Universitatea din București
Facultatea de Matematică și Informatică
Anul Universitar 2025-2026

PROIECT

~ Baze de date, de la NoSql la Vector DBs ~

T4 – Caching and data acceleration with Redis

Proiect realizat de: Mihăilă Denisa, Grupa 352
Nițe Dan-Alexandru, Grupa 344

Cuprins

0. Specificații Tehnice

Configurația Hardware

Model Procesor (CPU)	**11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz**
Memorie RAM	**16 GB DDR4**
Stocare	**512 GB NVMe SSD**
Sistem de Operare	**Windows 11**

Configurația Containerelor

Redis Container:

- **Imagine:** redis:latest
- **Limită Memorie:** 300 MB (--maxmemory 300mb)

- **Politică Evacuare:** allkeys-lru (Șterge cele mai vechi chei când se atinge limita de 300MB).
- **Persistență:** RDB face un point-in time snapshot la fiecare 60 de secunde dacă există cel puțin o modificare.

MongoDB Container:

- **Imagine:** mongo:latest
- **Port Map:** 27017:27017
- **Volum:** Persistent (mongo_data:/data/db)

Versiuni utilizate

Categorie	Tehnologie / Librărie	Versiune	Rol în Arhitectură
Limbaj	**Python**	3.13.3	Limbajul principal pentru API și scripturi
Framework	**FastAPI**	0.109.0	Framework web asincron de înaltă performanță
Server	**Uvicorn**	0.27.0	Server ASGI pentru rularea aplicației Python
DB Client	**PyMongo**	4.6.1	Driverul oficial Python pentru MongoDB
Cache Client	**Redis-py**	5.0.1	Interfața Python pentru comunicarea cu Redis
Utilitar	**Faker**	22.5.1	Generarea datelor sintetice pentru populare (Seeding)
DevOps	**Docker Desktop**	27.4.0	Platforma de containerizare

1. Tema Proiectului

Proiectul are ca obiectiv implementarea unei arhitecturi software, capabilă să optimizeze performanța interogărilor, să gestioneze volume mari de date și să reducă solicitarea bazei de date principale a unei platforme de tip **E-commerce** (Studiu de caz: Magazin de Cadouri).

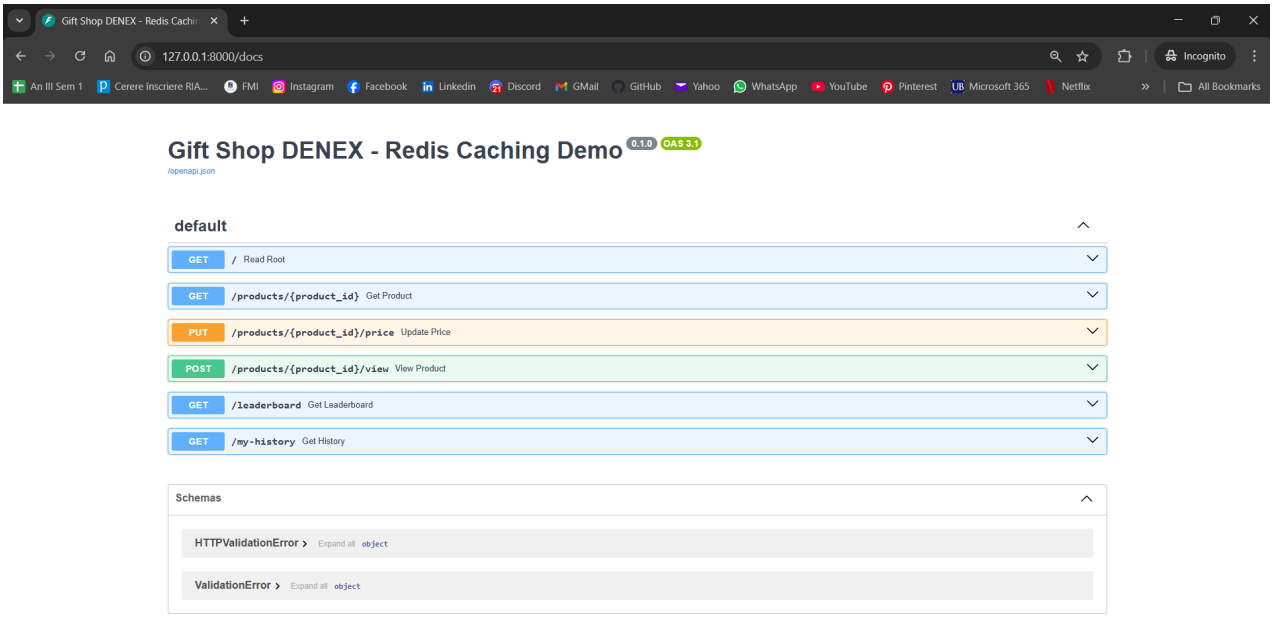
Soluția tehnică propusă integrează un strat de stocare de tip "In-Memory" (**Redis**) cu o bază de date persistentă NoSQL (**MongoDB**). Lucrarea analizează comparativ trei strategii de caching distincte: **Cache-Aside**, **Write-Through** și **Write-Behind**, evidențiind avantajele și compromisurile fiecăreia în ceea ce privește consistența datelor și viteza de procesare.

2. Arhitectura Sistemului

Pentru a ușura procesul de rulare a întregii arhitecturi și pentru a izola procesele distincte între ele. Întregul mediu de execuție este orchestrat prin intermediul **Docker**, configurația serviciilor și a rețelei interne fiind definită în fișierul *docker-compose.yml*.

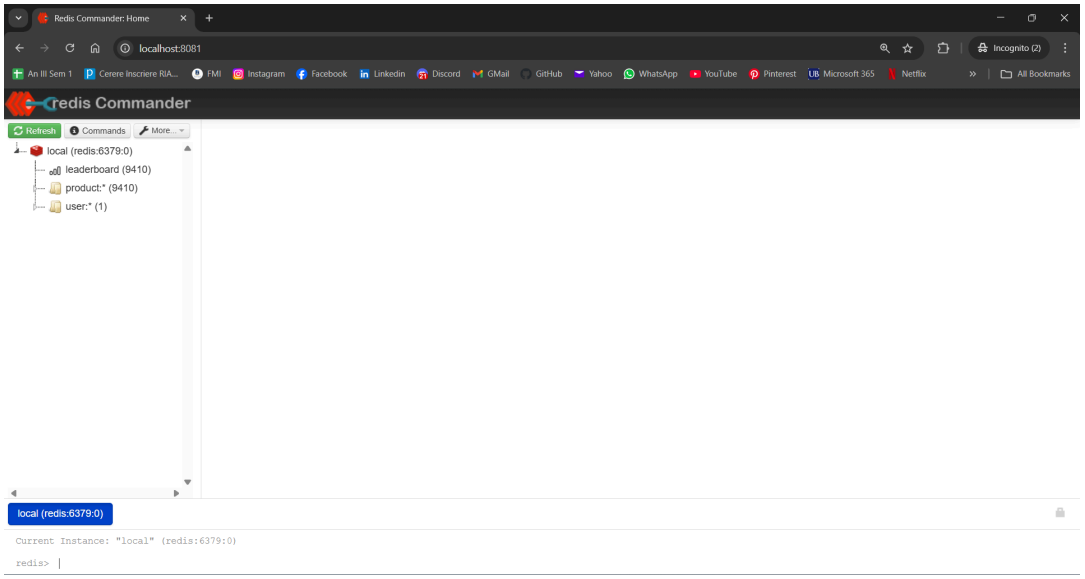
Sistemul este compus din patru servicii interconectate:

- **Nivelul de Aplicație (API Gateway & Business Logic)**
  Este implementat în Python, utilizând framework-ul FastAPI și serverul ASGI Uvicorn. Acest strat acționează ca punct central de control, având rolul de a:
  - Gestiona cererile HTTP ale clienților
  - Implementa cele trei strategii de caching
  - Coordona fluxul de date dintre memoria volatilă și stocarea persistent



Interfața Swagger UI (FastAPI) expunând endpoint-urile REST pentru gestionarea produselor și testarea strategiilor de caching.

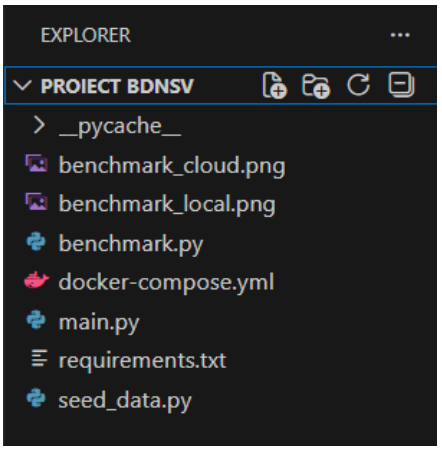
- **Nivelul de Cache (In-Memory Storage)**
  Este construit pe baza Redis, o bază de date care stochează datele accesate frecvent direct în memoria RAM. Rolul său este de a minimiza latența de citire și de a prelua încărcarea(load-ul) de pe baza de date principală.
- **Nivelul de Persistență (Persistent Storage)**
  Acesta utilizează MongoDB, o bază de date NoSQL orientată pe documente, ce permite reținerea datelor permanent pe disk, asigurând durabilitatea și integritatea datelor pe termen lung.
- **Nivelul de Monitorizare (UI)**
  Include serviciul **Redis Commander**, o interfață grafică web ce permite vizualizarea în timp real a structurilor de date din Redis.



Panoul de control Redis Commander, utilizat pentru monitorizarea cheilor și a structurilor de date stocate în memorie.

```
PS D:\Proiect BDNSV> docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED      STATUS                  PORTS                      NAMES
f03646b22fbf   rediscommander/redis-commander:latest   "/usr/bin/dumb-init …"   2 days ago   Up 2 hours (healthy)    0.0.0.0:8081->8081/tcp     bdnsv_redis_gui
6b134260c1eb   redis:latest                         "docker-entrypoint.s…"   2 days ago   Up 2 hours              0.0.0.0:6379->6379/tcp     bdnsv_redis
28278630b9df   mongo:latest                         "docker-entrypoint.s…"   2 days ago   Up 2 hours              0.0.0.0:27017->27017/tcp   bdnsv_mongo
```

Containerele active afișate în terminal prin comanda docker ps.

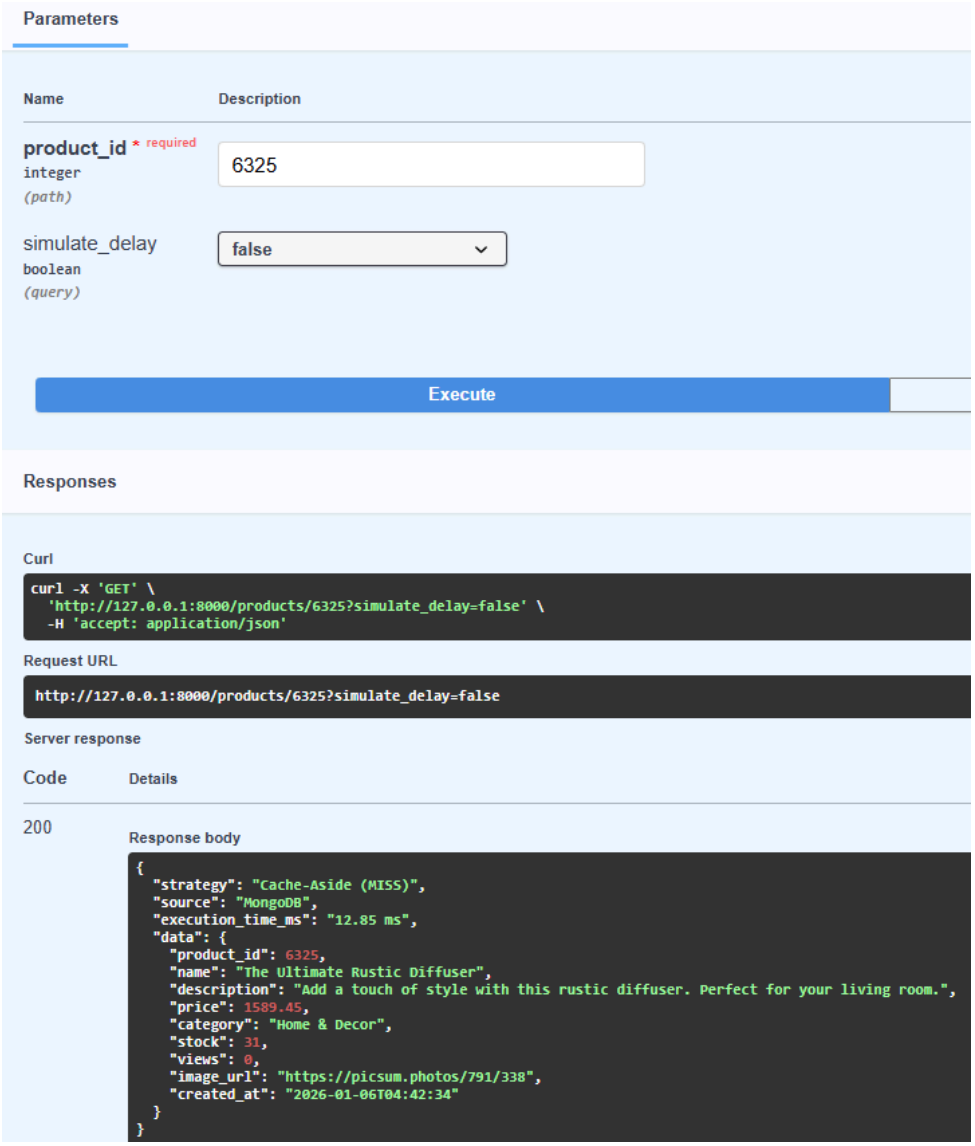


Organizarea fișierelor proiectului.

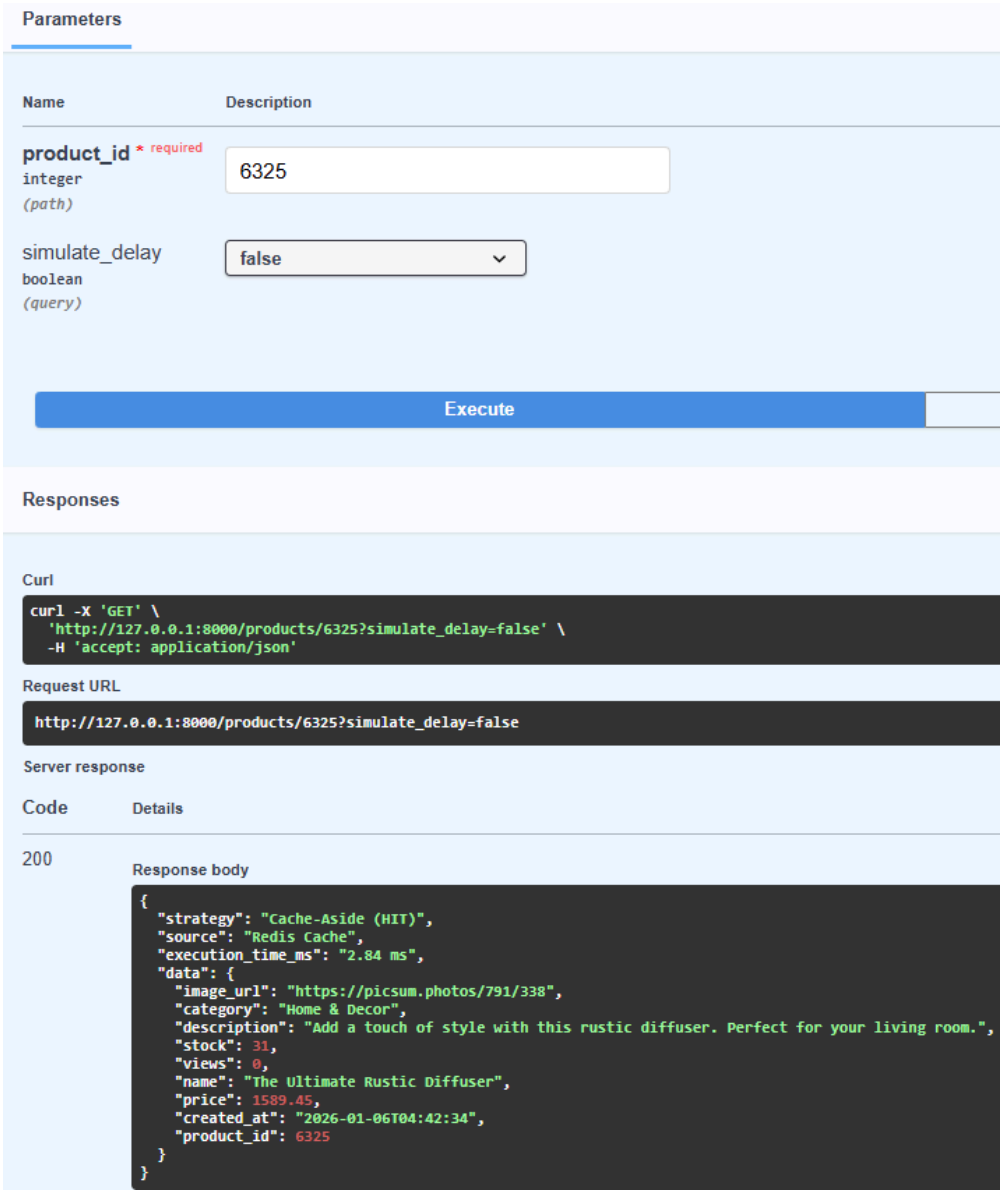
3. Strategiile de Caching implementate

Pentru a demonstra adaptabilitatea arhitecturii și a analiza impactul asupra performanței în scenarii diferite de utilizare, am implementat trei modele distincte de caching:

- **Cache-Aside (Lazy Loading)**: Această strategie este fundamentală pentru optimizarea operațiunilor de citire a detaliilor produselor (GET /products/{id}). Aplicația interoghează inițial cache-ul Redis. În cazul în care datele sunt găsite în cache ("HIT"), acestea sunt returnate instant. În cazul în care datele nu se află încă în cache ("MISS"), aplicația preia datele din MongoDB și le returnează clientului, adăugându-le de asemenea și în Redis, pentru o posibilă viitoare accesare mai rapidă. Pentru a preveni umplerea cache-ului cu date învechite, a fost implementat mecanismul Time-To-Live (TTL) de 30 de minute.



Răspuns Cache MISS - Datele sunt preluate din MongoDB. Execution time: 12.85 ms



Răspuns Cache HIT - Datele sunt returnate instantaneu din Redis. Execution time: 2.84ms

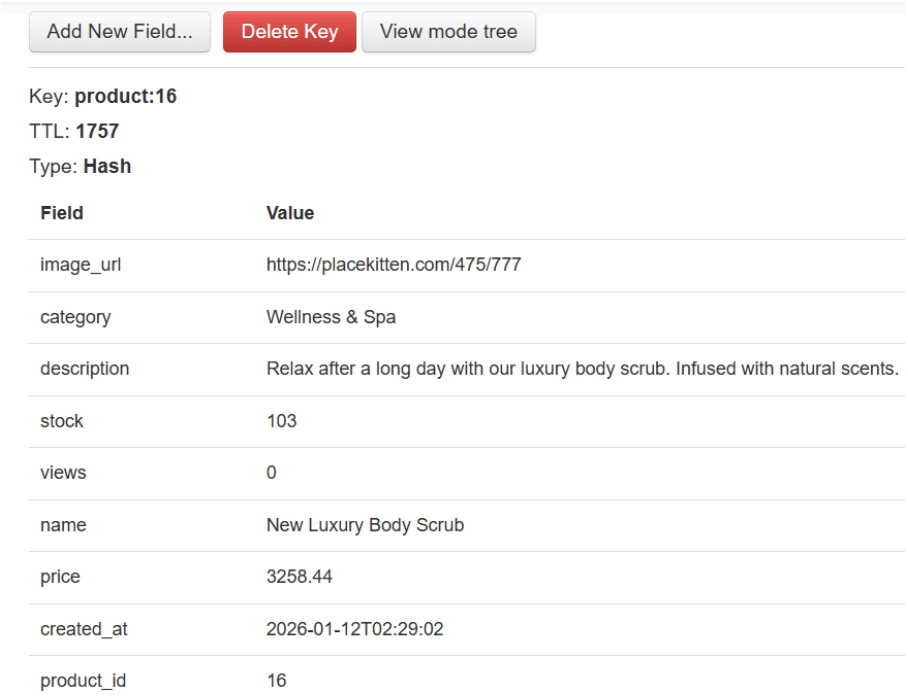
- **Write-Through (Strong Consistency)**: Aceasta este aplicată în cazul actualizărilor de preț (PUT /products/{id}/price), prioritizând integritatea datelor în defavoarea vitezei. Modificările sunt scrise sincron mai întâi în MongoDB, pentru a se asigura că produsul căutat este unul existent, iar apoi în Redis. Deși această metodă este una mai lentă, ea garantează că vechiul preț nu va mai exista în niciunul dintre straturile de stocare.
- **Write-Behind (Eventual Consistency)**: Această metodă este folosită în cadrul proiectului pentru a contoriza vizualizarea produselor (POST /products/{id}/view). Fiind o strategie asincronă, aplicația actualizează imediat contoarele în memoria Redis și returnează un răspuns de succes clientului, obținând o latență mică. În același timp, în fundal rulează procese care modifică contoarele și în MongoDB, însă clientul nu mai așteaptă un răspuns pentru finalizarea acestui proces.

4. Structuri de Date

Pentru a optimiza procesarea datelor direct în memorie și a utiliza cât mai mult capacitățile avansate ale Redis, proiectul utilizează patru structuri de date complexe:

Hash

- Comenzi utilizate: HSET, HGETALL
- Utilizare: Stocarea detaliilor despre produse
- Justificare Tehnică: În loc să serializăm întregul obiect produs într-un singur șir JSON (ceea ce ar necesita decodare completă la fiecare citire), am utilizat structura **Hash**. Aceasta funcționează similar unui dicționar, mapând câmpuri specifice (preț, stoc, descriere) la valori. Acest lucru permite actualizarea individuală a atributelor (de exemplu, modificarea doar a prețului prin HSET) fără a suprascrie întregul obiect, reducând astfel overhead-ul de procesare.



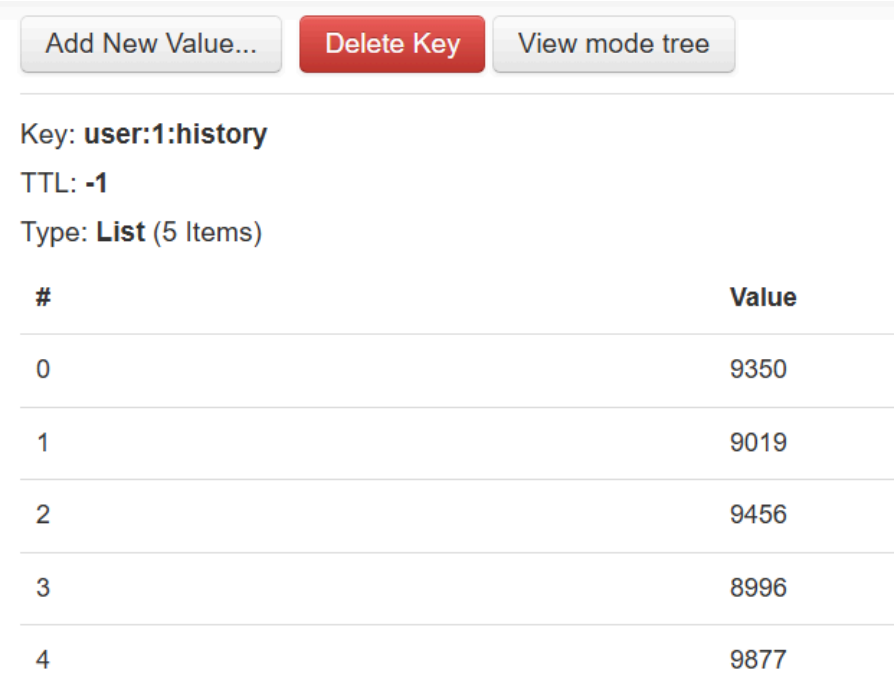
Structura HASH în Redis Commander stocând detaliile unui produs.

String

- Comenzi utilizate: INCR
- Utilizare: Contorizarea vizualizărilor per produs
- Justificare Tehnică: Deși Redis le numește "String-uri", acestea pot stoca numere întregi pe care le poate manipula atomic. Comanda INCR transformă valoarea textului în număr, o incrementează și o salvează la loc. Această operațiune este atomică, garantând că niciun click nu se pierde, chiar și în cazul accesului simultan, o problemă frecventă în bazele de date clasice care necesită mecanisme complexe de locking.

List

- Comenzi utilizate: LPUSH, LTRIM, LRANGE
- Utilizare: Coadă de tip FIFO pentru ultimele produse vizualizate
- Justificare Tehnică: Structura de Listă dublu înlănțuită permite inserarea extrem de rapidă (O(1)) a noilor elemente la începutul listei (LPUSH). Combinată cu comanda LTRIM, am implementat un mecanism automat care păstrează doar ultimele 5 intrări. Aceasta previne creșterea necontrolată a consumului de memorie per utilizator.



Structura LIST utilizată pentru istoricul de navigare.

Sorted Set

- Comenzi utilizate: ZINCRBY, ZREVRANGE
- Utilizare: Generarea topului celor mai populare produse (leaderboard)
- Justificare Tehnică: Aceasta este cea mai avansată structură utilizată. Sorted Set menține elementele unice, ordonate automat în funcție de un "scor" (numărul de vizualizări). Spre deosebire de o bază de date SQL care ar necesita interogări costisitoare de tip ORDER BY la fiecare cerere, Redis returnează topul instantaneu, deoarece ordinea este menținută la momentul inserării (ZINCRBY).

Add New Member... | Delete Key | View mode tree

Key: **leaderboard**
TTL: **-1**
Type: **Sorted Set** (9410 Members)

#	Score	Value
0	23	product:9934
1	20	product:31
2	16	product:2026
3	15	product:8543
4	7	product:6507
5	7	product:6131
6	7	product:5570
7	7	product:3624
8	7	product:1817
9	6	product:9936
10	6	product:9432
11	6	product:9281

Clasamentul produselor în timp real folosind Sorted Sets.

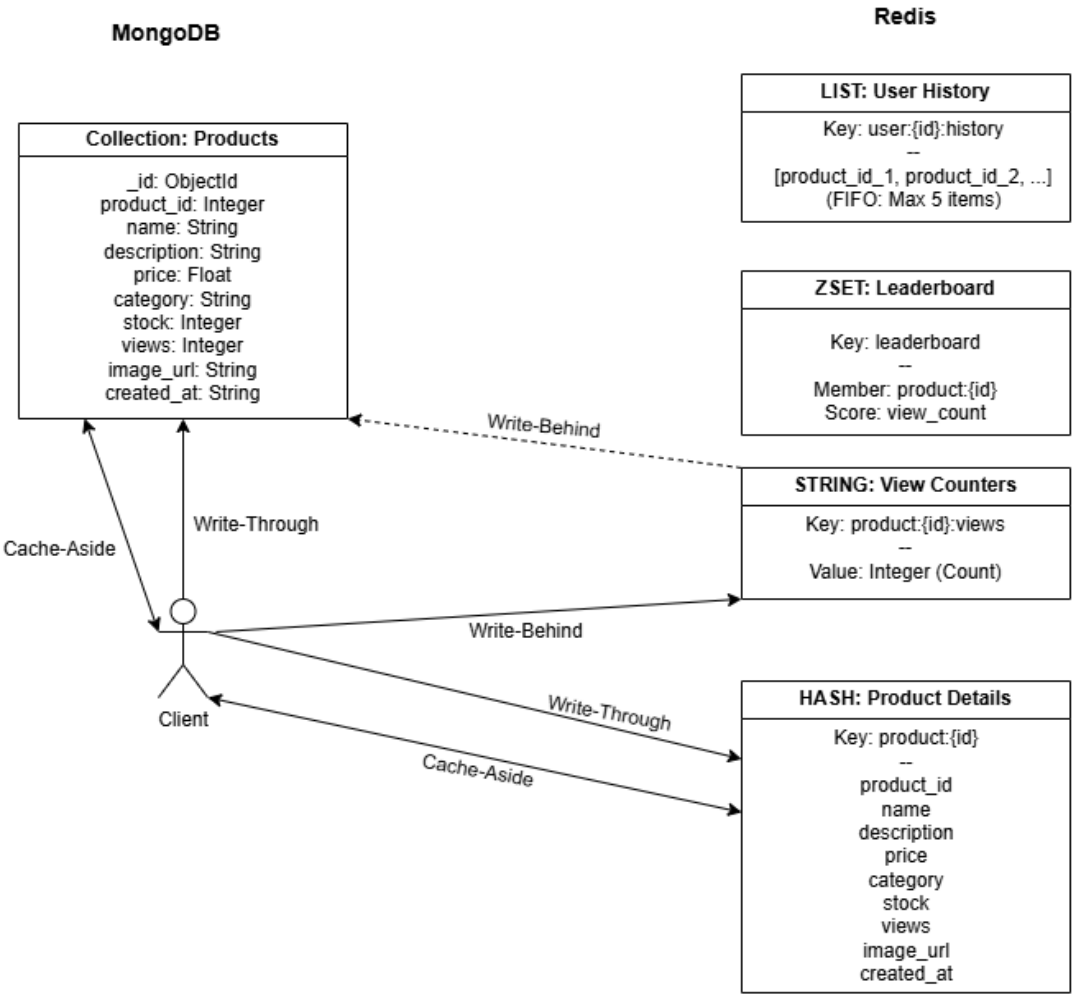


Diagrama fluxurilor de sincronizare între MongoDB și Redis

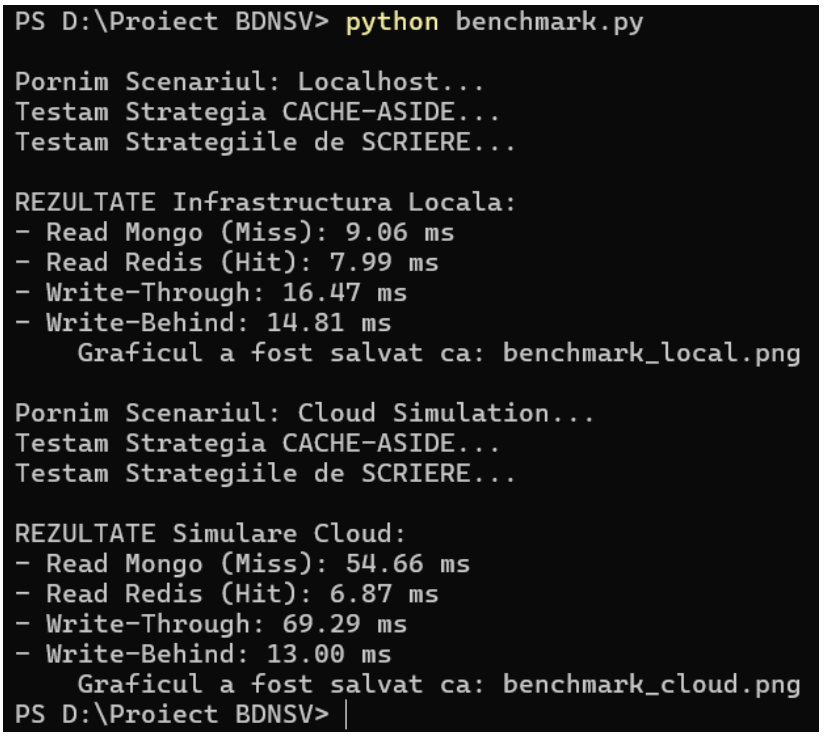
5. Analiza Performanței

În timpul realizării acestui proiect, am remarcat faptul că rularea locală atât a bazei de date MongoDB, cât și a stratului de caching Redis, poate duce la obținerea unor rezultate nerealiste, cu diferențe mult prea mici de timp.

Așadar, am preferat să abordăm două situații complet diferite:
a) Toată infrastructura se află local (fără latență)
b) Simulare Cloud: Când baza de date (**persistence layer**) se află la o distanță mai mare, caz în care există o latență semnificativă atunci când este interogată MongoDB

Pentru a putea compara performanțele metodelor de caching, am creat scriptul *benchmark.py* în care rulăm următoarele teste pentru cele două ipoteze:
a) Infrastructura locală Alegem aleatoriu 1000 de ID-uri între 1 și 5000. Pentru fiecare dintre cele 1000 de produse aplicăm mai întâi strategia **CACHE-ASIDE** și măsurăm care a fost media de timp atunci când produsul a fost găsit în Redis ("Hit") față de când a fost nevoie să fie preluat din MongoDB ("Miss"). Mai apoi, comparăm cele două strategii de scriere, **WRITE-THROUGH** și **WRITE-BEHIND**, observând timpul de așteptare al clientului.

b) Infrastructura cloud simulată Alegem aleatoriu 1000 de ID-uri între 5001 și 10000. Am ales să împărțim ID-urile în două părți egale pentru a ne asigura că la începutul testului niciunul dintre produse nu se află deja în Cache, caz care ar fi favorizat incorect rezultatele. În cazul simulării cloud, a fost stabilită artificial o latență de **50 de milisecunde** la fiecare interogare către baza de date MongoDB. După configurare, am continuat cu exact aceleași testări și comparații ca și în cazul infrastructurii locale.



Execuția scriptului automatizat de benchmarking în terminal.

6. Generarea și Popularea Datelor

Pentru a putea realiza teste de performanță relevante și pentru a valida scenariile de scalabilitate, nu a fost suficient să avem o bază de date cu un număr redus de intrări. Aveam nevoie de un volum semnificativ de date care să simuleze cât mai bine un catalog real de E-commerce.

Așadar, am creat scriptul automatizat *seed_data.py*, care are rolul de a popula baza de date MongoDB cu **10.000 de produse** unice.

Abordarea Semantică: Am observat că generarea pur aleatoare de text face dificilă urmărirea și debug-ul aplicației. De aceea, am implementat o logică de generare semantică, bazată pe dicționare predefinite pentru fiecare categorie (ex: *Gadgets, Books, Home & Decor*).

Scriptul funcționează astfel:

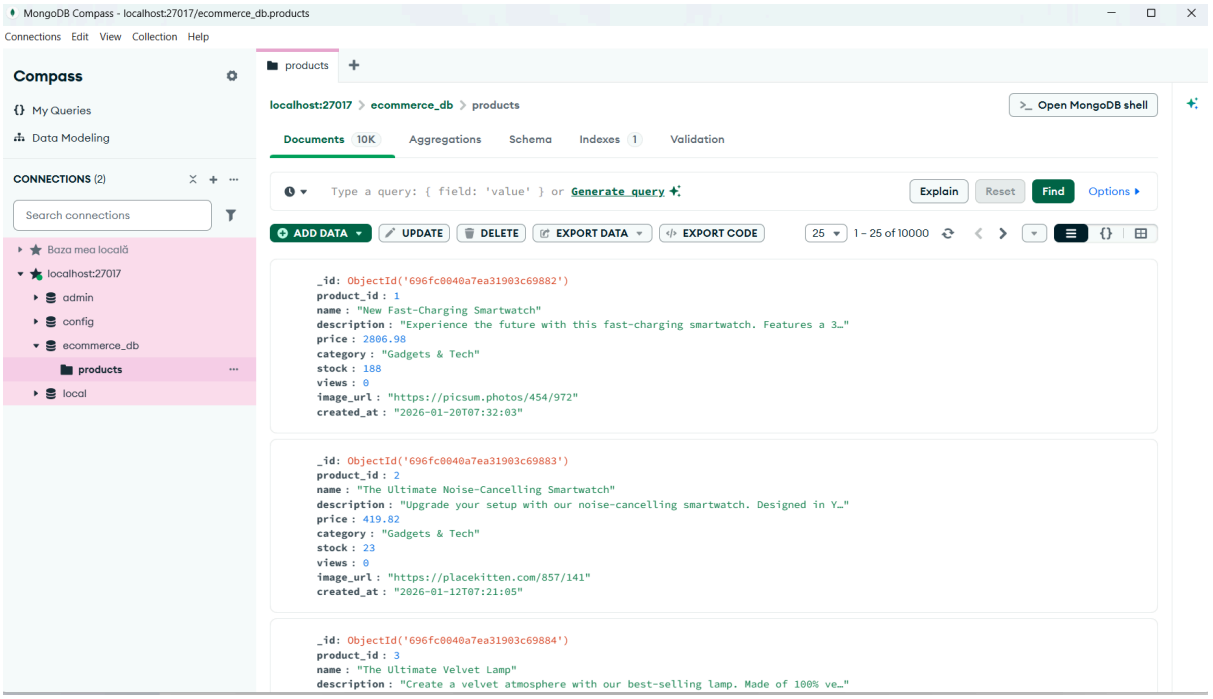
1. **Curățare:** La fiecare rulare, scriptul șterge datele anterioare din MongoDB pentru a asigura un mediu de testare curat.
2. **Construcția Numelui:** Folosește un algoritm care combină aleatoriu un **Adjectiv** (ex: "Wireless", "Ergonomic") cu un **Substantiv** specific categoriei (ex: "Mouse", "Keyboard") și un sufix de **Model/Serie**.
    - *Exemplu generat:* "High-Performance Headphones 'Titan' Series".
3. **Atribute Realiste:** Prețurile, stocurile și descrierile sunt generate folosind librăria **Faker**, dar respectând limite logice (de exemplu, prețuri între 15 și 3500 RON).

4. **Inserare în Masă:** După generarea celor 10.000 de obiecte în memorie, acestea sunt inserate printr-o singură operațiune optimizată (insert_many) în MongoDB.

Acest proces ne-a asigurat un set de date consistent, pe care am putut rula ulterior testele de benchmark.

```
PS D:\Proiect BDNSV> python seed_data.py
Stergem datele vechi...
Generam 10.000 de produse...
    ... s-au generat 2000 produse
    ... s-au generat 4000 produse
    ... s-au generat 6000 produse
    ... s-au generat 8000 produse
    ... s-au generat 10000 produse
Inseram datele in MongoDB...
SUCCES! Produsele au fost inserate in 1.42 secunde.
```

Output-ul scriptului de populare a bazei de date.

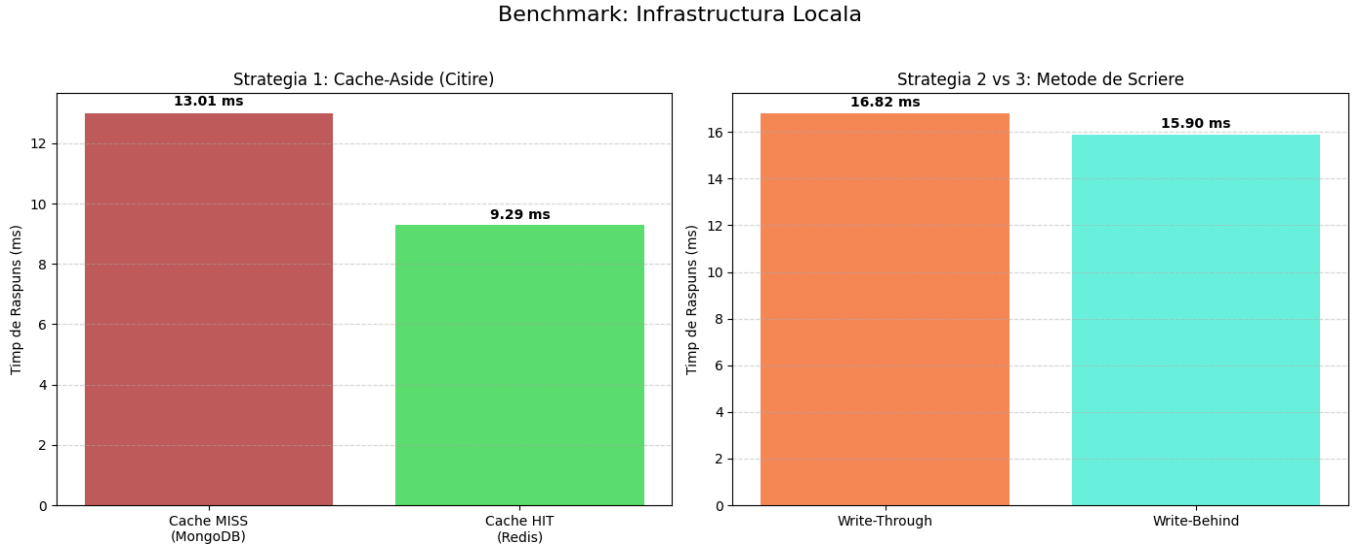


Validarea celor 10.000 de produse în MongoDB Compass.

7. Rezultate

În urma rulării suitei de teste automatizate (benchmark.py), am generat grafice comparative pentru cele două scenarii ipotetice. Rezultatele obținute validează premisa inițială a proiectului: stratul de caching oferă beneficii exponențiale pe măsură ce latența către baza de date crește.

Scenariul A: Infrastructură Locală



În acest scenariu, ambele servicii (Redis și MongoDB) rulează pe aceeași mașină, eliminând latența de rețea. Diferențele de performanță observate sunt cauzate exclusiv de mediul de stocare (Memorie RAM vs. Disk I/O).

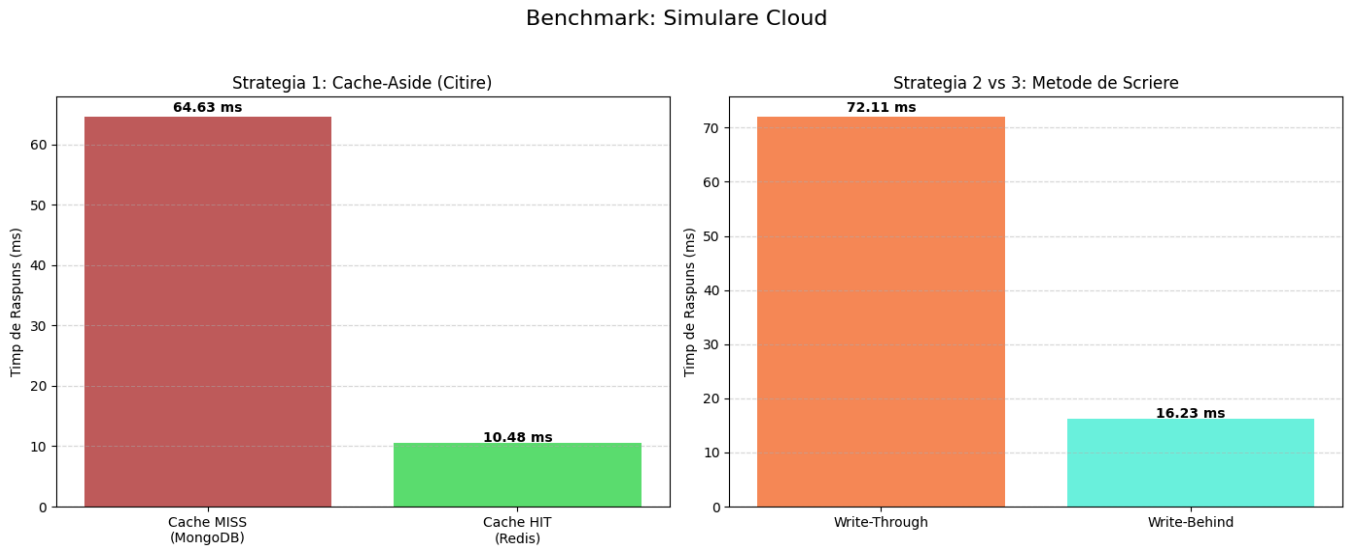
Citire:

- **Cache MISS (MongoDB):** 13.01 ms
- **Cache HIT (Redis):** 9.29 ms
- **Interpretare:** Chiar și în absența latenței de rețea, Redis a oferit o îmbunătățire de aproximativ **30%**. Aceasta se datorează accesului direct la memoria RAM, care este ordine de mărime mai rapidă decât accesarea discului SSD necesară pentru MongoDB.

Scriere:

- Strategiile **Write-Through** (16.82 ms) și **Write-Behind** (15.90 ms) au înregistrat timpi similari, deoarece scrierea locală în baza de date este extrem de rapidă, nepermițând strategiei asincrone să se diferențieze semnificativ.

Scenariul B: Simulare Cloud



Acest scenariu reflectă o arhitectură distribuită reală, unde baza de date se află la distanță. Am simulat o latență de rețea de 50ms, iar rezultatele evidențiază impactul major al arhitecturii de caching.

Citire:

- **Cache MISS (MongoDB):** 64.63 ms (include latența de rețea).
- **Cache HIT (Redis):** 10.48 ms.
- **Interpretare:** Utilizarea Redis a redus timpul de răspuns de **6.1 ori**. Deoarece datele sunt servite din memoria cache locală (sau apropiată de aplicație), latența rețelei către baza de date este complet eliminată pentru utilizatorul final.

Scriere:

- **Write-Through:** 72.11 ms. Această strategie cumulează latența rețelei și timpul de scriere pe disc, fiind cea mai lentă, dar cea mai sigură metodă.
- **Write-Behind:** 16.23 ms. Fiind o strategie asincronă, aceasta a fost de **4.4 ori mai rapidă** decât Write-Through. Clientul primește confirmarea imediat după scrierea în Redis, în timp ce sincronizarea cu MongoDB are loc în fundal, fără a bloca experiența utilizatorului.

8. Referințe

[1] Redis Documentation, https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/ , Accesat la: 15.01.2026.
[2] MongoDB Compass Documentation, https://www.mongodb.com/products/tools/compass , Accesat la: 15.01.2026
[3] FastAPI Documentation, https://fastapi.tiangolo.com/ , Accesat la: 15.01.2026.
[4] Cursul 7 - Caching, Materia ”Baze de date, de la NoSql la Vector DBs”
[5] Gemini, https://gemini.google.com/ , Generat la: 17.01.2026
[6] Amazon Web Services (AWS), Caching Patterns https://docs.aws.amazon.com/whitepapers/latest/database-caching-strategies-using-redis/caching-patterns.html , Accesat la: 15.01.2026