MongoDB представлена на рисунку:



Knowledge Check

Які з перелічених типів застосунків найкраще підходять для документоорієнтованих баз даних?

A. Логування подій, наприклад, покупок в інтернет-магазині
B. Зберігання складних реляційних даних із нормалізованою структурою
C. Рекомендаційні системи в інтернет-магазинах
D. Зберігання фінансових даних, що вимагають суворих ACID-транзакцій
E. Онлайн-блогінг, де кожен пост та коментар можуть бути окремим документом
F. Навігаційні програми для розрахунку маршрутів

Knowledge Check

Які з перелічених застосувань найкраще підходять для документоорієнтованих баз даних?

A. Логування подій, наприклад, покупок в інтернет-магазині

B. Зберігання складних реляційних даних із нормалізованою структурою

C. Рекомендаційні системи в інтернет-магазинах

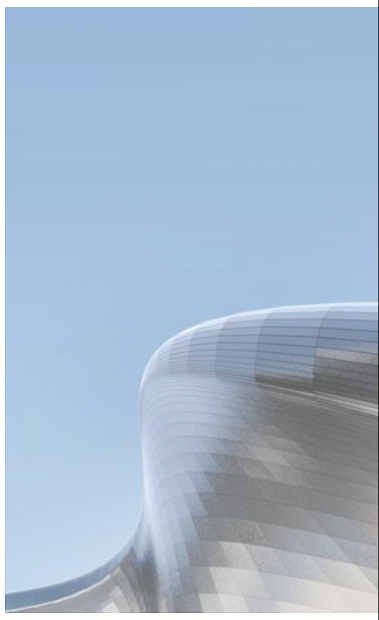
D. Зберігання фінансових даних, що вимагають суворих ACID-транзакцій

E. Онлайн-блогінг, де кожен пост та коментар можуть бути окремим документом

F. Навігаційні програми для розрахунку маршрутів

PART 02

СУБД типу Ключ-Значення



СУБД типу Ключ-Значення

Відмінною рисою цих СУБД є проста модель даних – **асоціативний масив або словник**, що дозволяє працювати з даними по **ключу**.

- Основна задача подібних сховищ – **максимальна продуктивність**, тому жодна **інформація про структуру** значень **не зберігається**.
- **Легко масштабуються**.
- Цілком **відсутні відношення між сутностями**.
- **Використовується для:**
  - роботи з даними в реальному часі,
  - кешування результатів тривалих операцій,
  - зберігання інформації про сесії користувачів,
  - контроль за кількістю запитів.

Rank			DBMS	Database Model	Score		
Jul 2025	Jun 2025	Jul 2024			Jul 2025	Jun 2025	Jul 2024
1.	1.	1.	Redis	Key-value, Multi-model	149.72	-2.01	-7.05
2.	2.	2.	Amazon DynamoDB	Multi-model	84.15	+0.81	+13.20
3.	3.	3.	Microsoft Azure Cosmos DB	Multi-model	21.68	-0.75	-5.44
4.	4.	4.	Memcached	Key-value	16.37	+0.67	-1.37
5.	5.	5.	etcd	Key-value	6.89	-0.16	-0.10

АТД Словник (асоціативний масив)

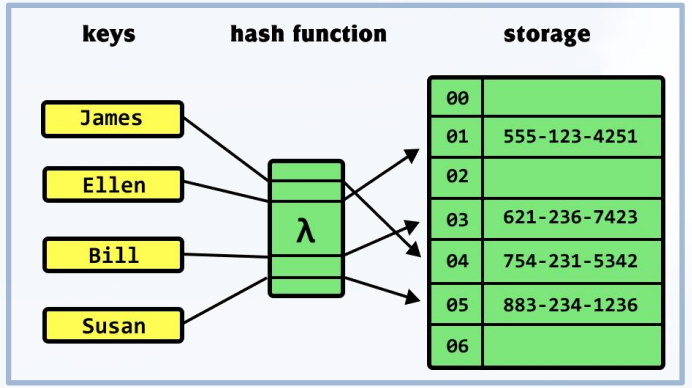
Абстрактний тип даних **Словник** - це **тип даних**, який дозволяє зберігати пари “**Ключ-Значення**”. В цих парах кожному ключу відповідає певне значення, “проасоційоване” з ним.

Цей тип даних передбачає наступні **операції:**

- **додавання пари** (ключ, значення),
- **видалення пари** (ключ, значення) **за заданим ключем**,
- **пошук значення за заданим ключем**.

Для реалізації **Словника** (та Множини, яка є частковим випадком Словника, в якому з пари (ключ, значення) зилишається тільки (ключ)) використовуються наступні структури даних:

- дерева пошуку,
- ***хеш-таблиці***,
- списки з пропусками.



Хеш-таблиця – це **масив ключів** з особливою логікою, що складається з:

- **Обчислення хеш-функції**, яка перетворює ключ пошуку в індекс.
- **Вирішення конфліктів** (колізій), коли два і більше різних ключа перетворюються в один і той же індекс масиву.

В середньому випадку операції пошуку, додавання, видалення в хеш-таблицях мають **алгоритмічну складність O(1)**.

⚠ По своїй суті **СУБД типу “Ключ-Значення” реалізовані** за принципом **хеш-таблиці**, тому операції з даними в них дуже ефективні за часом.

Redis (**RE**mote **DI**ctionary **S**erver) – це **резидентна** система управління базами даних класу NoSQL з відкритим вихідним кодом, що працює зі структурами даних типу «**ключ - значення**» і зберігає дані **в оперативній пам'яті**.

Резидентною називається така СУБД, яка зберігає всі дані **в оперативній пам'яті** (а не на жорсткому диску), що дозволяє робити доступ до даних **максимально швидким** порівняно з іншими базами даних.

Перший випуск - 8 травня 2009 р.
Остання версія - 8.0.3 від 6 липня 2025 р.

Написана на мові C.

Ліцензія:

1) AGPLv3 (GNU Affero General Public License),
2) SSPLv1 (Server Side Public License),
3) RSALv2 (Redis Source Available License v2).

Підтримує більшість мов програмування високого рівня, таких як Python, JavaScript, Java, C/C++, C# та PHP. Ці мови мають бібліотеки, які можуть взаємодіяти з Redis.

Що ж таке Redis?

Redis дозволяє зберігати дані у **високорівневих структурах даних**, таких як рядки, хеш-таблиці, списки, множини.

Іншими словами можна сказати, що Redis — це **сервер структур даних**.

Основна ідея розробників:

Реляційні СУБД **vs** **Redis**

Традиційні операції з рядками бази даних, їх перебирання, сортування, впорядковування

З самого початку для зберігання інформації використання **тих структури даних, які потрібні програмісту**

Спочатку Redis використовували практично так само, як Memcached (в першу чергу, для кешування результатів запитів).

Але, з розвитком Redis, ця СУБД знайшла застосування й у багатьох інших ситуаціях.

Зокрема — у реалізаціях механізму **видавець/передплатник**, завдання **потокової обробки даних**, системи, де потрібно працювати з **чергами повідомлень**.

Redis: Основні характеристики

Підтримка різних типів даних: Redis підтримує складні типи даних, такі як рядки, хеші, списки, множини, відсортовані множини, бітові карти, гіперлогологи та геопросторові індекси.

Атомарні операції: Redis підтримує атомарні операції на складних типах даних, що дозволяє виконувати операції без ризику переривання та порушення цілісності даних.

Транзакції: Redis підтримує транзакції за допомогою команд MULTI, EXEC, DISCARD та WATCH, що дозволяє групувати команди для послідовного виконання.

Публікація та підписка: Redis має вбудовану систему публікації та підписки, яка дозволяє клієнтам підписуватись на канали та отримувати повідомлення, що публікуються в цих каналах.

Персистентність: Хоча Redis є системою, орієнтованою для зберігання даних у пам'яті, він пропонує налаштування для **збереження даних на диск**, що дозволяє відновити стан бази даних після перезавантаження. Redis пропонує різні опції для збереження даних на диск, такі як точкові знімки (snapshots) та журналювання транзакцій за допомогою AOF (Append Only File).

Реплікація: Redis підтримує майстер-слейв реплікацію, дозволяючи створювати копії даних для резервного копіювання або масштабування читання.

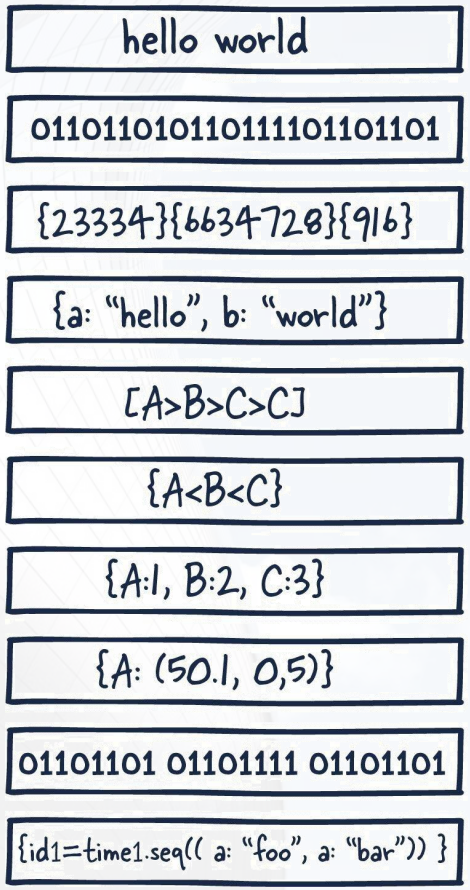
Lua скриптинг: Redis дозволяє виконувати скрипти на Lua для виконання складних операцій на сервері.

Керування пам'яттю: Redis надає різні політики витіснення для керування пам'яттю, коли доступний простір вичерпано.

Кластеризація: Redis підтримує кластери для забезпечення високої доступності та розподілу даних по **кількох вузлах.**

Безпека: Redis пропонує функції безпеки, такі як автентифікація паролем та підтримка SSL для захисту даних.

Redis: Типи даних



- Рядок (String)
- Бітовий масив (Bitmap)
- Бітове поле (Bitfield)
- Хеш-таблиця (Hash)
- Список (List)
- Множина (Set)
- Впорядкована множина (Sorted set)
- Геопросторові дані (Geospatial)
- Структура HyperLogLog (HyperLogLogs)
- Потік (Stream)

Для кожного типу даних в Redis передбачено свій набір операцій і своя логіка роботи з відповідними даними.

Redis: CRUD-операції на рядках

Створення рядків відбувається за допомогою оператора SET:

```
SET 6.1212.nz-123456 “Savchenko Olena Vadymivna” // створення ключа 6.1212.nz-123456 зі значенням
                                                 // “Savchenko Olena Vadymivna”
```

Якщо треба вказати **час життя** змінної, в кінці команди треба додати EX (для секунд) або PX (для мілісекунд):

```
SET 6.1212.nz-123456 “Savchenko Olena Vadymivna” EX 3600
```

За допомогою запиту TTL 6.1212.nz-123456 можна дізнатись час, який залишився до моменту вилучення вказанного ключа з відповідним значенням з оперативної пам’яті (TTL - Time To Live).

Читання даних реалізується за допомогою оператора GET (або його аналогів):

```
GET 6.1212.nz-123456
```

Оновлення даних можна зробити за допомогою команди GETSET, яка встановлює для заданого ключа нове значення і повертає старе значення.

```
GETSET 6.1212.nz-123456 “Kovalevska Olena Vadymivna”
```

Якщо ж треба оновити значення ключа, то можна скористатися командою RENAME:

```
RENAME  6.1212.nz-123456  6.1213.nz-123456
```

Видалення даних по ключу відбувається за допомогою команди DEL:

```
DEL 6.1213.nz-123456
```

Redis: Списки

Списки (Lists) – це впорядковані колекції рядків, які підтримують вставку та вилучення елементів як з початку, так і з кінця. Списки у Redis реалізуються як **двозв'язні списки**.

Вони часто використовуються для реалізації черг, стеків та інших структур даних, де важливим є порядок елементів.

1. **Реалізація черги** (FIFO):

```
LPUSH queue "task1"
LPUSH queue "task2"
RPOP queue            // Поверне "task1"
```

2. **Реалізація стеку** (LIFO):

```
LPUSH stack "task1"
LPUSH stack "task2"
LPOP stack            // Поверне "task2"
LTRIM stack 0 99      // Залишає останні 100 елементів
```

Поради щодо використання:

Атомарність операцій: Операції зі списками Redis атомарні, що робить їх безпечними для використання в **багатопотокових** середовищах.

Сценарії використання: Списки можна використовувати для реалізації черг повідомлень, журналів подій та інших завдань, де важливий порядок елементів.

Redis: Хеш-таблиці, множини, впорядковані множини

1. **Хеш-таблиці** в Redis (Hashes) - це колекції пар "ключ-значення", де ключі можуть бути рядками, а значення - рядками або навіть складними структурами. Це зручний тип даних для зберігання об'єктів із кількома полями:

```
HSET user:1000 name "Alice" email "alica@example.com" age 30   // Створення хеш-таблиці user:1000
HGET user:1000 name    // "Alice"
HGETALL user:1000      // name "Alice" email "alica@example.com" age 30
HDEL user:1000 email
```

2. **Множини** (Sets) – це невпорядковані колекції унікальних елементів:

```
SADD languages "Python" "JavaScript" "Ruby"
SMEMBERS languages                  // "Python" "JavaScript" "Ruby"
SREM languages "Ruby"               // видаляємо вказаний елемент з множини
SISMEMBER languages "Ruby"          // 0 (перевіряємо наявність елемента в множині)
```

3. **Впорядковані множини** (Sorted Sets) - це колекції елементів, кожен з яких має свій "оціночний" бал (score). Елементи сортуються за цим балом, що дозволяє ефективно використовувати їх для завдань із пріоритетами або для створення рейтингів.

```
ZADD leaderboard 100 "Alice" 200 "Bob" 150 "Charlie"
ZRANGE leaderboard 0 -1 WITHSCORES      // "Alice" 100 "Charlie" 150 "Bob" 200
ZINCRBY leaderboard 150 "Alice"         // збільшуємо бал на вказане значення
ZREVRANGE leaderboard 0 -1 WITHSCORES   // "Alice" 250 "Bob" 200 "Charlie" 150
ZREM leaderboard "Bob"                  // видаляємо Боба
ZREMRANGEBYSCORE leaderboard -inf 200   // видаляємо всіх з балами від -∞ до 200
ZRANGE leaderboard 0 -1                 // "Alice"
```

0 і -1 є індексами діапазону елементів, які потрібно витягти: 0 – перший елемент, 1 – другий, -1 – останній, -2 – передостанній.

Redis: Геопросторові дані

Redis підтримує зберігання **географічних координат** (довгота, широта) та виконання запитів, пов'язаних з географією:

```
GEOADD cities 35.1387 47.8393 "Zaporizhzhia"
GEOADD cities 25.9356 48.2925 "Chernivtsi"
GEODIST cities "Zaporizhzhia" "Chernivtsi" km   // поверне 685.5046
```

Остання команда поверне відстань між Запоріжжям та Чернівцями у кілометрах: 685.5046

Команда GEOSEARCH дозволяє знайти всі елементи у вказаному радіусі (або прямокутнику) від зазначеної географічної точки або об’єкта, який вже присутній в базі даних:

```
GEOADD cities 30.5245 50.4504 "Kyiv"
GEOADD cities 24.0297 49.8397 "Lviv"
GEOSEARCH cities FROMMEMBER "Zaporizhzhia" BYRADIUS 500 km DESC  //  поверне Kyiv, Zaporizhzhia
```

GEOSEARCHSTORE працює так само, як GEOSEARCH, але замість того, щоб просто повернути результати, вона зберігає їх у новому Sorted Set, з яким надалі можна працювати як з окремим об’єктом:

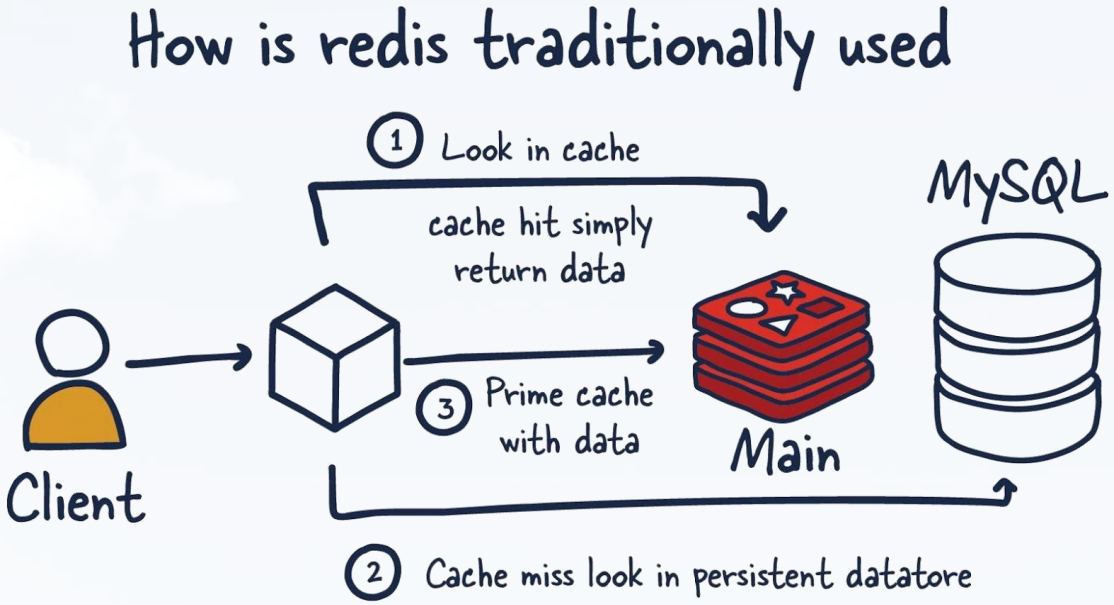
```
GEOSEARCHSTORE nearest_cities_to_zp cities FROMMEMBER "Zaporizhzhia" BYRADIUS 1000 km STOREDIST
ZRANGE nearest_cities_to_zp 0 -1 WITHSCORES  //  поверне Kyiv 443.7595099212779
                                                 Chernivtsi 685.504615881281
                                                 Lviv 842.2380732557223
```

Redis: Кешування

Однією із задач, які вирішує Redis, є ефективне кешування даних. У цьому випадку він часто працює разом з реляційними СУБД, наприклад, MySQL, PostgreSQL. Кешування дозволяє швидко завантажувати частозапитувані дані або результати складних аналітичних запитів.

Redis служить буферною СУБД і, у відповідь на запит користувача, виконує перевірку по ключу, **не чіпаючи основну БД**. Це в рази знижує навантаження на ресурси з високою відвідуваністю (від кількох тисяч користувачів на годину).

Кешовані дані Redis можуть мати **термін життя** (TTL - Time To Live), після чого вони **автоматично видаляються**. Це допомагає контролювати обсяг кешу та уникати застарілих даних.



Стратегії кешування даних

Глобально всі стратегії кешування даних у БД можна розділити на **2 категорії**: при читанні та при записі.

1. Кешування **при читанні** (on read): Визначають, як дані завантажуються в кеш при запиті на читання.

Cache-Aside (Завантаження збоку)

- **Ідея**: Додаток спочатку перевіряє кеш. Якщо даних немає, бере з БД, кладе в кеш, потім повертає.
- **Сценарії**: **Найпоширеніша**. Для даних, що **часто читаються**, але **рідко змінюються** (профілі, статті, каталоги). Додаток контролює кеш.

Read-Through (Наскрізне читання)

- **Ідея**: Додаток завжди запитує дані з кешу. Якщо даних немає, кеш сам бере їх з БД, зберігає, потім повертає додатку.
- **Сценарії**: Коли кеш інтегрований як частина ORM/провайдера. Спрощує логіку додатка.

Стратегії кешування даних (продовження)

2. Кешування **при записі** (on write): Визначають, як дані записуються в кеш та основне сховище.

Write-Through (Наскрізний запис)

- **Ідея:** Дані записуються **одночасно** в кеш і в БД. Запис успішний, коли збережено в обох місцях.
- **Сценарії:** Для даних, що вимагають **високої цілісності та консистентності** (фінансові транзакції). Кеш завжди актуальний.

Write-Behind (Відкладений запис)

- **Ідея**: Дані записуються **спочатку лише в кеш**, підтвердження повертається одразу. У БД записуються асинхронно пізніше.
- **Сценарії**: Для **дуже швидких записів**, де невелика затримка консистентності або ризик втрати даних є прийнятним (лічильники, логи).

Write-Around (Запис навколо кешу)

- **Ідея**: Дані записуються **безпосередньо в БД, оминаючи кеш**. У кеш потрапляють лише при наступному читанні (**через Cache-Aside**).
- **Сценарії**: Для даних, що **записуються, але рідко читаються** (архівні дані, великі об'єкти). Економить місце в кеші.

Стратегії кешування при читанні

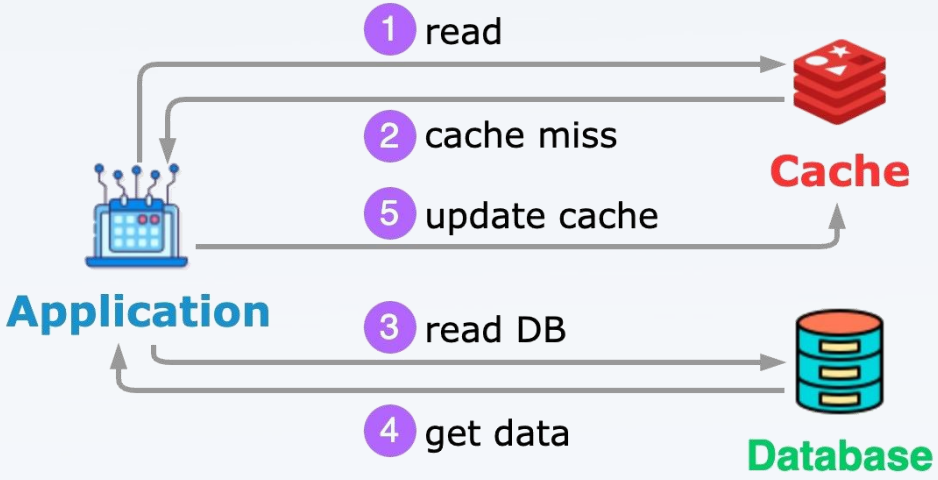
Cache Aside

Pros

1. update logic is on application level, easy to implement
2. cache only contains what the application requests for

Cons

1. each cache miss results in 3 trips
2. data may be stale if DB is updated directly



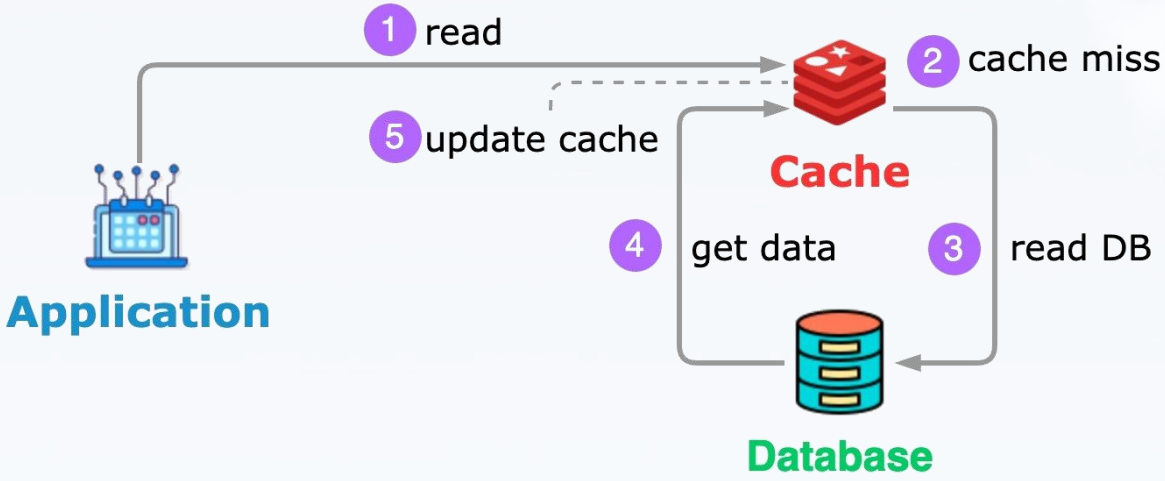
Read Through

Pros

1. application logic is simple
2. can easily scale the reads and only one query hits the DB

Cons

data access logic is in the cache, needs to write a plugin to access DB



Стратегії кешування при записі

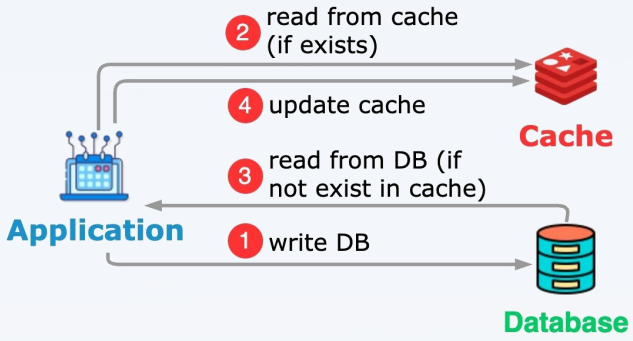
Write Around

Pros

1. the DB is the source of truth
2. lower read latency

Cons

1. higher write latency because data is written to DB first
2. the data in the cache may be stale



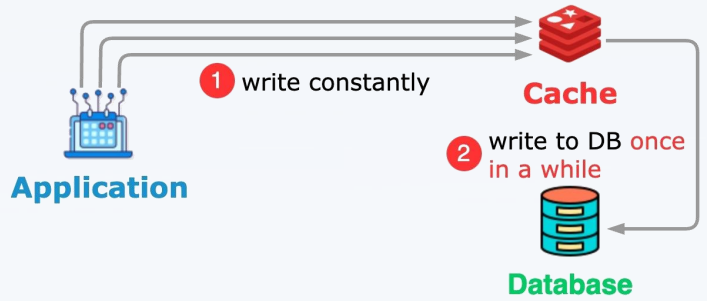
Write Back

Pros

1. lower write latency
2. lower read latency
3. the cache and DB are eventually consistent

Cons

1. there can be data loss if the cache is down
2. infrequent data is also stored in the cache



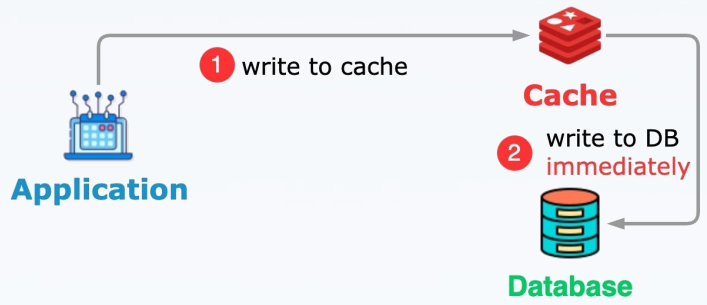
Write Through

Pros

1. reads have lower latency
2. the cache and DB are in sync

Cons

1. writes have higher latency because they need to wait for the DB writes to finish
2. infrequent data is also stored in the cache



Redis: Приклад кешування даних

Приклад реалізації кешування даних при роботі з MongoDB. На практиці частіше зустрічаються випадки використання комбінаці Redis + реляційна СУБД. Але у випадку використання досить великих БД, складних аналітичних запитів або баз даних, які знаходяться в хмарі, комбінація Redis + MongoDB також доволі робочій варіант.

```python
import redis, pymongo, json
from bson.objectid import ObjectId

# Setup Redis and MongoDB connections
redis_client = redis.StrictRedis(host='localhost',port=6379,db=0)
mongo_client = pymongo.MongoClient("mongodb://localhost:27017/")
mongo_db = mongo_client["learn"]
unicorns_col = mongo_db["unicorns"]

# CASHING ON READ
def get_unicorn_by_name(name):
    cache_key = f"unicorn:{name}"
    cached_data = redis_client.get(cache_key)

    if cached_data:
        return json.loads(cached_data)

    # Not in cache, fetch from MongoDB
    unicorn = unicorns_col.find_one({"name": name}, {"_id": 0})
    if unicorn:
        if '_id' in unicorn and isinstance(unicorn['_id'],ObjectId):
            unicorn['_id'] = str(unicorn['_id'])
        redis_client.set(cache_key, json.dumps(unicorn), ex=60)
    return unicorn
```

```python
# CASHING ON WRITE
def add_unicorn(unicorn):
    name = unicorn["name"]
    unicorns_col.insert_one(unicorn) # Insert into MongoDB

    # Immediately update cache
    cache_key = f"unicorn:{name}"
    if '_id' in unicorn and isinstance(unicorn['_id'],ObjectId):
        unicorn['_id'] = str(unicorn['_id'])
    redis_client.set(cache_key, json.dumps(unicorn), ex=60)

if __name__ == "__main__":
    new_unicorn = {
        "name": "Starlight",
        "gender": "f",
        "age": 5,
        "loves": ["adventure", "stars"],
        "weight": 420
    }
    add_unicorn(new_unicorn)
    result = get_unicorn_by_name("Starlight")
    print("Result:", result)
```

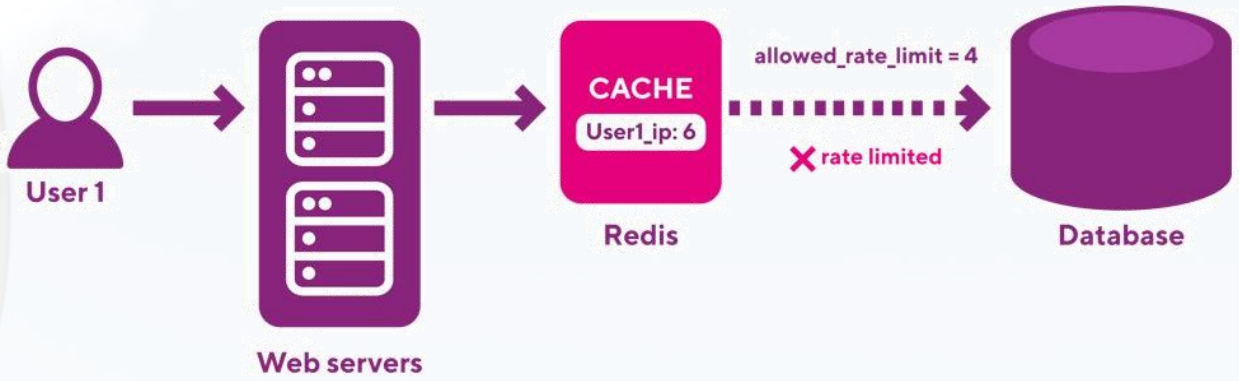
Перетворюємо тип даних ObjectId у звичайний рядок перед збереженням кешу, оскільки JSON не знає, як серіалізувати ObjectId.

Redis: Обмеження кількості запитів

Обмеження кількості запитів (rate limiting) — це техніка, яка використовується для контролю частоти запитів чи дій користувача у певний проміжок часу. Це корисно для захисту від зловживань, таких як спам або DDoS-атаки, а також забезпечення стабільності роботи системи.

Принципи роботи обмеження кількості запитів

1. **Підрахунок запитів за певний проміжок часу**: потрібно відстежувати кількість запитів, які робить користувач або клієнт за заздалегідь заданий проміжок часу (наприклад, за хвилину або за годину).
2. **Обмеження кількості запитів**: якщо користувач перевищив ліміт, система відхиляє запити, поки не закінчиться відведений час.
3. **Використання TTL** (Час життя): Redis підтримує команду для встановлення часу життя ключів (TTL), що допомагає автоматично видаляти інформацію про запити через заданий час.



Redis: Приклад обмеження кількості запитів

```python
import redis
import time

r = redis.StrictRedis(host='localhost', port=6379, db=0)

user_id = "user:1234"
limit = 5           # Maximum number of requests during the window time
window_time = 60    # TIME for tracking requests (in seconds, 1 minute)

key = f"{user_id}:requests_timestamps"    # key for storing user requests
current_time = int(time.time())

r.zremrangebyscore(key, 0, current_time - window_time)   # delete old records
r.zadd(key, {f"{current_time}-{time.time_ns()}": current_time}) # time_ns() for member uniqueness

if r.zcard(key) > limit:    # check the number of requests that remain in the window
    print(f"Rate limit exceeded. Try again later.")
else:
    print(f"Request processed.")

# Set the TTL for the entire key. This will help clear memory if the user no longer makes requests.
r.expire(key, window_time + 5)
```

Redis: Реплікації

Реплікація в Redis — це механізм копіювання даних з одного основного сервера (**Primary**, який раніше називався Master) на один або кілька підлеглих серверів (**Replicas**, раніше Slaves).

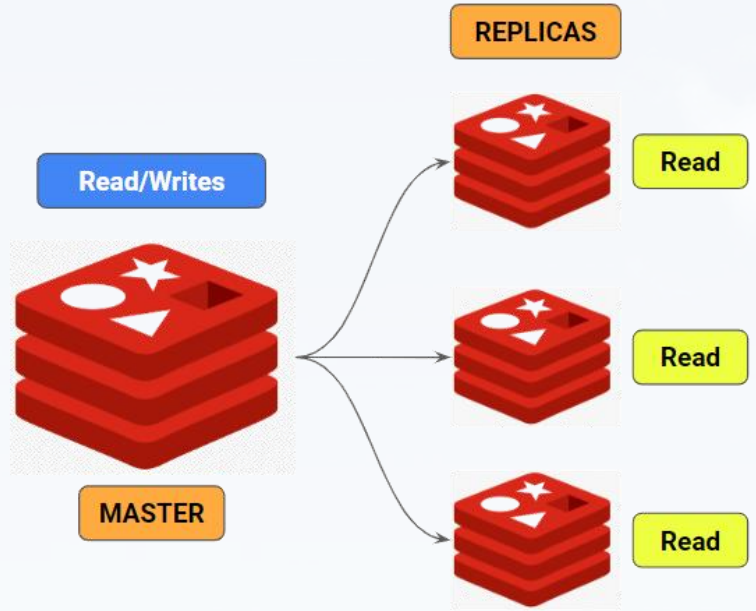
Реплікація підвищує **надійність та масштабованість**, дозволяючи розподіляти навантаження між серверами та надавати резервні копії даних.

Primary-вузол оброблює запити на оновлення даних. Оновлені дані передаються по ланцюжку від Primary-вузла до сусідніх Slave-вузлів, а від них – до інших Replica-вузлів (таким чином досягається децентралізація операції). Зміни завжди прості (з одним ключем), тому передаються **значення**, а не оператори.

Replica-вузли оброблюють записи на читання, розвантажуючи таким чином Primary-вузол. У разі відмови Primary-вузла функцію головного вузла **бере на себе один із Replica-вузлів**.

Реплікація в Redis **асинхронна**, що забезпечує високу продуктивність.

Однак при збої основного сервера можуть бути затримки синхронізації.



Redis: Шардинг

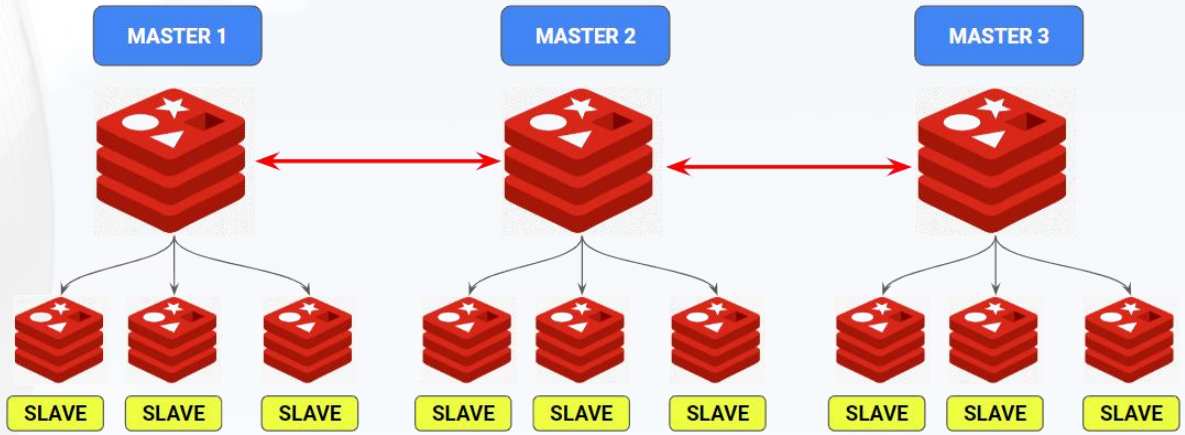
Шардинг (Sharding) – це процес розподілу даних між кількома незалежними інстансами Redis (нодами), кожний з яких зберігає лише **частину даних**.

Переваги шардингу:

- Збільшення загального обсягу доступної пам'яті.
- Підвищення продуктивності за рахунок розподілу навантаження.
- Незалежне масштабування окремих нод.

Реалізації шардингу:

- **Ручний** шардинг: Ви самі вирішуєте на якій ноді зберігати дані.
- **Автоматичний** шардинг із Redis Cluster: Використовується вбудований механізм розподілу ключів між нодами



Redis: Шардинг: приклади

Ручний шардинг:

```
import redis
redis1 = redis.StrictRedis(host='192.168.1.101', port=6379)
redis2 = redis.StrictRedis(host='192.168.1.102', port=6379)
def get_redis_instance(key):
    if hash(key) % 2 == 0:
        return redis1
    else:
        return redis2
redis_instance = get_redis_instance("user:1000")
redis_instance.set("user:1000", "data")
```

Автоматичний шардинг із Redis Cluster:

```
from rediscluster import RedisCluster
startup_nodes = [
    {"host": "192.168.1.101", "port": "6379"},
    {"host": "192.168.1.102", "port": "6379"},
    {"host": "192.168.1.103", "port": "6379"}
]
redis_cluster = RedisCluster(startup_nodes=startup_nodes, decode_responses=True)
redis_cluster.set("user:1000", "John Doe")
print(redis_cluster.get("user:1000"))
```

Redis: Розсилка повідомлень

Redis нативно підтримує 2 різні моделі обміну повідомленнями:

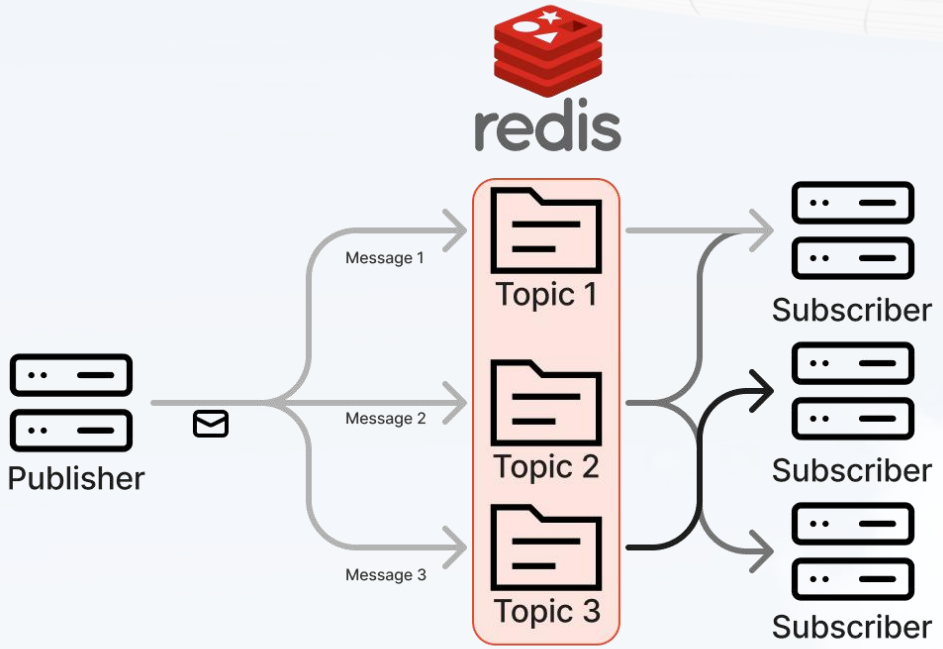
- **Pub/Sub**, працює за принципом **PUSH** (проштовхування):
  - Publisher (видавець) надсилає повідомлення в певний канал,
  - Subscriber (підписник) слухає канал і отримує повідомлення, коли видавець щось надсилає;
- **Streams**, працює за принципом PULL (витягування):
  - Producer (виробник) додає повідомлення в чергу повідомлень,
  - Consumer (споживач) витягує повідомлення з черги повідомлень.

Redis: Підписки (PubSub)

Як працює Pub/Sub в Redis:

- **SUBSCRIBE** — клієнт підписується на канали,
- **PUBLISH** — інший клієнт надсилає повідомлення в певний канал.

Redis автоматично доставляє це повідомлення всім підписникам відповідного каналу.



Redis не зберігає повідомлення — якщо в момент публікації немає підписників, повідомлення просто втрачається.

Наприклад,

```
SUBSCRIBE news                          // підписка на канал новин
PUBLISH news “Hello Subscribers!”       // публікація повідомлення в канал новин, яке миттєво буде доставлено
                                        // всім підписникам
```

Особливості використання:

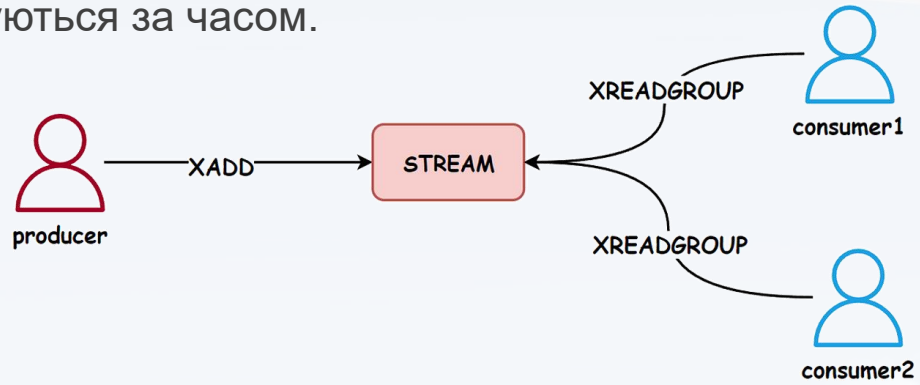
- Redis Pub/Sub **ідеально підходить** для чатів, повідомлень у реальному часі, сигналів між мікросервісами.
- Redis Pub/Sub **не підходить** для ситуацій, де потрібна гарантія доставки або обробка старих повідомлень (для цього краще використовувати Redis Streams або інші брокери повідомлень, наприклад, RabbitMQ або Kafka).

Redis: Streams

Redis Stream являє собою чергу з унікальними повідомленнями, які нумеруються за часом.

Кожне повідомлення в потоці має:

- **унікальний ID** (час + порядковий номер),
- набір полів і значень (**key-value pairs**), як у словнику.



Наприклад,

producer відправляє повідомлення у чергу:

```
XADD mystream * sensor-id 1234 temperature 19.8   // * — Redis сам поставить мітку часу
```

consumer прочитує наявні в черзі повідомлення:

```
XREAD STREAMS mystream 0    // 0 — зчитати все від самого початку
```

Для чого використовують Redis Streams?

- обробка подій у реальному часі (логування, аналітика),
- побудова черг задач,
- масштабовані системи з кількома робочими сервісами,
- збирання даних від пристроїв IoT,
- системи повідомлень з гарантією доставки.

Redis: Streams

Redis Streams дозволяє використання **груп споживачів**.

Наприклад,

створюємо групу споживачів:

```
XGROUP CREATE mystream mygroup 0     // 0 - група почне отримувати лише нові повідомлення,
                                     // що надходитимуть у потік після створення групи
```

читаємо повідомлення з черги повідомлень:

```
XREADGROUP GROUP mygroup Alice STREAMS mystream >     //  > означає, що клієнт хоче отримати тільки
                                                      // нові повідомлення, які ще не були прочитані іншими споживачами
```

після того, як споживач обробив повідомлення, він повинен підтвердити його обробку за допомогою команди XACK:

```
XACK mystream mygroup 12345    //  12345 — це ID повідомлення
```

Тепер це повідомлення буде вважатись відсутнім для всіх членів групи. Щоб повністю видалити повідомлення з потоку, використовується команда:

```
XDEL mystream 12345
```

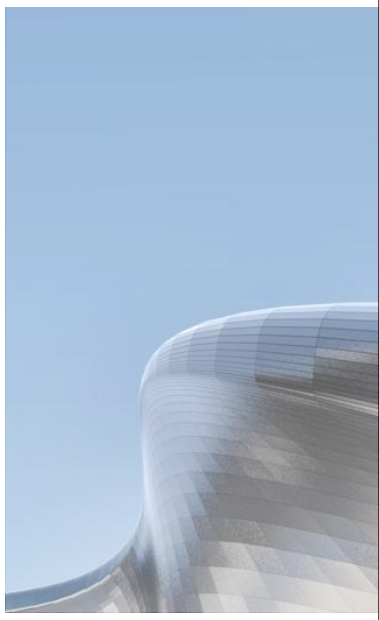
Redis: Підсумки

СУБД Redis призначена головним чином для виконання наступних завдань (але не обмежується ними):

- **Кешування даних**, що дозволяє суттєво знижувати навантаження на СУБД реляційного типу, якщо воно використовується паралельно.
- **Зберігання сесій користувачів**, у тому числі фрагментів сторінок сайтів та ряду інших елементів (наприклад, вмісту кошика інтернет-магазину, тому коли ви повертаєтеся на сайт і бачите, що товари в кошику залишилися, часто це реалізовано за допомогою Redis).
- **Робота з даними у реальному часі:** повідомлення на стіні користувачів у соцмережах, проведення голосування, Створення стрічок новин, групових чатів, блогів тощо.
- **Зберігання даних, до яких потрібно надавати швидкий доступ.** Це аналітична, комерційна та інша важлива інформація. Так, за допомогою Redis нерідко реалізують передачу даних із різних датчиків, які знімають та передають показання промислового обладнання та іншої техніки в режимі реального часу.
- **Обмеження кількості запитів** (rate limiting): контроль частоти запитів чи дій користувача у певний проміжок часу.

PART 03

Графові СУБД



Графові СУБД

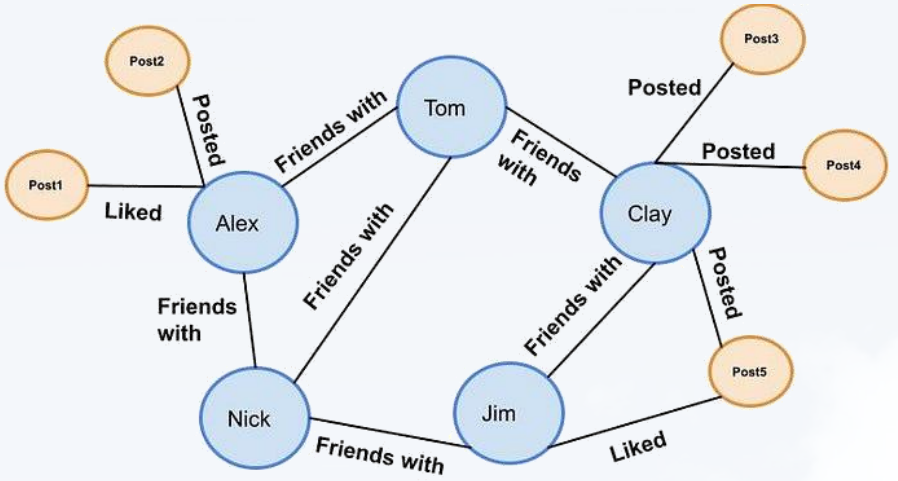
Призначені для зберігання **вузлів графів** та **зв'язків** між ними.

Більшість систем дозволяють задавати для вузлів та зв'язків набір довільних **атрибутів** та вибирати вузли та зв'язки за цими атрибутами.

Підтримують алгоритми **обходу графів** та **побудови маршрутів**.

Ефективно використовуються для завдань, пов'язаних із аналізом **соціальних мереж**, **вибором маршрутів** тощо.

Як правило, є ресурсовимогливими, оскільки для виконання алгоритмів обходу на графах повинні попередньо завантажувати граф (або його частину) в оперативну пам’ять.



Rank			DBMS	Database Model	Score		
Jul 2025	Jun 2025	Jul 2024			Jul 2025	Jun 2025	Jul 2024
1.	1.	1.	Neo4j	Graph	54.64	+3.30	+8.88
2.	2.	2.	Microsoft Azure Cosmos DB	Multi-model	21.68	-0.75	-5.44
3.	3.	3.	Aerospike	Multi-model	5.00	-0.26	-0.16
4.	4.	↑ 5.	ArangoDB	Multi-model	2.83	-0.06	-0.58
5.	5.	↓ 4.	Virtuoso	Multi-model	2.64	-0.07	-1.33

Графові СУБД: Особливості зберігання даних

На відміну від реляційних БД, де зв'язки реалізуються через зовнішні ключі та вимагають дорогих операцій JOIN, **графові БД зберігають зв'язки** безпосередньо як **вказівники** від одного вузла до іншого. Це називається "безіндексною суміжністю" (index-free adjacency).

Коли ви запитуєте зв'язані дані, СУБД не виконує JOIN. Натомість вона безпосередньо **"переходить" за цими вказівниками** від вузла до вузла. Це схоже на читання зв'язаного списку у пам'яті.

Алгоритмічна складність цієї операції складає O(1) для кожного стрибка від вузла до його безпосередніх сусідів.

Продуктивність запитів залежить **від глибини обходу** графа, а **не від загального обсягу** бази даних. Це означає, що запит "друзі друзів друзів" буде виконуватися практично з тією ж швидкістю, незалежно від того, мільйон або мільярд вузлів у базі даних, якщо глибина обходу залишається тією ж.

Особливості зберігання даних робить графові СУБД **надзвичайно швидкими** для виконання запитів, що стосуються зв'язків між даними, де **реляційні БД сповільнюються** через велику кількість **JOIN**-ів.

Різні графові СУБД надають різні мови запитів. Деякі з них є стандартизованими (Cypher, Gremlin, GraphQL, GQL), але багато СУБД їх не підтримують.

Найчастіше для графових СУБД розробляють **власні мови**, часто на основі синтаксису SQL.

Neo4j – **графова система управління базами даних** з відкритим вихідним кодом, реалізована на Java. Станом на 2025 рік вважається найпоширенішою графовою СУБД. Розробник – американська компанія Neo Technology, розробка ведеться з 2003 року.

Дані зберігає у власному форматі, пристосованому для представлення графової інформації.

Для обробки графа не потрібне його завантаження цілком в оперативну пам'ять обчислювального вузла, таким чином, можлива обробка **досить великих графів**.

Надає потужні засоби для виконання запитів, які засновані на **шляхах у графі**, що дозволяє знаходити певні шляхи або закономірності у даних.

Підтримує **ACID-транзакції**.

Реалізовано API для багатьох мов програмування, таких як Java, Python, Clojure, Ruby, PHP, C#, Go.

У СУБД використовується власна мова запитів **Cypher**, але запити можна робити й іншими способами, наприклад, безпосередньо через Java API і мовою Gremlin, створеною в проєкті з відкритим вихідним кодом TinkerPop.

Має зручну систему **візуалізації даних**.

Neo4j: Варіанти використання

Як і будь-яка графова СУБД, Neo4j надає потужні засоби для роботи з даними, організованими у вигляді графа, що обумовлює його широке застосування в різних галузях для аналізу, обробки та вилучення інформації зі складних та взаємопов'язаних даних.

Найбільш типові сфери застосування:

- **Логістика та транспорт**: Маршрутизація, оптимізація транспортних мереж.
- **Соціальні мережі**: Аналіз зв'язків між користувачами, рекомендації друзів та контенту.
- **Біоінформатика**: Аналіз геномів, метаболічних шляхів, взаємодії між білками.
- **Рекомендаційні системи**: Персоналізовані рекомендації товарів, фільмів, музики.
- **Знанієві бази**: Представлення та аналіз семантичних мереж та онтологій.

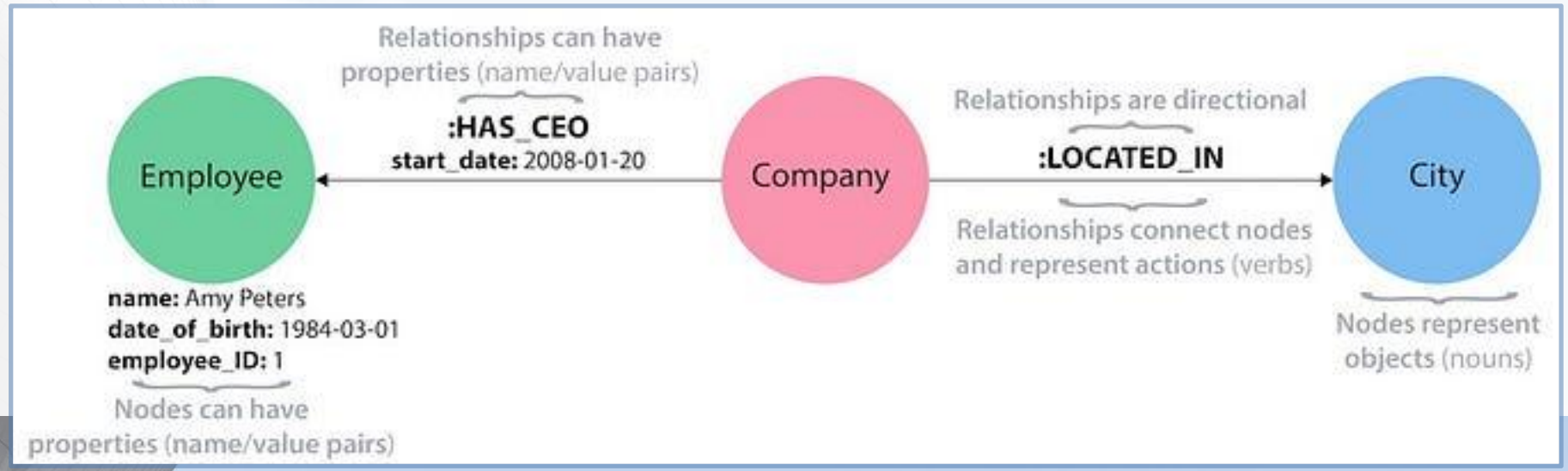
Neo4j: Основні поняття

Вузли (Nodes): є основною сутністю у графі Neo4j. Кожен вузол має унікальний ідентифікатор і може мати властивості, які описують цей вузол.

Зв'язки (Relationships): становлять відносини між вузлами у графі, мають напрямок (від одного вузла до іншого) і також можуть мати властивості, наприклад, вага, відстань, дата вступу на посаду тощо.

Властивості (Properties): представлені парами ключ-значення, які містять інформацію про вузли і зв'язки в графі. Наприклад, вузли можуть мати властивості, такі як ім'я, вік, а зв'язки – властивості, такі як дата створення, вага, відстань.

Мітки (Labels): використовуються для класифікації вузлів і зв’язків у графі. Вони дозволяють групувати вузли і зв’язки за типами і полегшують виконання запитів до графу, що стосуються певних типів вузлів/зв’язків.

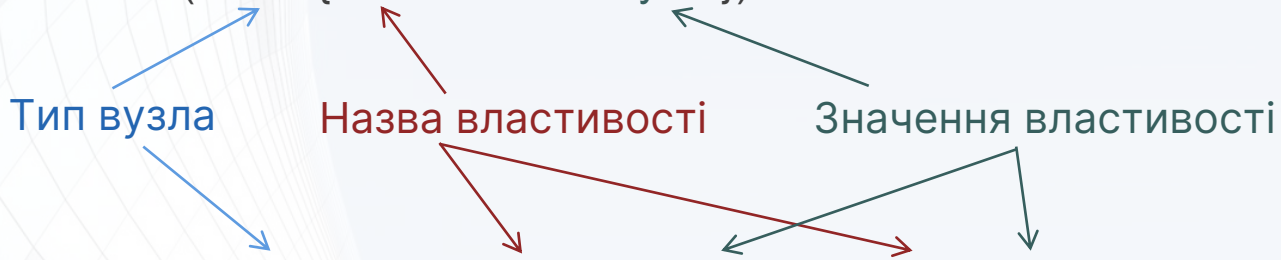


Neo4j: CRUD-операції

1. **Створення** вузлів і зв’язків:

   Ви можете створювати нові вузли та зв’язки у графі за допомогою ключових слів CREATE та MERGE.

   CREATE (:Artist {name: “Show Taylor”})



   CREATE (:Composition {name: “Heavy Soul”, type: “song”})

   MATCH (a:Artist {name: “Shaw Taylor”}), (c:Composition {name: “Heavy Soul”}) **CREATE** (a)-[:PERFORM]->(c)

2. **Читання** даних:

   Виконуйте запити для читання даних із графа за допомогою ключового слова MATCH, після якого необхідно вказати шаблон та критерії пошуку. Наприклад, можна знайти певний вузол чи зв'язок або здійснити пошук за шаблоном.

   MATCH (c:Composition {type: “instrumental”}) **RETURN** c

   Цей запит може бути переписаний у вигляді: MATCH (c:Composition) WHERE c.type= “instrumental” RETURN c

Neo4j: CRUD-операції

3. **Оновлення** даних:

   Оновлюйте існуючі вузли та зв'язки, а також їх властивості за допомогою ключових слів SET та REMOVE.

   MATCH (a:Artist {name: “Show Taylor”}) **SET** a.name= “Joanne Show Taylor”

   За допомогою SET можна також додавати нові властивості до вузлів та зв’язків.

   MATCH (a:Artist {name: “Peter Buka”})-[p:PERFOM]->(c:Composition {name: “Bella Ciao”}) **SET** p.by= “piano”

   За допомогою REMOVE можна видалити існуючу властивість.

   MATCH (a:Artist {name: “Peter Buka”})-[p:PERFOM]->(c:Composition {name: “Bella Ciao”}) **REMOVE** p.by= “piano”

4. **Видалення** даних:

   Видаляйте вузли та зв'язки із графа за допомогою ключового слова DELETE.

   MATCH (c:Composition {name: “Going Home”}) **DETACH DELETE** (c)

Neo4j: Аналіз даних

1. Вибірка слухачів, яким подобаються ті ж самі композиції, що й Іванці:

   MATCH (n {name: 'Ivanka'})-[]-(q)-[]-(u:User) **RETURN** n,q,u

2. Пошук композицій, які також можуть сподобатись Іванці:

   MATCH (n {name: 'Ivanka'})-[]-(q)-[]-(u)-[]-(c:Composition) **RETURN** n,c

   Цей запит може бути переписаний у більш компактній формі, з урахуванням кількості ребер між початковим вузлом і кінцевим:

   MATCH (n {name: 'Ivanka'})-[***3**]-(c:Composition) **RETURN** n,c

   Можна також вказувати діапазон:

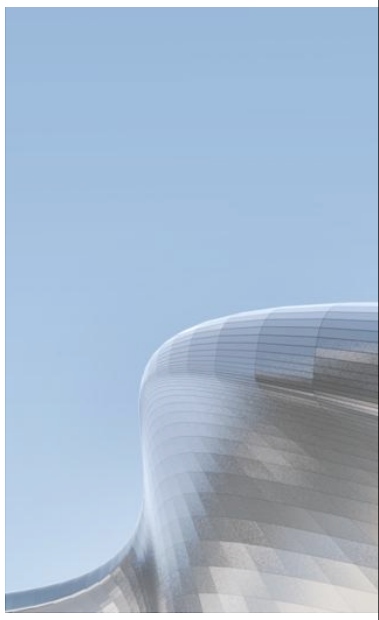
   MATCH (n {name: 'Ivanka'})-[***1..5**]-(c:Composition) **RETURN** n,c

3. Пошук найкоротшого шляху від одного вузла до іншого:

   MATCH p = **shortestPath**((o {name: "Oleg"})-[*]-(i {name:"Ivanka"})) **RETURN** p

PART 04

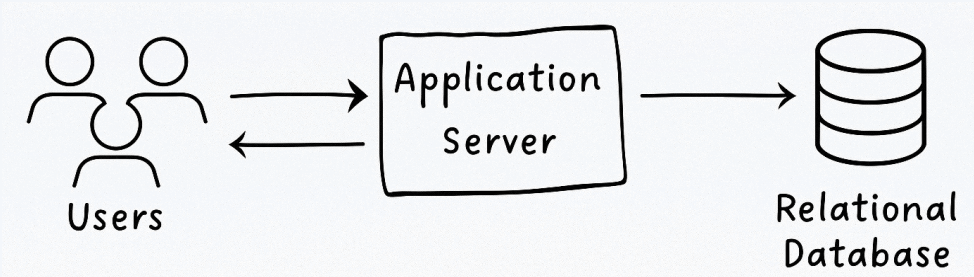
Пошукові СУБД



Пошукові СУБД

Уявимо собі задачу проєктування застосунку, по типу IMDb (Internet Movie Database), який дозволяє здійснювати **пошук** за назвою фільму, за акторами, які в ньому знімались, за режисером/продюсером або, в найбільш складному варіанті, за **описом природньою мовою**.

Припустимо, що ми вирішили використовувати реляційну СУБД.



Тоді пошук за назвою, жанром, актором міг би бути реалізований за допомогою запитів, подібних до наступного:

SELECT * FROM films WHERE title = “The Matrix”

і прблем з виконанням такого запиту не виникло б.

Якщо ж мова піде про пошук за описом фільму, то, теоретично, можна було б скористатись додатковими засобами, які пропонують реляційні СУБД: шаблонами або повнотекстовими індексами. І відповідний запит міг би мати вигляд:

SELECT * FROM films WHERE title LIKE “%serfing%” or description LIKE “%serfing%”

або скористатись повнотекстовим пошуком.

Пошукові СУБД

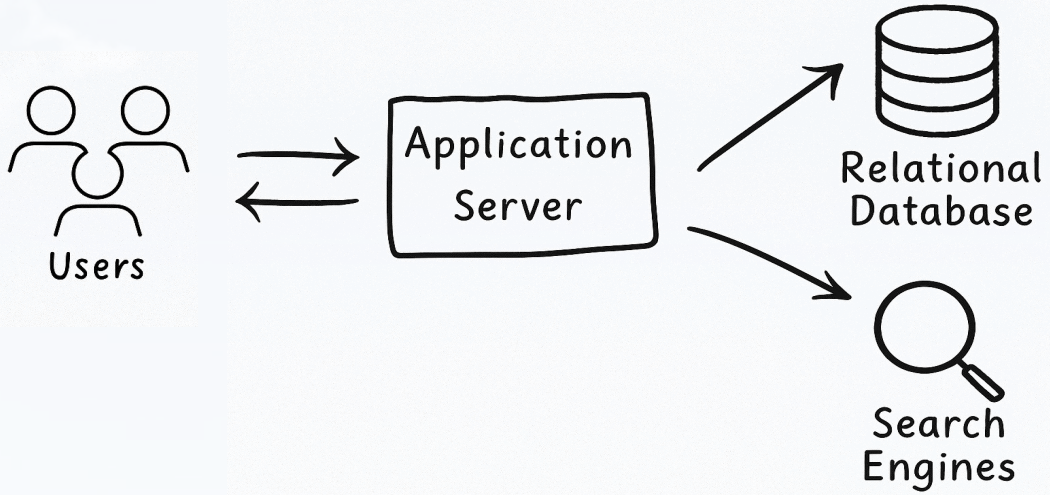
Але зазначений підхід має низку **недоліків**:

- запити можуть бути повільними при великих об’ємах даних;
- реляційні СУБД мають обмежений функціонал щодо повнотекстового пошуку, зокрема, семантичного пошуку (пошуку за змістом, а не співпадінням слів).

Що ж робити?

Використовувати **пошукові СУБД**

Таким чином, ми можемо використовувати **пошукову СУБД** для визначення id фільму/фільмів на основі **повнотекстового пошуку** по заданим критеріям, а в **основній СУБД** зберігати **структуровану** інформацію про ці фільми.



Пошукові СУБД

Оптимізовані для **повнотекстового** пошуку, **аналізу текстів** і фільтрації **великих обсягів неструктурованих** даних.

Ключові особливості:

- Зберігають документи (зазвичай у форматі JSON).
- Працюють на основі **інвертованого індексу** (а не b-дерев).
- Підтримують пошук із врахуванням морфології, нечіткості, автозаповнення.
- Забезпечують **високу швидкість** пошуку навіть у **терабайтах** тексту.
- Часто використовуються разом з іншими СУБД як додатковий пошуковий шар.

Приклади застосування:

- Повнотекстовий пошук у великих текстових масивах (сайти, документи)
- Пошук по логам (DevOps, SIEM-системи)
- Пошук товарів у e-commerce
- Аналітика з текстових даних у реальному часі

Rank			DBMS	Database Model	Score		
Jul 2025	Jun 2025	Jul 2024			Jul 2025	Jun 2025	Jul 2024
1.	1.	1.	Elasticsearch	Multi-model	118.83	-2.45	-12.00
2.	2.	2.	Splunk	Search engine	69.52	-0.10	-23.40
3.	3.	3.	Apache Solr	Search engine, Multi-model	34.17	+0.71	-4.71
4.	4.	4.	OpenSearch	Multi-model	17.30	+0.40	+0.66
5.	5.	↑ 7.	Algolia	Search engine	6.06	+0.29	+0.49

Інвертовані індекси

Пошукові СУБД базуються на **інвертованих** індексах, які є **основним способом зберігання** та обробки текстових даних.

Процес створення інвертованого індексу:

1. Розбиття на слова, фільтрація (видалення того, що не було розпізнано як слово).
2. Видалення стоп-слів (що зустрічаються у всіх документах, тому не можуть бути корисні при пошуку) та нормалізація.
3. Перетворення конвертованих даних в інвертований список слів.
4. Заповнення інвертованого індексу.

Розглянемо, **для прикладу**, поле, в якому зберігається перелік ключових слів наукової публікації і його можливий вміст:

1) MySQL, веб-розробка, мікросервіси, хмарні обчислення.

2) Бази даних, MySQL, індекси, хеш-таблиці, B-дерева.

3) Алгоритми, структури даних, хеш-таблиці, АВЛ-дерева.

Побудуємо для даних наборів ключових слів **інвертований список**, в якому відобразимо входження слів у кожен з 3-х документів.

Слова	Номера документів
MySQL	1, 2
веб-розробка	1
мікросервіси	1
хмарні	1
обчислення	1
Бази	1
даних	2, 3
індекси	2
хеш-таблиці	2, 3
B-дерева	2
Алгоритми	3
структури	3
АВЛ-дерева	3

elasticsearch

Elasticsearch − пошукова СУБД з відкритим вихідним кодом.

Загальна інформація:

- Розробник: компанія **Elastic NV**
- Мова реалізації: Java
- Перший реліз: 2 лютого **2010** р.
- Останній стабільний реліз: **8.16.0** (12 листопада 2024)
- Побудована на базі **Apache Lucene (повнотекстовий індексатор)**, який доповнений підтримкою кластеризації, реплікації, шардування, зручним **RESTfull/JSON API**.
- Ліцензія: Server Side Public License (SSPL) з 2021 р.

Загальна характеристика:

- Мова запитів: **Elasticsearch Query DSL** (JSON-подібна мова запитів)
- Масштабованість: **Розподілена архітектура з шардуванням і автоматичною реплікацією**
- ОС: **Кросплатформна**
- Візуалізація: Працює у зв’язці з **Kibana** (дашборди, графіки, лог-аналітика)

Elasticsearch: Особливості зберігання даних

Одиницею зберігання даних в Elasticsearch виступає **документ** у форматі **JSON**. Наприклад,

```
{
    "id": "12345",
    "title": "The Matrix",
    "description": "A young hacker Neo discovers that the world he lives in is a simulation...",
    "actors": ["Keanu Reeves", "Laurence Fishburne", "Carrie-Anne Moss"],
    "genres": ["Science Fiction", "Action"],
}
```

Документи, що мають схожу структуру чи призначення, об'єднуються у певний логічний простір, що в термінах Elasticsearch називається **індексом**. В реляційних СУБД це відповідає таблиці.

Приклад індексу: movies (для всіх фільмів), products (для всіх товарів), articles (для всіх статей), logs-2025-07 (для логів за липень 2025 року).

У реляційних базах даних існує поняття **бази даних** як верхнього рівня організації даних. У Elasticsearch **немає прямого аналога "бази даних"** в тому ж сенсі.

Elasticsearch	SQL	MongoDB
—	**Database**	**Database**
Index	**Table**	**Collection**
Field	**Column/Attribute**	**Field**
Document (JSON)	**Row/Tuple**	**Document (BSON)**

Elasticsearch: Особливості зберігання даних

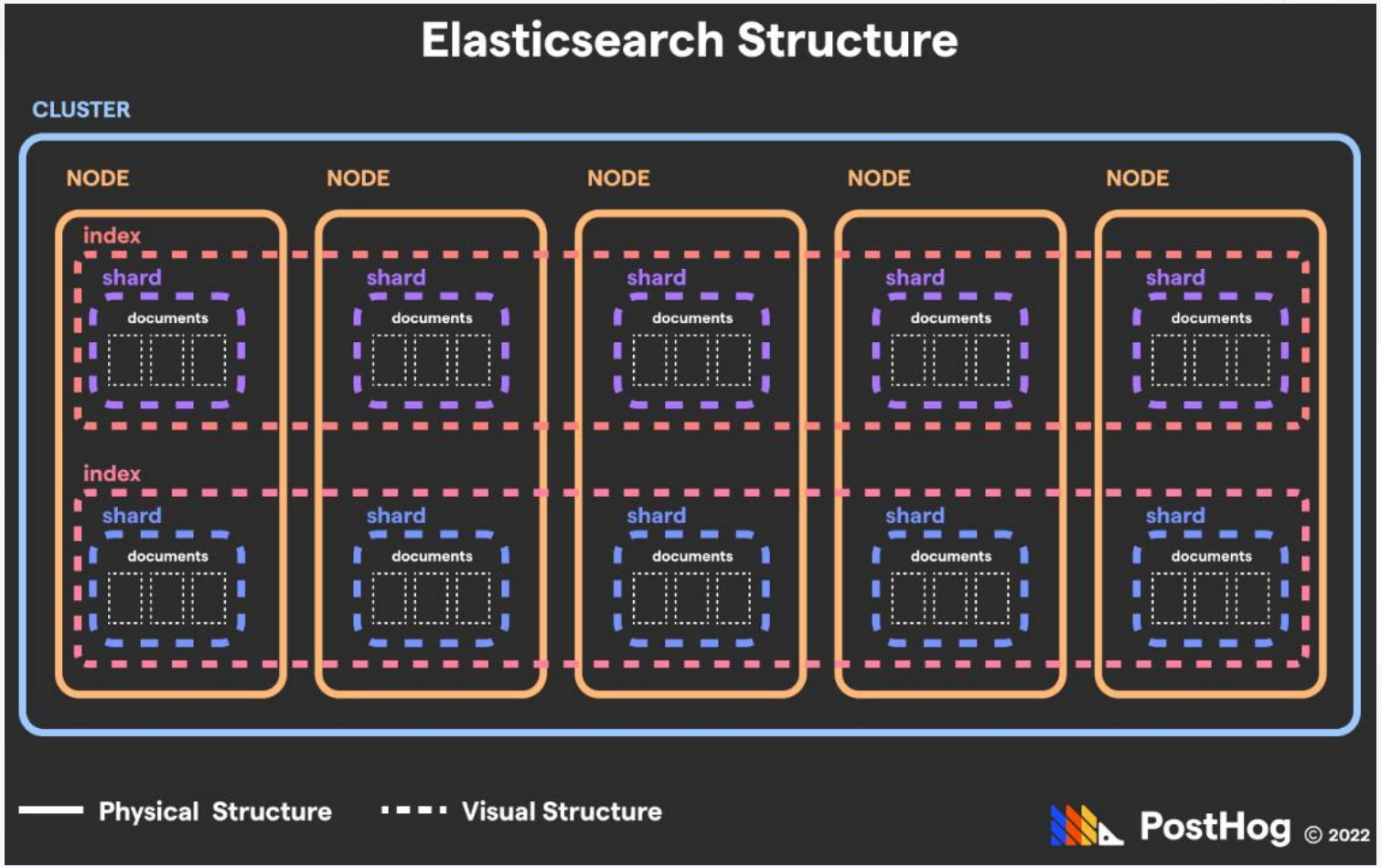
Node (Вузол): **Фізичний** або віртуальний сервер.

Shard (Шард): **Логічна** одиниця зберігання даних, яка є повноцінним ***Lucene Index***. Тобто, кожний шард містить якусь частину індексу (“таблиці”).

Index (Індекс): Множина шардів, які разом утворюють певний логічний простір однотипних документів (аналог таблиці).

Cluster (Кластер)**:** Інфраструктурна одиниця зберігання даних. Може містити декілька індексів (колекцій даних), і декілька нод (фізичних інстансів Elasticsearch).

Набір індексів може виконувати **роль бази даних** для конкретного застосунку.



Elasticsearch: Додавання даних

Процес додавання даних до Elasticsearch зазвичай складається з трьох ключових кроків:

1) створення індексу,
2) визначення його *мапінгу* (структури документів, що будуть в ньому зберігатись),
3) додавання документів.

1. **Створення індексу**.

Індекс – це логічний контейнер для документів, які мають схожу структуру. Всі пошукові запити виконуються до одного або кількох індексів.

Якщо ви намагаєтеся додати документ до **неіснуючого індексу**, Elasticsearch за замовчуванням **автоматично створить** цей індекс з **динамічним мапінгом**. Однак, для продакшн-систем і контролю над тим, як ваші дані індексуються, наполегливо **рекомендується створювати індекс** і **визначати його мапінг** вручну.

***Приклад** створення індексу 'movies_data':*

```
PUT /movies_data
```

2. **Визначення мапінгу** (відповідає етапу визначення схеми відночення (структури таблиці) в реляційних СУБД).

Мапінг визначає, як Elasticsearch повинен **зберігати** та **індексувати** кожне поле у ваших документах. Він вказує **тип даних поля** (наприклад, text, keyword, integer, date, geo_point) і як це поле має аналізуватися для пошуку.

Elasticsearch: Додавання даних

Правильний мапінг критично **важливий** для **продуктивності** пошуку, **точності** результатів та **ефективності** використання дискового простору.

Наприклад, якщо вам потрібен **повнотекстовий пошук** по полю, воно має бути типу **text**. Якщо вам потрібно **точний збіг** або сортування по рядковому полю, він має бути типу **keyword.**

Якщо ви не визначаєте мапінг вручну, Elasticsearch намагається **вгадати** тип поля на основі першого проіндексованого документа. Це зручно для швидкого старту, але може призвести до **неоптимальної поведінки** у довгостроковій перспективі.

***Приклад** визначення мапінгу для бази даних з фільмами*:

```
PUT /movies_data
{
    "mappings": {
        "properties": {
            "movie_id":          { "type": "keyword" },        // Точний ідентифікатор
            "title":             { "type": "text" },           // Повнотекстовий пошук за назвою
            "description":       { "type": "text" },           // Повнотекстовий пошук за описом
            "director_name":     { "type": "keyword" },        // Точна фільтрація за режисером
            "actors_names":      { "type": "text" },           // Повнотекстовий пошук за акторами
            "genres":            { "type": "keyword" },        // Фільтрація/агрегація за жанрами (масив ключових слів)
            "release_year":      { "type": "integer" },        // Числові операції, діапазони
            "rating":            { "type": "float" },          // Числові операції
            "keywords":          { "type": "text" }            // Додаткові теги для повнотекстового пошуку
        }
    }
}
```

Elasticsearch: Додавання даних

3. **Додавання документів** (відповідає додаванню нового рядка в таблицю або нового документу в колекцію).

***Приклад** додавання документу до індексу*

```
POST /movies_data/_doc
{
    "movie_id": "1",
    "title": "The Matrix",
    "description": "Young hacker Neo learns that the world he lives in is a simulation created by machines...",
    "director_name": "Lana Wachowski",
    "actors_names": ["Keanu Reeves", "Laurence Fishburne", "Carrie-Anne Moss"],
    "genres": ["Science Fiction", "Action"],
    "release_year": 1999,
    "rating": 8.7,
    "keywords": ["hacker", "simulation", "virtual reality"]
}
```

В даному прикладі щойнододаному документу СУБД привласнить певний id (ідентифікатор). За допомогою методу PUT можна додати/оновити документ з **конкретним ідентифікатором:**

```
PUT /movies_data/_doc/2
{
    "movie_id": "2",
    "title": "Inception",
    ...
}
```

Elasticsearch: Первинні ключі (id)

При **додаванні** документу через **POST** (наприклад, POST /movies_data/_doc {...}), Elasticsearch **автоматично генерує ID** для нового документа, який грає роль **первинного ключа**.

- **Формат ID**: **рядок довжиною 20 символів** у форматі Base64 URL-safe. Він генерується з використанням алгоритму Snowflake-like (раніше використовувався ULID).
- **Приклад**: ID може виглядати, наприклад, як u_2bYQZzL7gQeS9yT4c8 або Q-W92bYQZzL7gQeS9yT4c8.
- **Тип даних**: Всередині Elasticsearch ID документ завжди зберігається як **рядок**.

При **оновленні** документу через **PUT** із зазначенням **ID** (наприклад, PUT /movies_data/_doc/2):

- якщо вказаний у запиті ID вже **існує**, відбувається **оновлення** документу за цим ID;
- якщо документу із зазначеним ID в індексі **не існує**, документ **створюється** і йому **призначається зазначений ID**;
- При явному вказуванні ID (за допомогою PUT) можна використовувати **будь-який рядок**, який унікальний в межах індексу. Часто використовують цілі числа, тому що вони зручні, але все одно вони зберігаються як рядки.

Elasticsearch: Варіанти мапінгу

Розрізняють **статичний та динамічний мапінг** (mapping).

Під **статичним мапінгом** розуміють той, який задається **до моменту додавання** в індекс першого **документу**.

Якщо ж мапінг не був визначений, **СУБД сама намагається підібрати** оптимальний (з її точки зору) тип даних для кожного поля, що врешті решт буде впливати на спосіб індексування даних, що зберігаються в цьому полі, а також функціонал і швидкість пошуку цих данх. Цей спосіб називається **динамічним мапінгом**.

Існує 3 основних режими динамічного мапінгу:

- **dynamic: true** (за умовчанням)
  - Якщо ви індексуєте документ з полем, якого **немає** в поточному мапінгу, Elasticsearch автоматично додає це поле в індекс. Вона **намагається вгадати** тип даних поля на основі його значень.
- **dynamic: false**
  - Якщо ви індексуєте документ з полем, яке **відсутнє** в поточному мапінгу, Elasticsearch **проігнорує це поле**. Документ буде успішно проіндексований, але нове поле не буде додано до мапінгу і, отже, не буде індексуватися і не буде доступним для пошуку.
- **dynamic: strict**
  - Якщо ви індексуєте документ з полем, яке **відсутнє** в поточному мапінгу, Elasticsearch **відхиляє весь документ** і поверне помилку.

Elasticsearch: Типи даних

1. **Основні типи даних**

text:

- Призначений для **повнотекстового пошуку**.
- Значення цього поля аналізуються (розбиваються на токени, нормалізуються, стеммінгуються) під час індексації.
- Використовує **інвертований індекс**.
- Приклад: Опис фільму, текст статті, коментар.

keyword:

- Призначений для **точних збігів, фільтрації, сортування та агрегації**.
- Значення цього поля не аналізуються і зберігаються "як є".
- Приклад: ID продукту, назва міста, тег, ім'я користувача, жанр фільму.

date:

- Призначений для зберігання дат та часу.
- Підтримує запити за діапазонами та агрегації за часовими інтервалами.
- Приклад: Дата випуску фільму, час створення логу.

boolean:

- Зберігає логічні значення true або false.
- Приклад: Чи є товар у наявності, чи опубліковано статтю.

Elasticsearch: Типи даних

2. **Числові типи даних:** оптимізовані для діапазонних запитів та агрегацій.
   - long, integer, short, byte, double, float, half_float, scaled_float

3. **Складні типи даних:**
   - object, nested

4. **Географічні типи даних:**
   - geo_point, geo_shape

5. **Спеціалізовані типи даних:**
   - ip,
   - completion,
   - token_count,
   - dense_vector,
   - alias

```
PUT /movies_data                    // Приклад мапінгу
{
  "mappings": {
    "properties": {
      "movie_id":      {"type": "keyword"},
      "title":         {"type": "text"},
      "description":   {"type": "text"},
      ...
      "location":      {"type": "geo_point"},
      // Векторне представлення опису фільму для семантичного пошуку
      "embedding":     {"type": "dense_vector", "dims": 768},
      "crew": {                      // Приклад вкладеного об'єкта
        "type": "object",
        "properties": {
          "name": {"type": "text"},
          "role": {"type": "keyword"}
        }
      },
      "reviews": {                   // Приклад вкладеного об'єкта
        "type": "nested",
        "properties": {
          "author": {"type": "keyword"},
          "rating": {"type": "integer"},
          "comment": {"type": "text"},
          "date": {"type": "date"}
        }
      }
    }
  }
}
```

Elasticsearch: Особливості індексування даних

Elasticsearch складає **інвертований індекс** для **кожного поля** документа, яке позначено як **текст** (або аналогічне, призначене для **повнотекстового пошуку**).

Для інших типів даних використовуються **інші способи індексування**, зокрема, **Doc Values** (загальна назва для низки струтур даних, таких як хеш-таблиці, масиви, префіксні дерева тощо) для агрегацій та сортування, **B-Tree** для числових та датових діапазонів, **BKD-дерева** для гео-даних.

Всі поля документу Elasticsearch **індексує автоматично**, **базуючись на типах даних**, які були вказані **під час мапінгу** (визначення структури індексу-колекції). Тому треба досить ретельно обирати типи даних полів документу, оскільки це може суттєво вплинути на подальші характеристики пошукових запитів.

Якщо ж етап мапінгу був **пропущений**, Elasticsearch спробує **автоматично визначити типи полів** документів, доданих до індексу-колекції. Але для кращої продуктивності **рекомендується самостійно** (під час мапінгу) **визначати типи даних** в індексах-колекціях, щоб СУБД почала використовувати саме ті способи індексування, які **вам потрібні**.

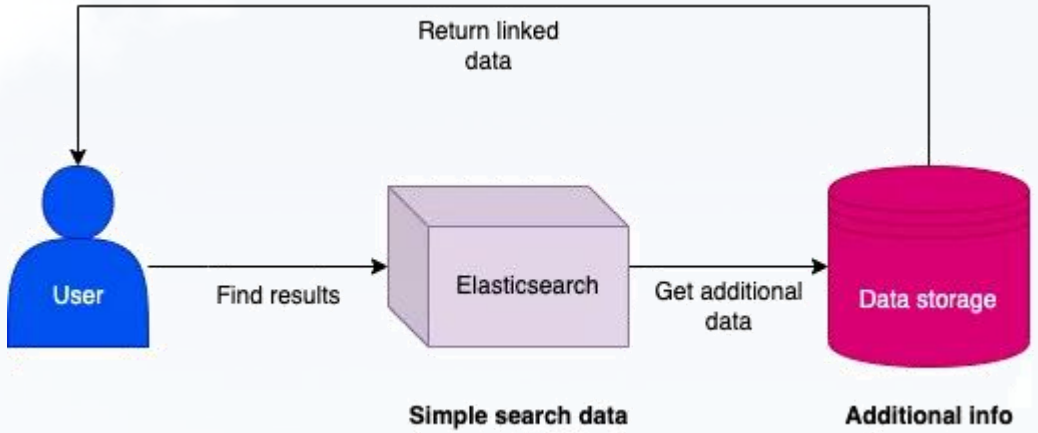
Elasticsearch: Особливості індексування даних

В Elasticsearch **немає поняття кластерного індексу**, як такого, який визначає **спосіб зберігання** даних. Самі дані зберігаються у **вигляді JSON-документу**.

Для підтримки всіх необхідних типів індексів створюються **окремі структури даних** (префіксні дерева, хеш-таблиці, b-дерева тощо). Тому пошукові СУБД зазвичай є **досить ресурсозатратними**, як з точки зору місця на жорсткому диску (для зберігання всіх необхідних для індексування структур даних), так і з точки зори оперативної пам'яті.

⚠ Якщо ви використовуєте Elasticsearch в комплексі з іншою СУБД, яка зберігає самі документи у вигляді структурованих даних, то **немає необхідності** зберігати **повноцінні копії** цих документи ще й в **Elasticsearch**.

Основним призначенням пошукових СУБД є швидкий пошук, тому має прямий сенс завантажувати в них лише ті **частини документів**, для яких цей пошук **треба оптимізовувати.**



Elasticsearch: Пошук даних

Основні принципи мови запитів Elasticsearch:

- **JSON-орієнтованість**: Всі запити надаються у вигляді JSON-об'єктів.
- **RESTful API**: Запити надсилаються на HTTP-ендпоінт /_search (або /<index_name>/_search) методом GET або PUT/POST.
- **Контекст запиту** (Query Context) vs. Контекст фільтра (Filter Context):
  - **Query Context** (query): Використовується для повнотекстового пошуку та умов, які мають впливати на оцінку релевантності (_score). Чим кращий збіг, тим вищий _score.
  - **Filter Context** (filter): Використовується для суворої фільтрації (так/ні), яка не впливає на оцінку релевантності. Фільтри кешуються для підвищення продуктивності.

Elasticsearch: Пошук даних

1. **Повнотекстовий пошук**: призначений для пошуку аналізованих текстових полів.

    1.1. **query match**: аналізує вхідний текст та шукає збіг токенів у вказаному полі.

    Приклад: знайти фільми, в описі яких є слова "computer" або "hacker"

```
GET /movies_data/_search
{
  "query": {
    "match": {
      "description": "computer hacker"
    }
  }
}
```

    1.2. **query match_phrase**: шукає точний збіг фрази.

    Приклад: знайти фільми, в назві яких є точна фраза "The Matrix"

```
GET /movies_data/_search
{
  "query": {
    "match_phrase": {
      "title": "The Matrix"
    }
  }
}
```

Elasticsearch: Пошук даних

1. Повнотекстовий пошук (продовження)

1.3. query multi_match: шукає за кількома полями одночасно.

Приклад: Знайти фільми, де слова "computer" або "hacker" зустрічаються в назві або описі

```
GET /movies_data/_search
{
  "query": {
    "multi_match": {
      "query": "computer hacker",
      "fields": ["title", "description"]
    }
  }
}
```

За замовчуванням Elasticsearch використовує логічний оператор "**OR**" між термінами у вашому запиті.

Якщо ви хочете, щоб були знайдені документи, які містять ВСІ слова з вашого запиту, ви можете явно вказати оператор "**AND**" у запиті match:

```
GET /movies_data/_search
{
  "query": {
    "match": {
      "description": {
        "query": "computer hacker",
        "operator": "and"
      }
    }
  }
}
```

Elasticsearch: Пошук даних

1. Повнотекстовий пошук (продовження)

1.4. Оскільки спеціалізацією Elasticsearch є саме пошук даних, то ця СУБД надає **багато можливостей** застосування спеціальних алгоритмів **рядкового порівняння** та алгоритмів **машинного навчання**.

Наприклад, ми можемо в запиті уточнити, що слова, які ми шукаємо, можуть містити **синтаксичні помилки**. Тоді для пошуку буде застосований алгоритм, який використовує поняття відстані Левенштейна (кількість односимвольних змін в слові, яка потрібна для того, щоб з одного слова отримати інше):

Приклад: Знайти фільми за описом, який містить синтаксичні помилки:

```
GET /movies_data/_search
{
  "query": {
    "match": {
      "description": {
        "query": "komputer haker",  // обидва слова містять синтаксичну помилку
        "fuzziness": "AUTO"
      }
    }
  }
}
```

Elasticsearch: Пошук даних

2. Запити за термінами: для пошуку за неаналізованими полями (keyword, integer, date, ...) для **точних** збігів.

2.1. query term: шукає точний збіг одного терміна (чутливий до регістру і НЕ аналізує текст).

Приклад: Знайти фільми, жанр яких в точності "Science Fiction"

```
GET /movies_data/_search
{
  "query": {
    "term": {
      "genres": "Science Fiction"          // або genres.keyword, якщо поле genres має тип text
    }
  }
}
```

2.2. query terms: шукає збіг з одним із кількох термінів

Приклад: Знайти фільми, жанри яких включають "Action" або "Thriller".

```
GET /movies_data/_search
{
  "query": {
    "terms": {
      "genres": ["Action", "Thriller"]    // або genres.keyword, якщо поле genres має тип text
    }
  }
}
```

Elasticsearch: Пошук даних

2. Запити за термінами (продовження).

2.3. query range: шукає значення в заданому діапазоні.

Приклад: Знайти фільми, випущені між 1990 і 2000 роками (включно).

```
GET /movies_data/_search
{
  "query": {
    "range": {
      "release_year": {
        "gte": 1990,
        "lte": 2000
      }
    }
  }
}
```

Elasticsearch: Пошук даних

3. Комбінування запитів

query bool: спосіб комбінування кількох запитів з використанням логічних операторів:

- **must**: запити, які повинні бути виконані, впливають на _score.
- **filter**: запити, які повинні бути виконані, не впливають на _score і кешуються, ідеально для строгої фільтрації.
- **should**: запити, які можуть бути виконані, впливають на _score, чим більше should збігів, тим вище _score.
- **must_not**: запити, які не повинні бути виконані, не впливають на _score.

Приклад: Знайти науково-фантастичні фільми, випущені після 2000 року, які не є трилерами, і мають рейтинг вище 8.0, а також, можливо, містять слово "future" в описі.

```
GET /movies_index/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {"description": "simulation"}}
      ],
      "filter": [
        {"range": {"release_year": {"gte": 1990, "lte": 2000}}},
        {"term": {"genres.keyword": "Science Fiction"}}
      ],
      "should": [
        {"match": {"actors.name": "Keanu Reeves"}},
        {"match": {"director": "Lana Wachowski"}}
      ],
      "must_not": [
        {"term": {"genres.keyword": "Comedy"}}
      ]
    }
  }
}
```

Elasticsearch: Пошук даних

4. Управління результатами (Pagination & Sorting).

size: кількість повернених документів.

from: зсув (offset) для пагінації.

Приклад: Знайти фільми з назвою "Matrix", починаючи з першого, і повернути 10 документів.

```
GET /movies_data/_search
{
  "query": {
    "match": { "title": "Matrix" }
  },
  "from": 0,
  "size": 10
}
```

sort: сортування результатів, можна сортувати за полями або за _score.

Приклад: Знайти фільми з назвою "Matrix", відсортовані за роком випуску (за спаданням), а потім за релевантністю.

```
GET /movies_data/_search
{
  "query": {
    "match": { "title": "Matrix" }
  },
  "sort": [
    { "release_year": { "order": "desc" } },
    { "_score":      { "order": "desc" } }
  ]
}
```

Elasticsearch: Видалення та оновлення даних

1. **Оновлення документу** (часткове або повне) за ID

```
POST /movies_data/_update/kf7_c5gBWqC7OuUtyKqs
{
  "doc": {
    "description": "Young hacker Neo learns that
                    the world he lives in is a
                    simulation created by machines.
                    He must choose between truth
                    and illusion.",
    "keywords": ["hacker", "simulation", "virtual
                  reality", "red pill", "blue
                  pill"]
  }
}
```

2. **Видалення індексу**

```
DELETE /movies_data
```

3. **Видалення документу:**

- **за ідентифікатором**:

```
DELETE /movies_data/_doc/kf7_c5gBWqC7OuUtyKqs
```

- **за певним критерієм**:

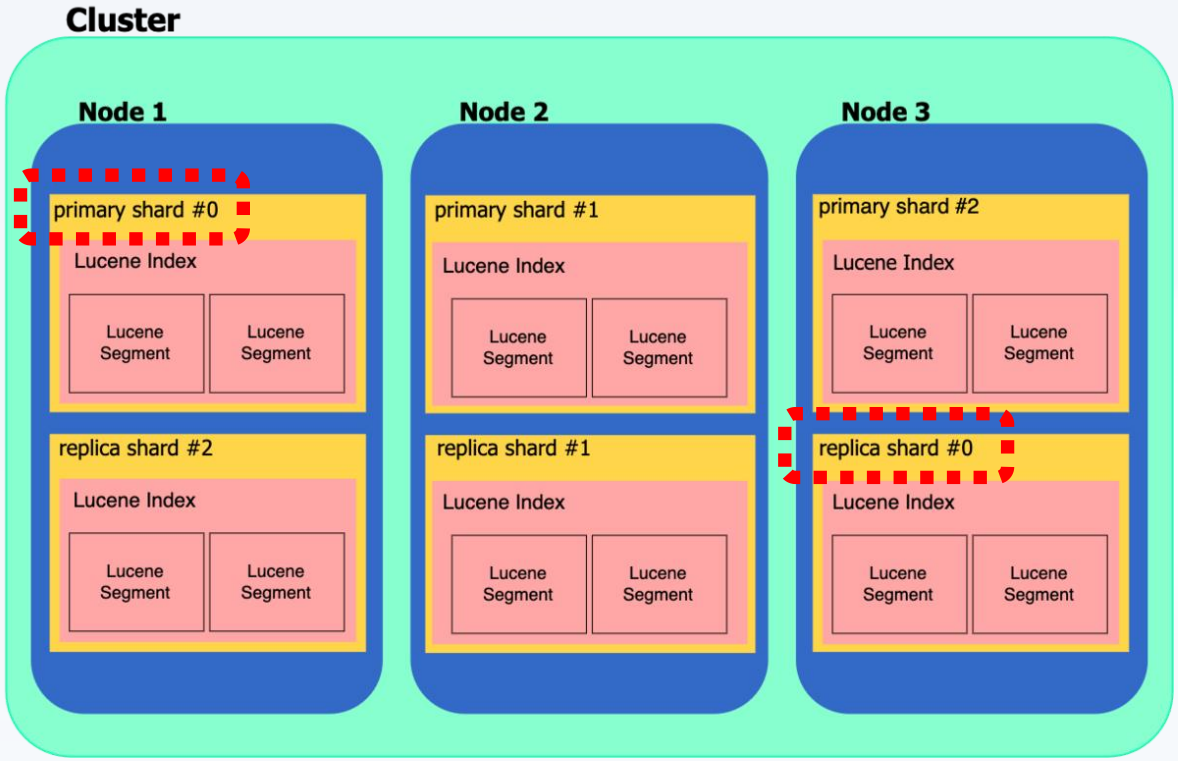
```
POST /movies_data/_delete_by_query
{
  "query": {
    "match": {
      "director": "Christopher Nolan"
    }
  }
}
```

Elasticsearch: Реплікації

Elasticsearch має досить високий рівень відмовостійкості завдяки механізму реплікації.

Репліка − це копія первинного шарду, яка призначена для:

- **Відмовостійкості** (Fault Tolerance): Якщо первинний шард або нода, на якій він розташований, виходить з ладу, репліка може взяти на себе його роль, забезпечуючи безперервну доступність даних.
- **Масштабованість читання**: Пошукові запити можуть виконуватися як на первинних, так і на репліка-шардах. Це дозволяє розподіляти навантаження на читання та збільшувати пропускну здатність кластера для пошукових операцій.



Ви можете налаштувати кількість реплік для кожного індексу. Зазвичай, рекомендується мати **принаймні одну репліку** на кожен первинний шард, щоб забезпечити відмовостійкість.

Elasticsearch: Log Storage

Одним з досить поширених застосувань СУБД Elasticsearch є його використання у якості **сховища для логів** подій в програмних застосунках. В цьому випадку Elasticsearch використовується для збереження логів.

Засоби пошуку, які пропонує Elasticsearch, дозволяють здійснювати різну **аналітику** по збережених логах **з метою**:

- **Діагностики та виправлення помилок**: Логи дозволяють точно визначити, де, коли і чому виникла помилка, оскільки вони містять трасування стека, значення змінних та послідовність подій, що призвели до проблеми.
- **Моніторингу та продуктивності**: Аналізуючи логи, можна відстежувати стан системи в реальному часі, виявляти "вузькі місця" у продуктивності, аномалії або несподівану поведінку до того, як вони стануть критичними проблемами.
- **Безпеки та аудиту**: Логи фіксують дії користувачів та системи, що є незамінним для виявлення несанкціонованих доступів, спроб атак, порушень безпеки або для проведення аудиту, щоб довести відповідність певним стандартам.
- **Аналізу поведінки користувачів та бізнес-аналітики**: Логи можуть містити інформацію про те, як користувачі взаємодіють з вашим додатком. Це дозволяє покращувати користувацький досвід, оптимізувати функції та приймати обґрунтовані бізнес-рішення.

У цьому, а також у багатьох інших випадках, Elasticsearch застосовується разом з таким інструментом як Kibana.

Kibana - це плагін з відкритим кодом для візуалізації даних з Elasticsearch

Загальна концепція журналювання подій

Системи логування в більшості сучасних мов програмування (таких як Python, Java, JavaScript, C#/.NET) мають схожу концептуальну основу.

Ось її ключові риси:

- **Рівні логування**: DEBUG, INFO, WARNING, ERROR, CRITICAL (або FATAL).
- **Логери**: об'єкти, які викликають логування, часто організовані ієрархічно. У великих застосунках може бути по декілька логерів, кожен з яких відповідає за логування подій певної частини застосунку (логер для модуля бази даних, логер для веб-інтерфейсу). У кожного логера повинен бути як мінімум один обробник.
- **Обробники** (Appenders/Handlers): механізми для направлення логів у різні місця (консоль, файл, мережа, база даних).
- **Форматування**: способи налаштування зовнішнього вигляду лог-повідомлень (мітка часу, рівень, ім'я логгера/класу/модуля, саме повідомлення). Форматування задається для кожного з обробників.
- **Конфігурація**: Можливість налаштування логування через конфігураційні файли (XML, YAML, JSON) або програмно.

Відповідно, щоб **налаштувати логер**, зазвичай потрібно:

1) створити логер/логери, вказавши базовий рівень логування;
2) створити один або декілька обробників, вказавши для кожного з них свій рівень логування та своє форматування;
3) прикріпити цей обробник(и) до логеру/логерів.

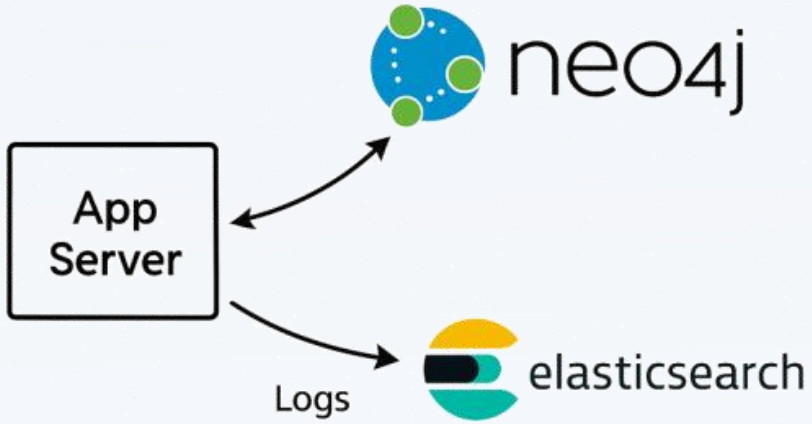
```
root (кореневий логер)
└── my_app (логер для всього додатка)
    ├── my_app.database (логер для роботи з базою даних)
    └── my_app.web (логер для веб-частини)
        └── my_app.web.auth (логер для автентифікації)
```

Приклад використання python-логеру

```
import logging
LOG_FILE_NAME = 'app_activity.log'
# --- 1. Створення та налаштування логгера з ім’ям “my_app_logger” ---
app_logger = logging.getLogger('my_app_logger')
app_logger.setLevel(logging.DEBUG)  # Встановлюємо мінімальний рівень для логгера на DEBUG
# --- 2. Створення та налаштування обробника логів для консолі (StreamHandler)
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.DEBUG)  # Встановлюємо рівень для консолі на DEBUG
# --- 3. Створення та налаштування обробника логів для файлу (FileHandler) ---
file_handler = logging.FileHandler(LOG_FILE_NAME, mode='w', encoding='utf-8')
file_handler.setLevel(logging.INFO)  # Встановлюємо рівень для файлу на INFO
# --- 4. Налаштування форматування (Formatter) ---
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
console_handler.setFormatter(formatter)  # Застосовуємо форматер до обох обробників
file_handler.setFormatter(formatter)
# --- 5. Додавання обробників до логгера ---
app_logger.addHandler(console_handler)
app_logger.addHandler(file_handler)
# --- 6. Демонстрація логування на різних рівнях --
app_logger.debug("This is a DEBUG message. You will see it only in the console.") # Буде виведено тільки в консоль
app_logger.info("Application started successfully.")  # Буде виведено і в консоль і в файл
app_logger.warning("Configuration file not found. Using default settings.")  # Буде виведено і в консоль і в файл
# --- 7. Демонстрація логування виключення (ділення на нуль) ---
try:
    numerator = 10
    denominator = 0
    app_logger.info(f"Attempting to divide {numerator} by {denominator}...")
    result = numerator / denominator
except ZeroDivisionError as e:
    app_logger.error(f"A division by zero error occurred: {e}", exc_info=True)  # exc_info=True додає до логу трасування стека викликів
except Exception as e:
    app_logger.critical(f"An unexpected critical error occurred: {e}", exc_info=True)
```

Elasticsearch: Приклад логування

Приклад використання Elasticsearch у якості сховища логів роботи засосунку, що використовує графову СУБД Neo4j для зберігання даних.




```
import neo4j
import logging
import datetime
import sys
from elasticsearch import Elasticsearch

# --- Конфігурація Neo4j ---
DATABASE_NAME = "neo4j"
URI = "bolt://localhost:7687"
USERNAME = "neo4j"
PASSWORD = "password_neo4j"

# --- 2. Конфігурація Elasticsearch ---
ES_HOSTS = ["http://localhost:9200"]
ES_USERNAME = "elastic"
ES_PASSWORD = "password_elastic"

LOG_INDEX_PREFIX = "neo4j_app_logs-"
```

Elasticsearch: Приклад логування

```
class ElasticsearchHandler(logging.Handler):  # Обробник логів для Elasticsearch.
    def __init__(self, es_client, index_prefix):
        super().__init__()
        self.es_client = es_client
        self.index_prefix = index_prefix

    def emit(self, record):   # Відправляємо запис логу до Elasticsearch.
        try:
            index_name = self.index_prefix + datetime.datetime.now().strftime("%Y.%m.%d")
            log_entry = self.format_record_to_dict(record)
            self.es_client.index(index=index_name, document=log_entry)  # Індексуємо документ
        except Exception as e:
            print(f"ERROR: Failed to send log to Elasticsearch: {e}")

    def format_record_to_dict(self, record):
        message = self.format(record)
        log_dict = {
            "@timestamp": datetime.datetime.fromtimestamp(record.created).isoformat(),
            "level": record.levelname,
            "logger_name": record.name,
            "message": message,
        }
        return log_dict

es_client = None
neo4j_logger = logging.getLogger('neo4j_app_logger')
neo4j_logger.setLevel(logging.INFO)
neo4j_logger.propagate = False # Не передавати логи до кореневого логгера
```

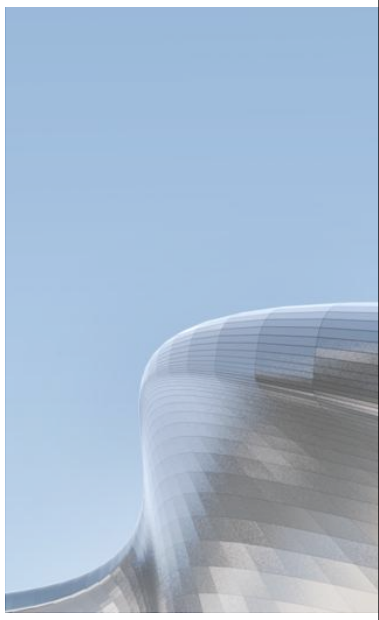
```
try:
    es_client = Elasticsearch(ES_HOSTS, basic_auth=(ES_USERNAME, ES_PASSWORD))
    es_client.info() # Перевірка з'єднання
    es_handler = ElasticsearchHandler(es_client, LOG_INDEX_PREFIX)
    es_handler.setLevel(logging.INFO)
    formatter = logging.Formatter('%(message)s') # Форматуємо тільки повідомлення
    es_handler.setFormatter(formatter)
    neo4j_logger.addHandler(es_handler)
    neo4j_logger.info("Application started. Connecting to Neo4j.")
except Exception as e:
    print(f"CRITICAL ERROR: Failed to connect to Elasticsearch: {e}")

def connect_neo4j():
    driver = None
    try:
        neo4j_logger.info(f"Attempting to connect to Neo4j at {URI}.")
        driver = neo4j.GraphDatabase.driver(URI, auth=(USERNAME, PASSWORD))
        driver.verify_connectivity()   # Явна перевірка підключення до Neo4j
        with driver.session(database=DATABASE_NAME) as session:
            neo4j_logger.info(f"Neo4j connection successfully opened.")
            query = "MATCH (item:Movie) RETURN item"
            neo4j_logger.debug(f"Executing Cypher query: {query}")
            result = session.run(query)
            neo4j_logger.debug("Querying data from Neo4j completed.")
            for record in result:
                print(f"Movie title: {record["item"].get("title", "N/A (Title Missing)")}")
    except Exception as e:
        neo4j_logger.error(f"Error during Neo4j operation: {e}", exc_info=True)
    finally:
        if driver:
            driver.close()
            neo4j_logger.info(f"Neo4j connection closed.")

connect_neo4j()
```

PART 05

СУБД часових рядів



СУБД часових рядів

СУБД часових рядів — це спеціалізовані системи управління базами даних, **оптимізовані для зберігання, обробки та аналізу даних, впорядкованих у часі** (метрики, показники сенсорів, фінансові дані).

Вони забезпечують:

- **ефективне стиснення та зберігання** великих обсягів часових даних;
- **швидкі операції агрегації** та **фільтрації** за часовими інтервалами;
- підтримку **високої швидкості запису** та **читання**;
- інтеграцію з інструментами **візуалізації** (наприклад, Grafana).

Рейтинг на основі технічної та ринкової складової (https://db-engines.com/)

Rank			DBMS	Database Model	Score		
Aug 2025	Jul 2025	Aug 2024			Aug 2025	Jul 2025	Aug 2024
1.	1.	1.	InfluxDB	Time Series, Multi-model	22.60	-0.05	-0.03
2.	2.	↑ 3.	Prometheus	Time Series	7.31	+0.18	+0.14
3.	3.	↓ 2.	Kdb	Multi-model	6.98	-0.12	-0.99
4.	4.	4.	Graphite	Time Series	4.67	+0.26	-0.68
5.	5.	5.	TimescaleDB	Time Series, Multi-model	3.91	+0.14	-0.16

Рейтинг на основі аналізу активності спільноти на GitHub

Monthly Ranking - Stars			
Jul 2025	Jun 2025	Repository	Stars
1	1	ClickHouse/ClickHouse	382 ↑14%
2 ↑7	9	questdb/questdb	381 ↑408%
3 ↓1	2	prometheus/prometheus	329 ↑4.1%
4 ↓1	3	netdata/netdata	263 ↑24.1%
5 ↑1	6	timescale/timescaledb	176 ↑36.4%

СУБД часових рядів

Найпопулярніші (відповідно до db-engines.com) **СУБД часових рядів:**

- **InfluxDB** — високопродуктивна TSDB з власною мовою запитів (Flux/InfluxQL), оптимізована для IoT і DevOps-метрик.
- **Prometheus** — TSDB для моніторингу та алертингу, тісно інтегрована з Kubernetes та екосистемою Cloud-Native.
- **Kdb+** — надшвидка колонкова СУБД для фінансових та біржових часових рядів, з мовою q.
- **Graphite** — проста у використанні TSDB для збору, зберігання та візуалізації метрик у реальному часі.
- **TimescaleDB** — розширення PostgreSQL, що додає ефективну роботу з часовими рядами та **сумісність зі стандартним SQL**.

Що ж обрати?

В деяких випадках оптимальним рішенням буде використання не спеціалізованої NoSQL-СУБД, а перевіреної роками **реляційної СУБД**.

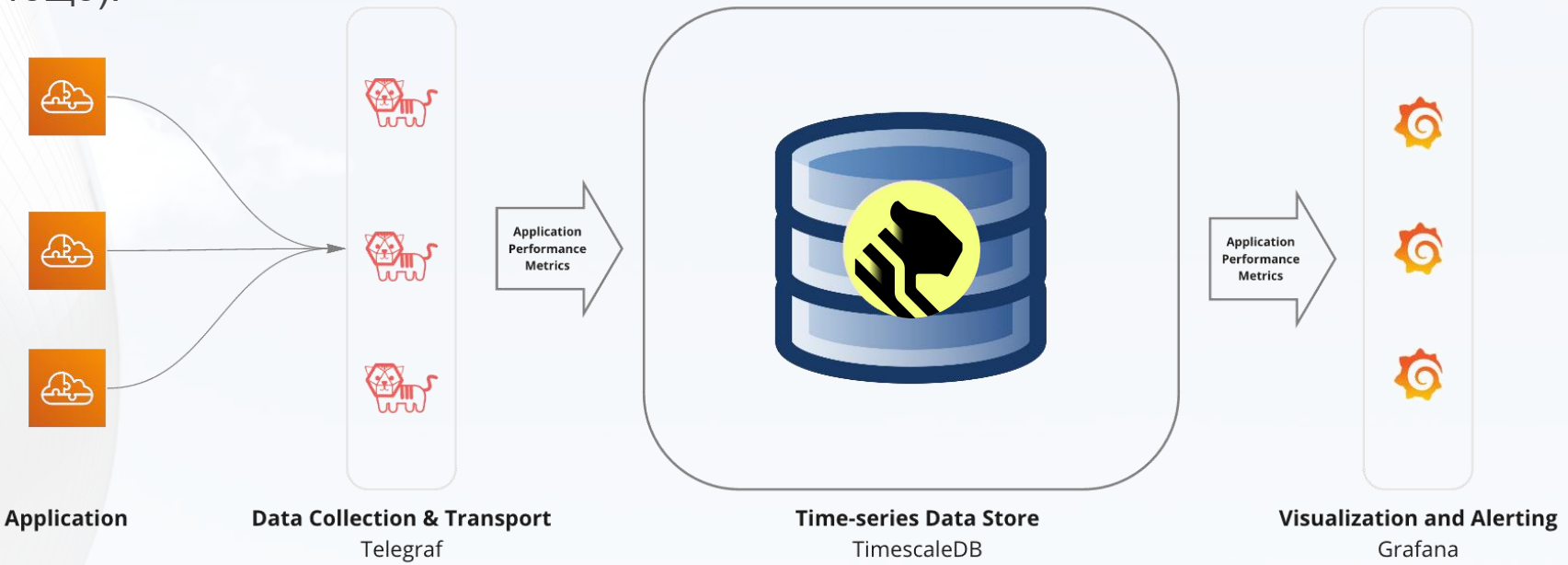
Зокрема, у світі IoT досить добре себе зарекомендувала СУБД **TimescaleDB** (практично не поступається InfluxDB).

Ще одним прикладом такої стратегії (використання спеціалізованих розширень для реляційних СУБД) слугує PostGIS (також розширення PostgreSQL), який став беззаперечним лідером у світі геопросторових даних.

TimescaleDB — це OpenSource-розширення для PostgreSQL, яке оптимізоване для роботи з часовими рядами (time-series) та великими обсягами даних:

- підтримує стандартний **SQL**, інтегрується з екосистемою PostgreSQL, полегшує роботу з часовими даними без втрати гнучкості реляційної БД;
- написана на мові **C,** перший реліз: 2018 р., поточний стабільний реліз: 2.15.0 (травень, 2024 р.);
- легко інтегрується з **Grafana** – потужним засобом візуалізації та аналізу часових рядів, який спеціально був розроблений для моніторингу даних, що змінюються в часі (метрики серверів, дані сенсорів, фінансові показники тощо).

Application perfomance monitoring stack



TimescaleDB: Особливості зберігання даних

Збір даних часових рядів (наприклад, показань датчиків) часто призводить до **ГІГАНТСЬКИХ таблиць**. Звичайний PostgreSQL **сповільнюється**, коли таблиця стає занадто великою.

TimescaleDB для зберігання часових даних використовує **гіпертаблиці**.

- Для користувача гіпертаблиця виглядає і працює як звичайна таблиця PostgreSQL. Ви можете вставляти, оновлювати, видаляти та вибирати дані, використовуючи стандартний SQL.
- Однак **всередині** TimescaleDB автоматично **розбиває** цю логічну таблицю на **безліч менших фізичних таблиць**, які називаються «**чанками**» (chunks). TimescaleDB автоматично створює нові чанки в міру надходження даних.
- Коли ви робите запит на дані за певний період часу, TimescaleDB шукатиме інформацію лише в кількох невеликих чанках **замість сканування всієї бази даних**.

Вбудований в TimescaleDB механізм партиціонування (partitioning) даних за часом, дозволяє виконувати запити **набагато швидше**, оскільки двигун шукає дані лише у відповідних чанках, а не у всій таблиці.

Партиціонування **за часом є обов'язковим**, але можна також партиціонувати дані за **іншими колонками**, наприклад, за ID пристрою, геолокацією тощо.

- Припустимо, що у вас є дані з різних пристроїв, тоді має сенс налаштувати TimescaleDB таким чином, щоб вона створювала чанки як за часом, так і за ID пристрою. І коли потрібні будуть дані по конкретному пристрою чи групі пристроїв, що знаходяться в єдиній геолокації, СУБД достатньо буде здійснювати пошук лише в конкретних чанках.

TimescaleDB: Гіпертаблиці

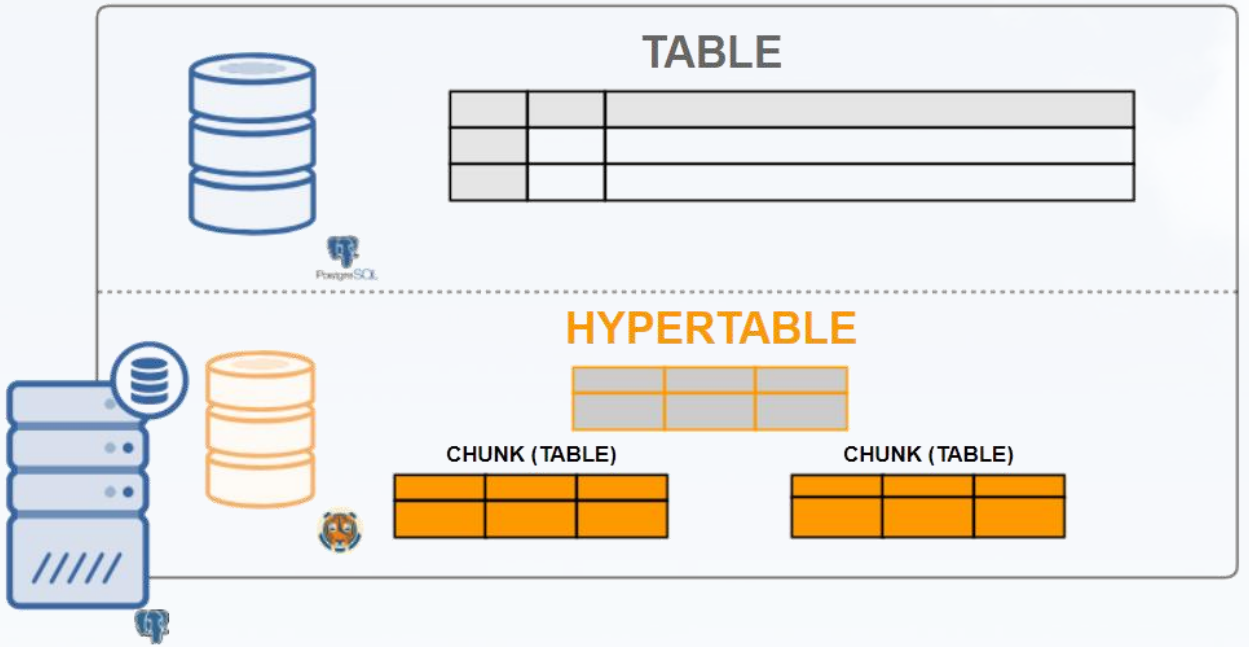
Таблиці в TimescaleDB

Гіпертаблиці:

- ідеально підходять для даних, що постійно надходять (наприклад, показників з датчиків)
- можуть посилатися на звичайні таблиці за допомогою зовнішніх ключів (FOREIGN KEY). Це дозволяє створювати повноцінні реляційні моделі
- наприклад, у вас може бути гіпертаблиця з показниками датчиків, а кожне показання може бути пов'язане з конкретним пристроєм у звичайній таблиці devices.

Звичайні таблиці:

використовуються для зберігання довідкової інформації (типи датчиків, їх місцерозташування тощо)



TimescaleDB: Створення гіпертаблиць

Процес створення гіпертаблиць є двохетапним:

1-й етап: **Створення звичайної таблиці** (CREATE TABLE): Ви створюєте звичайну, стандартну таблицю PostgreSQL. Ви визначаєте колонки, їх типи даних, первинні та зовнішні ключі, індекси тощо.

2-й етап: **Перетворення на гіпертаблицю** (SELECT create_hypertable): Після того, як базова таблиця створена, ви викликаєте функцію create_hypertable(), передаючи їй ім'я таблиці та назву часової колонки.

Чому такий підхід?

- **Сумісність з PostgreSQL:** TimescaleDB є розширенням для PostgreSQL. Такий підхід дозволяє зберігати повну сумісність з інструментами, мовами та ORM, які вже існують для PostgreSQL. Будь-який застосунок, що працює з PostgreSQL, може легко працювати і з TimescaleDB.
- **Гнучкість:** Ви можете спочатку створити складну схему, додати всі необхідні обмеження та індекси, а потім, коли будете готові, перетворити її на гіпертаблицю.
- **Прозорість:** Це чітко розділяє визначення схеми даних (на першому етапі) від логіки розподілу даних (на другому етапі).

TimescaleDB: Приклад створення гіпертаблиць

```sql
-- Створення звичайної таблиці
-- для зберігання інформації про пристрої
CREATE TABLE devices (
    device_id SERIAL PRIMARY KEY,     -- ID пристрою
    device_name TEXT NOT NULL,        -- Назва пристрою
    location TEXT NOT NULL            -- Місцезнаходження
);

-- Створення гіпертаблиці для зберігання показань з датчиків
-- 1-й етап
CREATE TABLE sensor_readings (
    time TIMESTAMPTZ NOT NULL,    -- Часова мітка,
                                  -- обов'язкова для гіпертаблиць
    device_id INTEGER NOT NULL,   -- ID пристрою
    temperature NUMERIC,          -- Температура
    humidity NUMERIC              -- Вологість
    PRIMARY KEY (time, device_id)
);

-- Додаємо зовнішній ключ, щоб пов'язати гіпертаблицю
-- з таблицею `devices`.
ALTER TABLE sensor_readings
ADD CONSTRAINT fk_device
FOREIGN KEY (device_id) REFERENCES devices (device_id);

-- Перетворення звичайної таблиці на гіпертаблицю
-- з партиціонуванням по колонці `time`
-- 2-й етап
SELECT create_hypertable('sensor_readings', 'time');
```

```sql
-- Додавання індексу для прискорення запитів
-- за конкретним пристроєм
-- (індекс буде автоматично перенесено на всі чанки)
CREATE INDEX idx_device_id_time
ON sensor_readings (device_id, time DESC);

-- Вставка даних у звичайну таблицю `devices`
INSERT INTO devices (device_name, location) VALUES
('Термометр-1', 'Київ'),
('Гігрометр-2', 'Львів');

-- Вставка даних у гіпертаблицю `sensor_readings`
INSERT INTO sensor_readings
(time, device_id, temperature, humidity) VALUES
(NOW() - INTERVAL '1 hour', 1, 22.5, 65.2),
(NOW() - INTERVAL '30 minutes', 1, 23.1, 64.8),
(NOW() - INTERVAL '15 minutes', 2, 20.9, 70.1),
(NOW(), 2, 21.3, 71.5);

-- Запит на отримання температури з назвою пристрою
-- та його місцезнаходженням
SELECT d.device_name, d.location, s.time, s.temperature
FROM sensor_readings s
JOIN devices d ON s.device_id = d.device_id
WHERE d.location = 'Київ'
ORDER BY s.time DESC;
```

TimescaleDB: Налаштування гіпертаблиць

Одне з найважливіших питань щодо продуктивності TimescaleDB – налаштування гіпертаблиць.

Зокрема, важливу роль відіграє **розмір чанків**. Хоча для чанків є значення за замовчуванням, яке зазвичай добре працює на початковому етапі, для серйозних проєктів **критично важливо** налаштовувати розмір чанків **вручну**.

На що впливає ***розмір чанка***?

- **Швидкість вставки даних** (Data Ingestion Rate):
  - *Маленькі чанки*: Якщо ви вставляєте дані дуже швидко, і чанки створюються занадто часто, це може створити додаткове навантаження на систему. Кожен новий чанк — це нова таблиця, яка потребує ресурсів для створення;
  - *Великі чанки*: Якщо чанки занадто великі, це може призвести до надмірної фрагментації індексу.
- **Шаблони запитів** (Query Patterns):
  - *Запити на "останні" дані*: Якщо ваші запити найчастіше стосуються даних за останній день або тиждень, варто налаштувати розмір чанка саме на цей інтервал. Це полегшить для СУБД пошук даних.
  - *Запити на великі діапазони*: Якщо ви регулярно запитуєте дані за місяці або роки, чанки можуть бути трохи більшими, але важливо, щоб кожен чанк не був настільки великим, щоб його індекс ставав неефективним.
- **Прибирання старих даних** (Data Retention):
  - Коли ви видаляєте старі дані, TimescaleDB може видалити цілий чанк одразу, якщо він повністю вийшов за межі терміну зберігання. Набагато ефективніше видаляти цілі чанки, ніж мільйони рядків з одного величезного чанка.

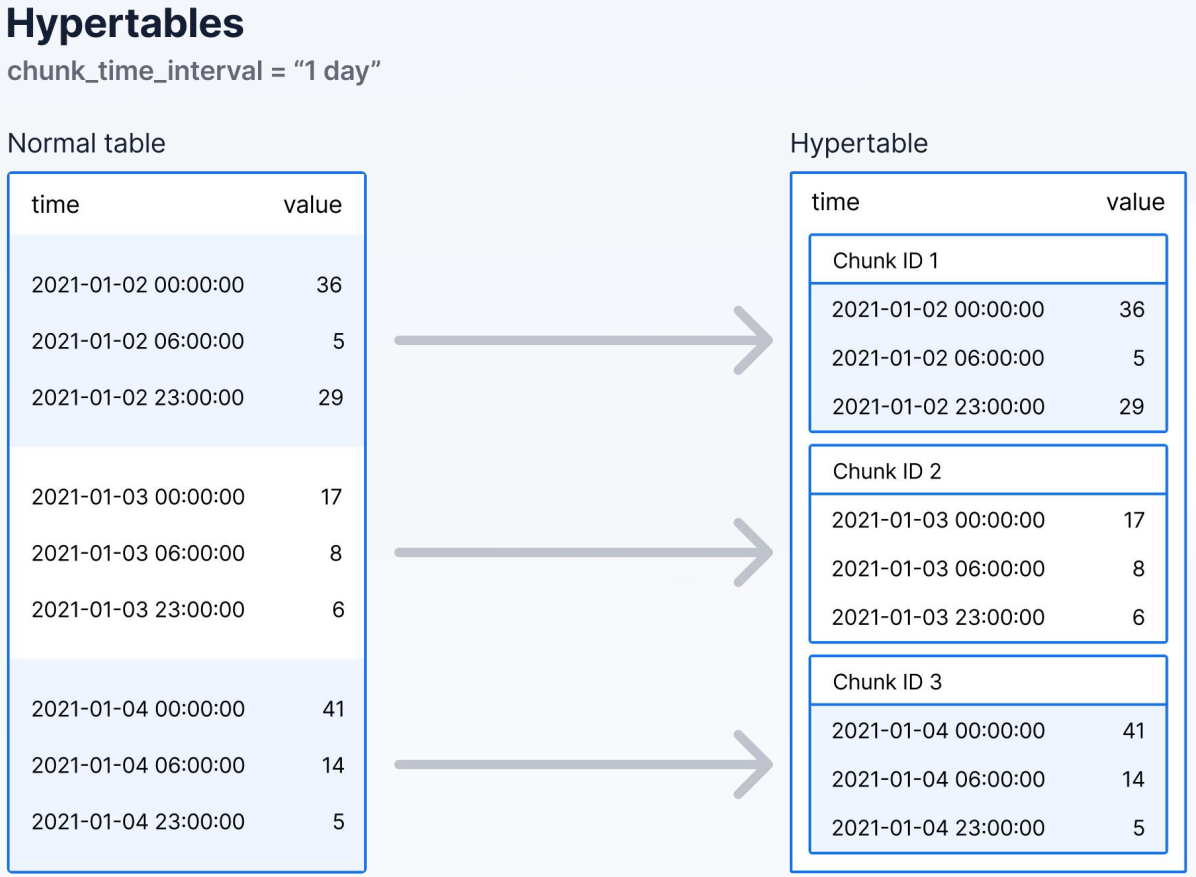
TimescaleDB: Налаштування гіпертаблиць

Розмір чанка задається при створенні гіпертаблиці:

```
SELECT create_hypertable(
    'sensor_readings',
    'time',
    chunk_time_interval => INTERVAL '7 days'
);
```

Зазвичай, гарною відправною точкою є **вибір chunk_time_interval** на основі вашого типового **часового діапазону** запитів.

Якщо ви часто запитуєте дані за останній день, встановіть інтервал в INTERVAL '1 day'. Якщо за останній тиждень, встановіть INTERVAL '7 days'. Це значно покращить продуктивність.



TimescaleDB: Багатовимірні чанки

TimescaleDB дозволяє створювати чанки не тільки за часом, а й за додатковими стовпцями. Це називається **багатовимірним чанкуванням** (multi-dimensional chunking). Для цього в create_hypertable використовується параметр **chunk_on**, де ви вказуєте стовпець, за яким потрібно розділити дані.

Наприклад,

```
SELECT create_hypertable(
    'sensor_readings',
    'time',
    chunk_time_interval => INTERVAL '1 day',
    chunk_on => 'device_id'
);
```

Фізично це працює так:

- TimescaleDB спочатку ділить дані на чанки за часом (наприклад, на кожен день).
- В межах кожного такого "часового" чанку дані додатково діляться на **під-чанки** за значеннями **другого стовпця** (наприклад, device_id).

Це значно покращує продуктивність, оскільки:

- Запити з фільтрацією за device_id та time будуть звертатися лише до невеликої кількості конкретних під-чанків.
- Дані для одного пристрою будуть зберігатися разом, що прискорює операції читання з диска.
- Зменшується навантаження на індекси, оскільки кожен **під-чанк** має свій **власний**, менший за розміром **індекс**.

TimescaleDB: Багатовимірні чанки

TimescaleDB використовує **дворівневу ієрархію чанкування**:

- **Основний рівень (завжди час)**: Спочатку дані діляться на великі чанки за часом (chunk_time_interval).
- **Другий рівень (за додатковими вимірами)**: Всередині кожного часового чанка дані діляться на під-чанки за комбінацією всіх додаткових стовпців, вказаних у chunk_on (яких може бути **декілька**).

Якщо другий рівень представлений комбінацією стовпців, то в параметр chunk_on ви повинні передати масив назв стовпців:

```
SELECT create_hypertable(
    'sensor_readings',
    'time',
    chunk_time_interval => INTERVAL '1 day',
    chunk_on => ARRAY['device_id', 'location']
);
```

Багатовимірне чанкування є потужним інструментом, що дозволяє **оптимізувати базу даних під конкретні шаблони запитів**.

Це ідеальне рішення для сценаріїв Інтернету речей (IoT), де дані надходять з великої кількості окремих пристроїв.

TimescaleDB: Первинні ключі та індексування

Найкращою практикою є поєднання концепцій первинного ключа та чанкування. Так, **первинний ключ** для гіпертаблиці має включати **стовпець часу** та будь-які додаткові **стовпці**, за якими ви **чанкуєте.**

Включення time та device_id в первинний ключ (наприклад, PRIMARY KEY (time, device_id)) не лише забезпечує унікальність, але й дозволяє базі даних створювати індекси, які **ідеально узгоджуються з фізичним розподілом** даних по чанках. Це забезпечує **максимальну ефективність** при виконанні запитів.

Концепція індексування в TimescaleDB загалом залішається незмінною.

Але! Кожен чанк у TimescaleDB є окремою, стандартною PostgreSQL таблицею. Коли ви виконуєте команду CREATE INDEX на гіпертаблиці, TimescaleDB **не створює** один величезний індекс на всю таблицю. Замість цього, він автоматично створює **ідентичний індекс** на **кожній** існуючій **під-таблиці** (чанку), як правило, на основі b-дерев.

Ця модель має як **переваги**, так і **деякі особливості**, які важливо розуміти.

Переваги

Оптимізація запитів: пошук лише в цих конкретних чанках

Зменшення блокування: операції індексування, обслуговування та видалення даних відбуваються на менших, ізольованих чанках

Потенційні недоліки

Об'єм дискового простору: може трохи збільшуватись при великій кількості маленьких чанків

Запити без часового фільтра: Будуть виконуватись повільніше, оскільки СУБД змушена буде сканувати індекси у всіх чанках

TimescaleDB: Неперервні агрегати

Неперервні агрегати — це функція TimescaleDB, яка автоматизує створення та інкрементальне оновлення **матеріалізованих представлень** для часових даних. Це дозволяє вам попередньо обчислювати агреговані дані (наприклад, середні значення, суми, кількості) і зберігати їх у **фізичній таблиці**.

Головна ідея: Замість того, щоб щоразу, коли ви робите запит, сканувати мільярди рядків сирих даних, ви **запитуєте вже готові**, попередньо розраховані результати.

Приклад:

```
-- 1. Створення таблиці
CREATE TABLE sensor_readings (
  time TIMESTAMPTZ NOT NULL,
  device_id TEXT NOT NULL,
  temperature DOUBLE PRECISION
);

-- 2. Перетворення її на гіпертаблицю
SELECT create_hypertable('sensor_readings', 'time');
```

```
-- 3. Створення неперервного агрегату
CREATE MATERIALIZED VIEW hourly_sensor_summary
WITH (timescaledb.continuous) AS
SELECT
  time_bucket('1 hour', time) AS hour,
  device_id,
  AVG(temperature) AS avg_temp,
  MAX(temperature) AS max_temp
FROM sensor_readings
GROUP BY hour, device_id;
```

```
-- 4. Дуже швидкий запит для отримання середньої температури за останні 24 години
SELECT hour, device_id, avg_temp
FROM hourly_sensor_summary
WHERE hour > NOW() - INTERVAL '24 hours'
ORDER BY hour DESC;
```

TimescaleDB: View vs Materialized View vs Continuous Aggregates

Характеристика	Звичайне представлення (CREATE VIEW)	Матеріалізоване представлення (CREATE MATERIALIZED VIEW)	Неперервний агрегат TimescaleDB
Зберігання даних	Ні. Це лише збережений запит.	Так. Це фізична таблиця на диску.	Так. Це фізична таблиця на диску.
Швидкість запитів	**Низька**, оскільки запит виконується щоразу.	**Висока**, оскільки ви запитуєте вже обчислені дані.	**Висока**, оскільки ви запитуєте вже обчислені дані.
Свіжість даних	Завжди актуальна (оскільки запит виконується наживо).	Дані можуть бути застарілими.	Дані майже **завжди актуальні**, завдяки автоматичному, інкрементальному оновленню.
Обслуговування	Не потребує.	Потребує ручного оновлення (REFRESH). Це може бути повільним.	Оновлюється автоматично **у фоновому режимі та дуже швидко** (інкрементально).
Використання	Спрощення складних запитів.	Прискорення запитів на статичних або рідко оновлюваних даних.	Ідеально для часових агрегацій. Прискорення запитів на величезних потоках даних.

TimescaleDB: Політики зберігання даних

Політики зберігання даних (Data Retention Policies) у TimescaleDB — це механізм, який автоматично та ефективно видаляє застарілі дані з гіпертаблиць і, за необхідності, з пов'язаних неперервних агрегатів.

Після створення політики зберігання, перевірка та видалення застарілих даних відбувається у **фоновому режимі**.

Основна мета:

- **Економія дискового простору:** Видаляючи дані, які більше не потрібні, ви звільняєте місце на диску. Це критично для систем, що працюють з великими потоками даних.
- **Оптимізація продуктивності:** Менший обсяг даних означає, що запити виконуються швидше, а процеси обслуговування (наприклад, резервне копіювання) стають більш керованими.

Приклад створення політики, яка буде видаляти чанки з даними, старішими за один рік, з гіпертаблиці sensor_readings:

```
SELECT add_retention_policy('sensor_readings', INTERVAL '1 year');
```

Можна також перевірити наявні політики:

```
-- Перегляд політик для таблиці sensor_readings
SELECT * FROM timescaledb_information.jobs
WHERE proc_name = 'policy_retention'
  AND config ->> 'hypertable_name' = 'sensor_readings';
```

TimescaleDB: Стиснення даних

TimescaleDB пропонує потужні вбудовані механізми **стиснення**, які значно **зменшують обсяг дискового простору**, необхідного для зберігання часових даних, одночасно покращуючи продуктивність запитів.

Основний підхід полягає в тому, що стиснення застосовується до "чанків" (chunks) даних, а не до окремих рядків, що дозволяє використовувати високоефективні алгоритми.

Стиснення в TimescaleDB налаштовується **на рівні гіпертаблиці**. Ви можете увімкнути його, вказавши, які колонки слід стискати, за допомогою команди ALTER TABLE:

```
ALTER TABLE my_hypertable SET (
    timescaledb.compress,
    timescaledb.compress_segmentby = 'device_id',
    timescaledb.compress_orderby = 'time'
);
```

Найбільш зручним способом є **автоматизація стиснення** за допомогою **політик**. Ви можете створити політику, яка буде автоматично стискати чанки, старші за певний проміжок часу:

```
SELECT add_compression_policy('my_hypertable', INTERVAL '30 days');
```

TimescaleDB зберігає стиснуті чанки як **імутабельні** (незмінні). Це означає, що ви **не можете виконати** UPDATE, DELETE або ALTER TABLE ADD COLUMN на стиснутому чанку.

Якщо вам потрібно **змінити дані** в старому чанку, ви повинні спочатку **розпакувати** його.

TimescaleDB: Типові випадки використання

TimescaleDB був створений для ефективної роботи з даними, що надходять у часі, що робить його ідеальним для наступних сфер:

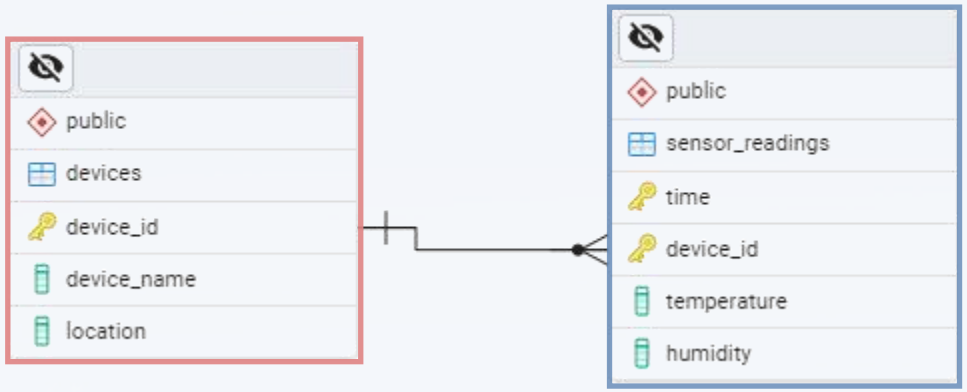
- **IoT (Інтернет речей):** Зберігання та аналіз даних з мільйонів датчиків, пристроїв та сенсорів (наприклад, показники температури, вологості, тиску) для моніторингу, предиктивного обслуговування та аналітики.
- **Системний моніторинг:** Збір метрик та журналів (logs) з серверів, хмарних інфраструктур, баз даних та мережевого обладнання для відстеження продуктивності, діагностики проблем та побудови дашбордів.
- **Фінансові дані:** Аналіз високочастотних біржових котирувань, цін акцій, валютних курсів та торгових транзакцій для побудови фінансових моделей та алгоритмічної торгівлі.
- **Аналітика веб-застосунків:** Відстеження активності користувачів (кліків, переглядів сторінок, взаємодій), аналіз трафіку, і відстеження поведінки в реальному часі для покращення продуктивності сайту та UX.
- **Енергетика:** Збір даних з "розумних" лічильників (smart meters), моніторинг електромереж, аналіз споживання енергії та прогнозування попиту.

Ключові **переваги** в цих сценаріях:

- **Висока швидкість запису** великих обсягів даних.
- **Швидке виконання** складних **аналітичних** запитів.
- **Економія місця** завдяки ефективному стисненню даних.

TimescaleDB: Приклад моделювання

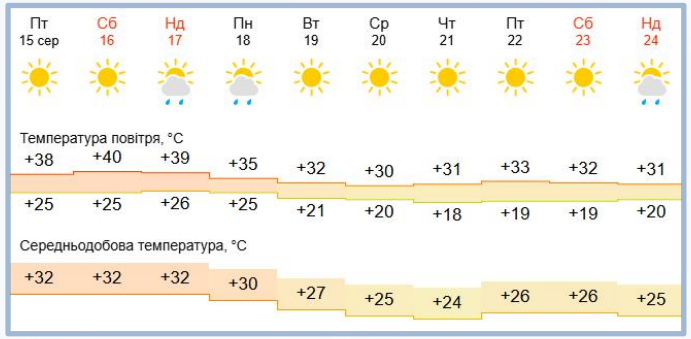
Вдосконалимо приклад, з яким раніше працювали (з температурою, вологістю та інформацією про пристрої). Нижче наведена його схема даних.



Якщо подивитись на типовий прогноз погоди, то на ньому різна температура, зазвичай, буде підсвічена різним кольором. Зручно було б в базі даних зберігати порогові значення для кожного з рівнів температури (норма, вище норми, нижче норми). Зберігати все це в базовій таблиці було б недоречно, це призвело б до надлишковості даних.

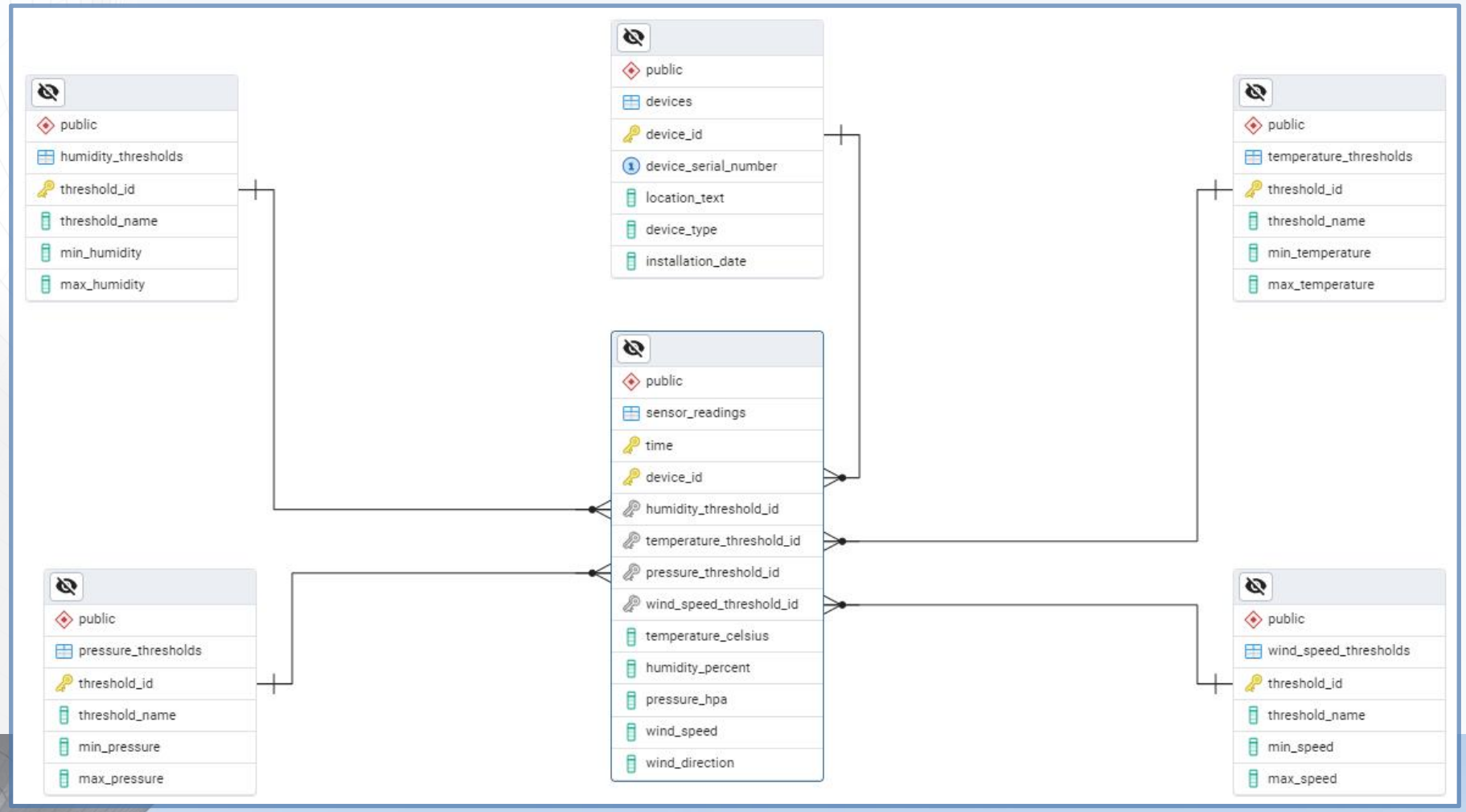
Крім того, при зміні норм треба було б вносити зміни у величезну таблицю, яка зберігає всю історію спостережень. І навіть якщо це лише доба або тиждень, а дані збираються з великої кількості локацій і, наприклад, щохвилинно, то це досить велике навантаження на базу даних.

Набагато зручніше було б зберігати всі ці норми в окремих таблицях.



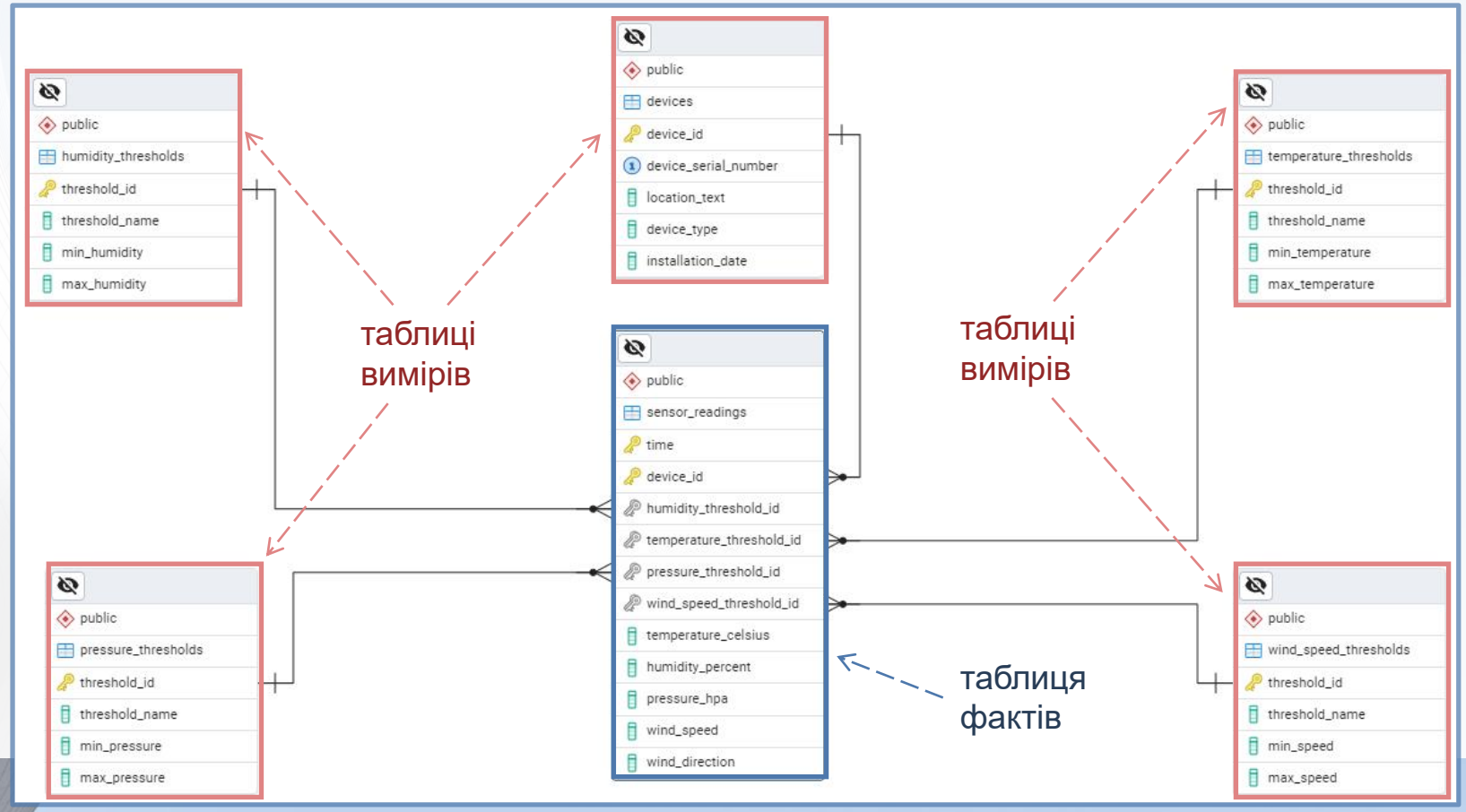
TimescaleDB: Приклад моделювання

Додамо також в базу даних інформацію про атмосферний тиск та швидкість вітру разом з інформацією про градації кожного з вимірювань. Наприклад, для вітру це можуть бути границі для категорій “Штиль”, “Штормове попередження”, “Шквальний вітер” тощо. В результаті отримаємо наступну схему даних.



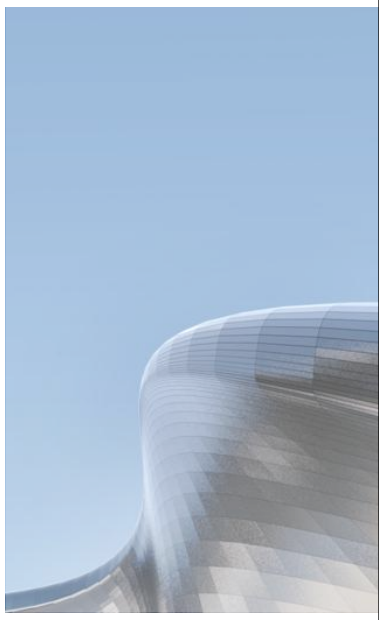
TimescaleDB: Приклад моделювання

Така схема даних є типовою для сфери **Big Data**. Назва такої схеми - **зіркова**, у зв'язку з геометричною схожістю. Центральну таблицю в таких схемах називають **таблицею фактів**. Вона містить конкретні значення показників датчиків, фінансових показників, результатів спостережень тощо. А таблиці, які з нею з'єднані за допомогою зовнішнього ключа, називаються таблицями вимірів. Вони містять довідникову інформацію про кожен з вимірів таблиці фактів.



PART 06

Data Analytics



Data Analytics

Data Analytics — це напрямок знань, який поєднує в собі математику, статистику, програмування та глибокі знання бізнесу. Його головна мета — **трансформувати сирі дані в цінні інсайти** для прийняття обґрунтованих рішень.

Це не просто збір даних, а цілий цикл:

- **Збір і очищення**: Підготовка даних для аналізу.
- **Аналіз**: Виявлення тенденцій, закономірностей та аномалій.
- **Візуалізація**: Перетворення складних даних у зрозумілі графіки та діаграми.
- **Створення звітів**: Надання інсайтів для бізнесу.

Основні **типи** аналітики:

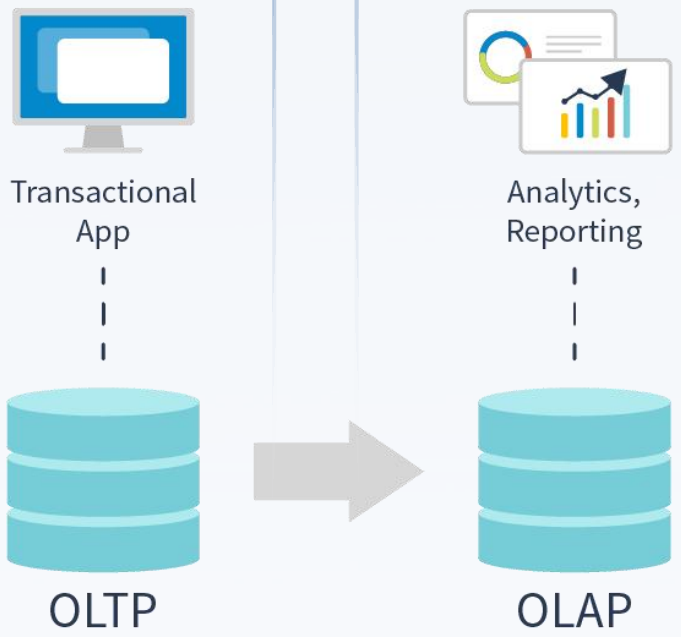
- **Описова** (Descriptive): Відповідає на питання: "Що сталося?" (наприклад, "який відсоток електроенергії зараз країна імпортує?").
- **Діагностична** (Diagnostic): Відповідає на питання: "Чому це сталося?" (наприклад, "чому останніми роками попит на електроенергію збільшився?").
- **Прогностична** (Predictive): Відповідає на питання: "Що може статися?" (наприклад, "який обсяг споживання електроенергії ми очікуємо найближчими роками?").
- **Приписова** (Prescriptive): Відповідає на питання: "Що потрібно зробити?" (наприклад, "які обсяги електричної, атомної, теплової елекроенергії потрібно генерувати найближчими роками, щоб обходитись без імпорту?").

Data Analytics: OLAP / OLTP системи

В системах аналізу даних, зазвичай, можна виділити 2 окремі підсистеми, які відповідають за різні етапи роботи з даними: **OLTP** та **OLAP**.

OLTP (**Online Transactional Processing**)-системи відповідають за етап **збору** даних:

- Їх основне завдання — **обробляти велику кількість транзакцій** у реальному часі. Це, по суті, системи, які генерують сирі дані про повсякденну діяльність.
- Наприклад, коли ви робите онлайн-замовлення або реєструєтеся на сайті, саме OLTP-система фіксує цю транзакцію, що є початком циклу даних.



OLAP (**Online Analytical Processing**)-системи належать до етапів **аналізу, візуалізації та створення звітів**.

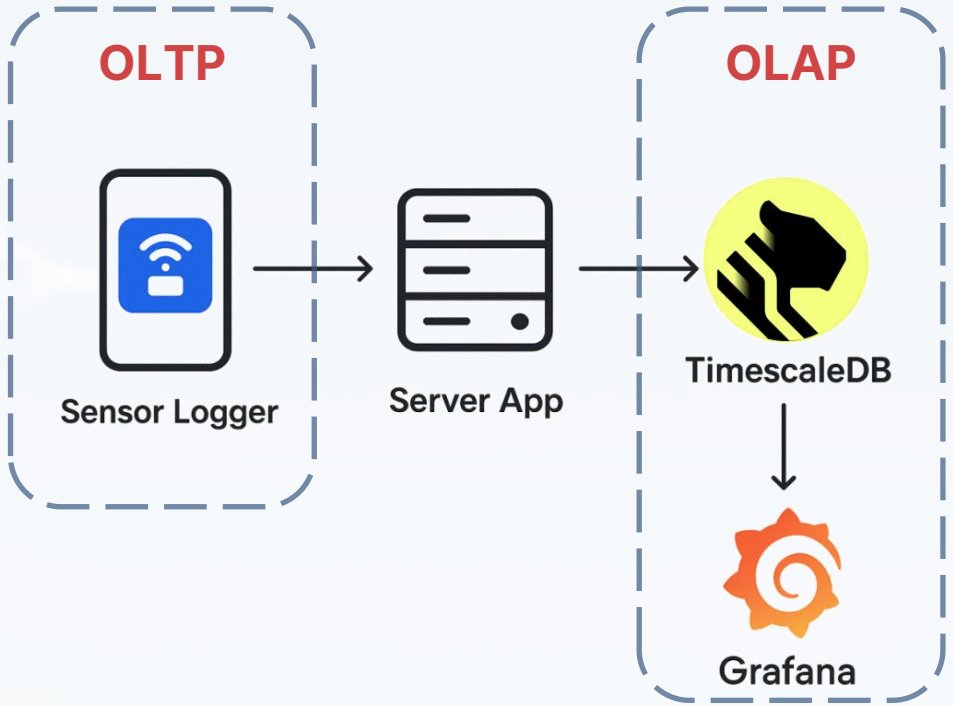
- Вони не створюють дані, а використовують історичні дані, які були зібрані (часто з OLTP-систем).
- OLAP-системи оптимізовані для виконання складних запитів, які дозволяють аналізувати великі обсяги даних, виявляти тренди та закономірності.
- На основі цих даних аналітики створюють звіти та візуалізації, які допомагають ухвалювати стратегічні рішення.

Data Analytics: OLAP / OLTP системи

Наприклад, розглянемо систему, яка збирає дані з датчиків мобільного телефону та передає їх для зберігання у базу даних TimescaleDB, а для аналізу даних використовує Grafana.

В такій системі **мобільний додаток**, який збирає та передає дані з датчиків (в даному випадку Sensor Logger), уособлює в собі **OLTP** систему, тобто відповідає за збір даних.

Зв'язка **TimescaleDB + Grafana** відповідають за аналіз, візуалізацію та створення звітів, тобто представляють собою приклад **OLAP** системи.

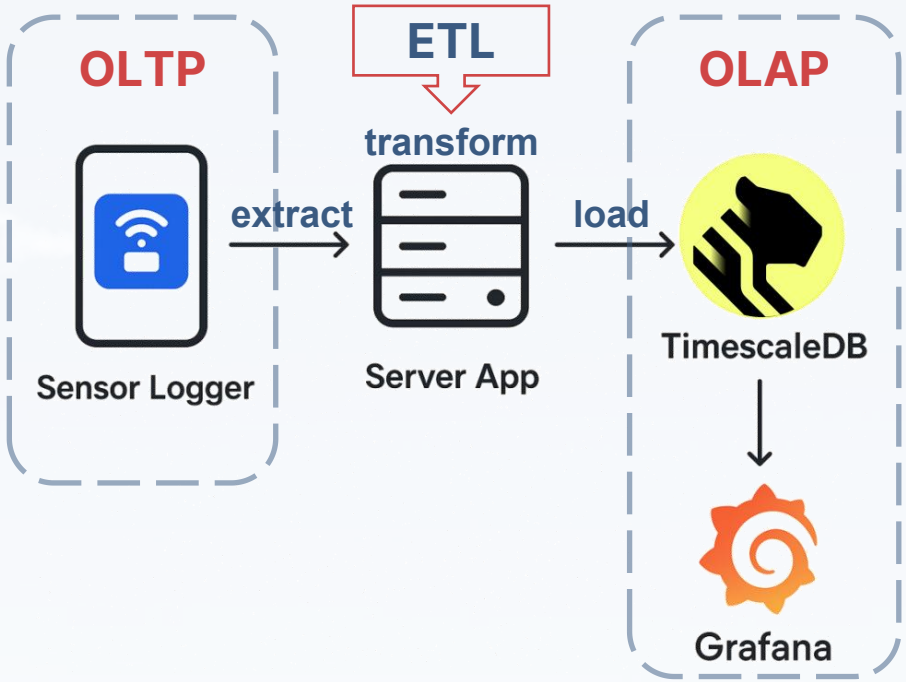


Data Analytics: ETL / ELT процеси

У світі Data Analytics серед процесів обробки даних виділяють **ETL** та **ELT** процеси.

ETL (**Extract, Transform, Load**) — це класичний процес в аналітиці даних, що складається з трьох ключових етапів. Його мета — інтегрувати дані з різних джерел, очистити їх, перетворити в єдиний формат і завантажити в центральне сховище для подальшого аналізу.

В системі аналізу даних з датчиків мобільного телефону роль ETL процесу відіграє програмний застосунок, який **трансформує** дані, прийняті від Sensor Logger, в форму, яка відповідає структурі гіпертаблиць в TimescaleDB.

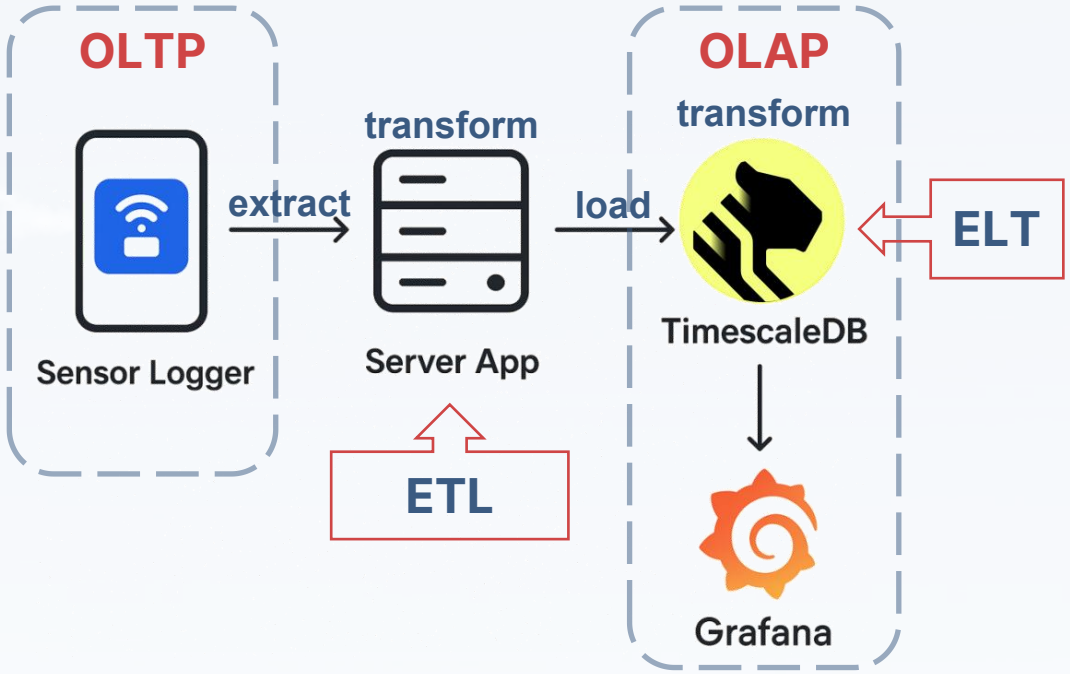


Data Analytics: ETL / ELT процеси

ELT (**Extract, Load, Transform**) — це сучасна альтернатива традиційному ETL-процесу. Його ключова роль полягає в швидкій інтеграції великих обсягів даних у сховища (наприклад, Data Warehouse або Data Lake) та використанні обчислювальної потужності цих сховищ для подальшої трансформації даних.

Цей підхід дозволяє **швидко завантажувати великі обсяги сирих даних**, а потім, використовуючи потужність самої СУБД, перетворювати їх для аналітики.

В системі, що розглядається, роль **ETL** процесу відіграє **TimescaleDB**, яка, зокрема, дозволяє створювати неперервні агрегати для швидкої аналітики (шляхом трансформації) на великих обсягах даних.

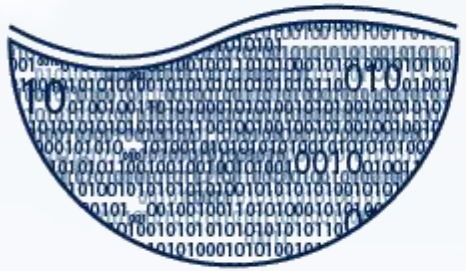


Data Analytics: Data Warehouse vs Data Lake

У сучасному світі Data Analytics та Big Data ключову роль відіграють концепції **Data Warehouse** та **Data Lake.**

Data Lake (Озеро даних)

- Це сховище, яке дозволяє зберігати **величезні** обсяги **сирих, неструктурованих, напівструктурованих** і структурованих даних у їхньому нативному форматі.
- Є основним інструментом для роботи з **Big Data** через свою гнучкість і здатність зберігати дані у їхньому нативному форматі.
- Data Lake дозволяє зберігати **величезні обсяги даних**, які Data Warehouse просто не може обробити або які не мають визначеної структури.

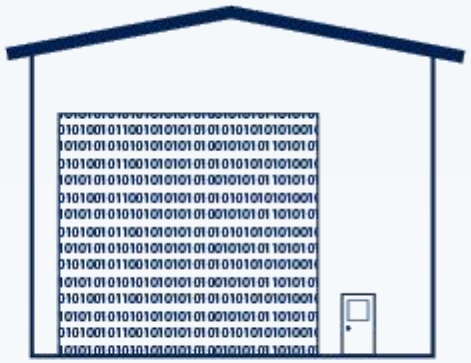


Data Lake

VS

Data Warehouse (Сховище даних)

- Це централізоване сховище, яке об'єднує дані з різних джерел в єдину, **структуровану** базу даних.
- Data Warehouse розроблено спеціально для **аналітики та формування звітів**.
- Він допомагає перетворити величезні масиви сирих даних, зібраних у Data Lake, на **зрозумілі та корисні** для бізнесу інсайти.



Data Warehouse

Data Analytics: Data Lake

Особливості **Data Lake**:

- **Сирі дані**: Зберігає дані "як є", без попередньої обробки чи трансформації.
- **Схема даних**: Використовує підхід "**schema-on-read**" (схема на момент читання). Структура даних визначається лише тоді, коли дані зчитуються для аналізу.
- **Гнучкість**: **Дуже гнучке** сховище, що дозволяє зберігати будь-які типи даних (текст, зображення, відео, дані з сенсорів, логи).
- **Призначення**: Він є ідеальним місцем для розвідувальної аналітики, машинного навчання (ML), штучного інтелекту (AI) та прогнозного аналізу. Коли Data Scientists (фахівці з даних) шукають приховані закономірності в неструктурованих даних (наприклад, у текстах відгуків клієнтів, фотографіях чи даних з датчиків), вони працюють саме з Data Lake.
- **Приклад**: Зберігання логів з веб-серверів, потокових даних із соціальних мереж, даних з IoT-пристроїв.

Data Analytics: Data Warehouse

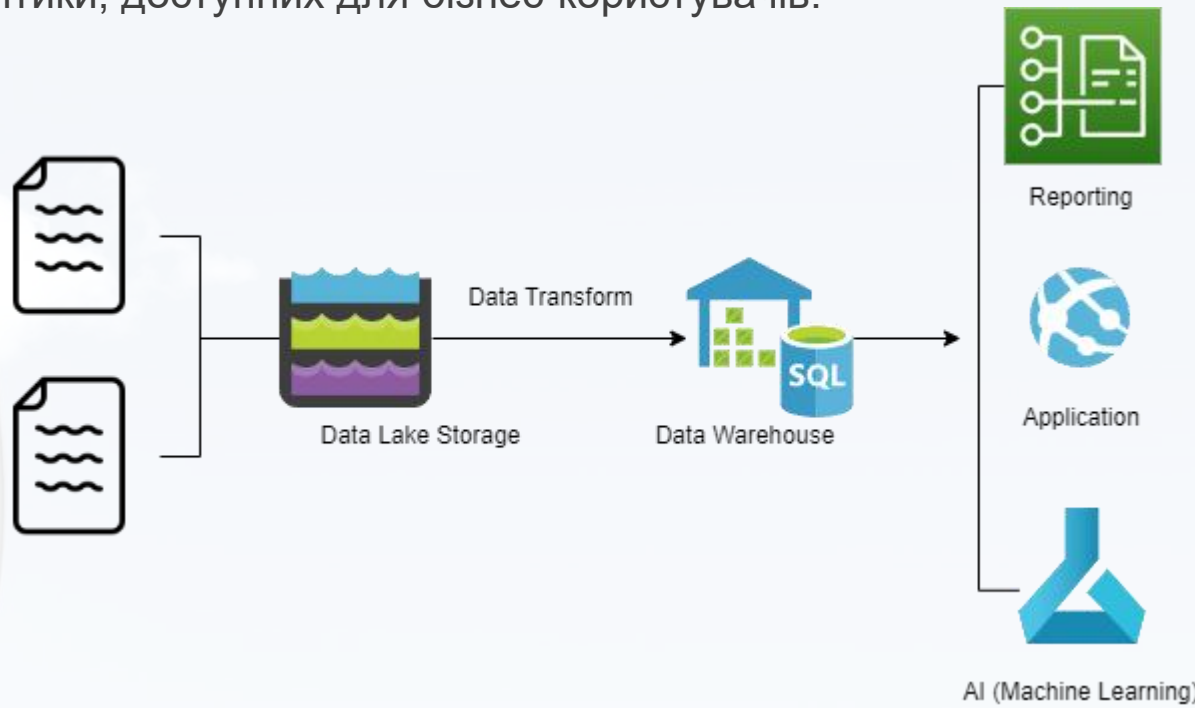
Особливості **Data Warehouse**:

- **Структуровані дані**: Переважно містить структуровані дані, які вже були очищені, трансформовані та завантажені у певний формат.
- **Схема даних**: Використовує підхід **"schema-on-write"** (схема на момент запису). Це означає, що структура даних (схема) визначається до того, як дані будуть завантажені в сховище.
- **Якість даних**: **Висока** якість і узгодженість даних, оскільки вони проходять через процеси ETL (Extract, Transform, Load) або ELT (Extract, Load, Transform).
- **Призначення**: Він ідеально підходить для **Business Intelligence** (BI) та **звітів**, які вимагають швидкого доступу до структурованих, очищених даних. Якщо бізнес-аналітикам потрібно щодня або щотижня отримувати чіткі відповіді на питання "Скільки ми продали за останній місяць?" або "Який наш прибуток у порівнянні з минулим роком?", Data Warehouse забезпечить це швидко і ефективно..
- **Приклад**: Типове використання — аналіз продажів, фінансових даних, клієнтських транзакцій.

Data Analytics: Синергія Data Warehouse та Data Lake

Найефективніший підхід — це **комбінація** обох сховищ:

- Дані надходять з різних джерел у Data Lake у сирому вигляді.
- Фахівці з даних (Data Scientists) проводять глибокий аналіз, шукають інсайти та розробляють моделі машинного навчання.
- Очищені, структуровані та найбільш цінні дані потім переміщуються у Data Warehouse для регулярних звітів, дашбордів та BI-аналітики, доступних для бізнес-користувачів.



Data Analytics: SQL

SQL має досить потужні засоби для здійснення аналітики на структурованих даних у вигляді **віконних функцій**.

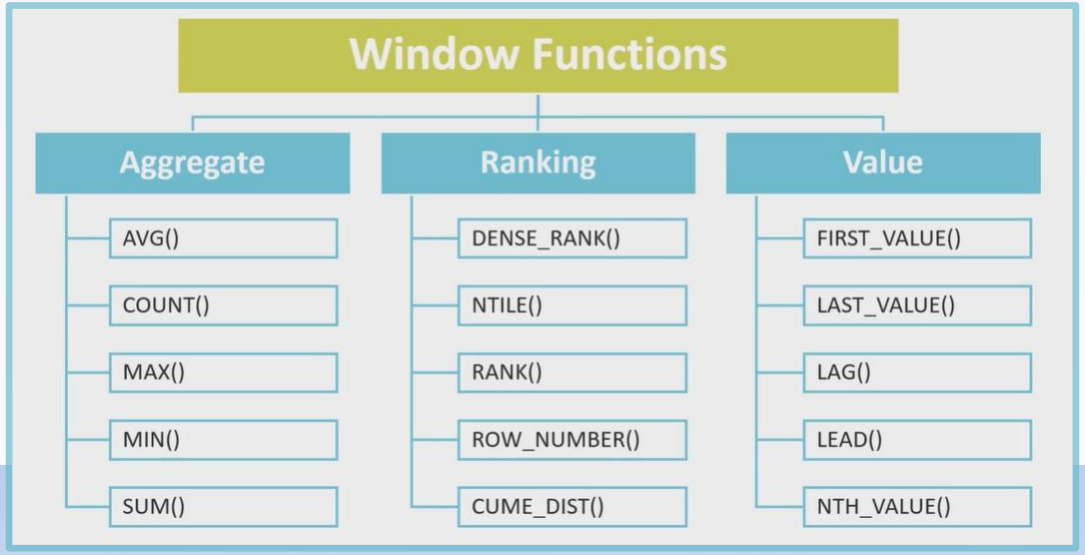
Віконна функція SQL - це функція, яка працює з виділеним набором рядків (вікном) і виконує обчислення для цього набору рядків в окремому стовпці.

При використанні звичайних агрегатних функцій операція GROUP BY скорочує кількість рядків у запиті за допомогою їх групування. У разі використання віконних функцій **кількість рядків у запиті не зменшується** порівняно з вихідною таблицею.

Агрегатні функції також можуть бути використані як віконні функції, потрібно лише додати вираз визначення "вікна". Область застосування віконних функцій найчастіше пов'язана з **аналітичними запитами, аналізом даних**.

Віконні функції прийнято розділяти на 3 класи:

- **Агрегуючі** (Aggregate)
- **Ранжуючі** (Ranking)
- **Функції зсуву** (Value)



Data Analytics: SQL - віконні функції

Загальний синтаксис віконної функції:

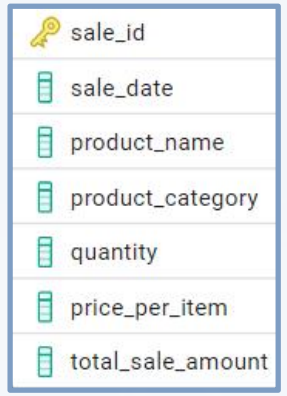
```
Function_name(expression) OVER (
    [PARTITION BY column1, column2, ...]
    [ORDER BY columnA, columnB, ...]
    [Frame Clause]
)
```

- Function_name(expression): Сама функція, яку ви використовуєте, наприклад AVG(sales) або RANK().
- `OVER()`: Ключове слово, яке перетворює звичайну функцію на віконну. Це воно створює "вікно" або набір рядків для обчислення.
- `PARTITION BY`: Розділяє дані на групи. Обчислення виконується незалежно в кожній групі. Це схоже на GROUP BY, але рядки **не згортаються, а залишаються окремими**.
- `ORDER BY`: Сортує рядки в межах кожної групи. Це обов'язково для більшості віконних функцій, особливо для ранжування та функцій зсуву.
- `Frame Clause`: Визначає, які рядки з вікна будуть включені в обчислення для поточного рядка (наприклад, `ROWS BETWEEN 2 PRECEDING AND CURRENT ROW`).

Data Analytics: SQL - агрегуючі віконні функції

Для прикладу використаємо таблицю, яка містить дані про продажі:

	sale_id [PK] integer	sale_date date	product_name character varying (100)	product_category character varying (50)	quantity integer	price_per_item numeric (10,2)	total_sale_amount numeric (10,2)
1	1	2024-01-01	Laptop	Electronics	1	1200.00	1200.00
2	2	2024-01-01	Mouse	Electronics	2	25.00	50.00
3	3	2024-01-01	Coffee	Food & Drinks	3	8.50	25.50
4	4	2024-01-01	T-shirt	Apparel	1	25.00	25.00
5	5	2024-01-02	Headphones	Electronics	1	80.00	80.00
6	6	2024-01-02	Jeans	Apparel	1	60.00	60.00



Обчислити сумарні щоденні прибутки магазину можна за допомогою агрегуючої віконної функції SUM():

```
SELECT sale_date, product_name, product_category, total_sale_amount,
      SUM(total_sale_amount) OVER (PARTITION BY sale_date) AS sale_per_day,
FROM sales
ORDER BY sale_date;
```

	sale_date date	product_name character varying (100)	product_category character varying (50)	total_sale_amount numeric (10,2)	sale_per_day numeric
1	2024-01-01	Laptop	Electronics	1200.00	1300.50
2	2024-01-01	Mouse	Electronics	50.00	1300.50
3	2024-01-01	Coffee	Food & Drinks	25.50	1300.50
4	2024-01-01	T-shirt	Apparel	25.00	1300.50
5	2024-01-02	Headphones	Electronics	80.00	210.00
6	2024-01-02	Jeans	Apparel	60.00	210.00
7	2024-01-02	Science Book	Books	25.00	210.00
8	2024-01-02	Keyboard	Electronics	45.00	210.00

Data Analytics: SQL - агрегуючі віконні функції

Середнє значення щоденного прибутку можна отримати за допомогою запиту:

```
WITH daily_sales AS (
    -- КРОК 1: Обчислюємо загальну суму продажів за кожен день
    SELECT sale_date, SUM(total_sale_amount) AS daily_revenue
    FROM sales
    GROUP BY sale_date
)
-- КРОК 2: Обчислюємо середнє значення продажів за весь час
-- за допомогою віконної функції
SELECT sale_date, daily_revenue, AVG(daily_revenue) OVER () AS average_daily_revenue
FROM daily_sales
ORDER BY sale_date;
```

Коли не вказано обмежуюче вікно (PARTITION BY), розрахунок відбувається для всієї таблиці

	sale_date date	daily_revenue numeric	average_daily_revenue numeric
1	2024-01-01	1300.50	627.790322580645161
2	2024-01-02	210.00	627.790322580645161
3	2024-01-03	670.00	627.790322580645161
4	2024-01-04	224.00	627.790322580645161
5	2024-01-05	522.00	627.790322580645161
6	2024-01-06	86.00	627.790322580645161
7	2024-01-07	882.00	627.790322580645161

Data Analytics: SQL - Віконні функції - Фрейми

Фрейм (рамка) — це набір рядків у межах вікна (window partition), для яких виконується розрахунок віконної функції. По суті, фрейм визначає, які рядки з вашого вікна будуть включені до обчислення для кожного конкретного рядка.

Загальний синтаксис фрейму:

```
<frame_type> BETWEEN [start_bound] AND [end_bound] [EXCLUDE <frame_exclusion_clause>]
```

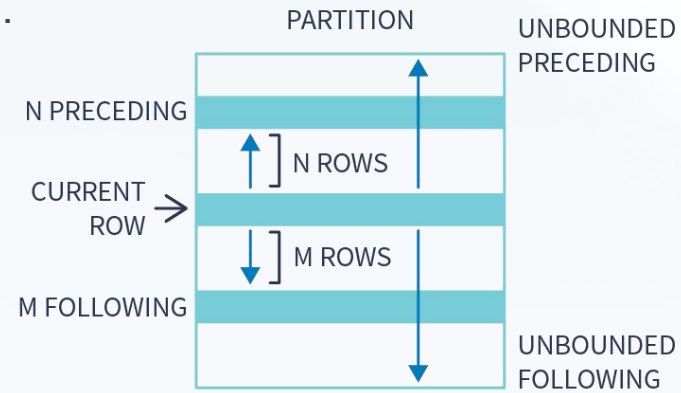
- **frame_type** може бути **ROWS, RANGE** або **GROUPS,**
- **start_bound** і **end_bound** визначають, де починається і закінчується фрейм,
- **EXCLUDE** дозволяє виключити певні рядки з обчислення.

Можливі значення для **start_bound**:

- **UNBOUNDED PRECEDING**: Фрейм починається з першого рядка у поточному вікні.
- **N PRECEDING**: Фрейм починається з N рядків/груп/значень до поточного.
- **CURRENT ROW**: Фрейм починається з поточного рядка/групи/значення.

Можливі значення для **end_bound**:

- **CURRENT ROW**: Фрейм закінчується на поточному рядку/групі/значенні.
- **N FOLLOWING**: Фрейм закінчується на N рядків/груп/значень після поточного.
- **UNBOUNDED FOLLOWING**: Фрейм закінчується на останньому рядку у поточному вікні.



Data Analytics: SQL - агрегуючі віконні функції - Фрейми

Найпоширеніші комбінації для **start_bound** і **end_bound:**

1. Ковзне середнє (Moving Average)

Логіка: Обчислити середнє значення для 3 попередніх рядків (включно з поточним).

Синтаксис: ROWS BETWEEN 2 PRECEDING AND CURRENT ROW

2. Накопичувальна сума (Cumulative Sum)

Логіка: Обчислити суму від початку вікна до поточного рядка.

Синтаксис: ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

Цей фрейм є поведінкою за замовчуванням для SUM() та AVG() з ORDER BY без фрейму.

3. Загальна сума/середнє (Total/Overall Aggregate)

Логіка: Обчислити суму/середнє для всього вікна.

Синтаксис: ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

Цей фрейм є поведінкою за замовчуванням, якщо ORDER BY не вказано.

4. Майбутнє середнє (Future Average)

Логіка: Обчислити середнє значення для поточного та 3 наступних рядків.

Синтаксис: ROWS BETWEEN CURRENT ROW AND 3 FOLLOWING

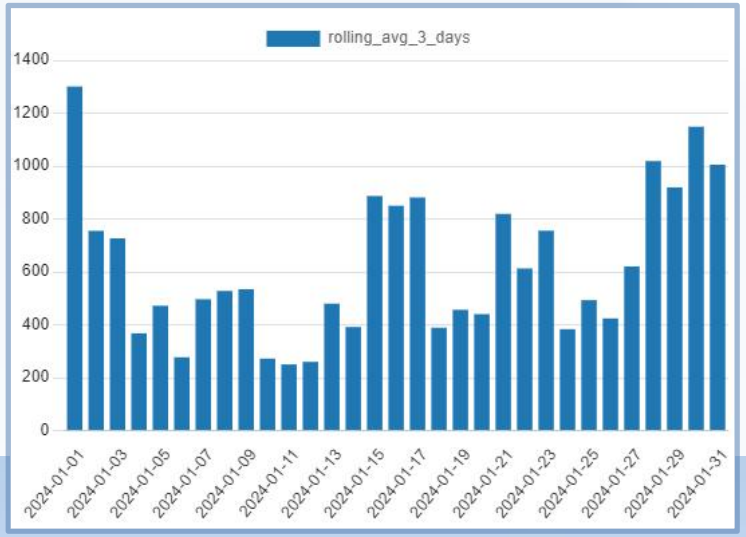
Data Analytics: SQL - віконні функції - типи фреймів

1. Фрейм **ROWS:** визначає рамку на основі фіксованої кількості **рядків,** відсортованих за значенням, зазначеним в **ORDER BY,** до або після поточного рядка. Він ідеально підходить для обчислення ковзних середніх (moving averages), сумарних накопичень (cumulative sums) та інших функцій, де важлива послідовність рядків.

Наприклад, розрахуємо **ковзне середнє** (за три дні) **по щоденним сумам**:

```
WITH daily_total_sales AS (
    -- КРОК 1: Обчислюємо загальну суму продажів за кожен день
    SELECT sale_date, SUM(total_sale_amount) AS daily_total
    FROM sales
    GROUP BY sale_date
)
-- КРОК 2: Застосовуємо віконну функцію до щоденних сум
SELECT
    sale_date, daily_total,
    -- Фрейм, що включає три рядки включно з поточним
        AVG(daily_total) OVER (
        ORDER BY sale_date
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS rolling_avg_3_days
FROM daily_total_sales
ORDER BY sale_date;
```

	sale_date date	daily_total numeric	rolling_avg_3_days numeric
1	2024-01-01	1300.50	1300.5000000000
2	2024-01-02	210.00	755.25000000000
3	2024-01-03	670.00	726.83333333333
4	2024-01-04	224.00	368.00000000000
5	2024-01-05	522.00	472.00000000000
6	2024-01-06	86.00	277.33333333333
7	2024-01-07	882.00	496.66666666666



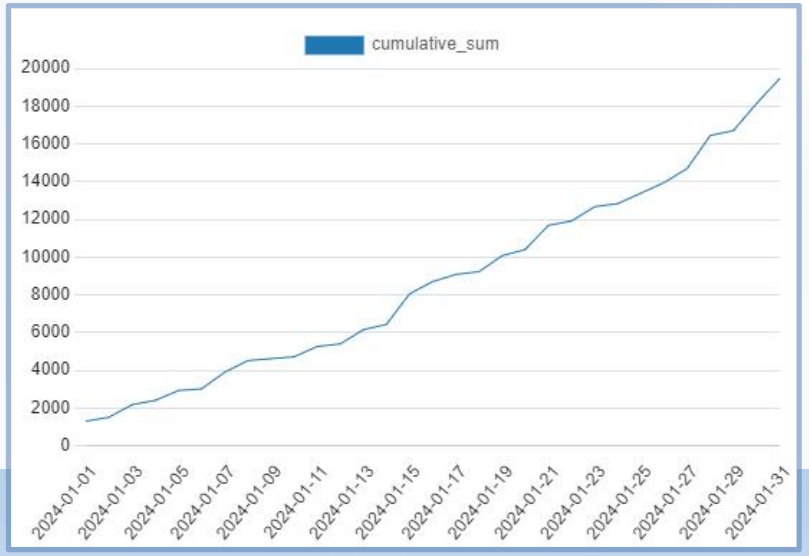
Data Analytics: SQL - віконні функції - типи фреймів

1. Фрейм **ROWS**

Приклад розрахунку **щоденної накопичувальної суми продажів**:

```
WITH daily_sales AS (
    -- КРОК 1: Обчислюємо загальну суму продажів за кожен день
    SELECT sale_date,
        SUM(total_sale_amount) AS daily_total
    FROM sales
    GROUP BY sale_date
)
-- КРОК 2: Обчислюємо накопичувальну суму
SELECT
    sale_date,
    daily_total,
    -- Фрейм, що включає всі рядки від початку вікна до поточного
    SUM(daily_total) OVER (
        ORDER BY sale_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS cumulative_sum
FROM daily_sales
ORDER BY sale_date;
```

	sale_date date	daily_total numeric	cumulative_sum numeric
1	2024-01-01	1300.50	1300.50
2	2024-01-02	210.00	1510.50
3	2024-01-03	670.00	2180.50
4	2024-01-04	224.00	2404.50
5	2024-01-05	522.00	2926.50
6	2024-01-06	86.00	3012.50
7	2024-01-07	882.00	3894.50



Data Analytics: SQL - віконні функції - типи фреймів

1. Фрейм **ROWS**

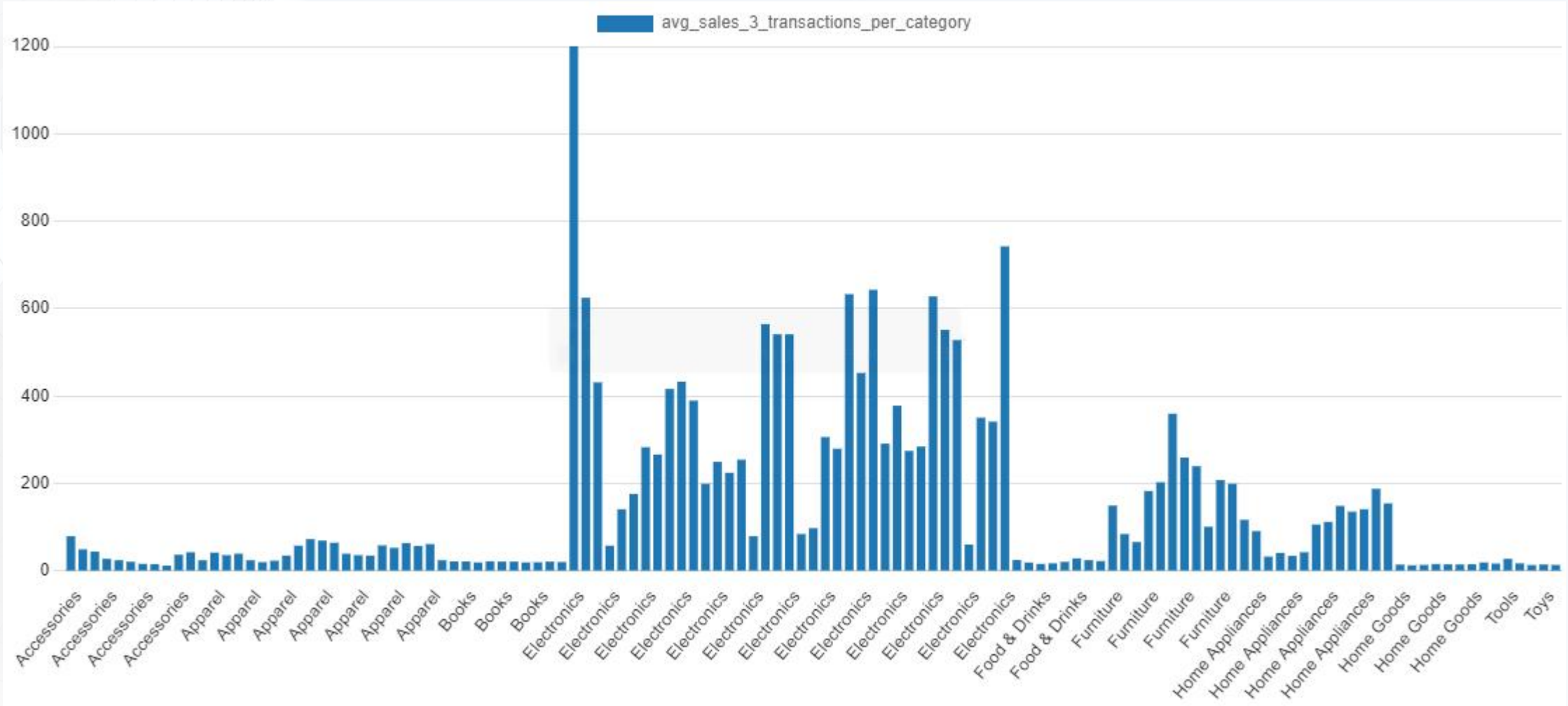
Приклад розрахунку **ковзного середнього по категоріях**:

```
SELECT
    sale_date,
    product_category,
    total_sale_amount,
    -- Ковзне середнє для останніх 3 транзакцій
     -- всередині кожної категорії
    AVG(total_sale_amount) OVER (
        PARTITION BY product_category
        ORDER BY sale_date
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS avg_sales_3_transactions_per_category
FROM
    sales
ORDER BY
    product_category,
    sale_date;
```

	sale_date date	product_category character varying (50)	total_sale_amount numeric (10,2)	avg_sales_3_transactions_per_category numeric
1	2024-01-05	Accessories	80.00	80.0000000000000000
2	2024-01-07	Accessories	20.00	50.0000000000000000
3	2024-01-10	Accessories	35.00	45.0000000000000000
4	2024-01-12	Accessories	30.00	28.3333333333333333
5	2024-01-15	Accessories	10.00	25.0000000000000000
6	2024-01-17	Accessories	25.00	21.6666666666666667
7	2024-01-18	Accessories	15.00	16.6666666666666667
8	2024-01-19	Accessories	8.00	16.0000000000000000
9	2024-01-23	Accessories	15.00	12.6666666666666667
10	2024-01-25	Accessories	90.00	37.6666666666666667
11	2024-01-25	Accessories	25.00	43.3333333333333333
12	2024-01-01	Apparel	25.00	25.0000000000000000
13	2024-01-02	Apparel	60.00	42.5000000000000000
14	2024-01-04	Apparel	25.00	36.6666666666666667

Data Analytics: SQL - агрегуючі віконні функції - Фрейми

Результат виконання попереднього запиту



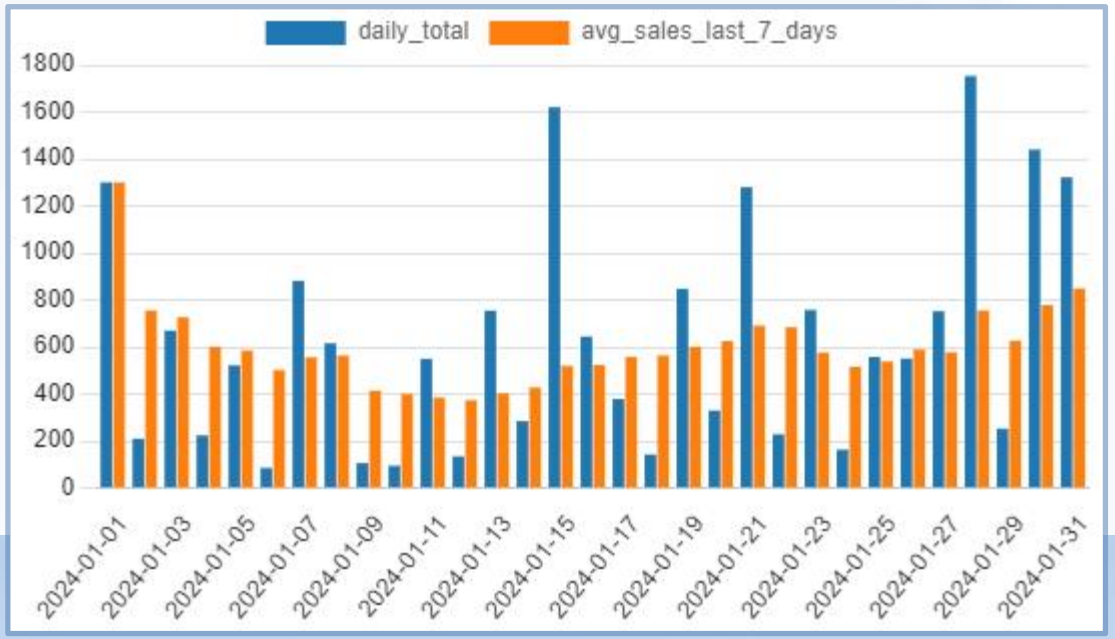
Data Analytics: SQL - віконні функції - типи фреймів

2. Фрейм **RANGE:** визначає рамку на основі діапазону значень у колонці, вказаній в ORDER BY, а не за кількістю рядків. Це особливо корисно, коли ви працюєте з датами, цінами, або іншими числовими значеннями.

Приклад: **Ковзне середнє продажів у діапазоні дат**

```
SELECT
    sale_date, SUM(total_sale_amount) AS daily_total,
    -- Ковзне середнє для всіх продажів за останні 7 днів
    AVG(SUM(total_sale_amount)) OVER (
        ORDER BY sale_date
        RANGE BETWEEN INTERVAL '7' DAY PRECEDING AND CURRENT ROW
    ) AS avg_sales_last_7_days
FROM sales
GROUP BY sale_date
ORDER BY sale_date;
```

	sale_date date	daily_total numeric	avg_sales_last_7_days numeric
1	2024-01-01	1300.50	1300.5000000000000
2	2024-01-02	210.00	755.25000000000000
3	2024-01-03	670.00	726.83333333333333
4	2024-01-04	224.00	601.12500000000000
5	2024-01-05	522.00	585.30000000000000



3. Фрейм **GROUPS:** менш поширений фрейм, який працює аналогічно до ROWS, але замість одиничних рядків оперує групами рядків, які мають однакові значення у колонці ORDER BY. Він формує групи на основі однакових значень з колонки, зазначеній в ORDER BY.

Data Analytics: SQL - віконні функції - EXCLUDE

Конструкція **EXCLUDE** є додатковою опцією у синтаксисі фреймів, яка дозволяє виключити певні рядки з обчислення.

Вона може включати одне з наступних значень:

- EXCLUDE CURRENT ROW: Виключити тільки поточний рядок з фрейму.
- EXCLUDE GROUP: Виключити поточний рядок і всі рядки, які мають таке саме значення, як поточний, в колонці ORDER BY.
- EXCLUDE TIES: Виключити всі рядки з тим же значенням, що й поточний, але залишити сам поточний рядок.
- EXCLUDE NO OTHERS: Це поведінка за замовчуванням. Всі рядки включені в обчислення.

Приклад запиту для **"центрованого" ковзного середнього** (по сусідніх днях)**:**

```
SELECT
    sale_date,
    SUM(total_sale_amount) AS daily_total,
    -- Середнє значення по сусідніх днях, виключаючи поточний
    AVG(SUM(total_sale_amount)) OVER (
        ORDER BY sale_date
        RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND INTERVAL '1' DAY FOLLOWING
        EXCLUDE CURRENT ROW
    ) AS avg_neighboring_days
FROM sales
GROUP BY sale_date
ORDER BY sale_date;
```

	sale_date date	daily_total numeric	avg_neighboring_days numeric
1	2024-01-01	1300.50	210.00000000000000
2	2024-01-02	210.00	985.25000000000000
3	2024-01-03	670.00	217.00000000000000
4	2024-01-04	224.00	596.00000000000000
5	2024-01-05	522.00	155.00000000000000

Data Analytics: SQL - Віконні функції - Фрейми

Розглянемо у якості ще одного інтегрального прикладу розрахунок відхилення від середнього у групі.

Уявіть, що ви керуєте аптекою і хочете знати, чи є **залишок** певного препарату на вашому складі **аномально високим або низьким** порівняно із середнім залишком у сусідніх аптеках. Для цього потрібно обчислити середній запас, але **виключити з розрахунку саму поточну аптеку**. Наступний запит обчислює середній запас ліків у сусідніх аптеках (на основі номера аптеки), порівнюючи його із запасом в поточній аптеці:

```
SELECT
    city, pharmacy_number, drug_name, stock_quantity,
    -- Середній запас у сусідніх аптеках (виключаючи поточну)
    AVG(stock_quantity) OVER (
        PARTITION BY city, drug_name
        ORDER BY pharmacy_number
        GROUPS BETWEEN 2 PRECEDING AND 2 FOLLOWING
        EXCLUDE CURRENT ROW
    ) AS avg_stock_in_neighbors,
    -- Різниця між запасом у поточній аптеці та середнім у сусідніх
    stock_quantity - AVG(stock_quantity) OVER (
        PARTITION BY city, drug_name
        ORDER BY pharmacy_number
        GROUPS BETWEEN 2 PRECEDING AND 2 FOLLOWING
        EXCLUDE CURRENT ROW
    ) AS stock_deviation
FROM pharmacy_stock
ORDER BY city, pharmacy_number, drug_name;
```

Data Analytics: SQL - ранжуючі віконні функції

Ранжуючі віконні функції призначені для присвоєння послідовного **номера** або **рангу** кожному рядку в межах певного вікна. Вони ідеально підходять для таких завдань, як:

- Вибір "топ-N" записів у кожній групі.
- Знаходження дублікатів.
- Створення порядкових номерів.

Найпоширеніші з них — **ROW_NUMBER()**, **RANK()**, **DENSE_RANK()**, **NTILE()**.

Наприклад, визначимо, які категорії є найбільш прибутковими. Для цього спочатку згрупуємо продажі за категоріями, а потім проранжуємо ці категорії на основі їхньої загальної виручки.

```
WITH category_sales AS (
    -- КРОК 1: Обчислюємо загальну суму продажів для кожної категорії
    SELECT product_category, SUM(total_sale_amount) AS total_sales
    FROM sales
    GROUP BY product_category
)
-- КРОК 2: Ранжуємо категорії на основі їхньої суми продажів
SELECT
    product_category, total_sales,
    RANK() OVER (
        ORDER BY total_sales DESC
    ) AS category_rank
FROM category_sales
ORDER BY total_sales DESC;
```

	product_category character varying (50)	total_sales numeric	category_rank bigint
1	Electronics	13851.00	1
2	Furniture	2105.00	2
3	Home Appliances	1553.00	3
4	Apparel	933.00	4
5	Accessories	353.00	5
6	Books	236.00	6
7	Food & Drinks	173.00	7
8	Home Goods	111.50	8
9	Tools	85.00	9
10	Toys	61.00	10

Data Analytics: SQL - віконні функції зсуву

Віконні функції зсуву використовуються для отримання значення з конкретного рядка у вікні, незмінного відносно поточного рядка. Вони не ранжують дані, а саме "витягують" значення з певної позиції. Це корисно для порівняння всіх інших значень з конкретним (першим, останнім, ...).

- **FIRST_VALUE**(): повертає значення з першого рядка у вікні.
- **LAST_VALUE**(): повертає значення з останнього рядка у вікні.
- **NTH_VALUE**(): повертає значення з N-го рядка у вікні.
- **LAG**(): дозволяє отримати значення з рядка, який передує поточному на певну кількість позицій.
- **LEAD**(): дозволяє отримати значення з рядка, який слідує за поточним на певну кількість позицій.

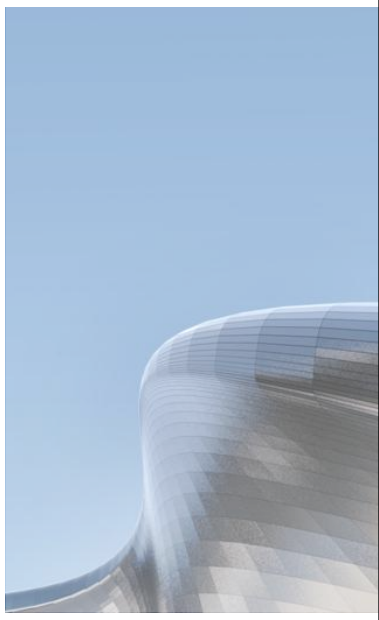
Приклад обчислення різниці в часі (у днях) між кожним продажем та попереднім (це може допомогти зрозуміти, наскільки часто відбуваються покупки):

```
SELECT sale_date, product_name, total_sale_amount,
    LAG(sale_date) OVER (ORDER BY sale_date) AS previous_sale_date,
    (sale_date - LAG(sale_date) OVER (ORDER BY sale_date)) AS days_since_last_sale
FROM sales
WHERE product_name = 'Jeans'
ORDER BY sale_date;
```

	sale_date date	product_name character varying	total_sale_amount numeric (10,2)	previous_sale_date date	days_since_last_sale integer
1	2024-01-02	Jeans	60.00	[null]	[null]
2	2024-01-21	Jeans	65.00	2024-01-02	19
3	2024-01-26	Jeans	62.00	2024-01-21	5
4	2024-01-30	Jeans	65.00	2024-01-26	4

PART 06

Колонкові СУБД



Ширококолонкові СУБД

Модель даних: База даних організована навколо **дворівневої** ключової структури:

- **Перший ключ:** Визначає, **на якому вузлі** фізично зберігаються дані (для шардування).
- **Другий ключ:** Визначає, **як сортуються** рядки всередині цього вузла (для швидкого пошуку).

Схема: Гнучка і розріджена (Sparse). Кожен рядок може мати **необмежену** кількість колонок, і на диску зберігаються лише ті, які містять дані. Рядки являють собою множину триплетів (назва колонки, значення, мітка часу).

Доступ до даних: Забезпечується **надзвичайно низька** затримка (Low-Latency) та **висока пропускна здатність** при доступі до даних за Першим ключем, навіть при величезних обсягах даних (**Big Data**).

Масштабування: Шардування (фрагментація) даних відбувається за **Першим ключем** для лінійної масштабованості.

Мова запитів: Використовується SQL-подібна мова (наприклад, CQL) з фокусом на **запитах по ключу**.

Варіанти використання колонкових СУБД

Типове застосування:

- **Збір подій** (Event Logging): Зберігання кліків, логів, телеметрії.
- **Оперативні профілі**: Швидкий доступ до історії активності користувача.
- **Розподілена Інвентаризація**: Відстеження запасів на великій кількості складів.
- **Геопросторові дані**: Зберігання даних про переміщення об'єктів та GPS-координат.
- **Зберігання метрик в реальному часі**: Системи моніторингу та діагностики продуктивності (APM).
- **Служби обміну повідомленнями**: Зберігання історії чатів та повідомлень з високою доступністю.
- Сценарії, де критично важливі **максимальна доступність** та **велика пропускна здатність** запису.

Rank			DBMS	Database Model	Score		
Jul 2025	Jun 2025	Jul 2024			Jul 2025	Jun 2025	Jul 2024
1.	1.	1.	Apache Cassandra	Wide column, Multi-model	108.76	+0.49	+9.63
2.	2.	2.	Apache HBase	Wide column	22.72	+0.04	-5.01
3.	3.	3.	Microsoft Azure Cosmos DB	Multi-model	21.68	-0.75	-5.44
4.	4.	↑ 5.	ScyllaDB	Wide column, Multi-model	3.98	+0.02	+0.15
5.	5.	↓ 4.	Datastax Enterprise	Wide column, Multi-model	3.66	+0.04	-1.44
6.	6.	↑ 7.	Microsoft Azure Table Storage	Wide column	2.61	+0.00	-0.80
7.	7.	↑ 8.	Google Cloud Bigtable	Multi-model	2.56	-0.03	-0.37

Apache Cassandra

Створена у 2008 році в компанії Facebook для забезпечення швидкого пошуку у власній системі вхідних повідомлень.

Основна мова: Написана переважно мовою Java.

Тип СУБД: Децентралізована, розподілена Wide-Column база даних. Оптимізована для Big Data та високої доступності, застосовує модель кінцевої узгодженості (Eventual Consistency).

Мова запитів: Використовує CQL (Cassandra Query Language) — декларативну мову, подібну до SQL.

Ліцензія: Розповсюджується під вільною ліцензією Apache 2.0.

Поточна версія: Актуальна стабільна версія — 4.1.x.

Екосистема: Часто використовується в парі з Apache Spark для виконання складної аналітики та обробки величезних масивів даних.