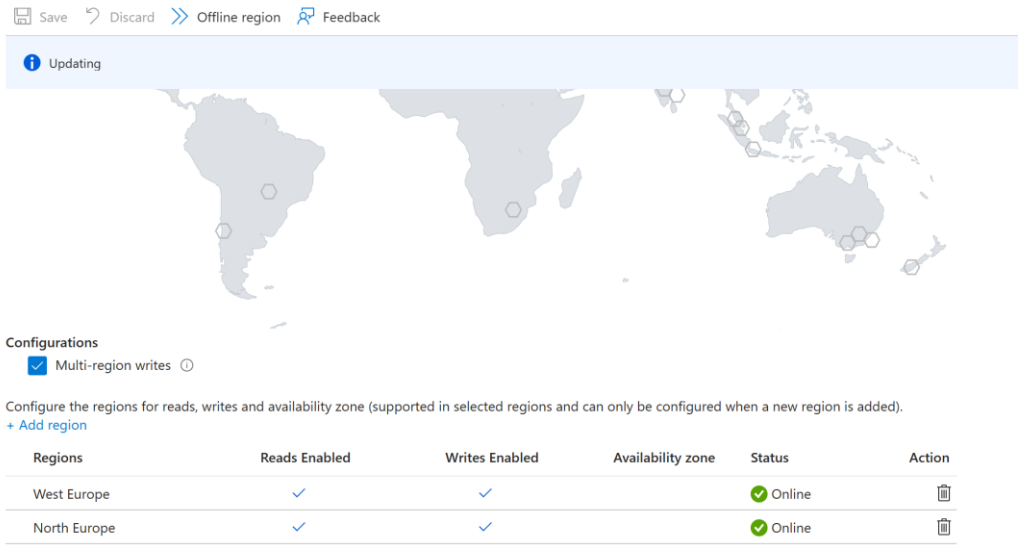
Лекція 7. NoSQL бази даних – Cosmos DB.

1. Тема 1. Принципи узгодженості в Cosmos DB.
2. Тема 2. Режими керування ресурсами — Provisioned Throughput Mode (заздалегідь виділений пропускний режим) та Serverless Mode (безсерверний режим).
3. Тема 3. Логічні та фізичні секції (partitions) у Cosmos DB.
4. Тема 4. Бібліотека azure.cosmos — ключова ієрархія об’єктів.

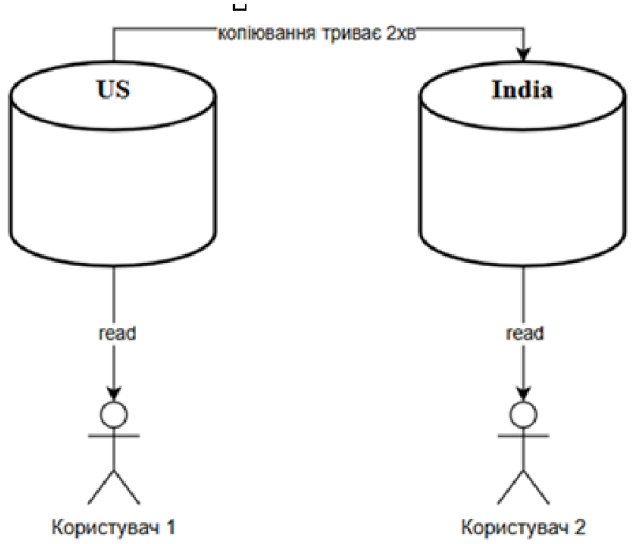


- ***Azure Cosmos DB - це розподілена база даних*** , тобто є сукупністю логічно взаємопов'язаних баз даних, які працюють разом і розподілені на різних серверах або у різних географічних регіонах. Це дозволяє швидше обробляти запити користувачів, навіть якщо вони знаходяться в різних країнах, оскільки база даних може бути доступна на найближчому сервері.
- ***Azure Cosmos DB*** використовує систему управління базами даних, яка гарантує, що всі ці розподілені бази даних будуть працювати злагоджено. Це включає в себе обробку запитів, оновлення даних, синхронізацію між різними репліками бази даних та забезпечення безпеки.

Multi-Region Write – це конфігурація в Azure Cosmos DB, яка дозволяє писати дані в кількох регіонах одночасно. В результаті, ваша база даних може бути доступною для читання і запису в кількох географічних точках одночасно, що підвищує відмовостійкість, швидкість доступу і масштабованість додатків.

Основні етапи роботи:

- ✓ Вибір регіонів для запису: вказуєте, в яких регіонах ваша база даних повинна дозволяти записувати дані. Це можуть бути декілька регіонів в різних частинах світу.
- ✓ Синхронізація: Кожен запис, зроблений в одному з регіонів, автоматично синхронізується з іншими регіонами. Це забезпечує консистентність даних в реальному часі.
- ✓ Реплікація даних: Копії зберігаються в усіх вибраних регіонах, що дозволяє отримати доступ до них швидше, особливо для користувачів, що знаходяться далеко від основного ценрта збереження даних.

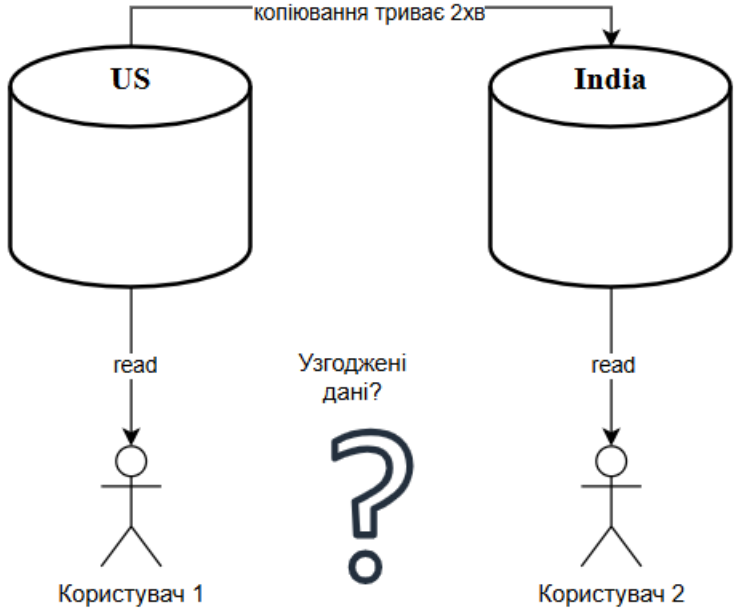


Якщо ваш додаток має користувачів по всьому світу і вимагає **низької затримки** при читанні та запису даних (наприклад, ігри, фінансові додатки, соціальні мережі), тоді **Multi-Region Write** буде рішенням.

*Які проблеми можуть бути повязані з **Multi-Region Write** функціональністю?*

Узгодженість даних

Узгодженість даних у контексті **Azure Cosmos DB** означає, наскільки однакова і консистентна інформація зберігається та доступна на всіх розподілених копіях бази даних в різних регіонах або серверах. Реплікація розподілених баз даних для забезпечення високого рівня їх доступності та низької затримки передбачає компроміс між узгодженістю читання та такими параметрами, як доступність, час затримки та пропускна спроможність.

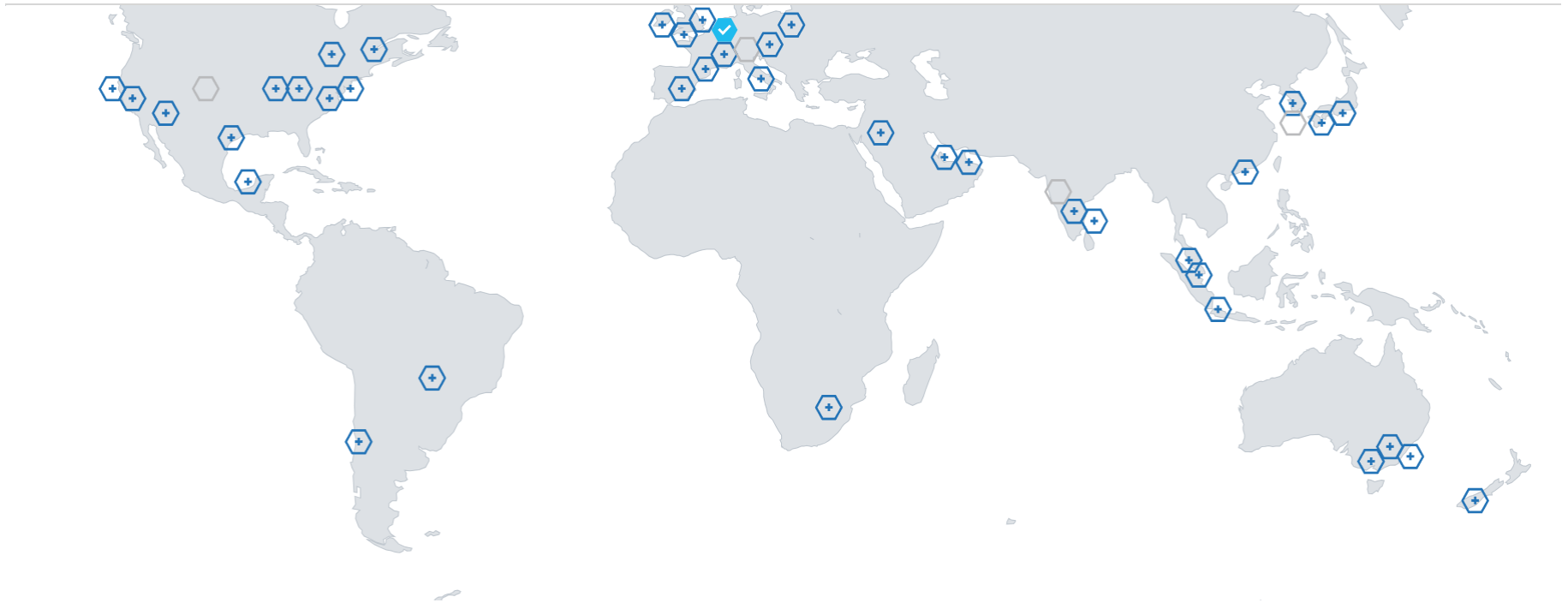


Приклад: Для копіювання даних з регіону US до регіону India потрібно 2хв, протягом цього часу, користувач з регіону US буде читати дані, які відрізняються від даних доступних для регіону India. Тобто дані будут не узгоджені.

Алгоритми для узгодженості даних

Рівень узгодженості	Характеристика
Strong	Повна узгодженість — завжди найсвіжіші дані
Bounded Staleness	Обмежене відставання (наприклад, «відставання на 5 операцій»)
Session	Сесійна узгодженість (гарантується узгодженість у межах сесії клієнта)
Consistent Prefix	Дані повертаються у правильному порядку, але не обов’язково найновіші
Eventual	Гарантовано узгодженість *з часом* (але не миттєво)

За замовчуванням база даних Cosmos DB створюється в одному регіоні з режимом узгодженості Session.



Configurations

Service managed failover Multi-region writes

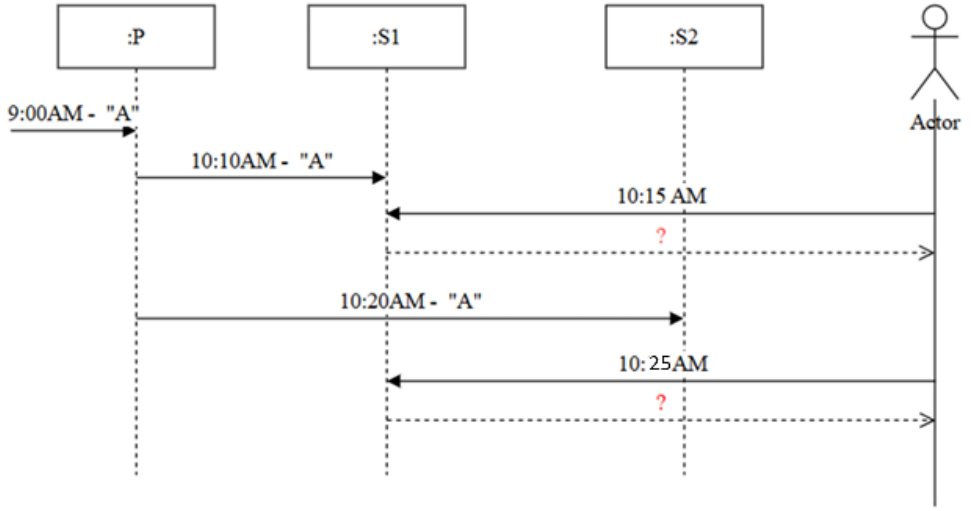
Configure the regions for reads, writes and availability zone (supported in selected regions and can only be configured when a new region is added).

+ Add region

Сувора узгодженість

Сувора узгодженість гарантує ***лінеаризацію***. Лінеарізація означає одночасне обслуговування запитів. Усі операції читання гарантовано повертають останню версію елемента. Клієнт ніколи не побачить не зафіксований або частково змінений запис. Тобто користувач не має змоги прочитані дані, які не підтверджено операцією commit.

Приклад: нехай головна база даних (“P”) отримала запис «A» в 9:00AM. Копіювання даних в інші регіони («S1», “S2”) закінчилась і операція commit для бази даних в регіоні “S2” закінчилась в 10:20 AM, в регіоні “S1” в 10:10 AM. Коли запис «A» стане доступним для читання в усіх регіонах (P, S1, S2)?



Основні характеристики суворої узгодженості:

1. **Низька доступність і висока латентність**: Оскільки сувора узгодженість вимагає підтвердження змін від усіх реплік у системі, перш ніж вони стануть видимими для читання, це збільшує затримки (латентність). У Azure Cosmos DB з суворою узгодженістю і кількома регіонами запис вважається завершеним лише тоді, коли він підтверджений у всіх регіонах. Через це затримка запису залежить від мережевої відстані між регіонами — чим далі один від одного регіони, тим довше триває запис.

   Затримка ≈ 2 × RTT (***Round-Trip Time*** - час, за який сигнал проходить від джерела до місця призначення і назад.) + 10 мс.

   Уявімо, у вас є Cosmos DB з трьома регіонами для копіювання даних:

   ✓ США (P)

   ✓ Європа (S1)

   ✓ Азія (S2)

   Найбільша затримка RTT між США і Азією — 150 мс.

   Тоді затримка запису ≈ 2 × 150 мс + 10 мс = 310 мс

   Отже, кожен запис в Cosmos DB завершиться лише через 310 мс, бо система чекає підтвердження від усіх регіонів.

Сценарії використання суворої узгодженості:

•**Фінансові транзакції**: Де важливо мати найсвіжіші дані для прийняття рішень і уникнення конфліктів при оновленнях.

•**Робота з системами управління інвентарем**: Коли потрібна гарантія того, що інформація про кількість товарів завжди актуальна.

Переваги та недоліки:

•**Переваги**: Найвищий рівень узгодженості даних, мінімізація можливих конфліктів між даними, повна впевненість в актуальності даних.

•**Недоліки**: Збільшення затримки через необхідність узгодження всіх реплік, зниження доступності у разі мережевих проблем або недоступності певних реплік.

Обмежене відставання (Bounded staleness).

“Обмежене відставання» - читання можуть відставати від останніх записів, але лише в межах певної затримки: за кількістю операцій (K), або за часом (T секунд). Усі клієнти бачать зміни в одному і тому ж порядку. В результаті - затримка менша, ніж в режимі "сувора узгодженість".

“Обмежене відставання», коли час запізнення дорівнює 0 являється «суворою» узгодженістю.

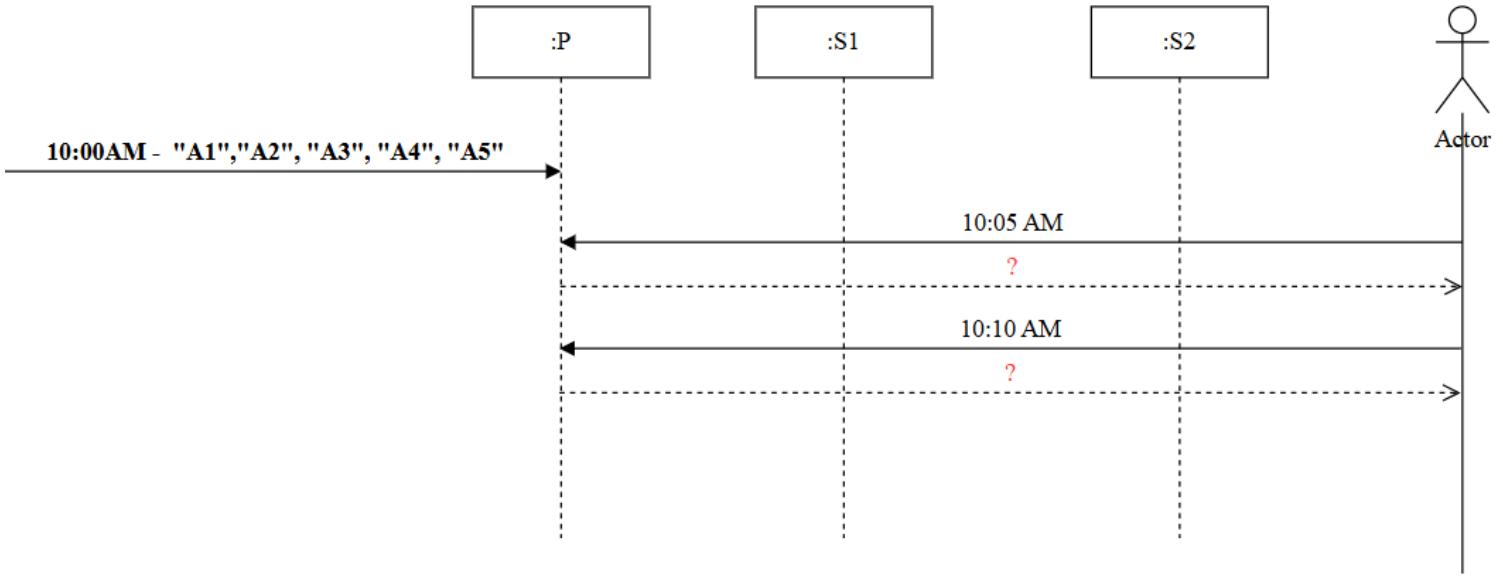
Приклад: обліковий запис Azure Cosmos DB налаштовано з обмеженою із запізненням узгодженістю з параметрами:

K = 5 (максимум 5 змін можуть бути пропущені)

T = 10 хвилин (або максимум 10 хвилин затримки)

О 10:00 AM в головному регіоні (P) виконується 5 послідовних записів: A1, A2, A3, A4, A5.

Користувач у вторинному регіоні (S) виконує запит на читання о 10:05 AM. Чи побачить користувач у регіоні S усі зміни (A1–A5) о 10:05 AM?



Переваги обмеженої із запізненням узгодженості:

1.Баланс між продуктивністю і узгодженістю: Оскільки система не завжди повинна негайно синхронізуватися між всіма репліками, це дозволяє покращити продуктивність, не вимагаючи сильної узгодженості.

2.Контрольована узгодженість: Користувач може точно визначити максимальне відставання, яке система дозволить, що надає гнучкість в залежності від вимог застосунку.

3.Глобальна реплікація: Цей рівень узгодженості особливо підходить для глобально розподілених баз даних, оскільки він дозволяє певне відставання між регіонами, що зменшує затримки при записах і покращує продуктивність для читачів.

Узгодженість сеансів

Узгодженість сеансів забезпечує, що всі операції, виконані в межах одного сеансу, будуть бачити актуальну інформацію, навіть якщо система копіює дані по різних регіонах.

Як це працює?

Кожен сеанс для Cosmos DB зазвичай асоціюється з унікальним ідентифікатором – наприклад, це може бути ID користувача або ID сесії.

Усі зміни, зроблені в рамках одного сеансу, будуть видимими для того самого користувача, навіть якщо база даних реплікується по кількох локаціях.

Якщо один користувач змінює документ, усі наступні запити цього ж користувача (протягом того ж сеансу) отримуватимуть актуальні дані, а інші користувачі можуть отримувати старіші версії даних до моменту, поки ці зміни не будуть синхронізовані.

Приклади використання:

•Інтернет-магазини: клієнт може бачити свої зміни в кошику або історії покупок, але не потрібно негайно бачити зміни, які зробили інші користувачі.

•Персоналізовані додатки: дані, які стосуються лише одного користувача, не потребують глобальної узгодженості, але важливо, щоб цей користувач завжди бачив актуальну інформацію у своїй сесії.

Приклад

```
 session_token_a = client_a.client_connection.last_response_headers.get('x-ms-session-token')
print("Session token A:", session_token_a)
```

Опис принципу узгодженості префіксів (Prefix Consistency)

Узгодженість префіксів означає, що читач завжди бачить події (оновлення, записи) у тому ж порядку, у якому вони були записані, навіть якщо бачить їх не всі.

Тобто, якщо система застосовує зміни послідовно: Операція 1 → Операція 2 → Операція 3 то клієнт може побачити:

тільки Операцію 1,

або Операції 1 і 2,

або всі 1, 2, 3,

але ніколи не побачить 2 без 1, чи 3 без 2.

Це гарантує логічну послідовність подій, навіть якщо дані ще не синхронізувалися повністю.

Якщо система має кілька регіонів (реплік), узгодженість префіксів означає: Репліка може відставати, але ніколи не плутає порядок оновлень.

Принцип підсумкової узгодженості (eventual consistency)

Основні принципи підсумкової узгодженості:

1.Затримка в узгодженні: Після того, як дані оновлено в одній репліці (вузлі), оновлення поширюється на інші репліки з деякою затримкою. Інші репліки можуть тимчасово бачити старі значення даних до того, як оновлення досягне їх.

2.Поступове досягнення узгодженості: З плином часу і після виконання достатньої кількості операцій всі копії даних зрештою синхронізуються, і всі вузли матимуть однакові версії даних.

3.Баланс між продуктивністю та узгодженістю: Підсумкова узгодженість надає більш високу продуктивність і менші затримки в порівнянні з моделями сильної, оскільки немає необхідності в синхронному оновленні всіх вузлів.

4.Невизначеність у проміжний момент: У короткий проміжок часу після оновлення різні клієнти можуть бачити різні версії даних, в залежності від того, до якого вузла або репліки вони звернулися.

Оцінка завантаженості

Кожна операція бази даних споживає системні ресурси. Споживання залежить від складності операції. ОЗ (одиниця запиту) в секунду – це одиниця продуктивності, яка обчислює системні ресурси (наприклад, ЦП, операції введення-виведення в секунду та пам'ять), необхідні для виконання операцій бази даних, що підтримуються Azure Cosmos DB.

$$1\ \text{ОЗ} = \text{читання}\ 1\ \text{КБ}$$

Якщо потрібно виконати операцію читання 100 разі в секунду, тоді маємо 100 ОЗ.

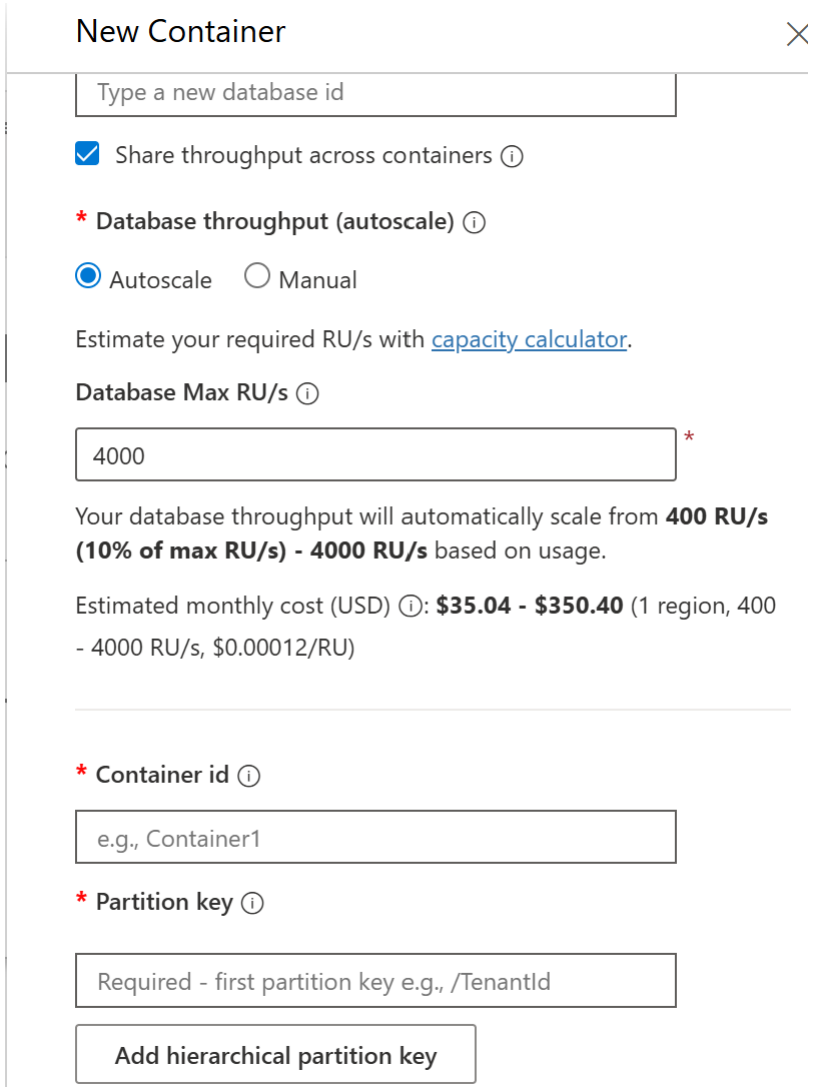
Якщо для облікового запису включено рівень «Безкоштовний», то надається 1000 ОЗ в секунду і 25Гб сховища.

Пропускна Здатність (Provisioned Throughput) - це попередньо задана кількість ОЗ/с (RU/s), яку ви виділяєте для своєї бази даних або контейнера.

Режими керування ресурсами

Основні режими:

- **Provisioned Throughput Mode (заздалегідь виділений пропускний режим)**
- **Serverless Mode (безсерверний режим)**



Provisioned Throughput Mode (заздалегідь виділений пропускний режим)

Основні види налаштувань:
✓ Автоматичне масштабування продуктивності(Autoscale)
Діапазон: від мінімального значення до 10× (наприклад: 100–1000 RU/s).
Переваги:
Оптимізовано для змінного навантаження.
Автоматичне масштабування вгору/вниз без втручання користувача.
Підходить для: сезонних піків, непередбачуваних змін у трафіку.
✓ Гарантована продуктивність — RU/s резервується заздалегідь.
Фіксоване значення RU/s, яке не змінюється автоматично.
Переваги:
Прогнозовані витрати.
Підходить для стабільного трафіку, наприклад, бекенд-сервісів.
Недоліки:
Не гнучко при змінному навантаженні.

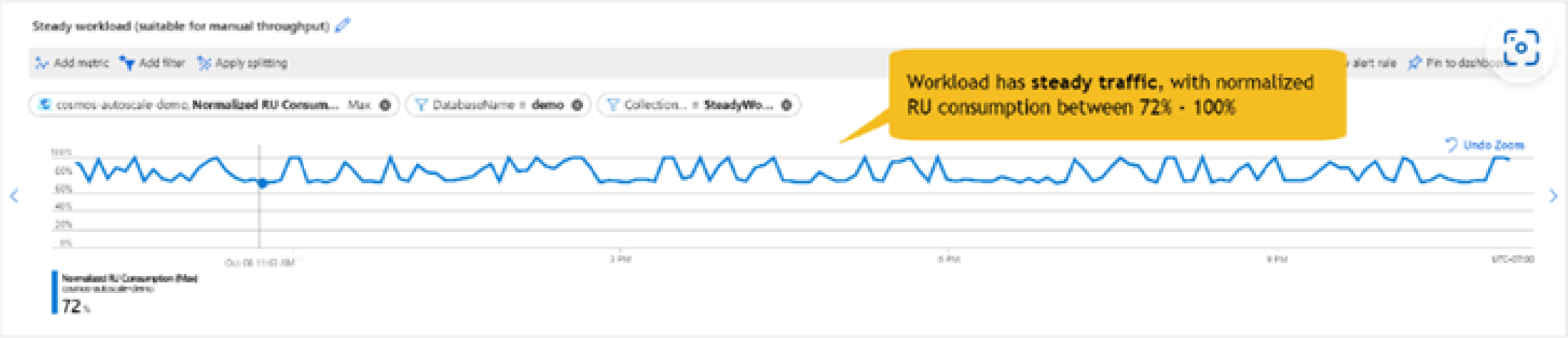
Serverless – це модель, де ви не задаєте фіксовану кількість RU/s. Замість цього ви платите лише за фактичні запити, які обробляються системою.

Переваги:

Гнучкість: Ви не платите за неактивні ресурси. Якщо база даних не використовується.

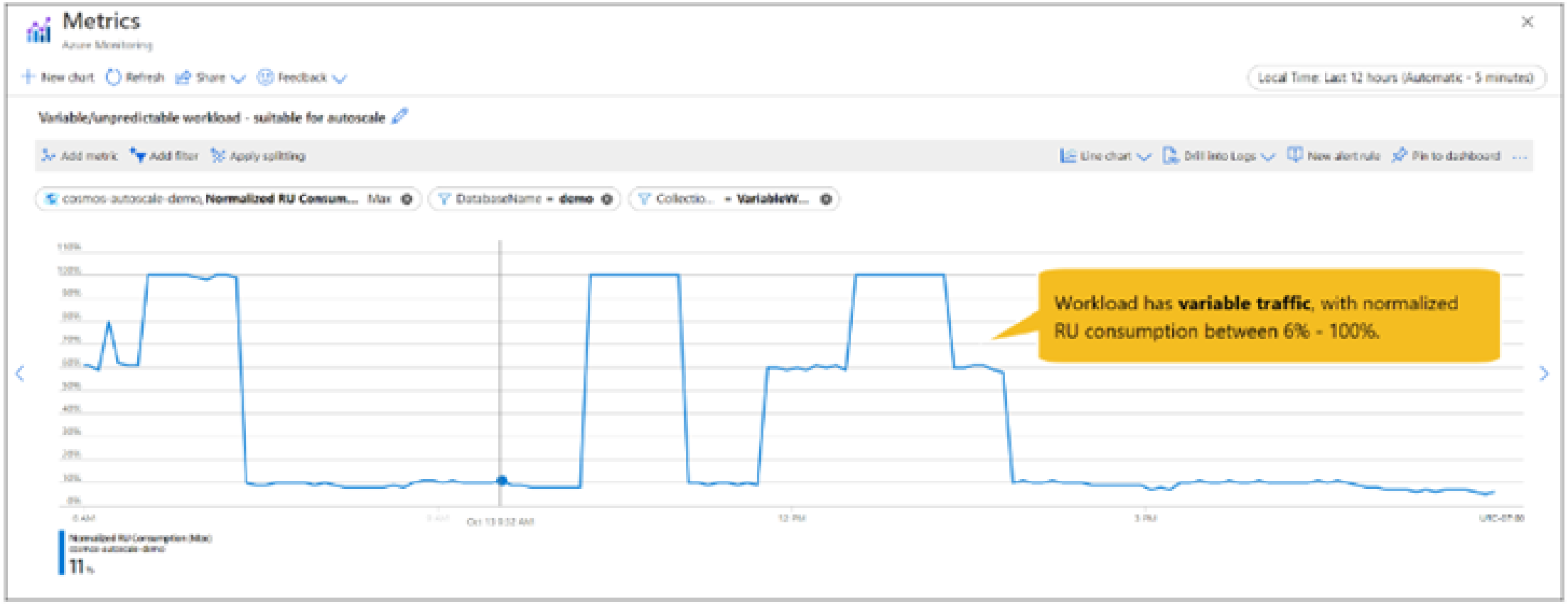
Недоліки:
Невизначені витрати: Якщо обсяг запитів різко зросте, ніж планували.
Підходить не для всіх сценаріїв: Для великих і постійно високих навантажень serverless може бути менш ефективним, оскільки вартість може зрости.

Стабільне навантаження



Наперед задається кількість ОЗ/с, яке не змінюється до тих пір поки, ви це не зміните самостійно.

Робоче навантаження зі змінним трафіком



Підготовлена максимальна пропускна здатність для автомасштабування: 4000 ОЗ/с (з масштабуванням від 400 до 4000.

Година 1: система масштабувала пропускну здатність до максимального значення 3500 ОЗ/с.

Година 2: система масштабувала пропускну здатність до мінімального значення в 400 ОЗ/с (завжди 10 % від Tmax) через невикористання.

Фактори, які впливають, на рівень споживання ресурсів

Розмір елемента.

Кількість властивостей елемента.

Тип узгодженості даних. Суворі рівні узгодженості та узгодженості з обмеженим старінням споживають приблизно вдвічі більше ОЗ при виконанні операцій читання порівняно з іншими відстроченими рівнями узгодженості.

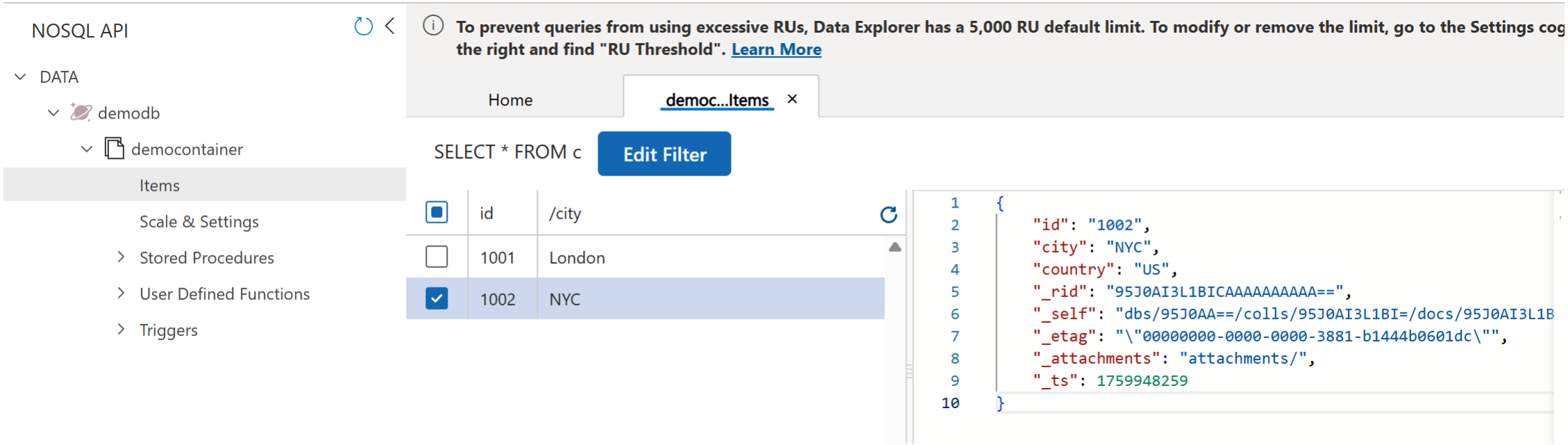
Тип операцій читання: точкове читання коштує менше ОЗ, ніж запити.

Шаблони запитів. Чинники, що впливають на вартість операцій запитів:

- ✓ Кількість результатів запиту.
- ✓ Кількість предикатів.
- ✓ Характер предикатів.
- ✓ Кількість функцій, що визначаються користувачем.
- ✓ Розмір вихідних даних.
- ✓ Розмір результуючого набору.
- ✓ Проекції.

Контейнери в Cosmos DB є основними одиницями зберігання та організації даних, які включають:

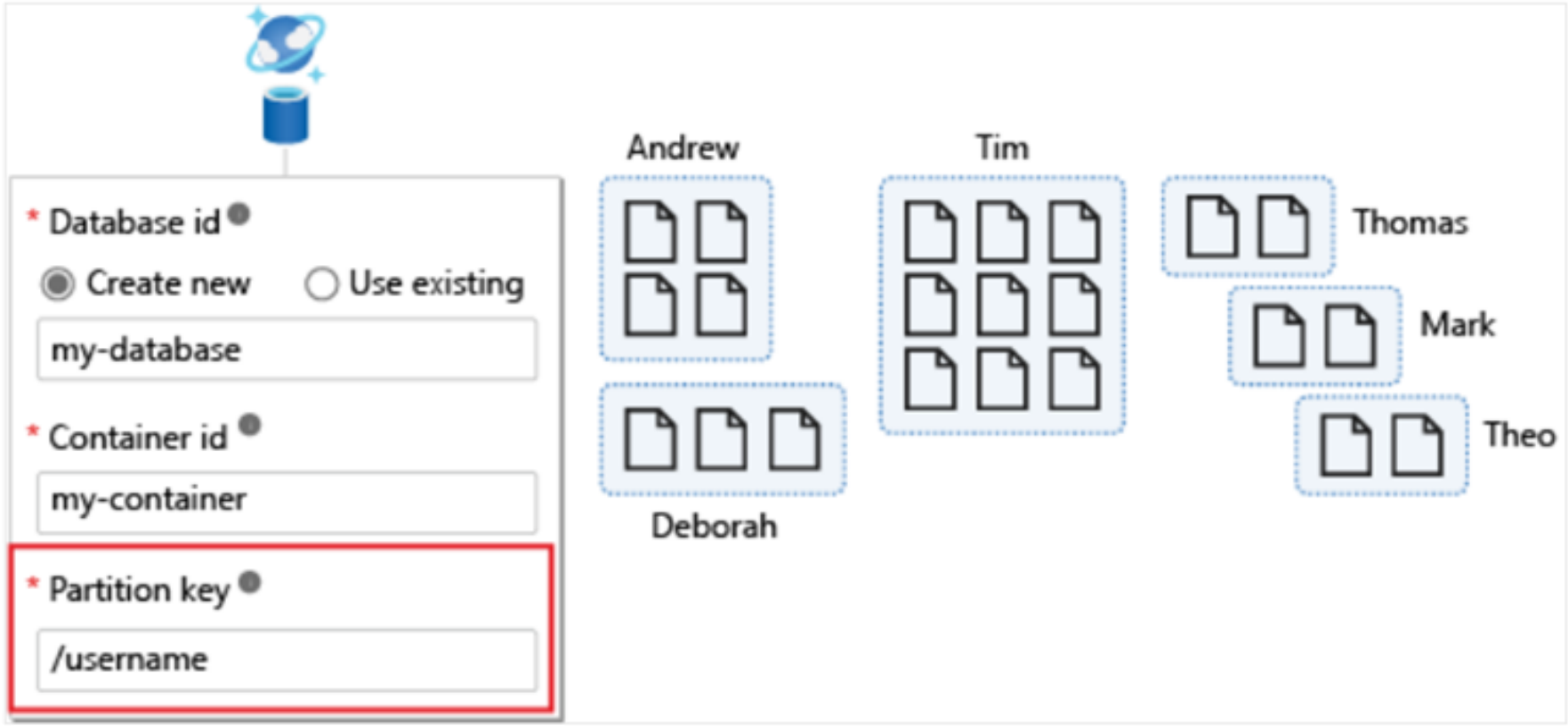
- ✓ Документи (JSON-об'єкти)
- ✓ Ключ розділу для масштабування
- ✓ Пропускну здатність (RU/s) для налаштування продуктивності
- ✓ Індексацію даних
- ✓ Системні метадані для управління версіями та доступом



Мета-дані (_rid, _self, _etag тощо) важливі для управління документами в базі даних. Вони використовуються для внутрішніх операцій Cosmos DB, таких як відновлення документів, перевірка на наявність змін, управління доступом. Мета-дані додають автоматично при збереженні документа.

- ✓ _rid (Resource ID): внутрішній ідентифікатор ресурсу в Cosmos DB
- ✓ _self (Self-Link): URL, який вказує на конкретний ресурс в Cosmos DB дозволяє отримати доступ до ресурсу (документа, контейнера і т.д.) за допомогою запитів або API.
- ✓ _etag: використовується для підтримки механізму контролю версій в Cosmos DB. допомагає уникнути конфліктів під час паралельних оновлень одного й того ж документа. Якщо документ змінюється, це поле оновлюється.
- ✓ _attachments: шлях до можливих вкладених файлів або додатків, які можуть бути прикріплені до документа.
- ✓ _ts (Timestamp): час, коли документ був останній раз змінений або створений в форматі UNIX timestamp (кількість секунд, що пройшли з 1 січня 1970 року).
- ✓ Приклад: "_ts": 1759948164

Контейнер – це ще одна абстракція для всіх даних, що зберігаються з одним і тим самим ключем секції. Ключ секції визначається під час створення контейнера. У цьому прикладі контейнер має ключ секції /username.



Горизонтальне та вертикальне масштабування

Реляційні бази даних зазвичай збільшуються шляхом збільшення розміру віртуальної машини або обчислювального середовища, де вони розміщені. В Cosmos DB дані одного контейнера розподіляються по різних «логічних секція» для забезпечення масштабування даних. Логічні секції формуються з урахуванням значення ключа секції (partition key), який є у кожного елемента у контейнері. Наприклад, для контейнеру, який включає «сутності» як показано в таблиці 1 та ключ секції: “city” – буде створено 4 логічні секції.

«сутності» items	Ключ секції – “city”
{ "id": "1001", "city":"London", "country":"UK" } { "id": "1002", "city": "NYC", "country": "US", } { "id": "1003", "city":"Paric", "country":"France" } { "id": "1004", "city": "Rome", "country": "Italy", }	

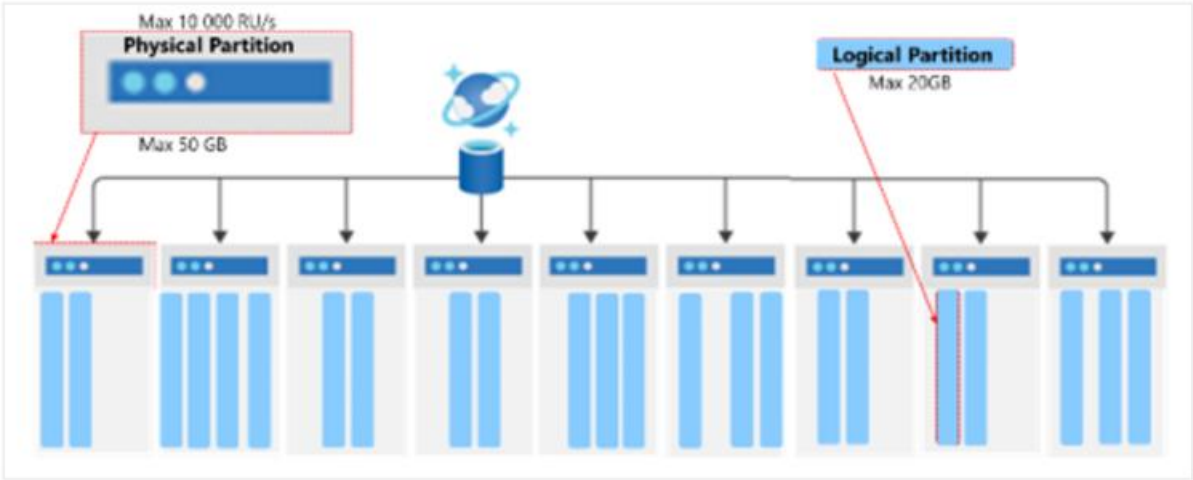
Крім ключа секції, що визначає логічну секцію, кожен елемент в контейнері також має ідентифікатор, який є унікальним в межах логічної секції. Поєднання ключа секції та ідентифікатора елемента створює індекс, що однозначно визначає елемент. Щоб збільшити масштаб бази даних NoSQL, потрібно додати додаткові сервери або вузли. Ці вузли також називаються **фізичними секціями** Cosmos DB.

Один контейнер може мати багато логічних секцій з однією фізичною. Фізичні секції створюють відповідно правил платформи Azure, а їх кількість залежить від характеристик:

1. Підготовлена пропускна спроможність. (Кожна окрема фізична секція може забезпечити пропускну здатність до 10 000 одиниць запитів за секунду.) Обмеження 10 000 одиниць запитів за секунду для фізичних секцій передбачає, що для логічних секцій також встановлено обмеження 10 000 одиниць запитів за секунду, оскільки кожна секція зіставлена лише з однією фізичною секцією.
2. Загальне сховище даних (у кожній фізичній секції може зберігатись до 50 ГБ даних).

Загальна кількість фізичних секцій у контейнері не обмежена.

Логічні секції об’єднані в фізичні розділи



Гаряче секціонування (Hot Partition) — це коли одна логічна секція отримує занадто багато даних або запитів, і як результат:

- ✓ починає перевантажувати фізичну секцію (сервер),
- ✓ знижується продуктивність,
- ✓ порушується баланс навантаження,
- ✓ порушується принцип масштабованості.

Гарячі секції сховища

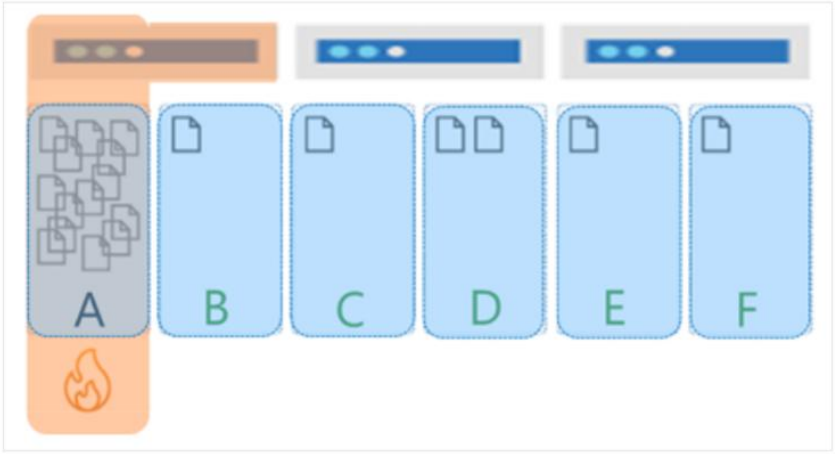
Як приклад розглянемо багатоклієнтську програму, яка використовує TenantId як ключ секції з п'ятьма клієнтами: від A до F. Клієнти B, C, D та E, D є маленькими. Клієнт A росте і швидко досягає межі 20 ГБ для своєї секції. В цьому випадку нам потрібен інший ключ секції, який розподілятиме обсяг сховища по кількох логічних секціях.

Клієнт A швидко розростається і досягає 20 ГБ – це максимальний обсяг даних для однієї логічної секції в Cosmos DB.

TenantId = "A" – це одна логічна секція.

Ця секція стає гарячою:

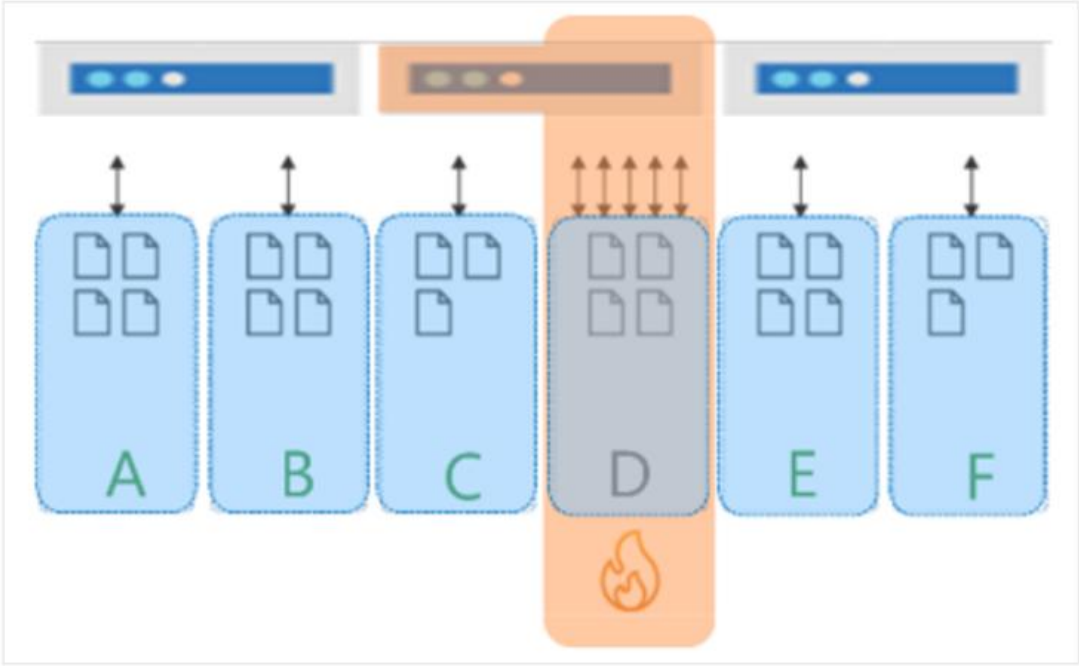
- ✓ перевищує ліміт у 20 ГБ,
- ✓ викликає більше запитів (високе навантаження),
- ✓ система вже не може розподілити дані для цього клієнта на інші фізичні секції.
- ✓ Інші клієнти – недовикористовують ресурси, бо їхні секції "холодні".



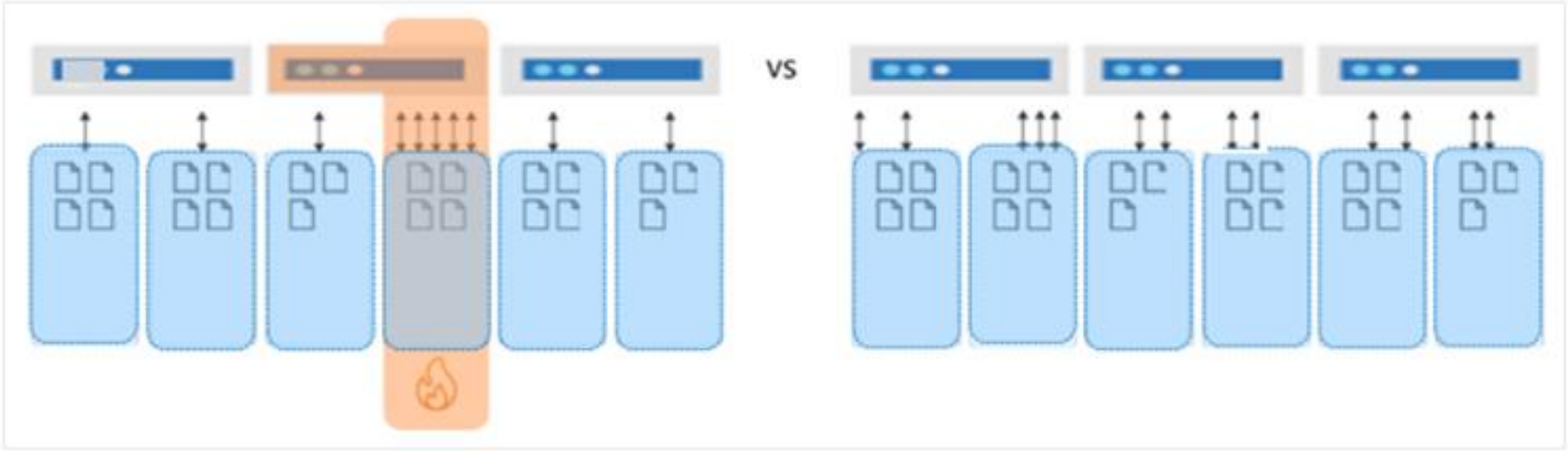
Гаряче секціонування пропускної спроможності

Наприклад, якщо у вас є контейнер з 30 000 ОЗ/с, робоче навантаження буде розподіллено між трьома фізичними секціями для тих самих шести клієнтів вище. Таким чином, кожна фізична секція отримує 10 000 ОЗ/с.

Якщо клієнт D споживає всі 10 000 ОЗ/с, його частота запитів буде обмежена, оскільки він може користуватися пропускної спроможністю, виділеної інших секцій. Це призводить до зниження продуктивності клієнтів **C і D (об’єднані в одну фіз секцію)** та невикористаної ємності обчислень в інших фізичних секціях та клієнтах, що залишилися. Зрештою, цей ключ секції призводить до створення структури бази даних, в якій робоче навантаження програми не може масштабуватися.



Якщо дані та запити розподіляються рівномірно, це гарантує, що зі зростанням бази даних буде забезпечено повне використання обсягу сховища та пропускної спроможності. Результатом буде максимально можлива продуктивність та підвищена ефективність. Загалом структура бази даних масштабуватиметься.



Як уникнути гарячих розділів?

При моделюванні даних для Azure Cosmos DB важливо, щоб вибраний ключ секції видавав рівномірний розподіл даних та запитів між фізичними секціями у контейнері.

Для розглянутого приклада:

Використовувати складніший ключ, наприклад - "partitionKey": "/tenantId-orderDate" або "partitionKey": "/tenantId/userId"

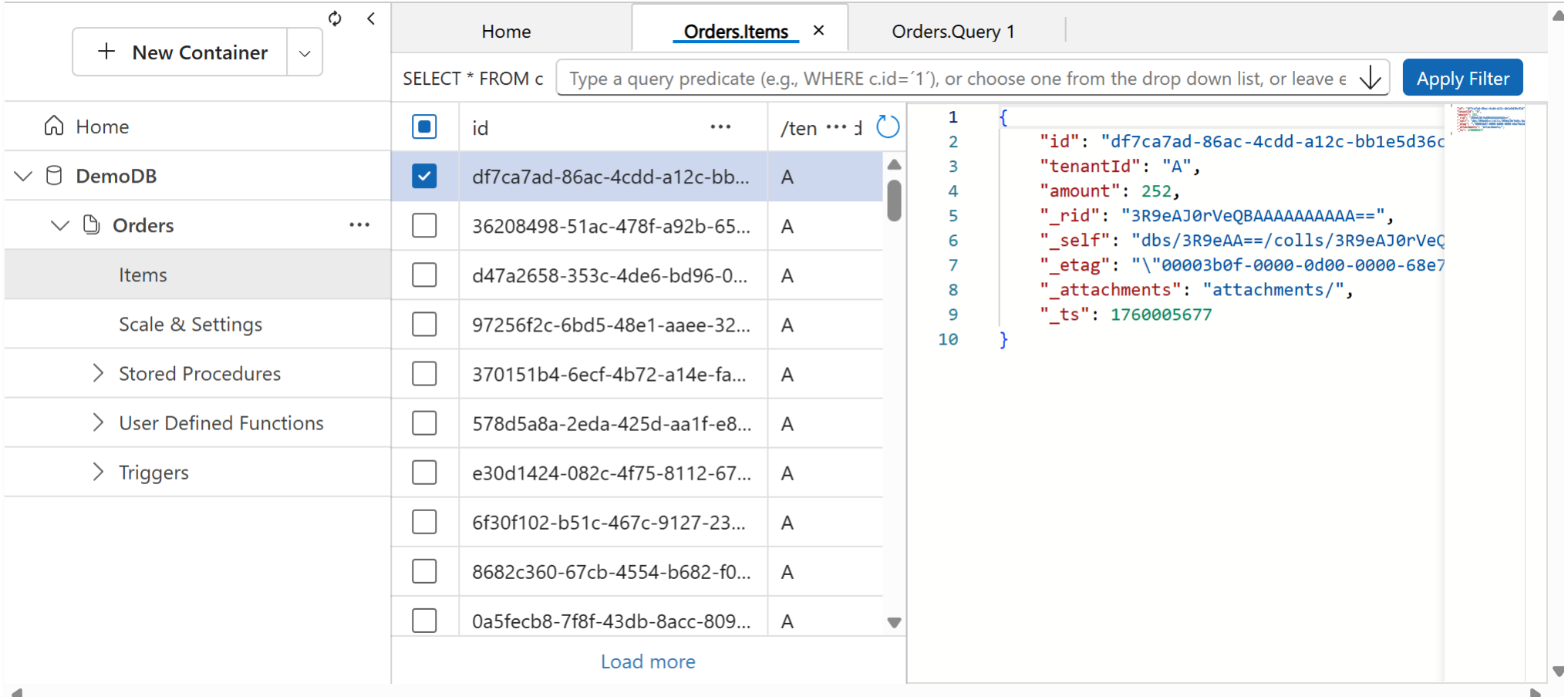
Це дозволяє розподілити навантаження на різні фізичні секції тобто дозволяє Cosmos DB масштабувати базу горизонтально.

Якщо не тестувати проект бази даних під навантаженням під час розробки, факт невдалого вибору ключа секції ***може залишатися непоміченим*** до тих пір, поки програма не опиниться в робочому середовищі і не буде записано багато даних.

Розглянемо приклад

Хід роботи

Створити контейнер
База даних: DemoDB
Контейнер: Orders
Partition key: /tenantId
Завантажимо дані, де "tenantId": "A" утворить "гарячу секцію" (90% — tenantId = "A")



Статистики запитів до документів

Параметр	Пояснення
Partition key range id	Ідентифікатор фізичної секції, яка обробила цей запит
Retrieved document count = 100	Скільки документів система отримала з диску
Retrieved document size (in bytes) = 15150	Розмір усіх документів, витягнутих з Cosmos DB (сирі)
Output document count = 100	Скільки документів реально повернуто вам у результаті запиту
Output document size (in bytes) = 27940	Розмір документів, повернутих у відповіді (може бути більшим, бо додаються метадані)
Index hit document count = 100	Скільки документів були знайдені через індекси (а не повним скануванням)
Index lookup time (ms) = 0	Скільки часу витрачено на пошук через індекс
Document load time (ms) = 0.23	Скільки часу витрачено на завантаження документів із диску
Query engine execution time (ms) = 0.04	Час, витрачений на обробку запиту всередині Cosmos DB
System function execution time (ms) = 0	Скільки часу витрачено на вбудовані функції (наприклад, IS_DEFINED())
User defined function execution time (ms) = 0	Час на виконання користувацьких функцій (UDF)
Document write time (ms) = 0.14	Якщо це був би запис, скільки часу на нього пішло. Але у вас був SELECT, тому показує 0 або статистику попередньої операції

Паралельні запити до даних контейнеру - Аналіз ресурсів при запиті до документа

```
Запускаємо паралельні запити до Cosmos DB...

[14:39:49] TenantId: C | Docs: 23 | RU: 3.40 | Time: 457.37 ms
[14:39:49] TenantId: E | Docs: 17 | RU: 3.24 | Time: 457.66 ms
[14:39:49] TenantId: D | Docs: 25 | RU: 3.46 | Time: 460.13 ms
[14:39:49] TenantId: F | Docs: 18 | RU: 3.27 | Time: 457.73 ms
[14:39:49] TenantId: B | Docs: 17 | RU: 3.24 | Time: 561.12 ms
[14:39:49] TenantId: A | Docs: 900 | RU: 26.58 | Time: 1284.62 ms

Завершено.
```

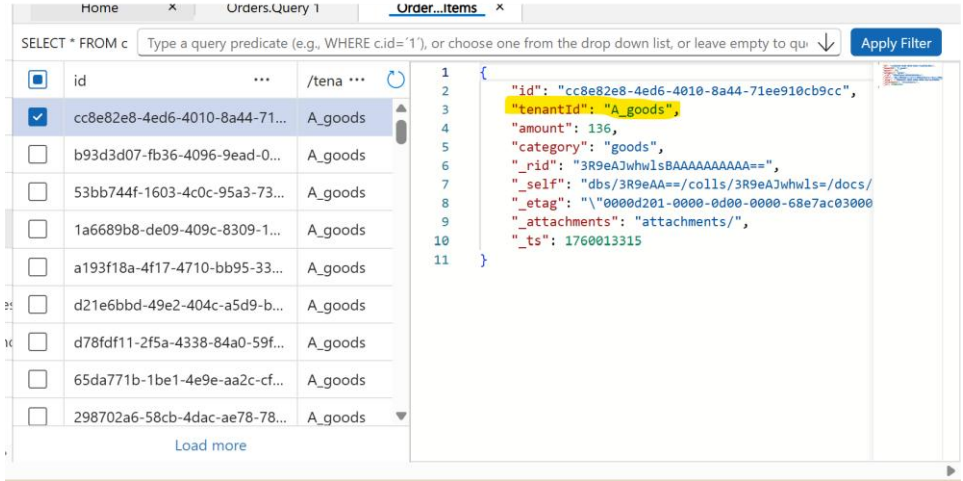
Секція TenantId = A – гаряча

Вона обробляє у 35–50 разів більше документів.

Спожила 26.58 RU, тоді як інші споживають ~3.2–3.5 RU.

Час запиту майже втричі вищий за середній.

Рішення: створити складний ключ.




```
Запускаємо паралельні запити до Cosmos DB...

[14:39:49] TenantId: C | Docs: 23 | RU: 3.40 | Time: 457.37 ms
[14:39:49] TenantId: E | Docs: 17 | RU: 3.24 | Time: 457.66 ms
[14:39:49] TenantId: D | Docs: 25 | RU: 3.46 | Time: 460.13 ms
[14:39:49] TenantId: F | Docs: 18 | RU: 3.27 | Time: 457.73 ms
[14:39:49] TenantId: B | Docs: 17 | RU: 3.24 | Time: 561.12 ms
[14:39:49] TenantId: A | Docs: 900 | RU: 26.58 | Time: 1284.62 ms

Завершено.
```

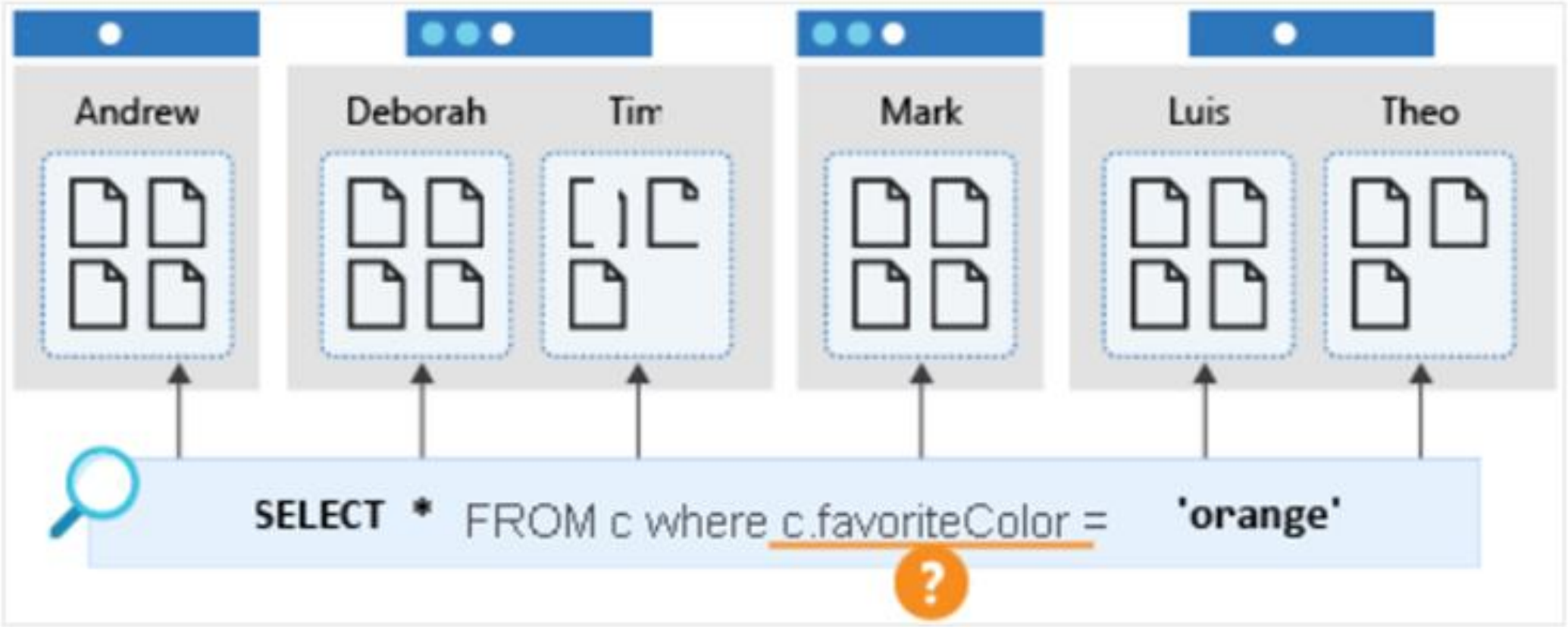
```
Паралельні запити до Cosmos DB...

[15:58:25] TenantId: D | Docs: 12 | RU: 3.12 | Time: 506.91 ms
[15:58:25] TenantId: E | Docs: 27 | RU: 3.52 | Time: 507.8 ms
[15:58:25] TenantId: F | Docs: 19 | RU: 3.30 | Time: 506.4 ms
[15:58:25] TenantId: B | Docs: 17 | RU: 3.25 | Time: 608.43 ms
[15:58:25] TenantId: C | Docs: 25 | RU: 3.46 | Time: 607.61 ms
[15:58:25] TenantId: A | Docs: 900 | RU: 26.88 | Time: 960.78 ms

Завершено.
```

Міжсекційні запити

Запит, який фільтрує за іншою властивістю, наприклад favoriteColor, викликає "розкид" по всіх секціях у контейнері. Це також називається міжсекційним запитом. Такий запит працюватиме нормально, якщо контейнер невеликий і займає лише одну секцію. Однак у міру зростання контейнера та збільшення числа фізичних секцій цей запит буде виконуватися повільніше і стане дорожче, оскільки йому потрібно перевірити кожну секцію, щоб отримати результати, незалежно від того, чи містяться у фізичній секції пов'язані із запитом дані.



Приклад - Міжсекційні запити

Час, витрачений на обробку запиту всередині Cosmos DB збільшився

	C	D	E	F	G	H	I	J	K
	SELECT * FROM c where c.tenantId = 'A_goods'								
unt	Retrieved document size (in bytes)	Output document count	Output document size (in bytes)	Index hit document count	Index looku	Document load time (ms)	Query engine execution time (ms)	System fun	User c
00	17253	100	30446	100	0.05	0.3	0.05	0	
	SELECT * FROM c where c.category = 'goods'								
	Retrieved document size (in bytes)	Output document count	Output document size (in bytes)	Index hit document count	Index looku	Document load time (ms)	Query engine execution time (ms)	System fun	User c
	17253	100	30446	100	0.05	0.28	0.06	0	

Бібліотека azure.cosmos є офіційним Python SDK від Microsoft для взаємодії з API for NoSQL в Azure Cosmos DB. Вона надає об'єктно-орієнтований інтерфейс для керування базами даних, контейнерами та елементами (документами).
Ключова ієрархія об'єктів:

- CosmosClient: Головний клієнт, точка входу для будь-яких операцій.
- DatabaseProxy: Представлення конкретної бази даних.
- ContainerProxy: Представлення контейнера, де зберігаються ваші дані.
- Елемент (Item): Ваші дані, зазвичай у форматі словника (dict) Python.

Встановлення та Ініціалізація

```
pip install azure-cosmos
```

Для підключення знадобляться Endpoint URL та Primary Key вашого облікового запису Cosmos DB.

```
from azure.cosmos import CosmosClient, exceptions

# --- Конфігурація ---
ENDPOINT = "https://your-account-name.documents.azure.com:443/"
KEY = "your-primary-key"

# --- Ініціалізація клієнта ---
# Рекомендується створювати один екземпляр CosmosClient (Singleton pattern)
client = CosmosClient(url=ENDPOINT, credential=KEY)
```

Основні Класи та їх Методи

CosmosClient - Клас, що представляє ваш обліковий запис Cosmos DB. Він є коренем для всіх операцій.
Основні методи:
get_database_client(database_id): Отримує об'єкт DatabaseProxy для існуючої бази даних, не перевіряючи її наявність на сервері.

```
db_client = client.get_database_client("MyDatabase")
```

create_database(id, ...): Створює нову базу даних. Якщо база даних з таким id вже існує, викине виняток exceptions.CosmosResourceExistsError.

```
try:
    new_db = client.create_database(id="MyNewDatabase")
    print("База даних створена.")
except exceptions.CosmosResourceExistsError:
    print("База даних вже існує.")
```

create_database_if_not_exists(id, ...): Створює базу даних, тільки якщо вона не існує. Зручно, але робить додатковий запит на перевірку.
list_databases(): Повертає ітератор для перегляду всіх баз даних в обліковому записі.

```
for db in client.list_databases():
    print(db['id'])
```

delete_database(database_id): Видаляє базу даних

```
client.delete_database("MyNewDatabase")
```

2.2. DatabaseProxy - Представляє конкретну базу даних. Використовується для керування контейнерами.
Основні методи:
get_container_client(container_id): Отримує об'єкт ContainerProxy для існуючого контейнера.

```
container_client = db_client.get_container_client("MyContainer")
```

create_container(id, partition_key, ...): Створює новий контейнер. partition_key є обов'язковим параметром.

```
from azure.cosmos import PartitionKey
try:
    container = db_client.create_container(
        id="Products",
        partition_key=PartitionKey(path="/categoryId") # Шлях до ключа секціонування
    )
    print("Контейнер створений.")
except exceptions.CosmosResourceExistsError:
    container = db_client.get_container_client("Products")
    print("Контейнер вже існує.")
```

- ✓ create_container_if_not_exists(id, partition_key, ...): Створює контейнер, якщо він не існує.
- ✓ list_containers(): Повертає ітератор для перегляду всіх контейнерів у базі даних.
- ✓ delete_container(container_id): Видаляє контейнер.
- ✓ query_containers(query, parameters=None): Виконує SQL-подібний запит для пошуку контейнерів.
- ✓ read(): Читає властивості самої бази даних.

2.3. ***ContainerProxy*** **-** клас для роботи з даними (елементами). Він надає методи для операцій CRUD (Create, Read, Update, Delete) та запитів.

Методи для роботи з елементами (CRUD):

create_item(body): Створює новий елемент. body – це словник Python. Словник повинен містити ключ id та ключ, вказаний як partition_key.

```
new_product = {
   "id": "PROD001",
   "categoryId": "Laptops",
   "name": "SuperBook Pro",
   "price": 1999.99
}
container_client.create_item(body=new_product)
```

read_item(item, partition_key): Читає один елемент за його id та partition_key. Це найшвидший спосіб отримати елемент.

```
item_id = "PROD001"
category_id = "Laptops"
product = container_client.read_item(item=item_id, partition_key=category_id)
print(product['name'])
```

read_all_items(): Читає всі елементи в контейнері. Не рекомендується для великих контейнерів, оскільки може споживати багато RU/s.
upsert_item(body): Оновлює елемент, якщо він існує, або створює новий, якщо не існує. Дуже зручна функція.

Cosmos DB Emulator

https://learn.microsoft.com/en-us/azure/cosmos-db/emulator

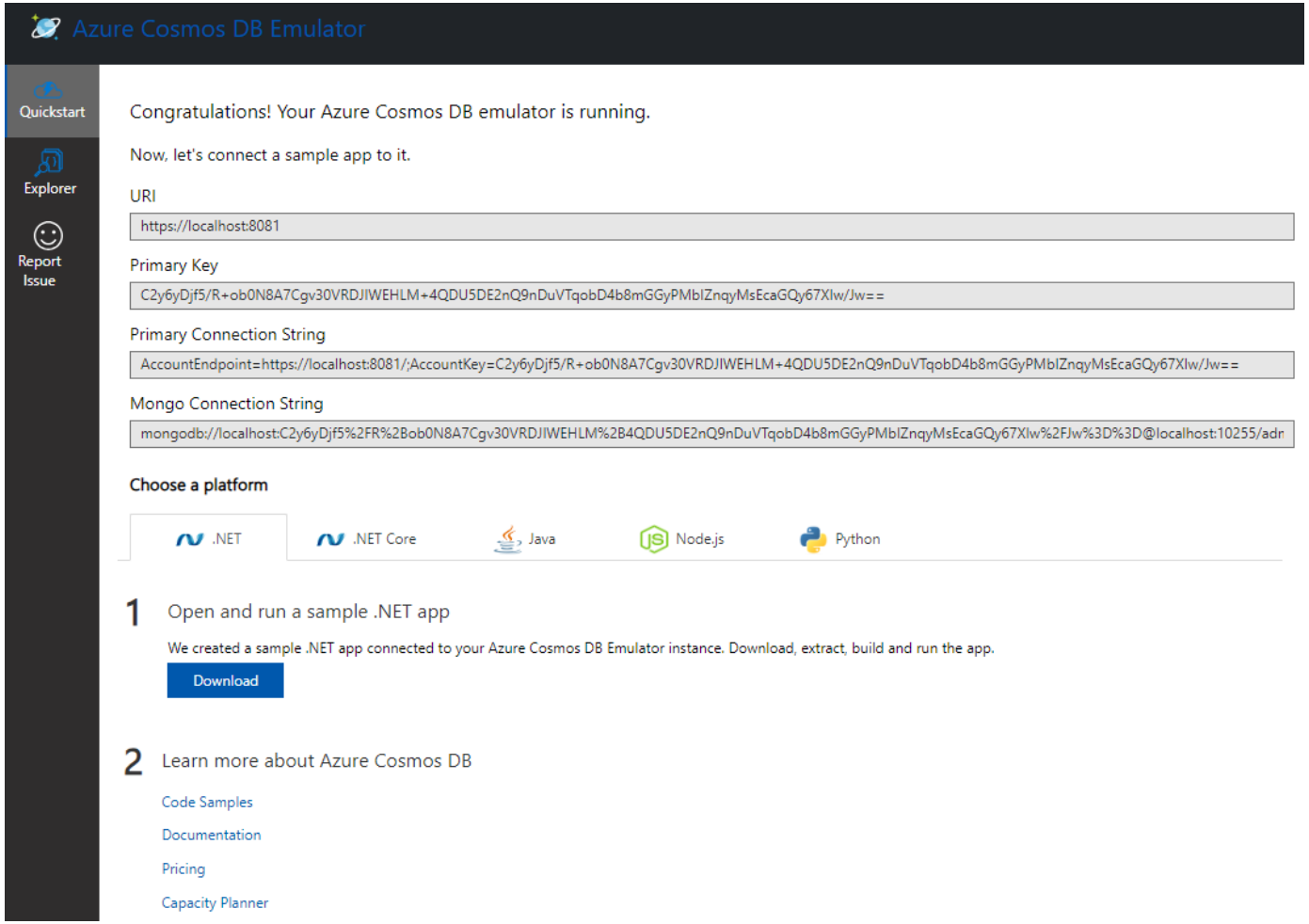
Емулятор забезпечує середовище у робочому просторі розробника.
Ключові відмінності у функціональності між емулятором і еквівалентною хмарною службою:

- Емулятор підтримує лише надану пропускну здатність. Емулятор не підтримує безсерверну пропускну здатність, тобто створюють модель розгортання бази даних, яка НЕ дозволяє обробляти запити без попереднього виділення фіксованих ресурсів для пропускної здатності.
- Під час запуску емулятор використовує добре відомий ключ. Ви не можете повторно згенерувати ключ для запущеного емулятора. Щоб використовувати інший ключ, потрібно запустити емулятор із указаним власним ключем.
- Емулятор не підтримує «надлишковість» даних
- Емулятор ідеально підтримує до 10 контейнерів фіксованого розміру зі швидкістю 400 RU/c або 5 контейнерів необмеженого розміру.
- Емулятор обмежує довжину унікального ідентифікатора елементів до 254 символів.
- Емулятор підтримує максимум п’ять операторів JOIN на запит.

Azure Cosmos DB emulator

https://learn.microsoft.com/en-us/azure/cosmos-db/how-to-develop-emulator?tabs=windows%2Cpython&pivots=api-nosql

cd C:\Program Files\Azure Cosmos DB Emulator
Microsoft.Azure.Cosmos.Emulator /port=8081



CosmosClient Class

CosmosClient(url: str, credential: TokenCredential | str | Dict[str, Any], consistency_level: str | None = None, **kwargs)

URL-адреса облікового запису Cosmos DB
credential - ключем облікового запису

Методи
create_database - Створіть нову базу даних із заданим ID (ім’ям).
create_database_if_not_exists - Створіть базу даних, якщо вона ще не існує.
delete_database - Видалити базу даних із вказаним ідентифікатором (ім’ям).
get_database_client – Отримати наявну базу даних (всі контейнери)
from_connection_string - Створіть екземпляр CosmosClient на основі Connection string
get_database_account - Отримати інформацію про обліковий запис бази даних.

Створити контейнер create_container(id, partition_key, indexing_policy=None, default_ttl=None, populate_query_metrics=None, offer_throughput=None, unique_key_policy=None, conflict_resolution_policy=None, **kwargs)

Параметри:
id - Вказує унікальний ідентифікатор контейнера всередині бази даних.
partition_key: Це обов'язковий параметр, який визначає, як дані будуть розподілятися між розділами (partitions) для горизонтального масштабування.
PartitionKey(path='/id', kind='Hash’) - розподіл даних по розділам (розділам) відбувається за допомогою хешування значень ключа
indexing_policy - керує індексуванням документів внутри розділів, де використовується ключ розділу для розбиття даних. Індексація – це механізм, який дозволяє швидше перейти і відфільтрувати дані за ключами або іншими атрибутами.
unique_key_policy - задання унікальних ключів у колекціях баз даних. Це означає, що в межах одного розділу не може бути двох документів з одиничними значеннями для полів, визначених як унікальних.
offer_throughput: Визначає кількість ресурсів, виділених для контейнера. Це може бути або кількість RU (Request Units) на секунду, або авто-масштабовані RU. Якщо не вказано, буде використовуватися параметри по замовчуванню для бази даних або автоматичне управління.
default_ttl: Визначає час життя (TTL – Time to Live) документів в контейнері в секундах. Після цього часу документ буде автоматично видалений.
Якщо значення встановлено в 0, то TTL вимкнено. Якщо вказати значення >0, документи автоматично видаляються через зазначений час після їх створення.

Властивості контейнера

etag — це унікальний ідентифікатор (строка), який автоматично генерується та присвоюється кожному документу у Cosmos DB при його створенні або зміні. Кожного разу, коли документ оновлюється, його etag також оновлюється, що робить його своєрідною "відміткою" версії документа.

Основні особливості etag:
Контроль версій:
Кожна зміна документа призводить до зміни його etag. Це дозволяє відслідковувати, чи був документ змінений між запитами на читання і запис.

etag дозволяє запобігати конфліктам при одночасних оновленнях документа. Перед оновленням або видаленням документа можна перевірити його поточне значення etag, щоб переконатися, що ніхто інший не змінив його після того, як ви його прочитали.
Якщо документ змінено іншим процесом, то значення etag зміниться, і ваш запит на оновлення або видалення може бути відхилений, щоб уникнути конфлікту.

enable_cross_partition_query

За замовчуванням: Запити в Cosmos DB обмежуються одним розділом, якщо параметр enable_cross_partition_query не встановлений в True. Це покращує продуктивність і зменшує затримки для запитів, оскільки обробка даних відбувається лише в межах одного розділу.

Перетин кількох розділів: Якщо ваш запит має охоплювати більше одного розділу (наприклад, для отримання даних без фільтрації за partition key або для агрегації даних по кількох розділах), ви повинні встановити параметр enable_cross_partition_query=True. Це дозволяє запиту виконуватися по кількох розділах, але може збільшити затрати на запит (по RU/s) та час виконання.

Коли використовувати enable_cross_partition_query:
Запити без фільтрації за partition key: Якщо ви не вказуєте значення для ключа розділу і ваші дані зберігаються в різних розділах, вам необхідно встановити цей параметр для отримання даних з усіх розділів.
Агрегації: Запити, що вимагають виконання агрегацій (наприклад, COUNT, SUM, AVG) часто потребують доступу до кількох розділів.
Великі обсяги даних: Якщо ваші дані розподілені по багатьох розділах, і вам потрібно виконати запит, який охоплює більше одного розділу.
Мінуси використання:
Вищі затрати RU: Крос-розділові запити часто вимагають більше ресурсів (Request Units), оскільки вони охоплюють кілька фізичних розділів для виконання запиту.
Збільшений час виконання: Запити, що виконуються по кількох розділах, можуть займати більше часу, ніж запити по одному розділу, через те, що потрібно звертатись до кількох частин бази даних.

upsert_item(body, populate_query_metrics=None, pre_trigger_include=None, post_trigger_include=None, **kwargs)

body: - документ, якмй треба додати або оновити
partition_key (опціональний): Це ключ партиції, який використовується для визначення розташування документа в кластері даних.
disable_automatic_id_generation (опціональний, за замовчуванням False): Цей параметр вказує, чи потрібно автоматично генерувати значення для поля id, якщо воно відсутнє у переданому документі. Якщо встановити в True, функція не генеруватиме id автоматично, і документ повинен мати явно вказане поле id.

Питання

1. Які існують рівні узгодженості в Azure Cosmos DB та чим вони відрізняються?
2. Чому рівень Strong Consistency забезпечує найвищу узгодженість, але впливає на продуктивність і глобальну доступність?
3. У яких сценаріях доцільно використовувати Eventual Consistency, і які ризики це створює для читання даних?
4. Що означає властивість read-your-own-writes і для яких рівнів узгодженості вона гарантується?
5. У чому полягає суть режиму Provisioned Throughput Mode, і як вимірюється пропускна здатність Cosmos DB?
6. Які переваги та недоліки використання заздалегідь виділеного пропускного режиму (Provisioned Throughput) для стабільних навантажень?
7. Як працює Serverless Mode, і в яких випадках він є оптимальним вибором?
8. Чому Serverless Mode може бути дорожчим для високих постійних навантажень порівняно з Provisioned Throughput?
9. Що таке RU (Request Units) і як їх споживання відрізняється між цими двома режимами?
10. Що таке логічна секція (logical partition) і яку роль відіграє Partition Key?
11. У чому полягає відмінність між логічною та фізичною секцією (physical partition)?
12. Яким чином Cosmos DB автоматично масштабує фізичні секції при збільшенні обсягу даних або навантаження?
13. Які критерії слід враховувати під час вибору ефективного Partition Key?
14. Які основні об’єкти становлять ієрархію SDK azure.cosmos (наприклад, CosmosClient, Database, Container) і яку роль виконує кожен із них?
15. Як відбувається взаємодія з документами за допомогою контейнера (Container) у бібліотеці azure.cosmos?
16. Яким чином створюється підключення до Cosmos DB у Python через CosmosClient та які параметри є обов’язковими?