Cinescope: Scalable Movie Genre Search And Analysis (DSC 202 Winter'25 Project)

Joel Polizzi
Halıcıoğlu Data Science Institute
University of California, San Diego
La Jolla, CA 92093
jppolizzi@ucsd.edu

Dongting Cai
Halıcıoğlu Data Science Institute
University of California, San Diego
La Jolla, CA 92093
docai@ucsd.edu

Xuanwen Hua
Halıcıoğlu Data Science Institute
University of California, San Diego
La Jolla, CA 92093
x2hua@ucsd.edu

1 Introduction

Scalable Genre Search and Relationship Analysis leverages scalable cloud-native infrastructure, distributed storage, SQL, Neo4J, Redis, a collaborative JupyterLab instance, and a Python-deployed Flask application. The Cinescope system is designed for analyzing the relations between genres of movies and the actors who star in the films. The datasets used in Cinescope have been comprised of TMDB's downloadable data and data scraped from the website utilizing their API. Additionally, our datasets are deployed and stored using a PostgreSQL database and the Neo4J Graph database with a horizontally scaled Redis cache, leveraging cloud-native infrastructure, CephFS distributed storage, and networking services to deliver a publicly accessible webpage, www.cinescope.nautilus-nrp.io.

In this project, we began with generating several datasets utilizing TMDB's API, then we built out our support infrastructure and software stack leveraging the Nautilus Kubernetes Cluster on the National Research Platform. Considerable efforts have been made to create a scalable and cohesive system where applications, backend databases, and services can connect to scale and support the final application. Once our software stack was in place and our database pods could communicate with our front-end application, we were able to begin structuring our SQL query and classify our relational data to return a list of the most popular movies per genre and each movies most popular cast members. In our graph database we then expand on this concept and input the SQL results and explore a graph of movies and various actor relationships for the selected genre.

Our experiments ultimately contained four different datasets, each imported into their own respective tables in SQL and Neo4J. Our dataset sizes have a broad range, from the smallest containing only 19 items, to the largest containing more than 6 million items. Through techniques learned in this course we created a final application which calls our SQL and Neo4J queries in Python and is deployed with replication as a flask application.

1.1 Motivation and Problem Formulation

TMDB allows the community to rate movies and actors to provide popularity scores and recommendations to users. TMDB uses metrics that they do not release publicly to sort and rank movies using the communities scores. Cinescope applies our own classification system to better understand methods that may be used to determine a movie recommendation, focusing on movies of specific genres. Furthermore, the project explores pipeline development and scalability for live database driven applications.

2 Infrastructure and Scalability

To support team collaboration, project scalability, and load balancing, we built our pipeline on the National Research Platform's Nautilus cluster. This cluster spans several academic networks, with our project primarily located in the western region along the CENIC networking backbone. The hardware and software-defined infrastructure allow us to define and leverage CPU and memory (RAM) resources, scaling our services across the cluster's western hardware. The Nautilus cluster also features networking capabilities between 10 Gbps and 400 Gbps, enabling high-speed pod-to-pod communication, even when pods are deployed at opposite ends of the CENIC backbone.

2.1 Micro-Service Architecture

Microservices offer a modern approach to packaging and deploying lightweight, portable applications using containers. Containers are ephemeral isolated Linux environments that ensure consistent application behavior across different systems by encapsulating both the software and its dependencies. To enhance portability, build tools are typically excluded from the final container, leaving a streamlined application for deployment on a given runtime. This reduces the need for an end user or developer to manually install dependencies.

We have utilized several prebuilt container images publicly available on Docker Hub including PostgreSQL, Redis, JupyterLab, and Ubuntu 22:04. The image used for Flask is custom built, beginning with a standard Flask image as the base image and layers on additional tools to expand the image functionality. The built image has been pushed to Dockerhub.

Microservices are typically deployed using an orchestration platform such as Kubernetes, which manages containerized applications at scale. Kubernetes orchestrates container abstractions called pods which are collections of one or more containers, associated networking services, and shareable storage resources. Multiple containers may coexist in a single pod and through networking configurations called services, pods can effortlessly communicate with one another.

2.2 Orchestration: Pods, Deployments, and StatefulSets

To orchestrate cinescope pods, we utilize two key Kubernetes deployment strategies called ***Deployments*** and ***StatefulSets***. Deployments ensure that a defined number of replicated pods are running, making them ideal for stateless applications. StatefulSets also manage a specified number of pods, however; a StatefulSet is designed for stateful applications and provide consistent pod identification. Both deployment strategies provide the ability to vertically and horizontally scale up or down the application pods and resources on a rolling basis [1]. When a pod update event is triggered for a workload increase, the scheduler will replicate pods up to the requested amount. Similarly, when a resource increase request is received the scheduler will increase the resources of a pod. When a decrease event is triggered, the scheduler will terminate pods or resources down to the desired amount.

$$desireReplicas = ceil \left\lceil currentReplicas \times \frac{currentMetricValue}{desiredMetricValue} \right\rceil$$

Figure 1: Representation of Calculating Desired Number of Replicas During a Scaling Event [1]

The configurations for pods, along with their associated Deployments, StatefulSets, Services, Ingress controllers, and Persistent Volume Claims, are all stored in a **tree-structured** key-value data format called YAML. YAML files are read top to bottom, starting from the root node, and as the parser encounters child nodes it reads them from left to right, following a hierarchical structure. This format allows for clear organization, deployment, and readability of Kubernetes objects.

```
kubectl apply -f *.yaml
```

Figure 2: Basic Kubernetes CLI Command to deploy a YAML

2.3 The Postgres Database

Pods deploy an instance of the Postgres image as a Deployment. This Deployment is assigned to run in the US-West region of the Nautilus cluster. The Deployment contains two containers, one that runs the Postgres Database and second that runs a Data Ingestion container. The Postgres container is provided resource requests and limits that allow the program to leverage between 10 and 20 CPU cores and between 16 and 32 Gigabytes of memory (RAM). The container is instantiated with port 5432 opened as a containerPort which will allow services and other pods to access the database over the port. Developers are also able to leverage port-forwarding of the pod to allow local applications, such as **DataGrip** to access the database.

kubectl port-forward postgres-864bff5d69-gm7lz 5432:5432

Figure 3: Forwarding from 127.0.0.1:5432 -> 5432

A volume is mounted to the container as well, which mounts a Persistent Volume Claim to the database data directory.

The second container in the Postgres pod is designed to ingest data from the data collection pods into the database. The Data Ingestion container also runs a Postgres image which provides the environment access to the standard Postgres CLI. Lower resources have been applied to this container; however, the same Persistent Volume Claim that is mounted in the Postgres container is mounted in the Data-Ingestion container. This allows data to seamlessly be pushed into the database from the data ingestion pod and become present in the Postgres database.

Rook CephFS (Ceph) was selected as the backend storage driver due to its distributed nature and support for the ReadWriteMany access mode. This enables multiple pods and applications to write to the file system simultaneously, making it ideal for pipelines and environments where development is integrated with production. Our Ceph partition is relatively small, with only 20 gigabytes allocated, but Ceph provides the flexibility to scale storage vertically if needed.

2.4 The Neo4J Graph Database

The Neo4J pods are launched using a Deployment containing the latest container image provided by Neo4J. The resource requests are set to scale between 10 and 20 CPU cores and between 16 and 32 gigabytes of memory (RAM). The deployment opens two ports. The 7474 port is opened to allow http access to the application. This allows the user to port forward the service and access the database interface at localhost:7474 of their browser. The 7687 port is also opened which allows for the bolt server to advertise the address for external clients to connect and use the database. Neo4J

kubectl port-forward svc/neo4j 7474:7474 7687:7687

Figure 4: port forwarding the neo4j service

stores its database information in /data therefore a persistentVolumeClaim has been created with CephFS to provide a consistent database storage backend that will persist through pod restarts. The deployment also mounts in the same shared CephFS volume that the JupyterLab and Flask pods mount in, providing the pod access to the main code base which contains csv datasets present on file system. Data is loaded from the main code base data directory. The deployment also mounts in a configMap which contains custom settings applied to the pods which relax the security defaults enabled by bolt. The configMap also contains basic information for authentication. A Service is used to expose the ports and allow for the pod to be accessed by other pods in the namespace.

2.5 Data Collection

TMDB is a daily updated movie database which contains a front-end for users to search the database and an API where developers can query for specific fields. For a user to pull specific movie information you are required to utilize the API for generating a dataset. Though TMDB provides daily updated identification datasets; we had to generate our own movies and actors/cast members datasets.
TMDB Direct Download Datasets:

- genre.json
- person_ids.json

Datasets which required API access and scraping:

- movies-master.json
- actor-movies-id-master.json

In order to pull the TMDB movies into a json file a python script was created which is designed to use the API to pull a movie and write the item to a file. Since the movie dataset has over 1 million entries, there is a high likelihood that the script could be interrupted during the pull process and fail. For this reason, when the script is initialized it will check if any entries have previously been written to the file. If the file has previously been written to, then the get_last_id function will return the last written movie ID. The value will then be used as the starting position when the script is re-ran. The script also uses an API call to get the latest added movie. Once the start and end positions are determined the script invokes the get_movies function which uses the movie ID to retrieve the movie data from TMDB. The write function, write_to_json then takes the input movie data and writes the line to a json file.

The second dataset we produced maps actors and cast members to the movies they have participated in. Since TMDB provides a dataset that links a person's ID to their name, our data pull focuses on associating movie IDs with each person's ID. This script is nearly identical to the movie pull script but makes a different API call and narrows the data that is pulled for each person id to only the movies they have participated in. The dataset contains over 5 million combinations of actors and crew members and movies which created a challenge to efficiently pull the data.

2.5.1 Parallel Processing

Considering that the actor-movie-ids dataset was too large to pull from an individual machine a StatefulSet of Ubuntu 22.04 based pods was created to pull the data in parallel. The Stateful set allowed for consistent naming of the pods which allowed for the script to be modified to identify which pod the script was deployed to. The script then assigned a range of data based on the data-pull pod name. This rapidly outperformed my local data pull scripts and allowed us to run over 5 million queries against TMDB and collect the data required for our project.

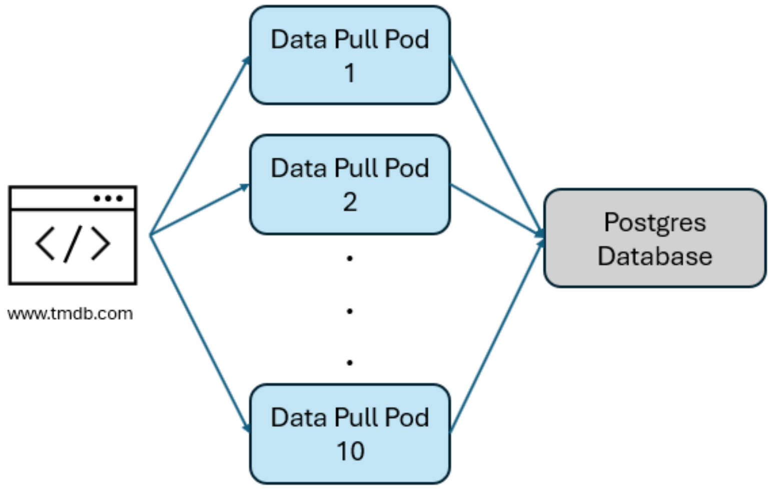


Figure 5: Data Pull Pipeline

2.5.2 Data Conversion: From JSON to CSV

When a TMDB query is made the output is in JSON format. This proved to be difficult to intake into the database and continually threw errors relating to the JSON format. To handle this a conversion script was created which converted each dataset from json to csv. As part of the conversion, we also handle pre-processing such as duplicate removal and removal of line breaks which caused import issues. Converting from JSON to CSV also provided us smaller files to work with which made upload to the database and to our Github repository easier.

2.5.3 Collaborative JupyterLab

To foster a collaborative team environment a collaborative JupyterLab instance was deployed on in the namespace that allows the Cinescope developers to simultaneously program in the same JupyterLab environment. Our instance is also deployed as a Deployment and has been configured with specific resource requests, securityContexts, and environment variables. The JupyterLab container is provided a basic entrypoint command which start jupyter when the deployment is launched. In the JupyterLab deployment we also introduce the concept of kubernetes secrets, which allow us to mount encrypted JupyterLab credentials. Similar to how our database is accessible over a containerPort, our JupyterLab instance has made port 8888 available for port forwarding. Our developers can each port-forward the pod to their independant machines and connect over a 127.0.0.1:888 url and begin to leverage the environment. A second CephFS data partition has been created to store the JupyterLab code and data, which consists of our code base available on Github. This CephFS partition is shared with the Flask application pods.

In JupyterLab we have several bash and python scripts, as well as protoyping Notebooks. Since our JupyterLab instance is deployed in the same namespace as our Postgres Database the pods are aware of one another. This allowed us to write a basic database.ini file that is used to connect to the database as if it were a 'host'. In the cinescope-db-cinnection.ipynb we were able to prototype our database connections and test queries that ultimately were included in our final flask application.

2.6 Flask Deployment and Watchdog Script

The Flask Application deployment has been scaled out to allow multiple replicas. This provides the Ingress endpoint the ability to load-balance the incoming user requests to different pods, providing horizontal scalability of the Cinescope application.

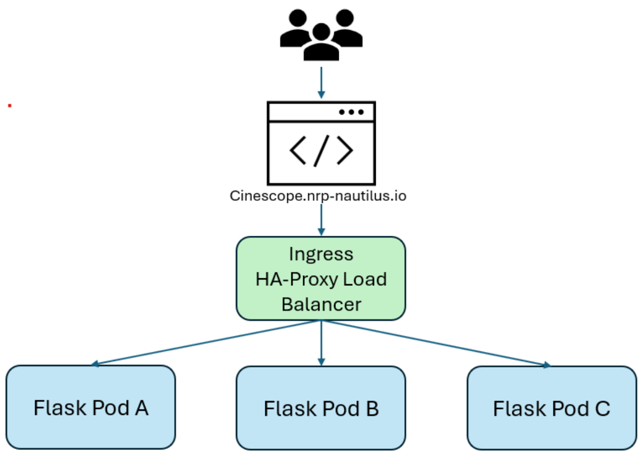


Figure 6: HA-Proxy Distribution of Load Balancing for Incoming Requests

Our flask application contains three object configurations:

1. Flask Pod Deployment
2. Flask Service

3. Flask Ingress with HA-Proxy
4. Ceph Distributed and Shared Storage (RWM)

Similar to our other software deployments, our Flask pods have defined resource requests and limits that have the ability to scale out vertically with rolling-updates in addition to the horizontal scale-out of pod replications. In order to facilitate a public webpage, our Flask pods include a Service that is configured to expose TCP port 8080 to our Ingress configuration. Ingress primarily provides Cinescope two functions: HA-Proxy load balancing and domain name servicing. As end-user interact with the webpage, HA-Proxy distributed the workload across the available Flask pods. This allows for an increase in the amount of traffic that the application can process without overwhelming any single pod. Additionally, the domain name servicing (DNS) ensures that users can access the application through a consistent URL, regardless of which pod is handling their request. This combination of load balancing and DNS improves the scalability and reliability of the application, providing a seamless experience for end users even during periods of increased traffic.

The Flask application pods also utilize a CephFS distributed storage pool which is shared across the JupyterLab application pods. As developers live code in the JupyerLab workspace the data volume is mounted across the Flask pods. A watchdog script is then implemented which checks for changes in the app.py (flask application) and re-launches the webpage with the developers updates.

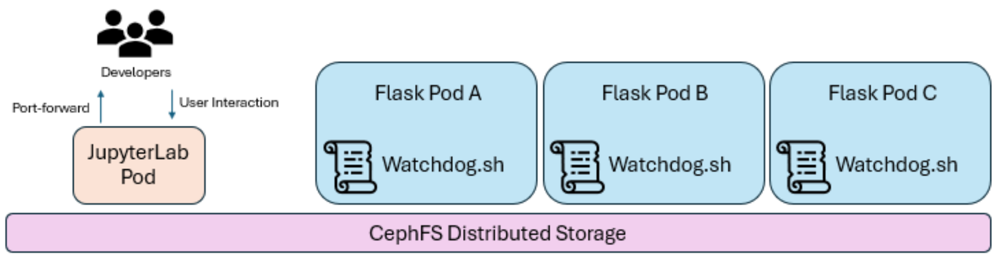


Figure 7: JupyterLab and Flask for push-to-production updates

The watchdog script primarily looks for changes in a file hash. As a the file is saved in the JupyterLab pod, the hash will be updated which then triggers the while loop to proceed and examine the hash once more. The while loop is set to check for hashing every 10 seconds ensuring that the webpage is up-to-date with the latest code implementations.

2.7 Redis Cache Database

Redis is deployed using a StatefulSet of six pods allowing the configuration to identify one of the replicated pods as a **Controller** while the remaining pods become configured as **Agents**. A CephFS storage backend of 20GB has been configured as well to share the /data volume across all the pods allowing for consistent data access. In addition, the Redis deployment contains a service which allows external access to port 6379.

Considering that our Postgres database only runs a single replication pod, Redis allows for horizontal scaling of the application and has rapidly increased the retrieval time for a cached query. As a query request is made from one of the Flask pods to the Postgres database, the application first will check if the Redis has the query cached. If the Redis controller does not have the query cached it will return a **cache miss**. In the Flask application, if a cache miss occurs, the program will proceed with the query taking place on the Postgres database. The application then directs the query result to be sent to the Redis controller and cached for future use.

When a **cache hit** occurs, the Flask application places a request to the Redis Controller, which distributes the workload to an available Agent. The Agent then returns the query results upstream where Flask can display the results.

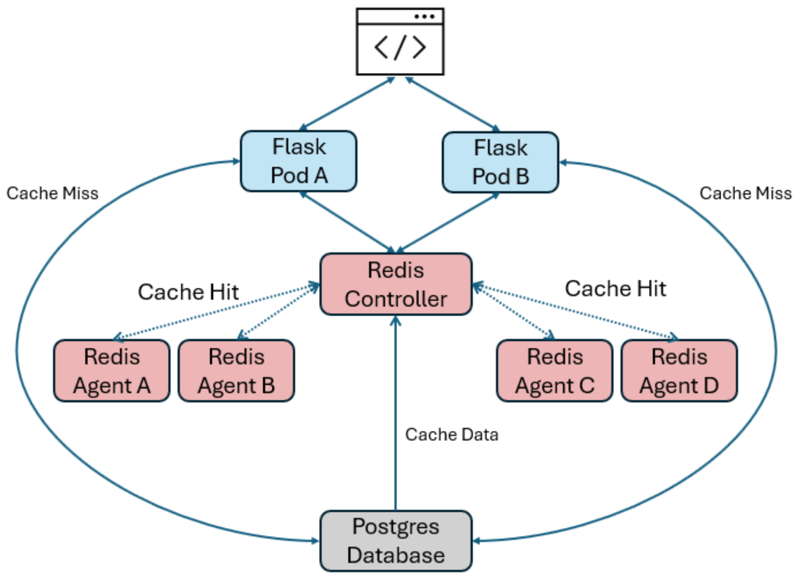


Figure 8: Redis: Caching, Hits, and Misses

2.8 Pipeline Composition, Workflow, and Software Stack

The Cinescope pipeline and infrastructure stack is a complex system designed to facilitate data pulling, pre-processing, and a developer-to-production workflow to automate the deployment of applications, updates, and scale the resources horizontally and vertically as needed. While end users will only interact with the final webpage, developers have the ability to interact with the various Databases and environments through services and port forwarding. For instance, a developer can port forward the postgres pod at port 5432 and use their local DataGrip IDE to write SQL queries, while the forwarding of ports 7474 and 7687 will allow the developer to access the Neo4J web interface at the localhost address 127.0.0.1:7474. Similarly, if a developer was to port-forward port 8080 of the JupyterLab pod then they can access the JupyterLab environment in their browser at 127.0.0.1:8080.

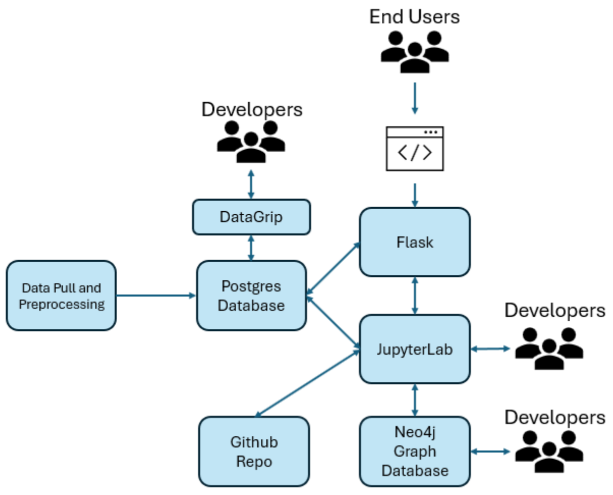


Figure 9: Developer workflow Flexibility

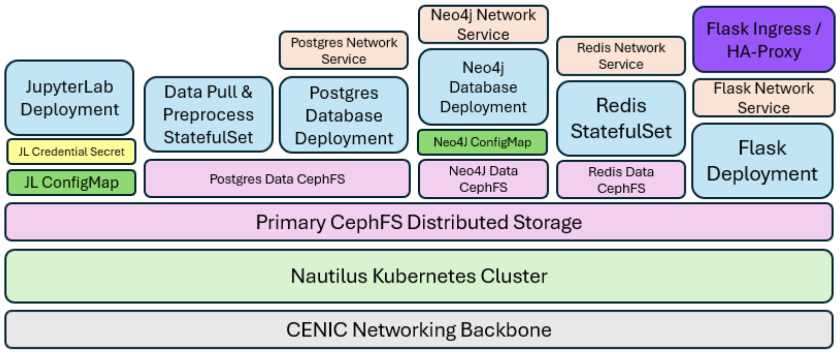


Figure 10: Infrastructure Stack

3 SQL Queries

Cinescope leverages the Postgres database and SQL query language for the exploration of relational data. The SQL queries form a basic content-based recommendation system based on a genre and language selection. This recommendation system also has elements of collaborative filtering where the top 5 movies for a selected genre are based off of a classification system which utilizes the amount of votes a movie has accumulated. A movie would be recommended for a genre if it meets the following criteria:

- The amount of users who have placed a vote is High according to our classification
- The TMDB provided popularity score is high

Looking at the total number of votes a movie has received and classifying the movies becomes important to rule out instances where a movie may have a score of 10/10 but only contain several votes.

The data has a broad range of vote totals in the movie dataset with the majority of movies containing a low number of votes. The dataset itself is comprised of 1,139,406 movie entries, many of which have had a low vote count according to TMDB. The movies range from having 0 votes to having 37,058 votes, with the average vote count being 21.6. This posed a challenge in creating a classification system to determine if a movie is considered to have a ’High’, ’Medium’, or ’Low’ amount of votes.

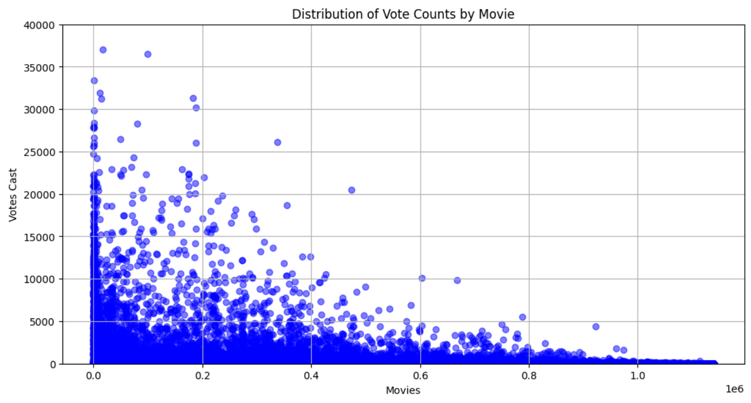


Figure 11: Vote Count Distribution

In order to address the broad disparity in votes, Cinescope applies logarithmic functions to classify movies into there appropriate category. The classification normalizes the data and makes it easier to compare movies with vastly different vote counts. Ultimately, the top five movies will be presented to the end-user.

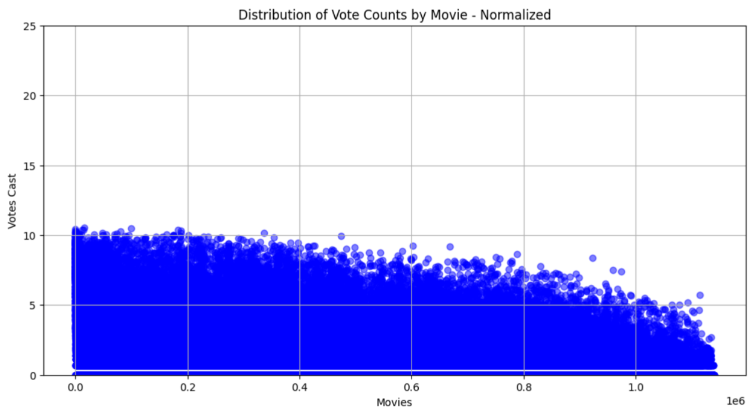


Figure 12: Vote Count Distribution Normalized

Building on the top five concept, Cinescope will also generate an array of the top five people associated with the film. To identify the most popular individuals per film, we developed a ranking system for each cast member based on their TMDB popularity score. This approach enables us to return an array of the top five people who participated in each movie.

3.1 SQL Schema

For each dataset that we collected, we established a SQL schema to efficiently manage our data. The schema is designed to improve the integrity of the data and normalize the data by reducing redundancies. Schema tables are designed with cascading deletions to maintain data consistency. Since TMDB regularly updates their database and occasionally removes entries, cascading deletions ensure that IDs across the data remain consistent when an updated dataset is uploaded to Cinescope.

Our schema totaled four different tables:

- movies: Detailed information about each movie.
- actors: List of all actors and cast members on file with TMDB. Includes An ID , name, and popularity score.
- actor_movies: An actor_id that associates with a particular movie_id. References Actors id column and movies id column.
- genre: Names of the genres available on TMDB and an associated ID number for the genre.

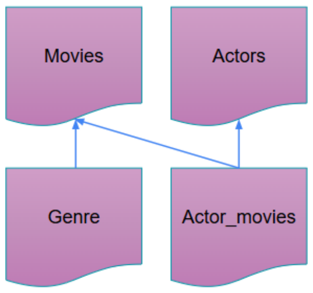


Figure 13: Schema Design

3.1.1 movies

The movies table is the most complex of the tables we introduce. It contains many different data types to comprehensively capture the attributes of each movie.

- **adult**: A *BOOLEAN* filed indicating if a movie is intended for adults only.

- **backdrop_path**: *TEXT* field to store the path to the movies backdrop image jpg.
- **belongs_to_collection**: *TEXT* field indicating if the movie is apart of a series of movies.
- **budget**: *BIGINT* integer value for how much the film cost to make.
- **genres**: The *JSONB* data type stores json data pertaining to the genres a movie is included in.
- **homepage**: A *TEXT* showing the url to the movies webpage
- **id**: Identification number set as the *SERIAL PRIMARY KEY* to uniquely identify a movie.
- **imdb_id**: *VARCHAR 15* used to store the imdb_id string containing letters and numbers.
- **origin_country**: *TEXT[]* is used to store a text array of the movies origin country.
- **original_language**: *VARCHAR 2* stores the two letter language identifier for each films original language.
- **original_title**: *TEXT* field to store the original movie title.
- **overview**: *TEXT* field to store the synopsis of the film.
- **popularity**: The *NUMERIC(10, 5)* allows for a floating point to be used for the TMDB popularity score.
- **poster_path**: *TEXT* field pointing to the jpg path on the TMDB server for the movies poster image.
- **production_companies**: *JSONB* data type used to store json data for a production companies id, name, logo_path, and origin_country.
- **production_countries**: *JSONB* used to store information about the country that the production took place in. Includes country name, ISO field, and country name abbreviation.
- **release_date**: *DATE* field indicating the data the movie was released.
- **revenue**: The amount of money the movie made stored as a *BIGINT* data type.
- **runtime**: *INTEGER* indicating the length of the movie in minutes.
- **spoken_languages**: *JSONB* containing the spoken language name, ISO number, language abbreviation, and the language spelled in English.
- **status**: A *VARCHAR 50* field indicating the movie's current status (e.g., released, in production).
- **tagline**: A *TEXT* field of the movies tagline.
- **title**: *TEXT* field for the title that the movie was relased under (as opposed to the original title).
- **video**: The video *BOOLEAN* field indicates if the movie is released on video.
- **vote_average**: A *NUMERIC(3, 1)* floating point value of a movies average viewer score. The scale ranges from 0 to 10 allowing for one decimal place.
- **vote_count**: An *INTEGER* indicating how many users have voted on the movie.

3.1.2 actors

The table is designed to store detailed information about individual actors. Each actor is assigned a unique id, which is used as the primary key for the table. The name, adult, and popularity fields capture the attributes of each actor. The table links to the actor_movies table to establish a relationship between each actor and the movies they have appeared in.

- **adult**: *BOOLEAN* value indicating if the actor is in adult films
- **id**: The actor's unique identification number which is used as the tables *INTEGER PRIMARY KEY*.
- **name**: *VARCHAR(255)* to store the actors name.
- **popularity**: *NUMERIC(10, 5)* is a floating point value indicating the actors popularity according to TMDB.

3.1.3 actor_movies

The actor_movies table creates an association between an actor id and a movie id. This table is required for referencing an actors id with the actors table to retrieve the actors name. It also references the movies table's id column to link actors with their respective movies. This connection is necessary because the movies table lists actors by their id's, rather than by their names.

- **actor_id**: An *INTEGER* value representing the actors id number.
- **movie_id**: An *INTEGER* value representing the movie id number.
- *PRIMARY KEY (actor_id, movie_id)*: Composite primary key to ensure that each combination of actor_id and movie_id is unique.
- *FOREIGN KEY (actor_id) REFERENCES actors (id) ON DELETE CASCADE*: Foreign key ensuring that if an actor is deleted from the actors table that the corresponding records in actor_movies table is also deleted.
- *FOREIGN KEY (movie_id) REFERENCES movies (id) ON DELETE CASCADE*: Foreign key ensuring that if a movie is deleted from the movies table that the corresponding records in actor_movies table is also deleted.

3.2 genre

The genre table contains two columns, the genera ID and the genre name. The Genre table links to the json data object in the Movies table, which contains a list of the genre ID's that each movie belongs to. The Genera table is used in our drop down menu system and the id that associates to a name is passed into a sub-query of our application.

- **id**: A *SERIAL PRIMARY KEY* was used to associate the ID number to a key. Serial key was used due to there being gaps in the numbers.
- **name**: *VARCHAR(50)* used to store the name of the genre string.

3.3 SQL Query

The project contains queries for two primary application functions. The first is used to populate our front-end genre drop down menu, while the second generates the relational genre content used in the project. Our SQL query contains several different CTE's which are designed to process specific tasks and send the output from one CTE to the next in order to process our final output.

3.3.1 Drop Down Menu Query

The drop down menu is created by two basic queries. The first of the queries returns the column names of the selected table. This is a done with a table selection and through python we provide the column name, schema and table as arguments to our getGenres function.

SELECT column FROM schema.table

The returned value is then passed into the getGenreId function as an argument.

The second query that is used by the drop down menu is used to get the genre id, which later is passed to the Relational Genre Content Query. This query is integrated into the getGenreId function of the app.py python script. Taking the schema, table, and above generated genre_name as arguments this function returns the genre ID that associates to a specific genre. The query selects the id from the specified schema and table where it name matches the input genre_name.

SELECT id FROM schema.table WHERE name = %s

3.3.2 Relational Genre Content Query: cinescope/sql/genre-lookup.sql

The first CTE, avg_vote takes the vote_average column for each movie in the movies table and finds the overall average of scores. Examining the dataset, we observe that the scores range from zero to ten. The CTE output is stored as a single column called avg_vote, storing this value.

The avg_votes_cast CTE takes the vote_count from the movies table and outputs two items, avg_count and max_count. The avg_count is calculated by applying the AVG function to the movies table vote_count column. This will return a single value that will be used for comparisons in the following query. The second value we create applies the MAX function to the vote_column to return the largest vote tally of any movie. The max_count value was used during the design and debugging of the query, but was omitted from being used in the final output.

Our classification system is developed in the avg_vote_score CTE. The table is initiated from the movies table and uses a cross join to incorporate both the avg_vote and avg_vote_cast CTE's. In the first CASE statement, we are doing a basic classification by comparing the vote_average for each movie to the total avg_vote that was calculated above. If a movies vote_average us greater than or equal to 1.5 times the overall average then the movie is considered to have a 'High' score. If this condition is not met then the CASE statement allows for the second condition to be evaluated. The second condition checks if the individual movies vote_average is greater than or equal to 0.5 times the calculated avg_vote total. If this condition is met, then the score is considered to be a 'Medium' score. If neither condition is met then the score will be considered 'Low'. This measurement set as the 'score' of the movie and is passed to the final output but is not used in further calculations. The second CASE statement outputs our vote_count and utilizes the natural logarithm (LN) function to classify the cote counts. The LN function allows us to normalize the data since there is a wide gap in distribution of votes. The first condition evaluates if the logarithm of the movies vote_count is greater than or equal to the logarithm of the average vote count multiplied by 10. This approximately states that if the logarithm of a particular movie multiplied by 10 is greater than or equal to 5.37 then it will be considered to have a 'High' vote count. Since LN function can not take a logarithm of a zero value, NULLIF is introduced to check if a value is zero and if true then replace it with 'null'. If this condition is not met then the second condition is checked. The second condition determines if a movie is considered to have a 'Medium' amount of votes. The condition is similar to the previous one however; the logarithm function utilizes a multiplier of 2 as opposed to 10. The logarithm of the avg_count multiplied by 2 is approximately 3.784; therefore, anything between 3.784 and 5.36 is a 'Medium' vote count. If neither condition is met, then the vote_count will be considered 'Low'.

Let vote_count = x Let avg_count = y

$$\sum_{i=1}^{n} \begin{cases} \text{'High'} & \text{if } \ln(\text{NULLIF}(\text{x}_i, 0)) \geq \ln(\text{y} \times 10) \\ \text{'Medium'} & \text{if } \ln(\text{NULLIF}(\text{x}_i, 0)) \geq \ln(\text{y} \times 2) \\ \text{'Low'} & \text{else} \end{cases}$$

The next CTE structures the ranking of a cast member. The cast members need to be ranked within an individual movie before we can populate the top five most popular actors. This table begins by inputting the actor_movies table and joining the table with the actors tables on the id's. We then create basic columns for the movie_id, the actor name, and the actor popularity. The last selection of the CTE uses a ROW_NUMBER() OVER function to apply a sequence for the ranking. In the function we use a PARTITION BY to segment the actor_movies table based on movie_id. We then apply ORDER BY to sort the actors by their popularity in descending (DESC) order. This approach partitions the movies by their ids and allows the query to rank the popularity of each actor in a movie.

The final CTE, cast_list takes the table created in cast_ranking and selects on the movie_id and creates an output array using the ARRAY_AGG function and applies an ORDER BY to order the items in the array by popularity in descending order. A FILTER function is applied to only populate the array with the top 5 most popular actors or cast members associated with the film.

The final results query begins from the movies template and joins ids on the avg_vote_score, and cast_list tables. The selections of the output will include the movies: Title, Viewer Score, Vote Average, Vote Classification, Vote Count (post normalization), Top Cast, and a movies overview as the Plot Description. The selection are gated by a WHERE EXISTS fclause that creates a sub-query to parse the JSONB_ARRAY_ELEMENTS for each films genres. The sub-query uses a WHERE clause to check if the genre id column matches the genre id provided by the getGenreId function of the app.py python script. If a match is found the the movie will be included in the results. Furthermore; the query uses AND to further narrow the query by the original_language that the film was released in. The language associations are created in a dictionary in our python code. In our flask application when a user selects a particular language the entry is passed as a variable to the query and the language selection is then applied. The GROUP BY clause is used to group the output results by specific columns. The ORDER BY clause first selects all movies with a 'High' vote count. It then orders these movies by their vote average in descending order. If there is a tie in the vote average, the movies are further ordered by their vote count, with the movie having the higher vote count appearing first. The final output is limited to the top 5 movies.

4 Neo4J Database

4.1 Data Modeling in Neo4j

The Neo4j graph schema for Cinescope uses three primary node labels and three types of relationships to model the movie domain.

4.1.1 Nodes

- `Genre` : Represents a movie genre (e.g., Action, Comedy). Each Genre node has at least an `id` and `name` property.
- `Movie` : Represents a film. Movie nodes include properties like a unique `id`, `title`, `popularity`, and `vote_average` (imported from the TMDB dataset).
- `Actor` : Represents an individual (actor). Actor nodes have an `id`, `name`, and attributes like `popularity` (popularity score) and an `adult` flag.

4.1.2 Relationships

- `ACTED_IN` : Connects an Actor to a Movie to indicate that ''`Actor X acted in Movie Y.`'' In the graph, this is a directed relationship from `Actor` to `Movie` (`Actor` $\rightarrow$ `Movie`). Every actor who appears in a movie will have an `ACTED_IN` edge to that movie node.
- `HAS_GENRE` : Connects a Movie to a Genre (`Movie` $\rightarrow$ `Genre`) to denote ''`Movie M is of Genre G.`'' Each movie typically has one or more `HAS_GENRE` edges linking it to the relevant genre nodes. This effectively tags movies with their genres in the graph.
- `COACTED_WITH` : Connects two Actor nodes who have performed together, i.e., who have co-acted in one or more common movies. This can be considered an undirected or bi-directional relationship (`Actor` $\leftrightarrow$ `Actor`). In implementation, such relationships can be derived by finding pairs of actors that share a movie.

  For example, if Actor A and Actor B both have `ACTED_IN` relationships to the same Movie node, we can create a `COACTED_WITH` relationship between A and B. In Cinescope's Neo4j database, co-acting relationships were generated for the interactive actor graph: the application finds actors who appeared in at least one (or multiple) common film(s) and links those actors with a relationship indicating they "shared" movies. (In the code, this is realized with a `SHARED` relationship carrying a property for the number of shared films, effectively serving the purpose of `COACTED_WITH`.)

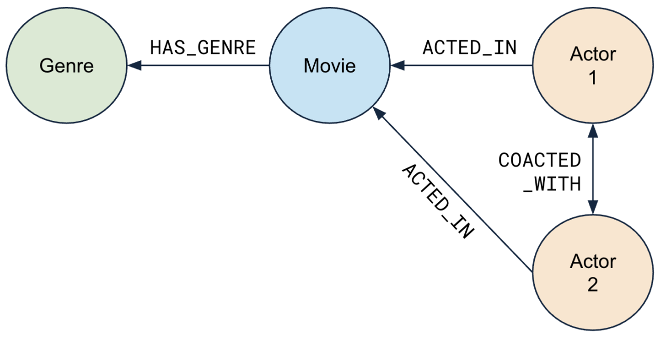


Figure 14: Example Relations in Cinescope's Neo4j Database

This data model mirrors a classic movie-actor graph: `Actor` → `Movie` → `Genre`, plus direct `Actor` ↔ `Actor` links for co-stars. It's a natural fit for Neo4j and allows queries to traverse, for example, from one actor to all their co-actors by hopping through a movie node, or from a genre to all movies and actors associated with it.

4.2 Integration with PostgreSQL and Flask

The integration is designed such that PostgreSQL handles initial data retrieval and filtering, and then Neo4j is used to organize that data into a graph structure for analysis and visualization. Flask acts as the glue, exposing API endpoints that coordinate these steps and deliver results to the front-end.

4.2.1 Data Flow

When a user selects a genre (and optional language filter) and submits a query via the web interface, the Flask backend first uses `psycopg2` (PostgreSQL driver) to fetch the top 5 movies of that genre from the SQL database (based on popularity and vote count criteria determined by the project's ranking logic). This is done by a function like:

```python
def getMoviesByGenreAndLanguage(genre_id, lang):
    # Fetches movies meeting the criteria
    return movie_list
```

Once this list of movies is obtained, the Flask route then opens a session with Neo4j and executes a Cypher query to fetch the graph elements around those movies. For example, one query takes the list of movie titles and matches the pattern of those Movie nodes, their Genre nodes, and any Actor nodes who acted in those movies. The result is a bundle of nodes and relationships (genres, movies, actors, and their connections) which represent a subgraph centered on the selected genre's top movies.

That data is then structured into a JSON response. In the code, this happens in an endpoint (e.g. `/api/graph`) that returns `graph_data` comprising lists of `nodes` and `edges`. Another endpoint (`/api/graph2`) is dedicated to the actor network: given the genre, it queries Neo4j for actors who share roles in multiple movies of that genre and returns their network of connections. The Flask session is used to pass the selected genre or its ID between requests (for example, storing the `genre_id` when the user selects a genre, so that the `/api/graph` call knows which genre's movies to retrieve).

By structuring the backend this way, the heavy lifting of popularity ranking is done in SQL, and Neo4j is then used for what it does best – relational exploration. Flask's role is to orchestrate these calls and ensure the front-end receives a unified response. Each API endpoint corresponds to a specific data need: one for the movie-actor graph, one for the actor-actor graph, and even others (like endpoints to get summary stats). For instance, there are small API endpoints that use Neo4j alone to fetch analytical summaries (like a list of genres with the most movies, or the most popular actor of each genre). All of these endpoints feed the front-end via AJAX or on-page scripts, demonstrating a clean integration of Neo4j with the Flask web framework and the existing PostgreSQL setup.

4.3 Implementation Details

Working with a large movie dataset (millions of records) presented challenges in terms of data volume and performance. We employed several strategies to ensure Neo4j could handle the load and deliver snappy query responses:

4.3.1 Data Import & Modeling

Instead of inserting all data at once, the project used batched imports. Movie and Actor data from TMDB were converted from JSON to CSV, then read in chunks (e.g. 10,000 rows at a time) and loaded into Neo4j via the Python driver. Using the `MERGE` operation for creating nodes (Movies, Genres, Actors) ensured that duplicates were not created even if an entity appeared in multiple batches.

For example, each unique Actor was merged by a consistent `id`. This approach, combined with Neo4j's ability to handle concurrent writes, allowed the team to import over 6 million data points without running out of memory in a single transaction. (The Neo4j deployment was configured with up to 32 GB RAM to accommodate the graph in memory). Creating appropriate indexes or constraints on node IDs was also important – indexing properties like `Movie.id` and `Actor.id` speeds up the `MERGE` and `MATCH` operations significantly, though these are set up behind the scenes when using Neo4j's primary key constraints.

4.3.2 Handling Large Result Sets

To keep the graph exploration efficient, Cinescope limits the scope of queries. The "Top 5 movies" concept means Neo4j is usually only fetching subgraphs of a few dozen nodes (5 movies, their genres, and their main actors) rather than the entire graph. Additionally, the Cypher queries include filters to reduce noise. For example, when collecting actors for the movie subgraph, the query filters out less relevant actors by requiring `a.popularity > 8`. This was a design choice to focus on principal cast members and avoid pulling in hundreds of bit-part actors that would clutter the graph.

Similarly, the co-actor graph only considers actors who have more than one shared movie in a genre, by using a `shared_movies > 1` condition. These thresholds act as simple yet effective performance optimizations, cutting down the number of nodes and relationships rendered, which in turn makes the front-end visualization faster and clearer.

4.3.3 Efficient Query Structuring

The Cypher queries are written to minimize unnecessary processing. Wherever possible, pattern matching is done in a single pass and results are aggregated in Neo4j before returning to Flask. For instance, instead of fetching all nodes and then filtering in Python, the query itself uses `WHERE` clauses to filter by genre or popularity, and uses `RETURN collect(...)` to bundle nodes in one go. This leverages Neo4j's ability to traverse relationships quickly in its index-free adjacency architecture.

In cases where a more complex logic was needed (like finding co-actors), the query uses a common Cypher pattern of matching a path `(a1)-[:ACTED_IN]->(m)<-[:ACTED_IN]-(a2)` and counting occurrences. The result is that Neo4j does the heavy graph computation internally (taking advantage of its optimized graph engine), and the Python code simply iterates over the returned records – a much lighter task.

4.3.4 Caching and Scaling

Beyond Neo4j itself, the project also incorporated a Redis cache and was deployed on a Kubernetes cluster for scalability. While these are infrastructure considerations, they ensured that repeated queries (e.g. requesting the same genre's graph multiple times) could be served quickly and that the Neo4j service could handle concurrent users by scaling up resources.

In Neo4j specifically, we monitored query performance and adjusted by adding constraints or refining queries as needed. Given the data size, memory management was crucial – the high-memory container for Neo4j, and possibly using Neo4j's query tuning (like terminating long-running queries or using `PROFILE/EXPLAIN` to optimize) were part of the implementation refinements.

4.4 Querying in Neo4j

The project makes use of Cypher, Neo4j's query language, to retrieve and manipulate the graph data. A variety of queries were written to support the features, ranging from simple pattern matches to more complex aggregations. Here are a few examples of how Cypher was used in Cinescope.

4.4.1 Fetching Top Movies per Genre

Given that each Movie node knows its genres (via `HAS_GENRE`) and has properties like `popularity` and `vote_average`, one can query Neo4j to find the top films of a particular genre. In Cinescope, the top-5 logic was primarily done in SQL, but we could achieve a similar result directly in Neo4j. For example, a Cypher query to get the five most popular Comedy movies might look like:

```
MATCH (m:Movie)-[:HAS_GENRE]->(g:Genre {name: "Comedy"})
WITH m ORDER BY m.vote_average DESC, m.popularity DESC
```

```
RETURN m.title AS title, m.vote_average AS score, m.popularity
LIMIT 5;
```

This query finds all movies labeled Comedy and orders them by rating and popularity, returning the top five. In the project's context, an equivalent Cypher query could use the precomputed "High/Medium/Low" vote classification if stored, or simply sort by the numeric score as shown. The result would be a list of movie titles which can then be used to pull related actors.

4.4.2 Retrieving a Movie-Actor-Genre Subgraph

Once the top movies for a genre are identified (by either SQL or the above method), Neo4j is queried for the subgraph of those movies, their genre, and actors. An actual query from the project looked like this:

```
MATCH (g:Genre)<-[:HAS_GENRE]-(m:Movie)
WHERE m.title IN $movie_list
MATCH (a:Actor)-[:ACTED_IN]->(m)
WHERE a.popularity > 8
RETURN collect(DISTINCT g) AS genres,
       collect(DISTINCT m) AS movies,
       collect(DISTINCT a) AS actors;
```

Here, `movie_list` is a parameter containing the titles of the top 5 movies (for the selected genre and language). The query finds those Movie nodes, then finds any Actor who `ACTED_IN` each of those movies, and filters actors by `popularity` $>$ `8` to get prominent actors. The result is aggregated into three collections: genres, movies, and actors.

4.4.3 Finding Co-Actors (Actor Networks)

A core use of Neo4j was to discover relationships between actors. Using the graph, it's straightforward to find actors who have worked together. A basic Cypher pattern for co-acting is:

```
MATCH (a1:Actor)-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(a2:Actor)
WHERE a1 <> a2
RETURN a1.name, a2.name, COUNT(DISTINCT m) AS movies_together;
```

This finds any two actors `a1` and `a2` who share at least one movie `m`, and counts how many movies they have in common. In Cinescope, a similar approach was used but further constrained to a specific genre and to pairs with more than one shared movie. The query for Graph 2 (Actor Co-acting Network) essentially adds a genre filter and a post-aggregation filter:

```
MATCH (a1:Actor)-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(a2:Actor),
      (m)-[:HAS_GENRE]->(g:Genre {name: $genre})
WHERE a1 <> a2
WITH a1, a2, COUNT(m) AS shared_movies
WHERE shared_movies > 1
RETURN a1.name AS Actor1, a2.name AS Actor2, shared_movies;
```

This finds actor pairs who have acted together in more than one movie of the given genre. The result might be, for example, that `‘‘Actor X and Actor Y have 3 movies together’’`.

4.4.4 Genre-based Aggregations

Beyond the interactive graphs, some analysis was done through Cypher queries to derive insights like "how many movies per genre" or "who is the most popular actor in each genre." For instance, one query in the code counted movies per genre:

```
MATCH (g:Genre)<-[:HAS_GENRE]-(m:Movie)
MATCH (a:Actor)-[:ACTED_IN]->(m)
WHERE a.popularity > 5
RETURN g.name, COUNT(DISTINCT m);
```

Another query grouped by genre to find the top actor:

```
MATCH (a:Actor)-[:ACTED_IN]->(m:Movie)-[:HAS_GENRE]->(g:Genre)
WHERE a.popularity > 5
WITH g.name AS genre, a ORDER BY a.popularity DESC
RETURN genre, collect(a.name)[0];
```

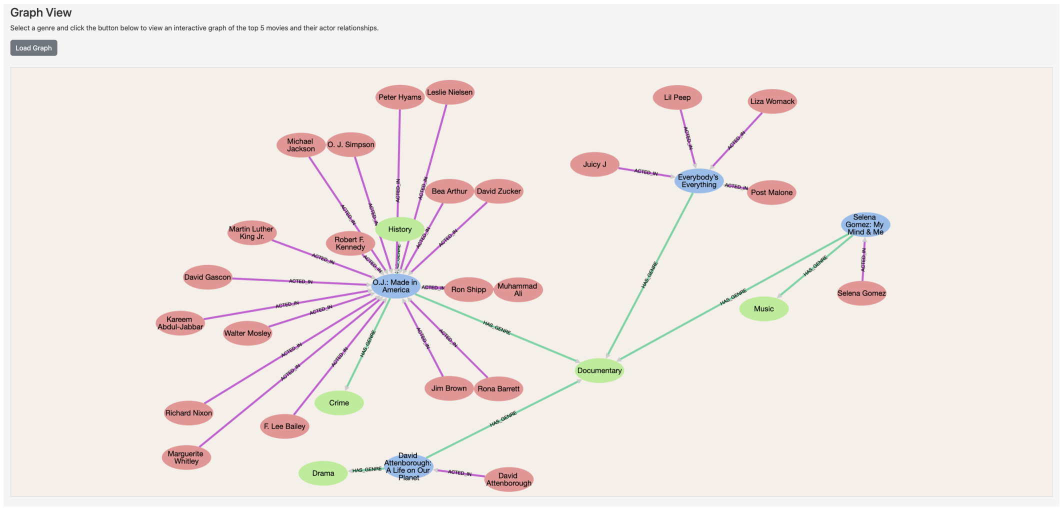
Each of these queries played a role in the project's features. Importantly, they demonstrate how expressive and concise Cypher is for graph data: exploring "who acted in what," "who worked together," and aggregating properties can all be done with pattern matching and built-in functions. By using Neo4j for these queries, the project benefited from the database's ability to navigate relationships quickly (e.g., following `ACTED_IN` links), which made these operations faster and easier to implement than equivalent operations in SQL or application code.

4.5 Visualization & User Interaction

One of the highlights of Cinescope is the interactive visualization of the movie graphs, made possible by the data in Neo4j. The front-end uses `Cytoscape.js` – a JavaScript library for graph visualization – to render two interactive graph views for the user. The integration between Neo4j's output and Cytoscape's visualization provides an intuitive way for users to explore the relationships in the data. Here's how the two graph modes are presented:

4.5.1 Graph 1: Movies and Actors of a Genre

After selecting a genre and clicking "Load Graph," the user sees a network graph of the **top 5 movies** for that genre and the actors in those movies. Genre, movie, and actor nodes are displayed, and relationships are drawn between them (movie–genre and actor–movie connections). Essentially, this graph is a visual representation of the subgraph returned by the `/api/graph` endpoint. The interface explains that this graph shows "the top 5 movies and their actor relationships."


Graph View

Select a genre and click the button below to view an interactive graph of the top 5 movies and their actor relationships.

Figure 15: Example of Graph 1 (Top Movies & Their Actors Relationship Network)

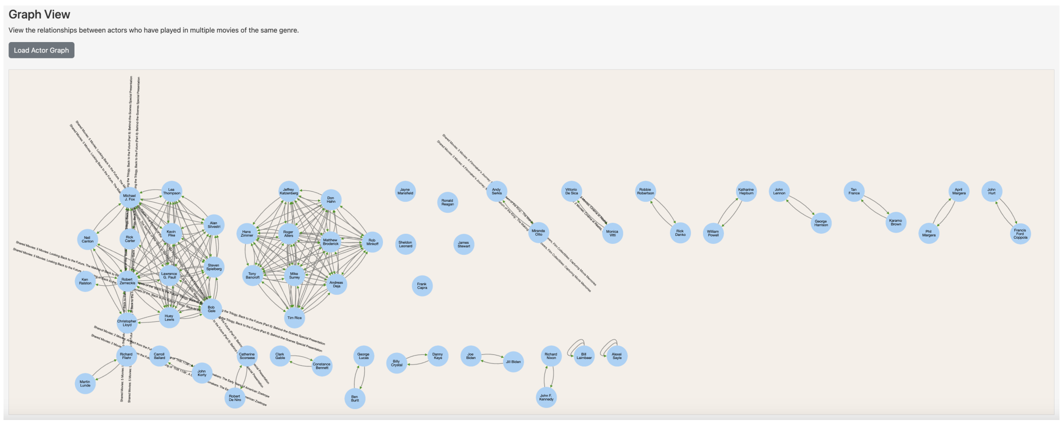
In the graph, the genre node is typically at the center (if shown) or at least connected to each movie via a `HAS_GENRE` edge. Each movie node is connected to the genre and to multiple actor nodes (via `ACTED_IN` edges from actors). This forms a hub-and-spoke pattern: genre in the middle, movies branching off, and actors connected to each movie. Users can hover over nodes to see their labels (movie titles, actor names, etc.), and can drag nodes around to rearrange the layout.

For example, if the selected genre is "Action," Graph 1 will show the top Action movies (e.g., *Mad Max: Fury Road*, *John Wick*, etc.), each linked to the "Action" node and to their main cast actors.

This allows the user to quickly see, say, which actors star in multiple of the top movies (those actors will be connected to more than one movie node, making the graph visually highlight that overlap).

4.5.2 Graph 2: Actor Co-Star Network

The second visualization focuses on actor-to-actor relationships within the selected genre. When the user clicks the "Load Actor" graph button, the app displays a network of actors who have worked together in that genre. Here, each node is an actor (`Actor`), and an edge between two actors indicates that those actors have co-starred in one or more movies of the chosen genre.


Graph View

View the relationships between actors who have played in multiple movies of the same genre.

Figure 16: Example of Graph 2 (Actor Co-acting Network)

In Cinescope, the edges were labeled with the number of shared movies (for example, an edge might read "Shared Movies: 2" if two actors did two films together). The interface describes this graph as "the relationships between actors who have played in multiple movies of the same genre." Technically, this corresponds to the `COACTED_WITH` relationships derived from Neo4j.

The graph tends to form clusters of actors: often you'll see small clusters corresponding to specific franchises or recurring collaborations. For instance, in a "Comedy" genre graph, you might see a cluster of actors who frequently appear together in ensemble comedies. Because the implementation filtered for actors with more than one shared movie, the graph emphasizes strong working relationships (regular co-stars) rather than single-film pairings.

Users can explore this by clicking on an actor node to see all their connections; the graph might show, say, that Actor A is connected to Actor B and C, meaning A has done multiple movies with B and with C. This second graph is particularly useful for discovering "ensembles" or frequent collaborators in a genre, which is a perspective not obvious from a table of movies alone.

4.5.3 Technology and User Experience

The project uses `Cytoscape.js` to render these graphs on an HTML5 canvas in the browser. Cytoscape provides pan and zoom interactions, so users can zoom in for details or pan around a complex graph. The nodes are likely styled differently by type (e.g., Genre nodes in one color, Movie in another, Actor in another) to help distinguish them visually. Labels on nodes and edges are displayed to convey names and relationship info (the "Shared Movies: N" label on co-actor edges is one example of making the relationship explicit).

This interactive visualization greatly enhances user experience by turning raw data into a navigable network. Instead of reading lists of names, users can literally see the movie's cast and how those casts overlap between movies. The two graphs complement each other and together provide both a content view and a people-centric view of the genre's landscape.

Finally, the surrounding interface (built with HTML/CSS/Bootstrap as mentioned in the project) ties it all together – there are dropdowns to select genre and language, a table listing the top 5 movies (with details like score and vote count), and then these graph sections below. Users begin by seeing the top movies in a table and can then choose to visualize them.

The result is an interactive exploratory tool: a user can pick a genre, perhaps notice in the table that certain actors recur, and then use the graphs to visually confirm that those actors indeed co-star frequently. The graph visualizations are a direct product of the Neo4j data and are what make the analysis results accessible to any user without requiring database knowledge.

5 Front End Development

In this project, the application is built by Python, Flask, and HTML templates, focusing on a user friendly interface and clean code organization. Sections below describe in details about how we use the Python code, Flask routes, and HTML templates together to create a smooth and interactive html website.

5.1 Python Code and Database Interaction

The core application logic is written in Python. The code uses the `psycopg2` package to connect PostgreSQL database. And other implementation are made:

- **retrieving genres:**
  - The `getgenres` function connects to the database and fetches the list of genre names from a specified schema and table.
  - It DYNAMICALLY builds the SQL codes to extract the desired column and returns a list of genres for later usage.
- **mapping genre names to IDs:**
  - The `getGenreId` function takes a genre name as input and retrieves its corresponding identifier from the database.
  - It uses parameterized SQL query to ensure safe and efficient access.
- **fetching language options:**
  - The `getLanguages` function query the database to obtain language codes from the movies table.
  - Then it maps each code to a descriptive name using a predefined language mapping dictionary.
- **filtering movies by genre and language:**
  - The `getMoviesByGenreAndLanguage` function combines the genre ID and selected language to query the database.
  - It reads an SQL script from an external file which is stored in our jupyter notebook environment, replaces placeholders with the appropriate values, and executes the query to get a list of movies.

These functions separate concerns and enhance the maintainability of the code by modularizing database interactions.

5.2 Flask Web Framework

Flask serves as the web framework which ties together the Python back-end and the front-end templates. The main route (`'/'`) handles both GET and POST functions:

- **GET request:**
  - When a user first accesses the application, the route initializes by retrieving all available genres and languages using the helper functions.
  - The initial page load provides the user with drop down menus for both genre and language. There's a search box under the language selection drop down menus, in case if users can not find the language that they are looking for.
- **POST request:**
  - When user click the SUBMIT button, the selected genre and language are captured from the HTML form.

  - The application then will get the genre ID corresponding to the selected genre and queries the database for movies matching both the genre and the language.
  - Finally, the page is re-rendered with the selected options and the top five movies displayed in a structured table.

Flask's use of the Jinja2 templating engine allows the server to put dynamic content directly into the HTML. This means that the movie data, genres, and languages are all integrated into the page without needing client-side rendering frameworks.

5.3 HTML Template, CSS, and JS Integration

The front end is built with HTML, CSS, and JS, augmented by popular packages and frameworks for a responsive interaction:

- **HTML structure:**
  - The template is structured with semantic HTML5 elements to ensure accessibility and readability.
  - Drop down menus are used for genre and language selection, with options populated dynamically via Jinja2.
- **CSS style with bootstrap:**
  - Boottrap is imported to ensure that the layout is responsive and consistent in different devices.
  - Custom styles are added to improve the visual representation, such as table formatting and hover effects on headers.
- **JS enhancement:**
  - The Select2 library is integrated for the language dropdown, allowing users to search through the language list easily.
  - Bootstrap's JS components provide additional interactivity, such as responsive modals and tooltip.

The design is both pleasing and functional. When the user submits the form, the page updates to reflect their choices, displaying key movie details such as title, the classification, viewer scores, vote counts, overview, and graph data to investigate actor relationships for movies within genres.

5.4 Overall Integration

The integration of Python back end logic, Flask routing, and front-end templating creates a dynamic application. Its modular design makes it easy to add extra filters or data points like release dates or user ratings with only minor changes on both the back end and the front end. Future updates could include more interactive features, such as AJAX calls that update the movie list without reloading the entire page. Although the current design focuses on server rendered HTML, the Flask routes can easily be modified to work as RESTful APIs for modern client side frameworks or mobile applications.

6 Achievements and Reflections

The final system allowed developers to create a scalable application from development to live production. The webpage cinescope.nrp-nautilus.io is up and accessible to any visitors of the page. Scalability is broadly achieved since of the resources are deployed on the Nautilus platform. The pods and the contained applications all have the ability to vertically scale while the front-end Flask pods and the back-end Redis database caching pods have the ability to scale horizontally. The SQL query effectively creates a genre recommendation system based on community feedback. The system uses the vote count of a film and creates a classification system to provide the end user the most accurate representation of a films popularity. The classification is then used in conjunction with a movies popularity to generate a list of the top five most popular movies. The application's scalability and the final query have proven successful. Utilizing ephemeral services, application pods can now restart with near-zero downtime.

Cinescope incorporates the following technologies to accomplish the project:

- Kubernetes Cloud-Native Infrastructure
  - Pod deployment strategies (Deployments, StatefulSets)
  - Rook CephFS Persistent Volume Claims
  - Networking Services
  - Secrets (Authentication)
  - ConfigMaps (Pod configuration)
  - Ingress (HA-Proxy and DNS)
  - Docker (Container build)
- Data Processing
  - Data pull and pre-processing pods
- Databases
  - Postgres
  - Neo4J
  - Redis
- Frameworks
  - Flask
- Developer Tools
  - DataGrip
  - Neo4J web interface
  - JupyterLab (Collaborative mode)
- Programming Languages
  - SQL
  - Cypher
  - Python
  - Bash
  - YAML
  - HTML/CSS/JS
- Data Formats
  - csv
  - json

Building a scalable system which integrates pre-processing, development, and production elements came with its challenges and there are necessary changes that would need to be made in the future if the project were to be continued:

- We are horizontally scaling our Redis pods to increase performance; However, our Postgres deployment only runs a single replication pod. Postgres would need to be reconfigured or a different distributor would need to be used to integrate high availability and replication.
- Neo4J cluster could have been explored for horizontal scaling fo Neo4J.
- The project currently contains too much redundancy in CephFS volumes. Reconfiguring the system to use a single CephFS volume would reduce complexity of the overall system and stack.
- Data ingestion would be further developed to leverage the data-sync/data-pull StatefulSet for nightly updates. One pod would download updates from the latest movie and actor id's available. A second pod would handle ingesting the updated data into the Postgres Database, while a thrid pod would push updates into the Neo4J Databse.

While reflecting on the the SQL query, there is a potential flaw in a movies popular actors:

- The top 5 most popular actors does not necessarily mean that the actors are starring in the production. In certain cases a cameo actor may have a higher actor rating and be the most popular actor to appear in the cast list. Further data would be required to allow the system to understand who is a top-listed cast member and who is not.

7 Acknowledgment

We would like to thank Professor Amarnath Gupta and our teaching assistant Zixi Chen for their support in the course. We also thank Nautilus Chief Architect John Graham, and P.I Tom DeFanti for permitting us to use the Nautilus platform for the project.

References

[1] T. K. Authors and T. L. Foundation. Horizontal pod autoscaling | kubernetes, 2025. Distributed under CC BY 4.0. The Linux Foundation has registered trademarks and uses trademarks.