FILIERA360

DIARIO

29/04/2025

- individuato file json per la memorizzazione dei dati (users)
- creato diagramma E-R
- ricerca per scegliere DBMS : MySql (con libreria pymysql)
- aggiunto container “mysql” modificando il codice in docker-compose.yaml
- creato connessione al db in app.py

30/04/2025

- scrittura delle query per la creazione delle tabelle nel db “filiera360” in file “mysql.sql” (tabelle create: users, otp_codes, liked_products, searches, models, invite_tokens)
- salvati utenti e OTP nel database: modificate le funzioni send_otp, save_users, load_users e le route /signup, /login e /verify-otp”

02/04/2025

- modificato le route /operators, /operators/add, /operators/delete e la funzione required_permissions

06/05/2025

- rimosso psw del db dal codice e inserita in .env (presente in .gitignore)

07/05/2024

- modificate find_producer_by_operator, verify_product_authorization, upload_product, update_product, add_certification_data, load_models, save_models, upload_model, get_model
- modificata tabella models—> script LONGTEXT e ID VARCHAR(50)

03/07/2025

- ho introdotto la funzione def get_db_connection()
- modificate route forgot-password, is_valid_invite_token, uploadBatch, updateBatch, addSensorData, addMovementsData, likeProduct, unlikeProduct, getLikedProducts, addRecentlySearched, getRecentlySearched

- modificata tabella searches
- aggiunto per test su scenari frequenti (login, sign up, verifica otp, like product, unliked product, recently searched) (ho installato axios)
- script per stress test con locust (ho installato locust)

07/07/2025

- modifiche per stress test (in app.py e in locust.py)

21/07/2015

- creato cartella database in cui ho definito le query e la connessione al db

22/07/2025

- aggiunta funzione per eliminare i token scaduti o usati; introdotto campi user_by e user_at nella tabella invite_token
- modificato query per eliminare otp scaduti o usati

23/07/2025

- modificato il db in modo che sia normalizzato

24/07/2025

- aggiunto query per memorizzare ogni prodotto nella tabella models
- modificato script per test query e stress test

WORKFLOW

1. **Analisi dei dati e modellazione del database relazionale**
    - Raccogliere tutti i file per la gestione dei dati
    - Individuare le entità principali e le loro relazioni (uno a molti, uno a uno…)
        - Realizzazione delle tabelle secondo lo schema relazionale definito in precedenza tramite DBMS scelto
    - Realizzare uno schema E-R
2. **Implementazione del database relazionale**
    - Scelta del DBMS da utilizzare (Mysql)

- Aggiungere il container per il database in Docker modificando il file docker-compose.yaml
- Effettuare connessione al database in app.py
- Modificare il codice in app.py per effettuare la migrazione della gestione dei dati con file JSON al database

3. **Verifica e valutazione delle prestazioni**
   - Effettuare test su scenari che si verificano frequentemente
   - Effettuare stress test con una mole di dati crescente
4. **Confronto dei risultati con quelli ottenuti nel 2.2**
   - Effettuare il confronto di entrambi i metodi di gestione dei dati, per analizzare alcuni aspetti come il tempo di accesso ai dati, la scalabilità e la facilità di aggiornare le informazioni.

DEVELOPMENT

Task 01

File per la gestione dei dati:

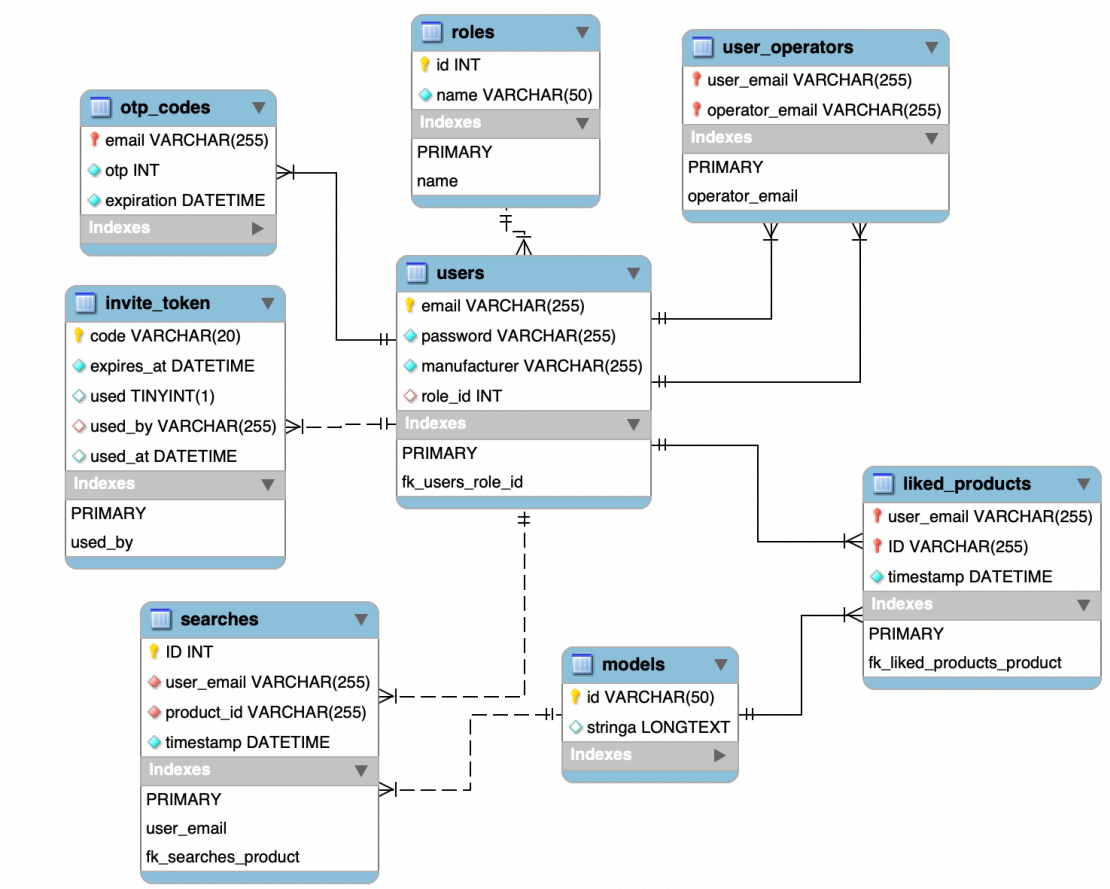
1. users.json
2. users-otp.json
3. searches.json
4. liked-products.json
5. models.json
6. invite_tokens.json

Entità principali e le loro relazioni:

1. Users (email chiave primaria, role_id chiave esterna)
2. Otp_codes (email chiave primaria e chiave esterna)
3. Searches (id chiave primaria, user_email e product_id chiave esterna)
4. liked_products (id e user_email chiave esterna e chiave primaria)

5. models (id chiave primaria)
6. invite_tokens (code chiave primaria, used_by chiave esterna)
7. roles (id chiave primaria)
8. user_operators (user_email e operator_email chiave esterna e chiave primaria)

Diagramma E-R con mysql workbench



MODIFICHE EFFETTUATE PER RENDERE LE TABELLE NORMALIZZATE:

- Separazione della tabella users dalla tabella roles a cui è collegata tramite foreign key role_id. Ogni utente ha un ruolo che punta a una riga nella tabella `roles`. I nomi dei ruoli (`user`, `operator`, `producer`) non vengono ripetuti in ogni riga utente.
- Nella tabella `liked_products` e `searches` resta solo l'ID del prodotto, non i dettagli (nome, manufacturer, creationDate). Quindi non è presente ridondanza e duplicazione di dati. Tutti i campi dipendono direttamente dalla chiave primaria della relativa tabella (quindi è garantita la 2NF). Poichè non sono stati memorizzati i campi `Name`, `Manufacturer`, `CreationDate` dentro `liked_products` o `searches`, sono evitate dipendenze transitive (è garantita la 3NF).

- Creazione della tabella `user_operators` per rappresentare correttamente la relazione tra producers e i propri operators. E' stato necessario effettuare questa modifica poichè non è possibile rappresentare una colonna con più valori nella tabella users (violazione della 1NF).

Task 02

DBMS da utilizzare: MySQL con PyMySQL

- Modificato il file requirements.txt: ho aggiunto le librerie pymysql, cryptography e pytz per ottenere il fuso orario di Roma

Modificato il file docker-compose.yaml:

- Ho aggiunto il container per il database (MySQL) e in particolare ho usato `healthcheck` per controllare se il container fosse funzionante, in quanto quest'ultimo necessitava di qualche secondo in più per avviarsi. Tuttavia, il backend cercava di connettersi a MySQL prima che il container fosse pronto, quindi la connessione falliva.
- Ho definito le variabili d'ambiente nel container del backend
- Ho impostato la dipendenza del backend a MySQL (il backend non può essere avviato se MySQL non è pronto)
- Ho aggiunto il volume mysql_data per far sì che i dati non vengano persi anche se il container dovesse essere interrotto o cancellato.

Creato db_connection.py per stabile la connessione con il db:

- Ho aggiunto le librerie pymysql, time (per generare una pausa tra un tentativo e l'altro) e pytz
- ho usato `pymysql.cursors.DictCursor` è un tipo di cursore che restituisce i **risultati delle query come dizionari** (anziché come tuple). Questo è utile perché, invece di lavorare con indici numerici per accedere ai dati, puoi utilizzare i nomi delle colonne. Questo garantisce maggiore leggibilità, maggiore gestione dei dati e facilità di modifica
- Ho introdotto la funzione `def get_db_connection()` perchè inizialmente la variabile globale connection era aperta una sola volta all'inizio e usata per tutte le rotte con `with connection.cursor()` as cursor:

  Questo funziona solo se viene eseguita una operazione per volta (single threaded).

  Ho riscontrato questo problema quando dovevo gestire le richieste recenti e i mi piace ai prodotti (multi-thread).

- Aggiunto parti di codice per connettermi al database in Docker e, se la connessione fallisce, ho considerato altri 10 tentativi (questo perché inizialmente la connessione falliva, in particolare il backend non riusciva a collegarsi al container del database perché MySQL necessita di più tempo per avviarsi)

Scrittura delle query:

OTP

- insert_or_update_otp:
  - ottiene il fuso orario di Roma (Italia) usando la libreria `pytz` (senza questo, l'orario di scadenza risultava essere 2 ore indietro rispetto a quello locale)
  - è aperta una connessione nel db tramite cursore per verificare l'esistenza dell'utente nel db attraverso una query. Se l'utente non è presente, restituisce una risposta JSON con messaggio di errore.
  - inserimento/aggiornamento dell'otp: viene eseguita una query per aggiungere una riga oppure se è già presente una riga in corrispondenza di quella email, il codice otp e la data di scadenza vengono aggiornati senza aggiungere un'ulteriore riga (`ON DUPLICATE KEY UPDATE`). infine viene eseguito il commit per salvare le modifiche nel db in maniera permanente
- get_otp_record
  - recupera dal db il codice OTP e la relativa data di scadenza associati a un utente, identificato dal suo indirizzo email.
- get_latest_otp (usato solo per recuperare otp in fase di stress test)
  - serve a **recuperare l'OTP più recente** associato a un determinato indirizzo email dalla tabella `otp_codes` .
- delete_otp
  - elimina otp scaduti o usati

USERS

- check_email_exists
  - controlla se esiste un utente registrato con quella email nel db
- check_manufacturer_exists
  - verifica se esiste già un produttore con quel nome (`manufacturer`) nella tabella `users` .

- insert_user
  - inserisce un nuovo utente nel db
- get_user_by_email
  - recupera tutti i dati dell’utente conoscendo l’email
- get_user_operators
  - restituisce la lista degli operatori associati all’utente
- get_user_role
  - restituisce il ruolo dell’utente
- get_manufacturer_by_email
  - restituisce il manufacturer associato ad un utente
- update_user_password
  - aggiorna la psw dell’utente
- add_operator_to_user
  - Aggiunge un operatore associato a un produttore.
- remove_operator_from_user
  - Rimuove un operatore associato a un produttore.

INVITE TOKEN

- fetch_invite_token_data
  - recupera le informazione del token
- mark_invite_token_used
  - segna un token come usato nel db
- delete_expired_or_used_tokens
  - elimina token usati o scaduti

ACCESS CONTROL

- find_producer_by_operator
  - trova il produttore associato a un operatore (se esiste)
- required_permissions

    - verifica se l’utente ha uno dei ruoli richiesti per svolgere determinate funzioni
- verify_product_authorization
    - verifica se l’utente ha il permesso per gestire un determinato prodotto

MODELS

- save_or_update_model
    - salva o aggiorna un modello 3D associato al prodotto
- get_model_by_product_id
    - recupera il modello 3D in base64 associato a un product_id
- insert_product_if_not_exists
    - inserisce un prodotto nella tabella models solo se non esiste già

LIKES

- has_user_liked_product
    - controlla se l’utente ha già messo mi piace a quel determinato prodotto
- add_product_like
    - aggiunge il prodotto piaciuto nella tabella liked_products (massimo 100)
- remove_product_like
    - rimuove il like al prodotto (quindi il prodotto viene rimosso dalla tabella liked_products)
- get_user_liked_products
    - restituisce i primi 100 prodotti a cui l’utente ha messo mi piace

SEARCHES

- add_recent_search
    - salva una ricerca recente dell’utente e mantiene solo le ultime 50 per ogni utente.
- get_recent_searches
    - recupera le ultime 50 ricerche effettuate dall’utente loggato

Task 03

METRICHE STANDARD PER ANALIZZARE I DATI OTTENUTI:

Nielsen Norman Group - LIMITI DI RISPOSTA

- 100 ms → risposta istantanea
- 1 s → l'utente percepisce un ritardo, ma non vi è perdita d'attenzione dello stesso.
- da 1 a 10 s → l'utente perde l'attenzione ed è difficile recuperarla quando la risposta viene fornita

Google – Core Web Vitals

- LCP – Largest Contentful Paint → misura le prestazioni di caricamento (`/login` , `/signup` , `/getLikedProducts` , `/getRecentlySearched`)
- INP – Interaction to Next Paint → misura l'interattività dell'utente (`/likeProduct` , `/addRecentlySearched` , `/verify-otp`)
- CLS – Cumulative Layout Shift → misura la stabilità visiva

Google Cloud - Throughput (RPS)

https://cloud.google.com/run/docs/about-concurrency

- Numero di richieste gestite al secondo → 5–20 RPS per istanza

K6 BLOG - Error Rate (%)

https://medium.com/qest/load-testing-with-k6-ef17a1f64def

- Percentuale di richieste fallite su totale → <0.1% eccellente, 0.1–1% accettabile, >1% critico.

TEST:

```
=== TEST USERS ===
Insert user: 0.0136s
Check email exists: 0.0020s
```

Check manufacturer exists: 0.0017s
Get user by email: 0.0022s
Update user password: 0.0036s
Get user role: 0.0019s
Get manufacturer by email: 0.0017s
Insert operator user: 0.0029s
Add operator: 0.0037s
Get user operators: 0.0022s
Remove operator: 0.0034s
TEMPO MEDIO USERS: 0.0035 s

=== TEST OTP ===
Insert or update OTP: 0.0298s
Get OTP record: 0.0064s
Get latest OTP: 0.0020s
Delete OTP: 0.0029s
TEMPO MEDIO OTP: 0.0411 s

=== TEST INVITE TOKENS ===
Insert token manually: 0.0019s
Fetch token: 0.0025s
Mark token as used: 0.0019s
Delete expired or used tokens: 0.0018s
TEMPO MEDIO INVITE TOKENS: 0.0020 s

=== TEST MODELS ===
Insert product if not exists: 0.0033s
Save or update model: 0.0035s
Get model by product ID: 0.0017s
TEMPO MEDIO MODELS: 0.0028 s

=== TEST LIKES ===
Add product like: 0.0035s
Has user liked product: 0.0027s
Get user liked products: 0.0016s
Remove product like: 0.0030s
TEMPO MEDIO LIKES: 0.0027 s

=== TEST RECENTLY SEARCHED ===
Add recent search 0: 0.0046s
Add recent search 1: 0.0108s
Add recent search 2: 0.0090s

Add recent search 3: 0.0093s
Add recent search 4: 0.0077s
Add recent search 5: 0.0096s
Get recent searches: 0.0083s
TEMPO MEDIO RECENTLY SEARCHED: 0.0085 s

TUTTI I TEST COMPLETATI IN 1.46 secondi

Le query che impattano su LCP sono: /login, /signup, /getLikedProducts, /getRecentlySearched, mentre le restanti impattano su INP

Analizzando il tempo necessario per l'esecuzione della singola query e gli standard proposti, è possibile concludere che il sistema è compatibile con le soglie dei Core Web Vitals in quanto tutte le operazioni rispettano gli standard ottimali.

STRESS TEST:

LOCUST→ test scritti in python; prevede interfaccia UI interattiva in cui è possibile avviare e stoppare il test, scegliere il numero di utenti simultanei e osservare come variano le metriche in tempo reale.

L'interfaccia utente presenta:

- numero di utenti in parallelo (attivi contemporaneamente)
- utenti avviati al secondo (quanti utenti si aggiungono ogni secondo fino ad arrivare al numero massimo definito precedentemente)
- host: indirizzo del sito

Alcune delle metriche che è possibile osservare sono:

— mediana: tempo affinchè il 50% delle richiesta sia elaborato.

— Il **95° percentile** è il tempo **sotto il quale il 95% delle richieste viene completato**.

— Il **99° percentile** è il tempo **sotto il quale il 99% delle richieste viene completato**.

In app.py, nella funzione send_otp è necessario commentare l'invio dell'email, e scrivere un codice per simularne l'invio. Il codice stampa l'otp relativo all'email e viene inviata una risposta positiva all'utente

Ho aggiunto `/get-latest-otp` che permette di ottenere l'ultimo OTP generato per un determinato indirizzo email. Verifico che l'email sia presente. Recupero otp relativo all'email, se l'otp non esiste ritorna errore

FREQUENZA:

- `@task(1)`: login, get_recently_searched, get_liked_products → meno frequente
- `@task(2)`: signup, like_product, add_recently_searched → più frequenti

In Locust, `self.client` è un oggetto che simula un client HTTP (come un browser) e permette di inviare richieste ai tuoi endpoint.

SCALABILITA':

Utenti in parallelo: 100

Utenti avviati al secondo: 4

Numero di richieste totali: 1881

Errori: 4

RPS (numero medio di richieste al secondo): 40.4

Request Statistics

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/addRecentlySearched	359	0	15.3	5	97	49	7.71	0
GET	/get-latest-otp?email=user6799%40gmail.com	100	0	14.31	3	56	15	2.15	0
GET	/getLikedProducts	173	0	9.27	3	40	61	3.72	0
GET	/getRecentlySearched?userEmail=user6799%40gmail.com	180	0	9.38	3	31	180	3.87	0
POST	/likeProduct	354	0	10.33	3	63	36	7.6	0
POST	/login	270	0	312.86	251	601	63	5.8	0
POST	/signup	345	2	316.74	9	640	42.95	7.41	0.04
POST	/verify-otp	100	2	17.9	4	50	472.14	2.15	0.04
	Aggregated	1881	4	111.33	3	640	81.78	40.4	0.09

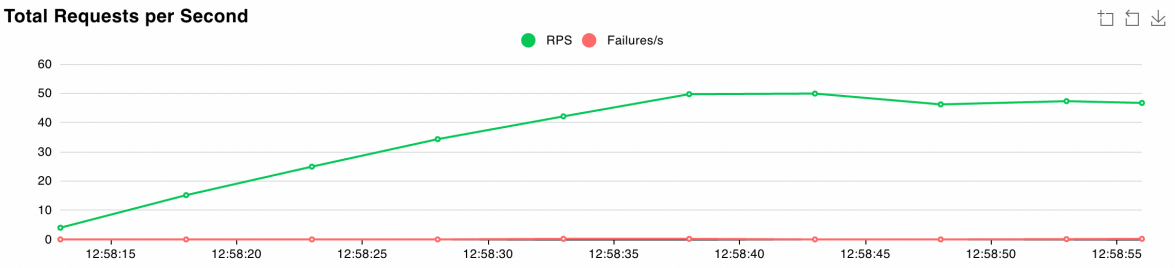
Response Time Statistics

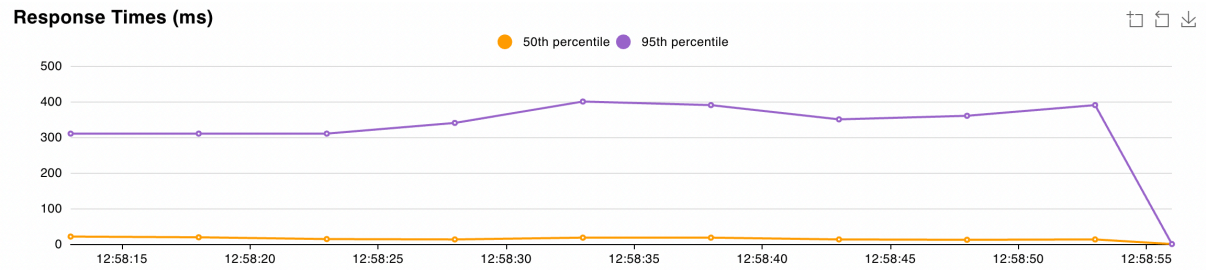
Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
POST	/addRecentlySearched	12	13	16	19	27	41	62	97
GET	/get-latest-otp?email=user6799%40gmail.com	11	17	19	20	24	37	56	56
GET	/getLikedProducts	8	9	10	12	16	19	39	40
GET	/getRecentlySearched?userEmail=user6799%40gmail.com	8	9	10	13	18	22	29	31
POST	/likeProduct	8	9	10	13	18	26	54	63
POST	/login	300	310	320	330	380	420	530	600
POST	/signup	300	320	330	350	400	440	540	640
POST	/verify-otp	18	19	21	26	28	32	50	50
	Aggregated	14	21	260	290	320	360	460	640

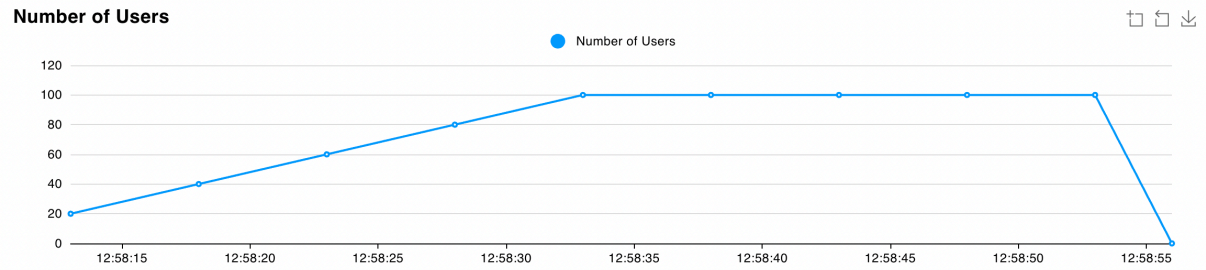
Failures Statistics

# Failures	Method	Name	Message
2	POST	/signup	HTTPError('409 Client Error: CONFLICT for url: /signup')
2	POST	/verify-otp	HTTPError('400 Client Error: BAD REQUEST for url: /verify-otp')

Charts







`/login` e `/signup` mostrano un ritardo maggiore.

Il numero di richieste al secondo per i singoli endpoint rientra negli standard proposti da ***Google Cloud.***

La percentuale di errori è bassa (0.04 → ottimo, sotto la soglia 0.1% del benchmark **K6**)

`/signup` genera 2 errori per conflitto (utente già esistente)

`/verify-otp` genera 2 errori per OTP errato o scaduto

CONFRONTO TRA DATABASE RELAZIONALE E DATABASE NON RELAZIONALE:

Categoria	Query	**MySQL (s)**	**MongoDB (s)**
USERS	Insert user	0.0136	0.0021
USERS	Check manufacturer exists	0.0017	0.0027
USERS	Get user by email	0.0022	0.0032
OTP	Insert or update OTP	0.0298	0.0069
OTP	Get OTP	0.0064	0.0018
OTP	Delete OTP	0.0029	0.0035
INVITE TOKENS	Insert token	0.0019	0.0239
INVITE TOKENS	Fetch token	0.0025	0.0066

Categoria	Query	**MySQL (s)**	**MongoDB (s)**
INVITE TOKENS	Mark token as used	0.0019	0.0077
MODELS	Insert product if not exists	0.0033	0.0033
MODELS	Save or update model	0.0035	0.0087
MODELS	Get model by product ID	0.0017	0.0037
LIKES	Add product like	0.0035	0.0100
LIKES	Has user liked product	0.0027	0.0023
LIKES	Get user liked products	0.0016	0.0019
LIKES	Remove product like	0.0030	0.0026
RECENTLY SEARCHED	Add recent search 0	0.0046	0.0151
RECENTLY SEARCHED	Add recent search 1	0.0108	0.0063
RECENTLY SEARCHED	Add recent search 2	0.0090	0.0057
RECENTLY SEARCHED	Add recent search 3	0.0093	0.0038
RECENTLY SEARCHED	Add recent search 4	0.0077	0.0053
RECENTLY SEARCHED	Add recent search 5	0.0096	0.0071
RECENTLY SEARCHED	Get recent searches	0.0083	0.0020

Dai risultati ottenuti è possibile affermare che il database relazione risulta essere più efficace in termini di tempo rispetto al database non relazionale poichè la maggior parte delle query vengono eseguite in un intervallo di tempo inferiore utilizzando come DBMS MySQL.