UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Databases — Graph Storage

Product Recommendation for Online Shop

Azure Cosmos DB Gremlin

Hamza Ouba

Table of Contents

Context

- Modern e-commerce platforms like Amazon or Zalando rely heavily on recommendation systems to increase conversion rates and average basket size.
- Almost every product page shows features such as “Similar products” or “Customers who bought this also bought”.
- These features are powered by systems that analyze past user interactions and relationships between products, which naturally form a graph-like structure.

Problem Statement

Goal design and evaluate the data models required to support product recommendations in an online shop.

based on:

- same category
- similarity metadata
- behavioral patterns (customers also bought)
- user-to-user similarity

real-world Constraints:

- scale
- responsiveness
- evolving catalog

Problem Statement

We must implement two different storage models for the same use case :

- a relational SQL database (classic e-commerce schema),
- a NoSQL graph database using Azure Cosmos DB with the Gremlin API.

We will then compare these two approaches in terms of

- **query complexity and expressiveness,**
- **performance on different data sizes**

Technologies Used

- **Python** to generate synthetic data and run benchmarks.
- **PostgreSQL** is used as the SQL database, accessed via psycopg2.
- **Azure Cosmos DB (Gremlin API)** For the graph database, with the gremlin-python driver.
- **pandas** is used to aggregate and analyze the benchmark results.

Benchmark Methodology

- **For each volume N:**
  - For each dataset size
  - we reset the database
  - generate the same synthetic data
  - rebuild both models
  - Execute the same 4 recommendation queries
- **Metrics collected:**
  - Build time
  - Query latency (seconds)
  - Number of results

Data Requirements

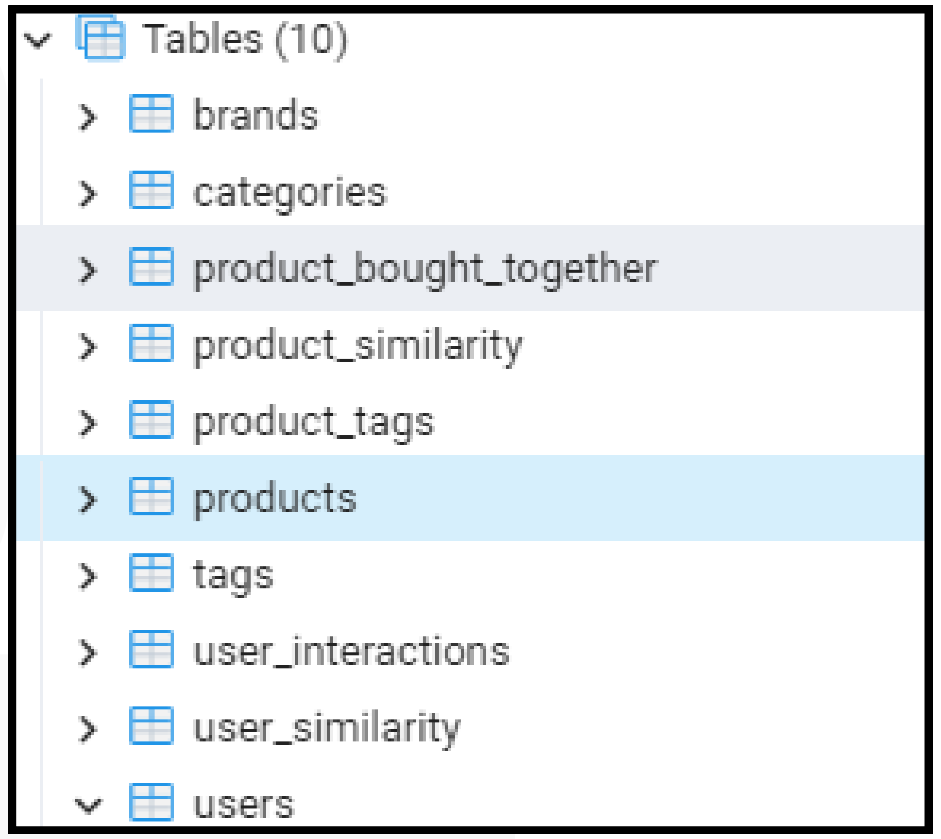
- **Entities:**
  - Product, User, Category, Brand, Tag
- **Relations (same semantics in SQL + Graph):**
  - Product → Category (IN_CATEGORY)
  - Product → Brand (HAS_BRAND)
  - Product → Tag (HAS_TAG)
  - Category parent-child (PARENT_OF)
  - User → Product interactions: VIEWED / BOUGHT / LIKED
  - Product similarity edges: SIMILAR_TO, BOUGHT_TOGETHER
  - User similarity edges: SIMILAR_USER

Data Generation

- **Base master data:**
  - Categories, brands, and tags are fixed master data.
- **Scaling rule:**
  - Products scale with N, and users scale proportionally: num_users = max(10, N/5)
- **Each product:**
  - Random brand/category/tag chosen
  - Random price in range
  - At least 1 tag per product
- **User interactions generation:**
  - User interactions are generated probabilistically to simulate realistic behavior.
    - Sample k products per user (VIEWED)
    - Random probabilities create BOUGHT / LIKED edges
- **Building the graph is expensive because it requires many network calls**
  - **We test with N = 500 / 1000 / 2000**

SQL Design

- **Tables:**
    - brands, categories, tags, users, products, product_tags
- **relationship tables (join tables.):**
    - user_interactions (VIEWED/BOUGHT/LIKED)
    - product_similarity, product_bought_together, user_similarity


Tables (10)
brands
categories
product_bought_together
product_similarity
product_tags
products
tags
user_interactions
user_similarity
users

SQL Indexing Strategy

- Primary keys:
    - include run identifiers to isolate benchmarks.
        - Most core tables use (run_pk, entity_id) as PRIMARY KEY
    - indexes are used to speed up joins and filters.
        - Ex : product_similarity(run_pk, src_product_id, score) → top similar by score
- Foreign keys : enforce integrity (products reference brands/categories, interactions reference users/products).

Queries Used

SQL (PostgreSQL)

- Simple joins :
  - **Similar_by_Category:** self-join products on (run_pk, category_id) + LIMIT 20
  - **Similar_by_SIMILAR_TO**: join product_similarity → products + ORDER BY score DESC
- self-joins
  - **Customers_Also_Bought:** join user_interactions with itself on same user + BOUGHT
- requires multiple joins and exclusions.
  - **User_Recommendations:** multi self-joins on user_interactions + exclusion subquery ("not already bought")

SQL queries used

Q1 Similar by Category

```
SELECT p2.*
FROM products p1 JOIN products p2
  ON p1.run_pk=p2.run_pk AND p1.category_id=p2.category_id
WHERE p1.run_pk=%s AND p1.product_id=%s AND
p2.product_id<>%s
LIMIT 20;
```

Q2 Similar by SIMILAR_TO

```
SELECT p_dst.*
FROM product_similarity s JOIN products p_dst
  ON p_dst.run_pk=s.run_pk AND
p_dst.product_id=s.dst_product_id
WHERE s.run_pk=%s AND s.src_product_id=%s
ORDER BY s.score DESC
LIMIT 20;
```

Q3 Customers also bought

```
SELECT DISTINCT p2.*
FROM user_interactions ui1
JOIN user_interactions ui2 ON ui1.run_pk=ui2.run_pk AND ui1.user_id=ui2.user_id
JOIN products p2 ON p2.run_pk=ui2.run_pk AND p2.product_id=ui2.product_id
WHERE ui1.run_pk=%s AND ui1.product_id=%s
  AND ui1.interaction_type='BOUGHT' AND ui2.interaction_type='BOUGHT'
  AND ui2.product_id<>ui1.product_id
LIMIT 20;
```

SQL queries used

Q4 User recommendations

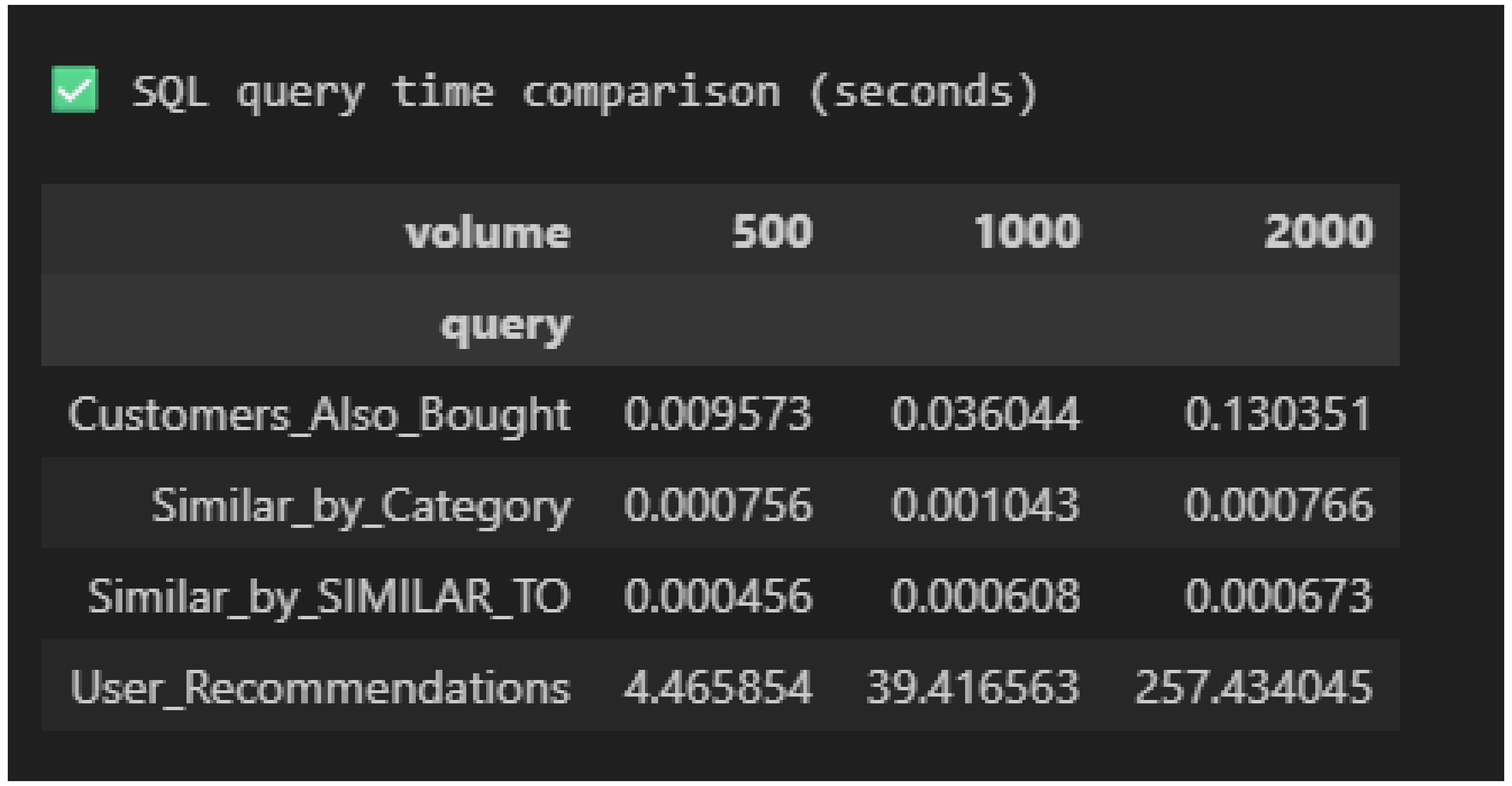
```
SELECT DISTINCT p3.*
FROM user_interactions ui_u
JOIN user_interactions ui_others ON ui_u.run_pk=ui_others.run_pk AND
ui_u.product_id=ui_others.product_id
JOIN user_interactions ui_rec ON ui_others.run_pk=ui_rec.run_pk AND
ui_others.user_id=ui_rec.user_id
JOIN products p3 ON p3.run_pk=ui_rec.run_pk AND p3.product_id=ui_rec.product_id
WHERE ui_u.run_pk=%s AND ui_u.user_id=%s
  AND ui_u.interaction_type='BOUGHT'
  AND ui_others.interaction_type='BOUGHT'
  AND ui_rec.interaction_type='BOUGHT'
  AND ui_rec.product_id NOT IN (
    SELECT product_id FROM user_interactions
    WHERE run_pk=%s AND user_id=%s AND interaction_type='BOUGHT'
  )
LIMIT 20;
```

SQL results summary

(N=500/1000/2000)

- Build (total): ~0.38s → ~0.99s → ~3.24s (scales reasonably with N)
- Query times:
- Similar_by_Category: ~0.0008–0.0010s (very fast)
- Similar_by_SIMILAR_TO: ~0.0005–0.0007s (very fast)
- Customers_Also_Bought: ~0.01s → 0.036s → 0.13s (grows with interactions)
- User_Recommendations: 4.5s → 39s → 257s (explodes)

- SQL performs very well for simple queries with proper indexing.
- the user-based recommendation query becomes extremely slow as data grows : due to join explosion, large intermediate results, and expensive DISTINCT and subqueries.



✅ SQL query time comparison (seconds)

volume	500	1000	2000
query			
Customers_Also_Bought	0.009573	0.036044	0.130351
Similar_by_Category	0.000756	0.001043	0.000766
Similar_by_SIMILAR_TO	0.000456	0.000608	0.000673
User_Recommendations	4.465854	39.416563	257.434045

NoSQL Graph Design

- **Vertex labels:** product, user, category, brand, tag
- **Edge labels:**
  - IN_CATEGORY, HAS_BRAND, HAS_TAG, PARENT_OF
  - VIEWED, BOUGHT, LIKED
  - SIMILAR_TO(score), BOUGHT_TOGETHER(support), SIMILAR_USER(score)
- Each vertex contains:
  - id (vertex id used in queries)
  - pk (partition key value = run identifier)

NoSQL reset/build cost

- Building the graph is expensive because it requires many network calls and is limited by Cosmos DB throughput.
- Write operations consume Request Units, and rate limiting can slow ingestion.
- Practical impact on our experiments: because graph build/reset already takes minutes at small volumes, we benchmarked N = 500 / 1000 / 2000 to keep total execution time feasible.

Queries Used

- Graph queries are written as **Gremlin** traversals.
- Each query starts from a node and traverses only a few hops.

Graph (Gremlin)

- **Similar_by_Category:** product → category → other products (limit 20)
- **Similar_by_SIMILAR_TO:** product → SIMILAR_TO edges ordered by score (limit 20)
- **Customers_Also_Bought:** product ← BOUGHT by users → other BOUGHT products (limit 20)
- **User_Recommendations:** user → BOUGHT → other users → BOUGHT → products (limit 20)

NoSQL queries used

Q1 Similar by Category

```
g.V([run_pk, product_id])
 .out('IN_CATEGORY')
 .in('IN_CATEGORY')
 .hasLabel('product')
 .dedup()
 .limit(20)
```

Q2 Similar by SIMILAR_TO

```
g.V([run_pk, product_id])
 .outE('SIMILAR_TO')
 .order().by('score', decr)
 .inV()
 .dedup()
 .limit(20)
```

Q3 Customers_Also_Bought

```
g.V([run_pk, product_id])
 .in('BOUGHT')
 .out('BOUGHT')
 .hasLabel('product')
 .dedup()
 .limit(20)
```

Q4 User recommendations

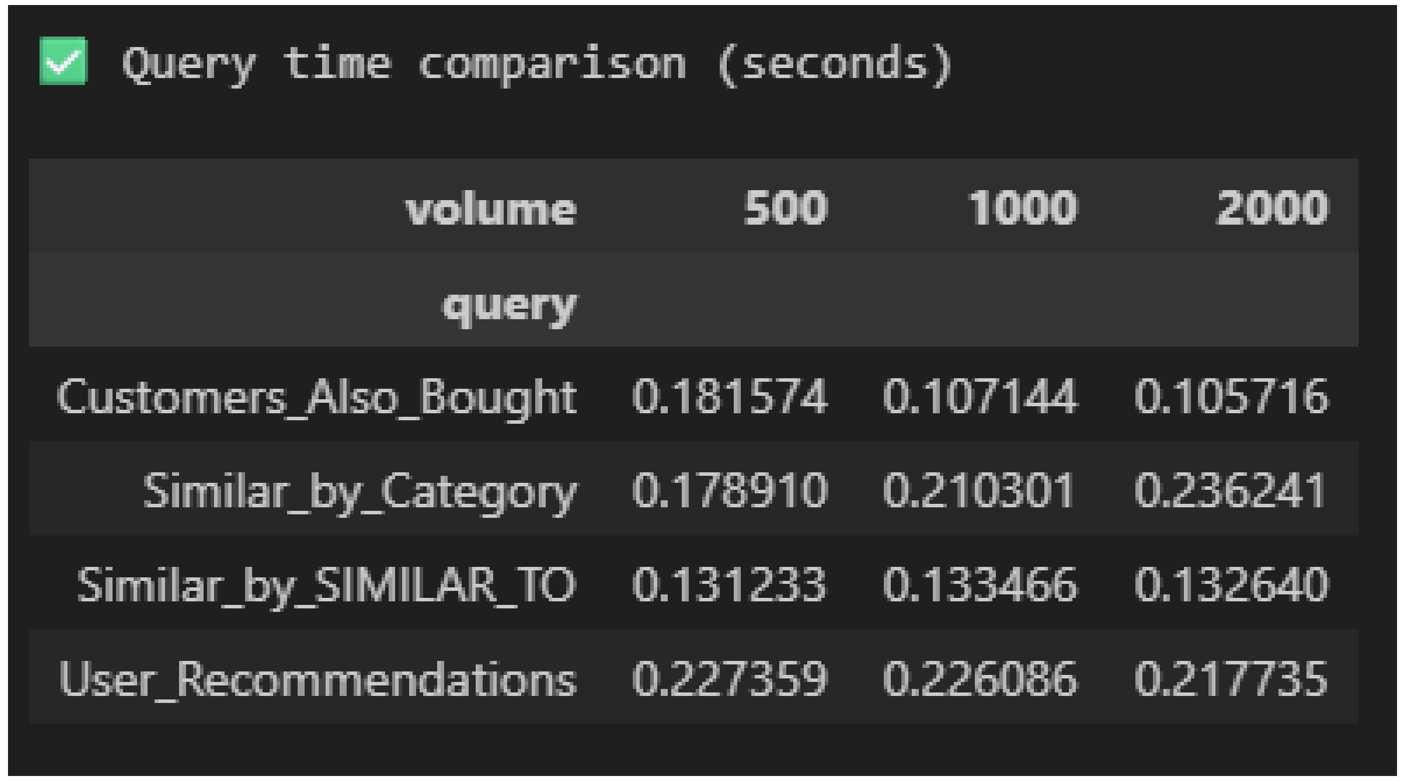
```
g.V([run_pk, user_id])
 .out('BOUGHT')
 .in('BOUGHT')
 .out('BOUGHT')
 .hasLabel('product')
 .dedup()
 .limit(20)
```

NoSQL results summary

(N=500/1000/2000)

- Build time: ~782s → ~1556s → ~3076s (huge; dominated by write + RU limits)
- Query times stayed around 0.10–0.24s across volumes:
- Similar_by_Category: ~0.18 → 0.21 → 0.24
- Similar_by_SIMILAR_TO: ~0.13 stable
- Customers_Also_Bought: ~0.18 → 0.11 → 0.11 (often 0 results because data is sparse)
- User_Recommendations: ~0.23 stable

- Graph query times remain stable as data grows.
  - Traversals are bounded in depth and avoid large intermediate results.
- Recommendations are graph-shaped problems, so the graph database handles them efficiently.



✓ Query time comparison (seconds)

volume	500	1000	2000
query			
Customers_Also_Bought	0.181574	0.107144	0.105716
Similar_by_Category	0.178910	0.210301	0.236241
Similar_by_SIMILAR_TO	0.131233	0.133466	0.132640
User_Recommendations	0.227359	0.226086	0.217735

Results Summary

Build time

- SQL build is very fast (sub-seconds to a few seconds)
- Graph build is very slow (minutes) and grows ~linearly with N

Query time

- **SQL:**
  - 3 queries remain extremely fast (sub-millisecond to milliseconds)
  - User_Recommendations becomes extremely slow as N increases
- **Graph:**
  - All 4 queries remain around ~0.10–0.24s and relatively stable

Trade-offs

- **SQL:**
  - strong consistency
  - good for transactions
  - joins become costly
- **Graph:**
  - natural recommendation modeling
  - fast traversal for relationship queries
  - depends on good data connectivity
- **Best architecture:**
  - Choose SQL for relational workloads + precomputed recos
  - Choose Graph DB when online traversals are core to the application

THANK YOU