Relational vs NoSQL

Performance and Efficiency in Comparison

Pedicillo Marco
1983285

Santavicca Federico
2003442

Why this study?

This project provides a comparative study between **PostgreSQL**, a relational database, and **Neo4j**, a graph database, using a large-scale dataset as benchmark.

The goal is to analyze **query performance** and **execution efficiency**, highlighting how certain queries are naturally optimized for one data model while becoming complex or inefficient for another.

Relational databases have long been the backbone of data management, ensuring stability and efficiency with structured information. The rise of graph databases reflects **the growing need to model complex connections** and has led to their increasing adoption.

MovieLens 32M

The MovieLens 32M dataset is a large-scale collection of movie ratings enriched with tags and metadata, widely used for evaluating recommender systems and database performance.

200,948
Distinct Users

87,585
Movies - **19** Genres

32,000,204
Ratings

1,993,727
Tags

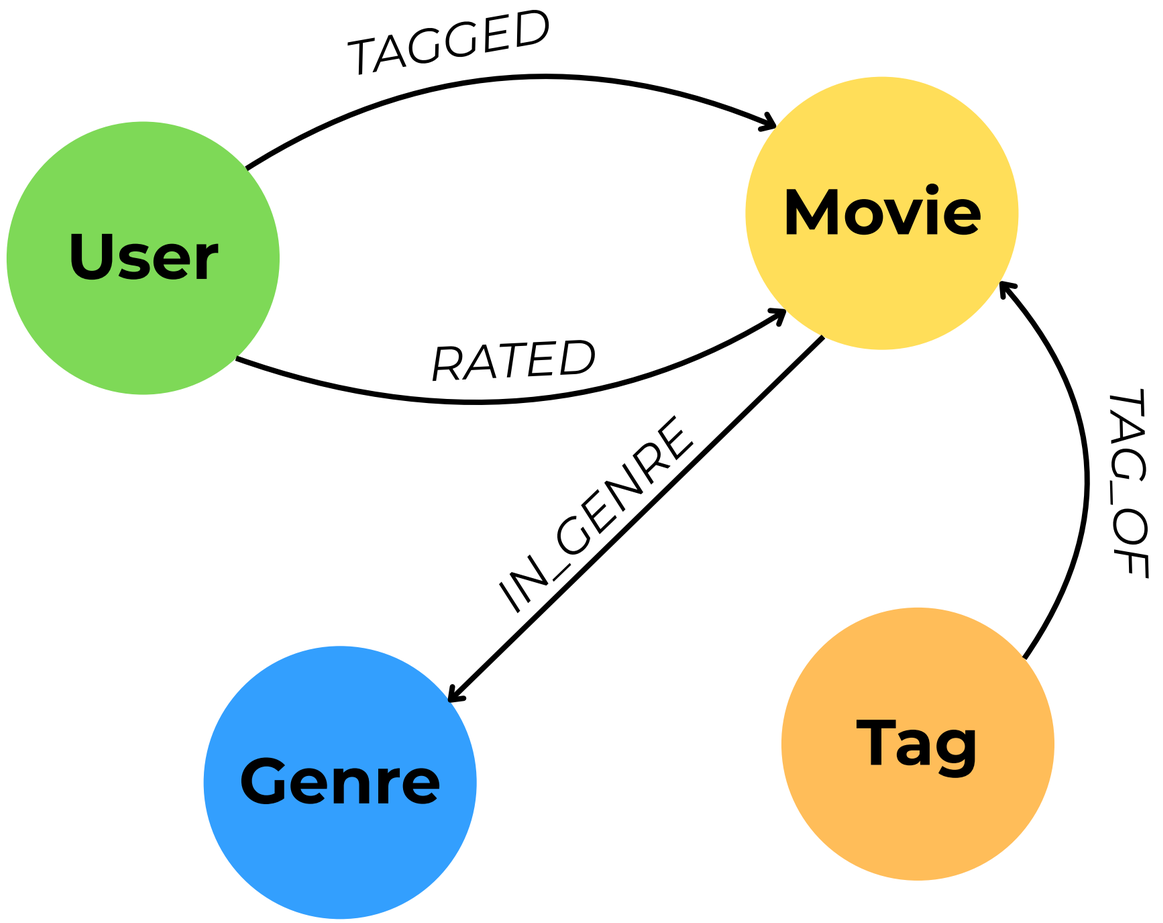
Building the Relational Model

The dataset was originally provided as CSV files, which were restructured into a relational schema in PostgreSQL. Separate tables were created for core entities (movies, users, genres) and bridge tables captured relationships such as ratings, tags, and movie–genre links. This normalization turned raw files into a consistent structure, enabling efficient querying and analysis.

Tables	Attributes
Users	userid
Movies	movieid, title, genres
Ratings	userid, movieid, rating, timestamp
Tags	tagid, name
Genres	genreid, name
Movie_genres	movieid, genreid
Tag_of	tagid, movieid
User_tags	userid, tagid, timestamp

Building the Graph Model

We imported the original CSV files into Neo4j. The graph was built with **429,097 nodes** in total, consisting of about 200k Users, 87k Movies, 140k Tags, and 19 Genres. Relationships capture user activity and metadata: in total ~**35 million relationships**, including ratings, tagging actions, tag assignments, and genre associations. This structure provides a clean and connected representation of the dataset, optimized for graph queries and traversal.



Performance Comparison

Goal

Assess efficiency, syntax, and structural differences in query execution between graph (Neo4j) and relational (PostgreSQL) databases.

Focus

Complex scenarios involving multi-hop traversals, user/movie similarity, recommendations, and indirect relationship analysis.

Method

- Implement the same queries in Cypher (Neo4j) and SQL (PostgreSQL).
- Compare execution times.
- Highlight when and why one model outperforms the other.

Queries Overview

1. One-Hop User-Based Recommendations

Finds users similar to the target by shared highly rated movies (≥4), then collects new movies they liked that the target has not seen, ranking candidates by user support and relevance.

2. Top Users Similar to Target Movie (by Genres)

Finds users who liked a given movie and identifies the top 20 with strong activity in its genres, ranked by total likes, genre coverage, and average rating.

3. Two-Hop Similar-User Recommendations with Genre Overlap

Recommends movies liked by second-degree similar users, excluding those already rated by the target, keeping only items with genre overlap and avg rating ≥ 4, ranked by popularity and rating.

4. Multi-Hop User Similarity Exploration

Finds users similar to a starting user by traversing up to three hops through movies with close ratings (≥4, difference ≤1), avoiding cycles. Returns the nearest similar users with their hop distance.

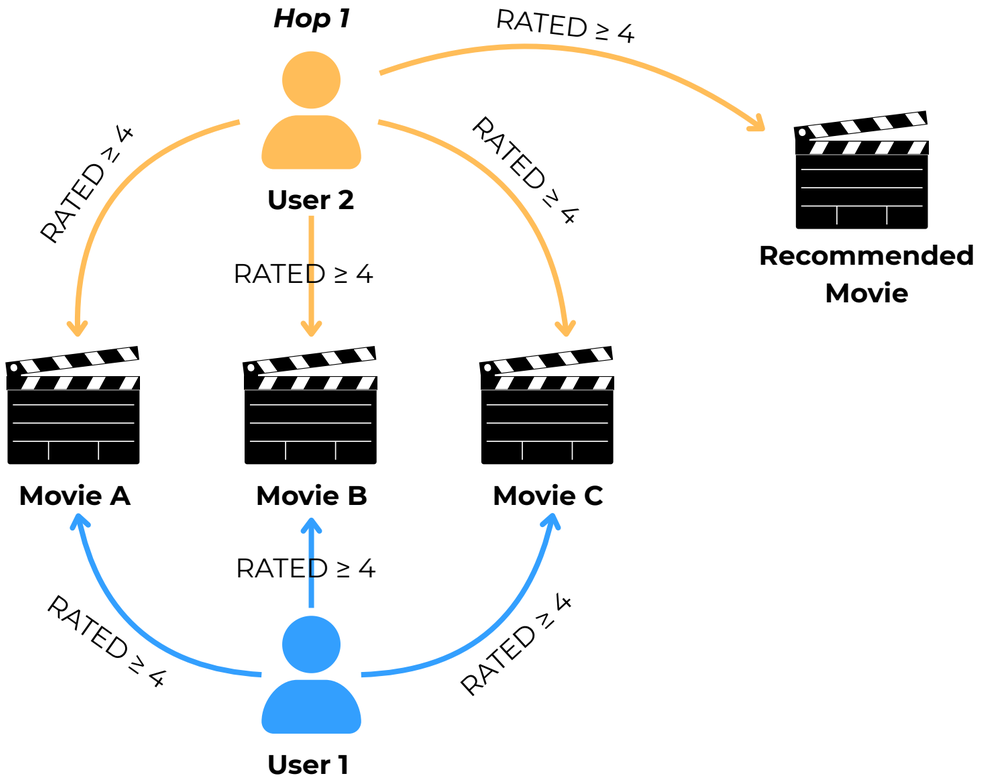
1: One-Hop User-Based Recommendations

Goal

Recommend unseen movies for a Target User by leveraging graph-based similarity: starting from the movies the user liked, identify other users with overlapping preferences and suggest new movies they highly rated, ranked by relevance.

Approach

The query identifies movies the target user liked, finds similar users with overlapping preferences (at least 3), and collects highly rated movies they enjoyed but the target has not seen, ranking the results by support and relevance.



1: PostgreSQL Query

1-3: We set the parameters.

4-8: Select all *movieid* the target has rated ≥ min_like.

9-17: Find other users who have rated the same movies liked by the target.

18-29: Collects candidate movies from similar users, aggregates them by candidate movie, and calculates the score as the sum of the 'weight-similarity' (the more movies a user had in common, the more they contribute).

30-34: Select title, sort by descending score, and take the top 20 recommended.

```
1  WITH params AS (
2    SELECT 123::int AS target_user, 4.0::numeric AS min_like, 3::int AS min_common
3  ),
4  user_likes AS (
5    SELECT r.movieid
6    FROM ratings r JOIN params p ON TRUE
7    WHERE r.userid = p.target_user AND r.rating >= p.min_like
8  ),
9  similar_users AS (
10   SELECT r.userid, COUNT(DISTINCT r.movieid) AS common_likes
11   FROM ratings r
12   JOIN user_likes ul ON ul.movieid = r.movieid
13   JOIN params p ON TRUE
14   WHERE r.rating >= p.min_like AND r.userid <> p.target_user
15   GROUP BY r.userid
16   HAVING COUNT(DISTINCT r.movieid) >= (SELECT min_common FROM params)
17 ),
18 candidates AS (
19   SELECT r.movieid, SUM(su.common_likes) AS score
20   FROM similar_users su
21   JOIN ratings r ON r.userid = su.userid
22   JOIN params p ON TRUE
23   WHERE r.rating >= p.min_like
24     AND NOT EXISTS (
25       SELECT 1 FROM ratings ru
26       WHERE ru.userid = p.target_user AND ru.movieid = r.movieid
27     )
28   GROUP BY r.movieid
29 )
30 SELECT m.movieid, m.title, c.score
31 FROM candidates c
32 JOIN movies m ON m.movieid = c.movieid
33 ORDER BY c.score DESC, m.title
34 LIMIT 20;
```

1: Neo4J Query

1: Select the Target User.

3-4: Select all liked movie the target has rated ≥ 4.

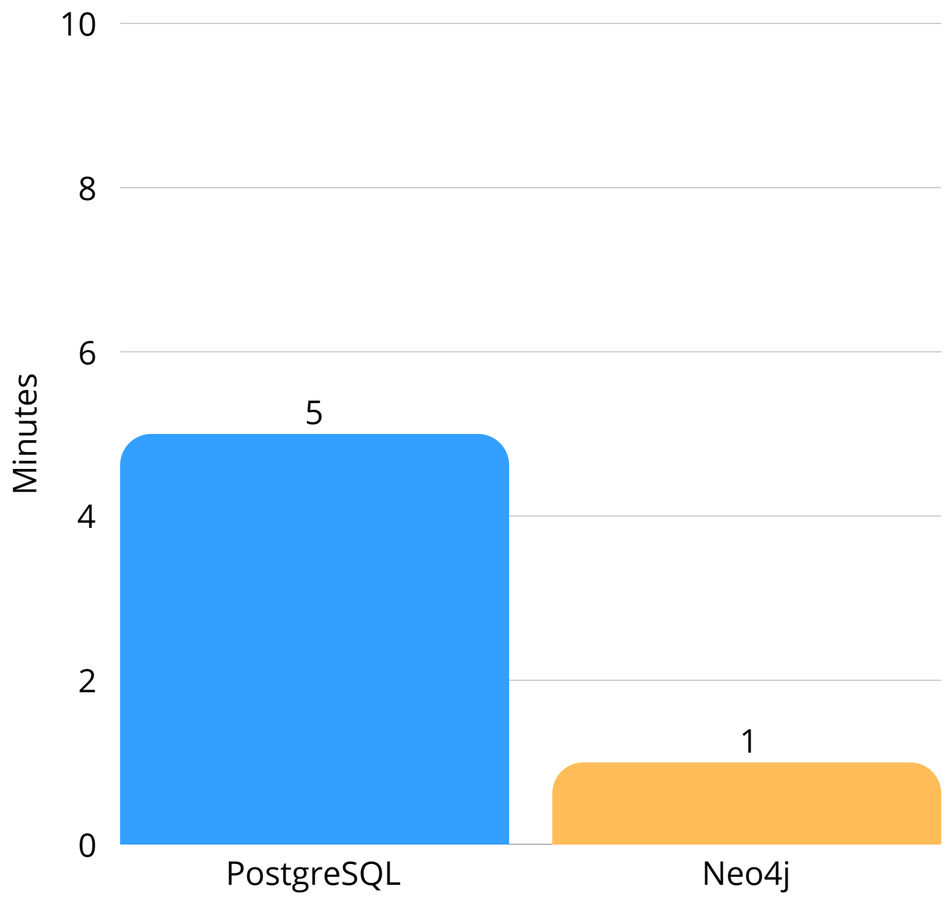
6-9: Finds users *u2* who rated ≥ 4 the same movies as the target and keeps those with at least three in common.

11-12: From similars *u2 we* take the films nominated *rec* that they have rated ≥ 4 but that the target has never seen.

14-18: It assigns each candidate a weight based on shared movies, sums these weights across peers, and returns the top 20 by score.

```
1  MATCH (u:User {userId: 123})
2
3  MATCH (u)-[r1:RATED]->(m:Movie)
4  WHERE r1.rating >= 4
5
6  MATCH (u2:User)-[r2:RATED]->(m)
7  WHERE r2.rating >= 4 AND u2 <> u
8  WITH u, u2, collect(DISTINCT m) AS common
9  WHERE size(common) >= 3
10
11 MATCH (u2)-[r3:RATED]->(rec:Movie)
12 WHERE r3.rating >= 4 AND NOT (u)-[:RATED]->(rec)
13
14 WITH rec, size(common) AS weight
15 RETURN rec.movieId AS movieId, rec.title AS title,
16        sum(weight) AS score
17 ORDER BY score DESC, title
18 LIMIT 20;
```

1: Comparison



Why PostgreSQL is slower?

- Multiple self-joins on massive tables

Why Neo4J is faster?

- Native graph model optimized for traversals
- Direct neighbor expansion without costly joins

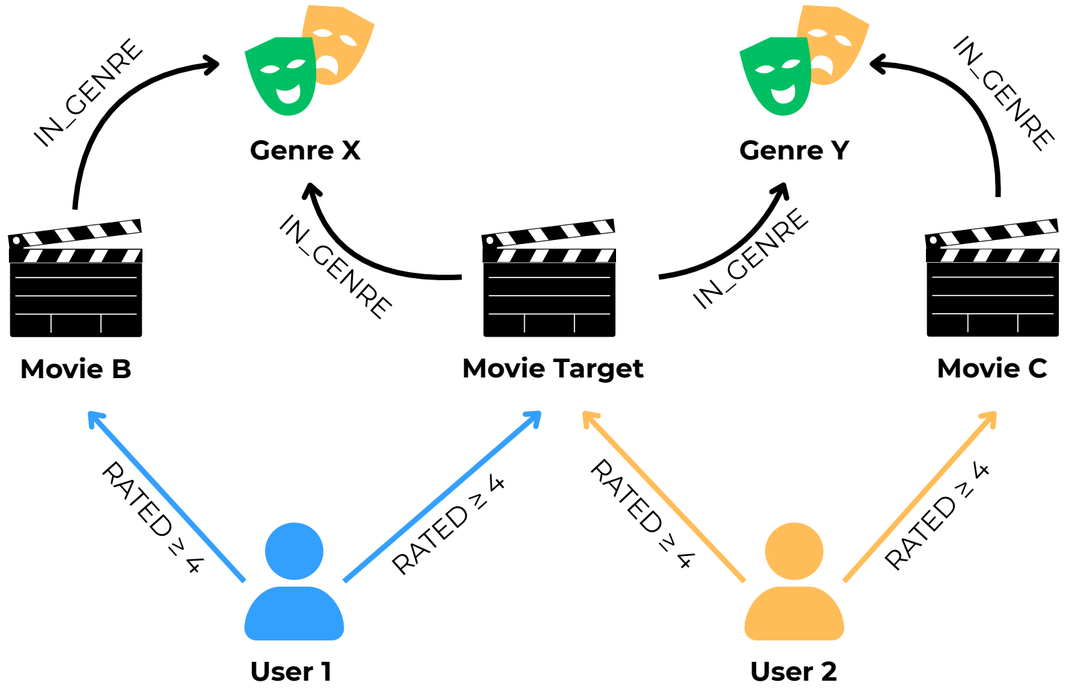
2: Top Users Similar to Target Movie (by Genres)

Goal

Identify the top 20 users most aligned with a Target Movie (e.g., Movie 50), based on their preferences in its genres.

Approach

The genres of the target movie are identified and users who rated it positively are selected. Their likes in the same genres are counted, keeping only those with enough per-genre likes, sufficient genre coverage, and a minimum total. The average rating is then computed, and the top 20 users are ranked by engagement and rating quality.



2: PostgreSQL Query

1-7: We set the parameters.

9-13: Extracts the *genreid* associated with the target movie.

15-20: Gets users who gave rating ≥ *min_like* to the target movie.

22-34: For each user who liked the target, consider only target genres and ratings ≥ *min_likes*, excluding the target film.
Count distinct liked films per <user, genre> (*likes_in_genre*) and keep pairs with *likes_in_genre* ≥ *min_per_genre*.
Result: target genres where each user is active and their like count in each.

```
1  WITH params AS (
2    SELECT 50::int AS target_movie,
3           4.0::numeric AS min_like,
4           3::int AS min_per_genre,
5           2::int AS min_coverage_genres,
6           20::int AS min_total_likes
7  ),
8
9  target_genres AS (
10   SELECT mg.genreid
11   FROM movie_genres mg
12   JOIN params p ON mg.movieid = p.target_movie
13 ),
14
15 users_liked_target AS (
16   SELECT DISTINCT r.userid
17   FROM ratings r
18   JOIN params p ON r.movieid = p.target_movie
19   WHERE r.rating >= (SELECT min_like FROM params)
20 ),
21
22 user_genre_counts AS (
23   SELECT r.userid,
24          mg.genreid,
25          COUNT(DISTINCT r.movieid) AS likes_in_genre
26   FROM ratings r
27   JOIN movie_genres mg ON mg.movieid = r.movieid
28   JOIN target_genres tg ON tg.genreid = mg.genreid
29   JOIN users_liked_target ult ON ult.userid = r.userid
30   WHERE r.rating >= (SELECT min_like FROM params)
31     AND r.movieid <> (SELECT target_movie FROM params)
32   GROUP BY r.userid, mg.genreid
33   HAVING COUNT(DISTINCT r.movieid) >= (SELECT min_per_genre FROM params)
34 ),
```

2: PostgreSQL Query

36-42: Group by user *coverage_genres* (number of target genres the user covers) and *total_likes_in_target_genres* (sum of likes_in_genre across those genres).

44-53: For each user, calculate the average ratings of films in the target genres, with rating ≥ *min_like*.

55-66: The query combines aggregated information for each user and applies a series of filters to select only those with a genuine affinity for the target film, returning the top 20 most aligned users

```
36  user_agg AS (
37    SELECT ugc.userid,
38           COUNT(DISTINCT ugc.genreid) AS coverage_genres,
39           SUM(ugc.likes_in_genre) AS total_likes_in_target_genres
40    FROM user_genre_counts ugc
41    GROUP BY ugc.userid
42  ),
43
44  user_avg AS (
45    SELECT r.userid,
46           AVG(r.rating) AS avg_rating_in_target_genres
47    FROM ratings r
48    JOIN movie_genres mg ON mg.movieid = r.movieid
49    JOIN target_genres tg ON tg.genreid = mg.genreid
50    WHERE r.rating >= (SELECT min_like FROM params)
51      AND r.movieid <> (SELECT target_movie FROM params)
52    GROUP BY r.userid
53  )
54
55  SELECT ua.userid,
56         ua.coverage_genres,
57         ua.total_likes_in_target_genres,
58         COALESCE(uav.avg_rating_in_target_genres, 0) AS avg_rating_in_target_genres
59  FROM user_agg ua
60  JOIN user_avg uav ON uav.userid = ua.userid
61  WHERE ua.coverage_genres >= (SELECT min_coverage_genres FROM params)
62    AND ua.total_likes_in_target_genres >= (SELECT min_total_likes FROM params)
63  ORDER BY ua.total_likes_in_target_genres DESC,
64           ua.coverage_genres DESC,
65           avg_rating_in_target_genres DESC
66  LIMIT 20;
```

2: Neo4J Query

1-2: We set the parameters.

4-6: Select the movie *m* and collect the list of genres it belongs to.

8-11: Select users who gave the target movie a rating ≥ *minLike*.

13-18: For each target's genre *tg*, count distinct films user *u* rated ≥ *minLike* (excluding the target). Keep genres with ≥ *minPerGenre* likes.
The result is the list of target genres in which the user is actually active and the related count.

```
1  WITH 50 AS targetMovieId, 4.0 AS minLike,
2       3 AS minPerGenre, 2 AS minCoverageGenres, 20 AS minTotalLikes
3
4  MATCH (m:Movie {movieId: targetMovieId})-[:IN_GENRE]->(g:Genre)
5  WITH m, collect(DISTINCT g) AS targetGenres, minLike, minPerGenre,
6       minCoverageGenres, minTotalLikes
7
8  MATCH (m)<-[rt:RATED]-(u:User)
9  WHERE rt.rating >= minLike
10 WITH m, targetGenres, minLike, minPerGenre, minCoverageGenres,
11      minTotalLikes, u
12
13 UNWIND targetGenres AS tg
14 MATCH (u)-[r:RATED]->(fm:Movie)-[:IN_GENRE]->(tg)
15 WHERE r.rating >= minLike AND fm <> m
16 WITH m, targetGenres, minLike, minPerGenre, minCoverageGenres,
17      minTotalLikes, u, tg, COUNT(DISTINCT fm) AS likesInGenre
18 WHERE likesInGenre >= minPerGenre
```

2: Neo4J Query

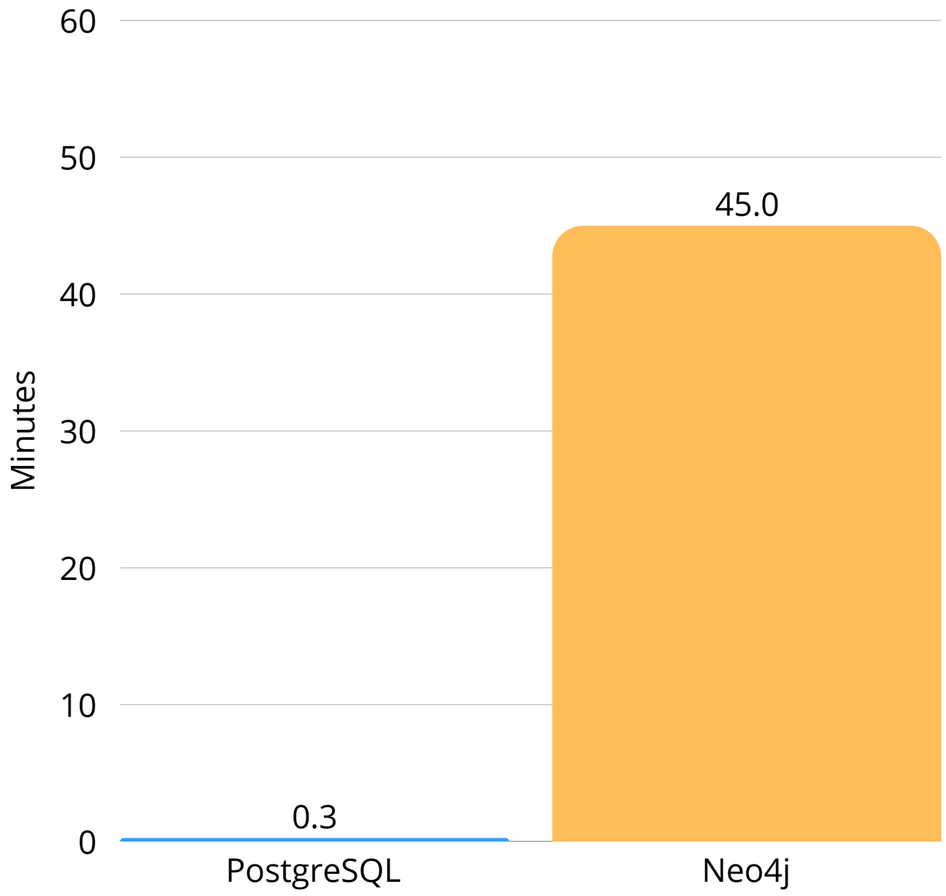
20-25: Group by user *coverageGenres* (number of target genres covered) and *totalLikes* (sum of likes in those genres).

27-29: Collects *m2* movies that the user rated ≥4 in the target genres.

31-40: Calculates the average of the user ratings on the movies collected in the previous point (*avgRating*) and and returns the Top 20 users most related to the target movie,

```
20 WITH m, targetGenres, minLike, minPerGenre, minCoverageGenres,
21      minTotalLikes, u,
22      COUNT(DISTINCT tg) AS coverageGenres,
23      SUM(likesInGenre)  AS totalLikes
24 WHERE coverageGenres >= minCoverageGenres AND
25       totalLikes >= minTotalLikes
26
27 MATCH (u)-[r2:RATED]->(m2:Movie)-[:IN_GENRE]->(g2:Genre)
28 WHERE r2.rating >= minLike AND m2 <> m AND g2 IN targetGenres
29 WITH u, coverageGenres, totalLikes, collect(DISTINCT m2) AS umovies
30
31 UNWIND umovies AS um
32 MATCH (u)-[r3:RATED]->(um)
33 WITH u, coverageGenres, totalLikes, avg(r3.rating) AS avgRating
34 RETURN u.userId AS userId,
35        coverageGenres,
36        totalLikes AS totalLikesInTargetGenres,
37        avgRating  AS avgRatingInTargetGenres
38 ORDER BY totalLikesInTargetGenres DESC, coverageGenres DESC,
39          avgRating DESC
40 LIMIT 20;
```

2: Comparison



Why PostgreSQL is faster?

- Powerful query planner for joins and groups
- Optimized aggregations on large tables

Why Neo4J is slower?

- UNWIND + DISTINCT cause row explosion
- Relationship filters applied after expansion

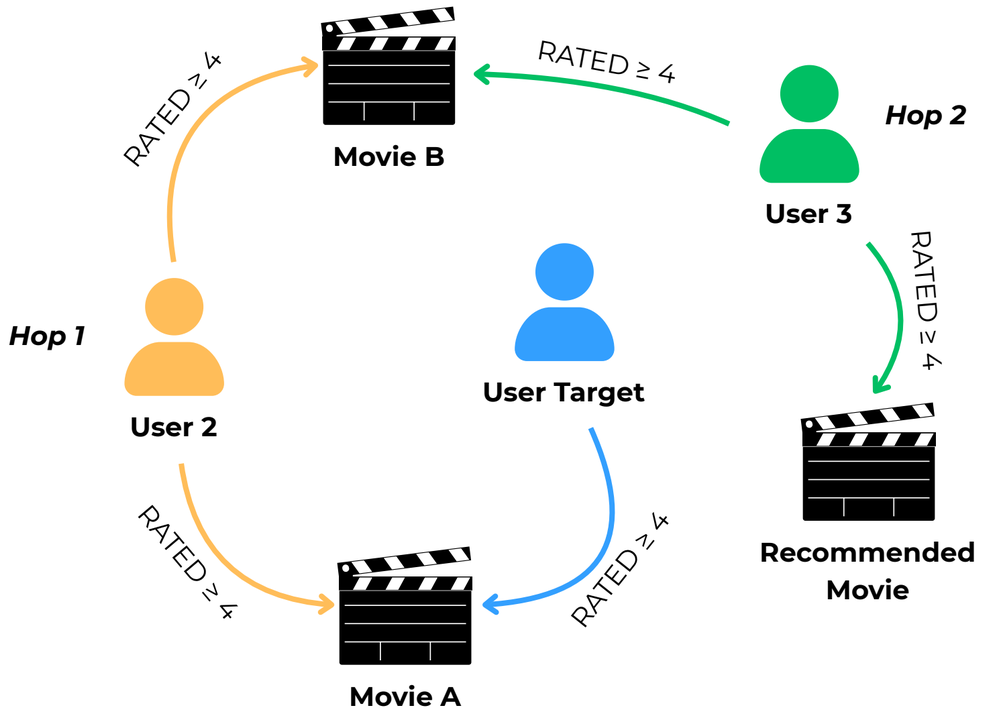
3: Two-Hop Similar-User Recommendations

Goal

Recommend unseen movies to a Target User by leveraging two-hop connections: starting from the user's liked movies → to similar users (first hop) → to their neighbors (second hop).

Approach

Identify the target user's positively rated movies and genres, find first-hop similar users with overlapping tastes, expand to second-hop users, and collect movies they liked but the target hasn't seen. Filter by rating and genre overlap, then rank the top candidates.



3: PostgreSQL Query

1-5: Select the movies the user liked.

6-11: Compute the target's frequented genres by counting how often each genre appears among the films the target liked.

12-19: Find top-level users similar to the target: users who rated the same movies as the target, for at least 5 movies in common.

20-30: Find similars of similars (second level): users who co-rate the same movies as lvl1, at least 5 co-ratings, excluding both the target and lvl1 users.

```sql
WITH target_high AS (
  SELECT r.movieid, r.rating
  FROM ratings r
  WHERE r.userid = 123 AND r.rating >= 4
),
target_genres AS (
  SELECT mg.genreid, COUNT(*) AS cnt
  FROM target_high th
  JOIN movie_genres mg ON mg.movieid = th.movieid
  GROUP BY mg.genreid
),
similar_lvl1 AS (
  SELECT r2.userid
  FROM ratings r2
  JOIN target_high th ON th.movieid = r2.movieid
  WHERE r2.userid <> 123 AND ABS(r2.rating - th.rating) <= 1
  GROUP BY r2.userid
  HAVING COUNT(*) >= 5
),
similar_lvl2 AS (
  SELECT r3.userid
  FROM ratings r3
  JOIN ratings rL1 ON r3.movieid = rL1.movieid
  WHERE rL1.userid IN (SELECT userid FROM similar_lvl1)
    AND r3.userid NOT IN (SELECT userid FROM similar_lvl1)
    AND r3.userid <> 123
    AND ABS(r3.rating - rL1.rating) <= 1
  GROUP BY r3.userid
  HAVING COUNT(*) >= 5
),
```

3: PostgreSQL Query

31-38: Collects the candidate films: these are the films rated ≥ 4 by lvl2 users, not yet voted by the target.

39-47: For each candidate film, calculate how many genres the film has (*movie_genres*) and how many genres of the film are also among the target genres (*overlap_with_user*).

48-58: Selects candidate films liked by second-level users, keeps those with an average rating ≥ 4 and at least two shared genres with the target, and ranks them by supporting users and average rating to return the top 10 recommendations.

```
31  candidates AS (
32    SELECT r.movieid, r.userid, r.rating
33    FROM ratings r
34    WHERE r.userid IN (SELECT userid FROM similar_lvl2)
35      AND r.rating >= 4
36      AND NOT EXISTS (SELECT 1 FROM ratings t WHERE t.userid = 123
37                      AND t.movieid = r.movieid)
38  ),
39  genre_coverage AS (
40    SELECT c.movieid, COUNT(DISTINCT mg.genreid) AS movie_genres,
41           COUNT(DISTINCT CASE WHEN tg.genreid = mg.genreid THEN mg.genreid END)
42           AS overlap_with_user
43    FROM candidates c
44    JOIN movie_genres mg ON mg.movieid = c.movieid
45    LEFT JOIN target_genres tg ON tg.genreid = mg.genreid
46    GROUP BY c.movieid
47  )
48  SELECT m.title,
49         COUNT(DISTINCT c.userid) AS num_lvl2_users,
50         AVG(c.rating) AS avg_rating,
51         gc.overlap_with_user
52  FROM candidates c
53  JOIN movies m ON m.movieid = c.movieid
54  JOIN genre_coverage gc ON gc.movieid = c.movieid
55  GROUP BY m.title, gc.overlap_with_user
56  HAVING AVG(c.rating) >= 4.0 AND gc.overlap_with_user >= 2
57  ORDER BY num_lvl2_users DESC, avg_rating DESC
58  LIMIT 10;
```

3: Neo4J Query

1-7: Set the parameters.

9-11: Select the *userId* and the movies rated ≥ *minLike*. Save the set as *liked*.

13-14: From *liked* films, we extracts the genres and collects them into *targetGenres*.

16-20: Find *u1* who co-rated the target's movies with ≥ *minLike*; keep those with *common1* ≥ *minCommon1* shared films and collect them as *hop1*.

22-28: For each *h1* ∈ *hop1*, find users *u2* who co-vote movies with *h1* with rating ≥ *minLike*.

```
1  :param uid => 123;
2  :param minLike => 4.0;
3  :param minCommon1 => 5;
4  :param minCommon2 => 5;
5  :param minGenreOverlap => 2;
6  :param minAvgRating => 4.0;
7  :param k => 10;
8
9  MATCH (u:User {userId: $uid})-[r0:RATED]->(m0:Movie)
10 WHERE r0.rating >= $minLike
11 WITH u, collect(DISTINCT m0) AS liked
12
13 MATCH (m0)-[:IN_GENRE]->(g0:Genre)
14 WITH u, liked, collect(DISTINCT g0) AS targetGenres
15
16 MATCH (u)-[r1:RATED]->(m:Movie)<-[r2:RATED]-(u1:User)
17 WHERE r1.rating >= $minLike AND r2.rating >= $minLike AND u1 <> u
18 WITH u, liked, targetGenres, u1, COUNT(DISTINCT m) AS common1
19 WHERE common1 >= $minCommon1
20 WITH u, liked, targetGenres, collect(DISTINCT u1) AS hop1
21
22 UNWIND hop1 AS h1
23 MATCH (h1)-[rh:RATED]->(m1:Movie)<-[rj:RATED]-(u2:User)
24 WHERE rh.rating >= $minLike AND rj.rating >= $minLike
25   AND u2 <> h1 AND u2 <> u AND NOT u2 IN hop1
26 WITH u, liked, targetGenres, u2, COUNT(DISTINCT m1) AS common2
27 WHERE common2 >= $minCommon2
28 WITH u, liked, targetGenres, collect(DISTINCT u2) AS hop2
```

3: Neo4J Query

30-33: From *hop2*, collect films rated ≥ *minLike* but not in *liked*; for each, gather supporters (hop2 users who rated it).

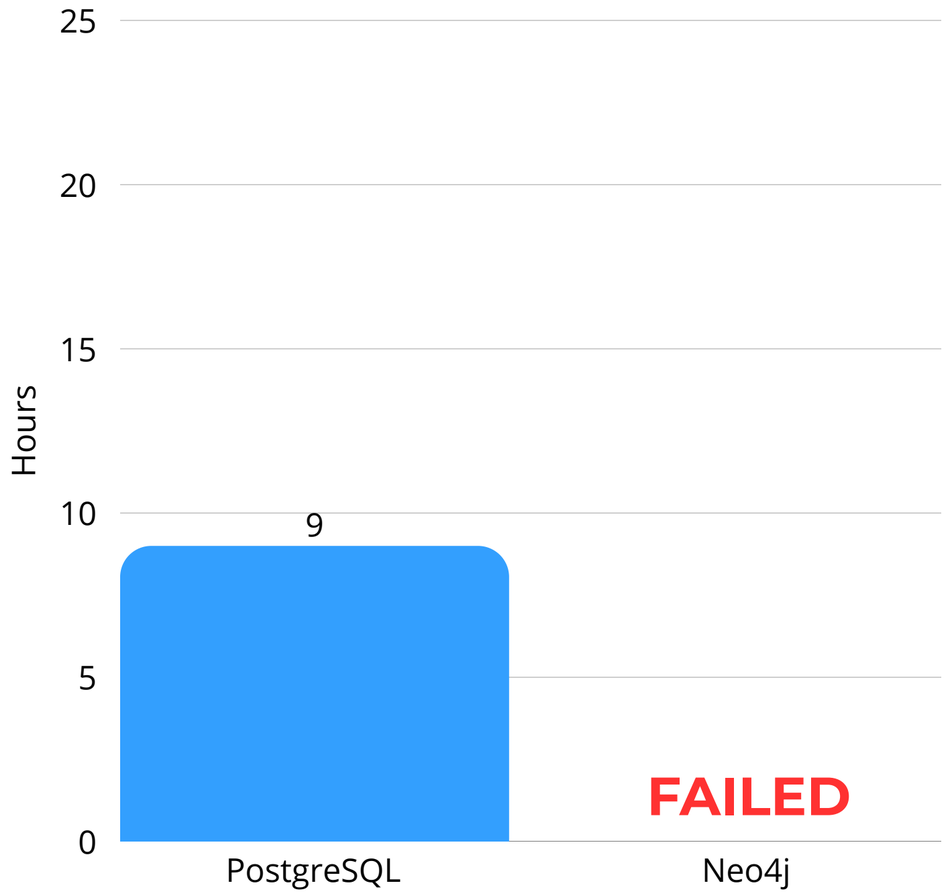
35-38: Computes how many movie genres belong to *targetGenres* (*genreOverlap*) and keeps only those with *genreOverlap* ≥ *minGenreOverlap*.

40-43: Computes the average supporters' rating for each *rec* (*avgRating*) and keeps only those with *avgRating* ≥ *minAvgRating*.

45-52: Return the top *k* movies with their supporters, av rating, and title.

```
30 UNWIND hop2 AS su
31 MATCH (su)-[r4:RATED]->(rec:Movie)
32 WHERE r4.rating >= $minLike AND NOT rec IN liked
33 WITH liked, targetGenres, rec, collect(DISTINCT su) AS supporters
34
35 MATCH (rec)-[:IN_GENRE]->(rg:Genre)
36 WHERE rg IN targetGenres
37 WITH rec, supporters, COUNT(DISTINCT rg) AS genreOverlap
38 WHERE genreOverlap >= $minGenreOverlap
39
40 UNWIND supporters AS s
41 MATCH (s)-[r5:RATED]->(rec)
42 WITH rec, supporters, genreOverlap, AVG(r5.rating) AS avgRating
43 WHERE avgRating >= $minAvgRating
44
45 RETURN
46   rec.movieId              AS movieId,
47   rec.title                AS title,
48   SIZE(supporters)         AS num_lvl2_supporters,
49   round(avgRating, 2)      AS avg_rating_lvl2,
50   genreOverlap
51 ORDER BY num_lvl2_supporters DESC, avg_rating_lvl2 DESC, title
52 LIMIT $k;
```

3: Comparison



Why PostgreSQL is better but slow?

- Heavy joins and groupings on tens of millions of ratings
- Mature optimizer manages memory usage

Why Neo4J fails?

- Path expansion explodes in memory
- Collect/UNWIND builds huge lists

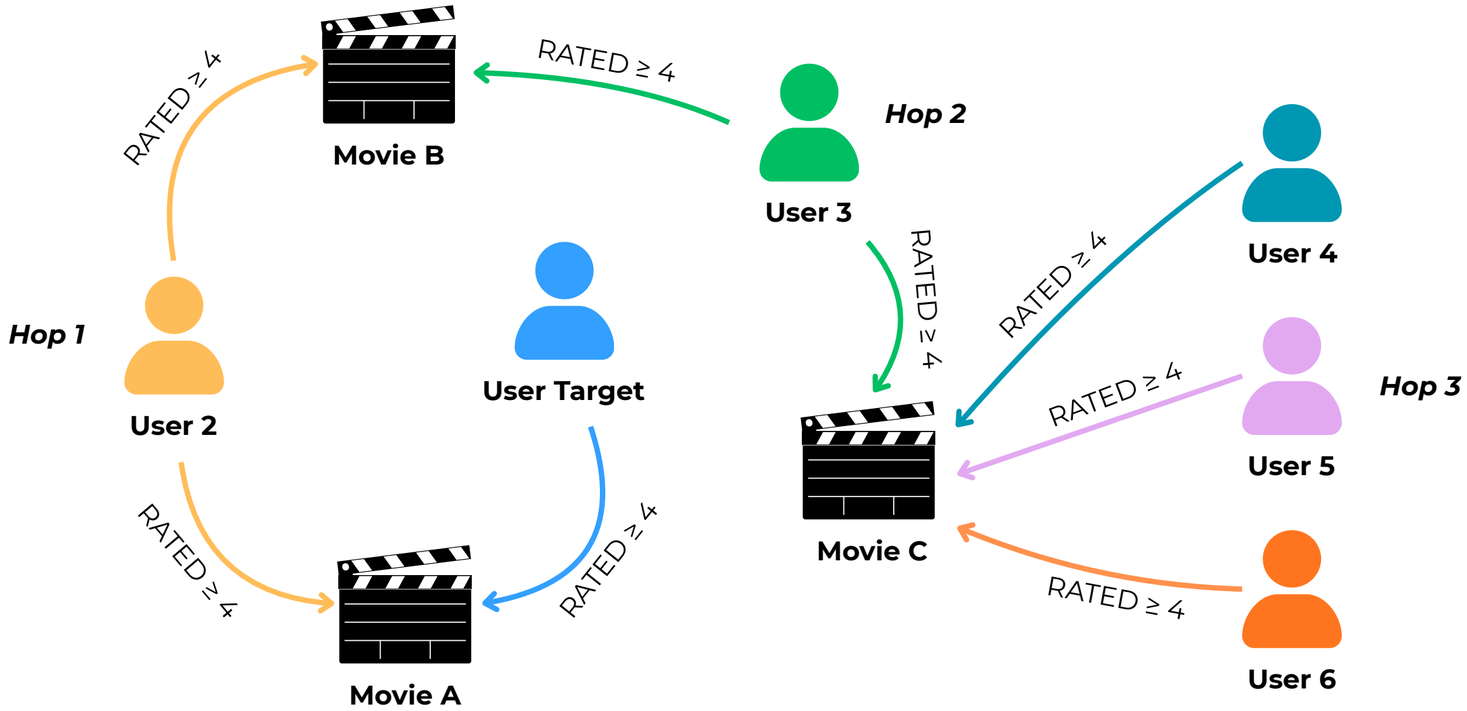
4: Multi-Hop User Similarity Exploration

Goal

Identify users related to the Target through up to 3 steps of common high-rated movies, while preventing loops.

Approach

The process starts from movies positively rated (≥4) by Target User, then moves to other users with similar scores (hop 1). It continues up to two more hops through users with comparable ratings, avoiding cycles. The result is a set of distinct users, each linked to the target by their minimum hop distance.



4: PostgreSQL Query

1-3: Defines the starting user.
Seed is the starting point of the traversal in the user graph.

4-18: For each film the target rated ≥ 4, it searches for users who rated the same film with a score no more than one point apart.
Each such pair is stored as a level-1 path (*depth = 1*) with the target, the similar user, and their connecting path (*users_path*), forming the basis for the next recursive expansions.

```
1  WITH RECURSIVE seed AS (
2    SELECT 1::int AS u0
3  ),
4  paths AS (
5    SELECT
6        1 AS depth,
7        ARRAY[seed.u0, r2.userid] AS users_path,
8        r2.userid AS current_user
9    FROM seed
10   JOIN ratings r1
11     ON r1.userid = seed.u0 AND r1.rating >= 4
12   JOIN ratings r2
13     ON r2.movieid = r1.movieid
14    AND r2.userid <> r1.userid
15    AND r2.rating >= 4
16    AND ABS(r1.rating - r2.rating) <= 1
17
18   UNION ALL
19
```

4: PostgreSQL Query

20-34: This block recursively expands the network of similar users.
From each *current_user*, it finds others who rated common movies ≥ 4 with a difference ≤ 1, increases depth (+1), and adds them to *users_path*.
The process continues up to depth 3, avoiding repeats, to build a multi-level graph of users with similar tastes.

35-41: Groups by each found user (*current_user*), keeping the minimum depth (*hop_distance*).
Sorts by distance (closest first) and then by ID, returning the 20 most similar users reached in the fewest hops.

```
20    SELECT
21        p.depth + 1,
22        users_path || r2.userid,
23        r2.userid
24    FROM paths p
25    JOIN ratings r1
26      ON r1.userid = p.current_user AND r1.rating >= 4
27    JOIN ratings r2
28      ON r2.movieid = r1.movieid
29     AND r2.userid <> r1.userid
30     AND r2.rating >= 4
31     AND ABS(r1.rating - r2.rating) <= 1
32    WHERE p.depth < 3
33      AND NOT r2.userid = ANY (users_path)
34  )
35  SELECT current_user AS similar_user,
36         MIN(depth)  AS hop_distance
37  FROM paths
38  WHERE depth >= 1
39  GROUP BY current_user
40  ORDER BY hop_distance, similar_user
41  LIMIT 20;
```

4: Neo4J Query

1-5: Starts from the Target User and finds users *u1* who rated the same movies *m0* with scores ≥ 4 and within 1 point of the target's rating, returning all directly similar users (first hop).

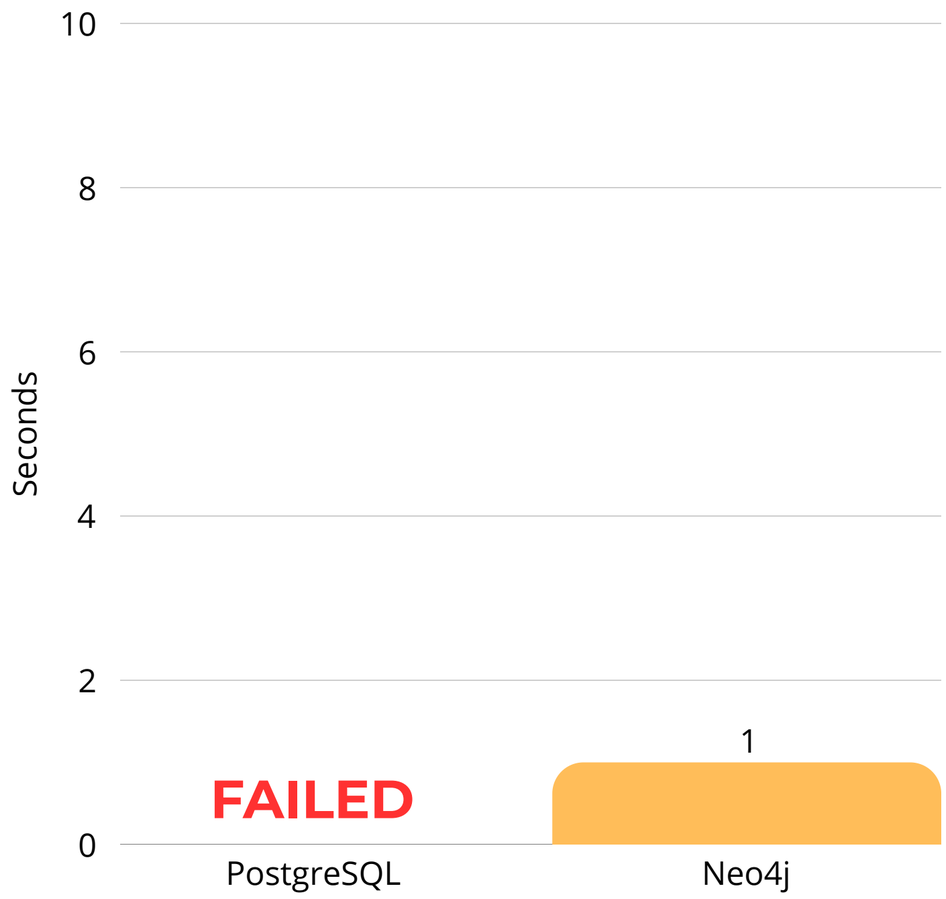
7-12: From each similar user *u1*, find other users *u2* who share similar liked movies and ratings (second hop).

14-20: Same scheme, but now it expands to the similar-of-similar-of-similar users *u3* (third hop).

22-27: Returns user paths up to the third hop, listing *startUser*, *hop1*, *hop2*, and *hop3*, with up to 20 distinct results.

```
1  MATCH (u0:User {userId: 1})-[r0:RATED]->(m0:Movie)<-[r1:RATED]-(u1:User)
2  WHERE r0.rating >= 4
3    AND r1.rating >= 4
4    AND abs(r0.rating - r1.rating) <= 1
5    AND u0 <> u1
6
7  OPTIONAL MATCH (u1)-[r2:RATED]->(m1:Movie)<-[r3:RATED]-(u2:User)
8  WHERE r2.rating >= 4
9    AND r3.rating >= 4
10   AND abs(r2.rating - r3.rating) <= 1
11   AND u1 <> u2
12   AND u0 <> u2
13
14 OPTIONAL MATCH (u2)-[r4:RATED]->(m2:Movie)<-[r5:RATED]-(u3:User)
15 WHERE r4.rating >= 4
16   AND r5.rating >= 4
17   AND abs(r4.rating - r5.rating) <= 1
18   AND u2 <> u3
19   AND u1 <> u3
20   AND u0 <> u3
21
22 RETURN DISTINCT
23   u0.userId AS startUser,
24   u1.userId AS hop1,
25   u2.userId AS hop2,
26   u3.userId AS hop3
27 LIMIT 20;
```

4: Comparison



Why PostgreSQL fails?

- Recursive self-joins on ratings cause massive intermediate results
- Huge GROUP BY and DISTINCT operations exceed memory and spill to disk

Why Neo4J is better?

- Direct graph traversal avoids costly joins
- Step-by-step pruning keeps the search space small

Graph Database Strengths & Weaknesses

+ Fast traversal of multi-hop relationships
+ Intuitive queries for complex patterns (e.g., similarity, recommendations)
+ Flexible schema, easy to evolve
+ Relationships are stored natively → no expensive JOINs

- Global aggregations are slow and memory-heavy
- Horizontal scaling and sharding are complex
- High write throughput (frequent inserts/updates) can be problematic
- Higher storage cost (each relationship stored as an object)

Relational Database Strengths & Weaknesses

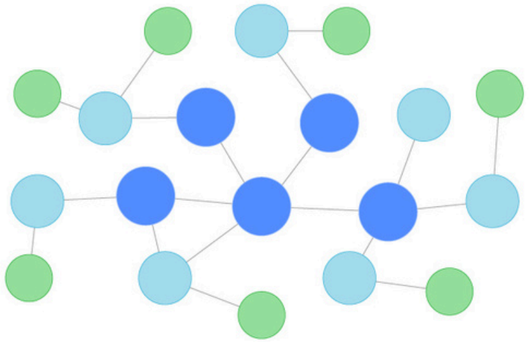
+ Excellent for aggregations, analytics, and reporting on large datasets
+ Strong ACID guarantees and transaction reliability
+ Mature ecosystem with advanced query optimization
+ Vertical scalability and replication well supported

- Queries with many JOINs become slow
- Modeling indirect or deep relationships is complex
- Schema changes can be costly
- Not suitable for graph-style queries (complex relationship patterns)

Conclusions

Neo4j (Graph DB)

- Optimized for path-based and relationship-focused queries (similarity, recommendations, multi-hop analysis).
- Ideal for real-time exploration of complex networks and relationship patterns.



PostgreSQL (Relational DB)

- Excels at aggregations, analytics, and large-scale reporting with advanced query optimization
- Best suited for heavy data processing and business intelligence workloads

In modern data stacks, Graph and Relational databases are often complementary, not competitors.