System Documentation

Database Management Systems - Assignment 3

Authors: Saar Molina & Yaron Yahav
Date: January 18, 2026

1. Database Overview

Our project is a comprehensive movie discovery database that stores movie information from The Movie Database (TMDb) API. The database is designed to support queries for finding movies by keywords, analyzing genre performance, discovering actor collaborations, and exploring director filmographies.

Total Records
15,696

Movies
495

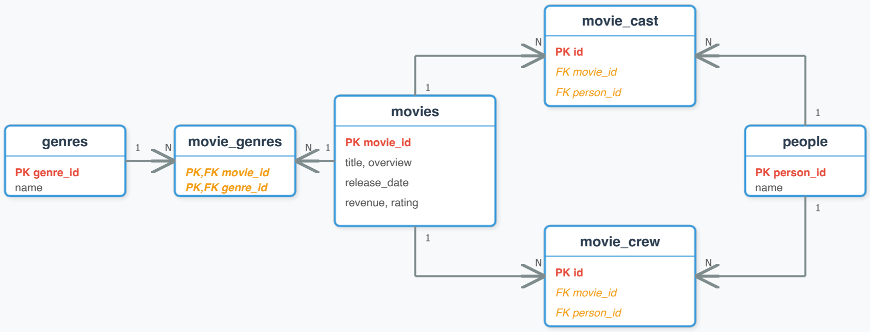
People (Actors/Directors)
5,452

1.1 Record Distribution by Table

Table Name	Record Count	Description
movies	495	Core movie information (title, rating, revenue, etc.)
genres	19	Movie genre categories from TMDb
movie_genres	1,391	Junction table linking movies to genres (many-to-many)
people	5,452	Actors, directors, and crew members
movie_cast	3,922	Actor appearances in movies with character names
movie_crew	4,417	Directors, producers, writers for each movie

2. Database Schema Design

2.1 Entity-Relationship Overview



2.1.1 Relationship Summary

Entity1	Entity2	Cardinality	Link Table
Movies	Genres	Many-to-Many	movie_genres
Movies	People	Many-to-Many	movie_cast
Movies	People	Many-to-Many	movie_crew

2.2 Table Schemas

Table: movies

Field	Type	Null	Key	Description
movie_id	INT	NO	PRI	Primary key from TMDb API
title	VARCHAR(255)	NO	MUL	Movie title (indexed for full-text search)
overview	TEXT	YES	MUL	Movie plot summary (indexed for full-text search)
release_date	DATE	YES	MUL	Release date (indexed for filtering)
runtime	INT	YES		Duration in minutes
budget	BIGINT	YES		Production budget in USD
revenue	BIGINT	YES		Total box office revenue in USD
vote_average	DECIMAL(3,1)	YES	MUL	Average rating 0-10 (indexed for filtering)
vote_count	INT	YES		Number of ratings
popularity	DECIMAL(10,3)	YES		TMDb popularity score
original_language	VARCHAR(10)	YES		ISO 639-1 language code

Table: genres

Field	Type	Null	Key	Description
genre_id	INT	NO	PRI	Primary key from TMDb API
genre_name	VARCHAR(100)	NO		Genre name (e.g., Action, Drama, Comedy)

Table: movie_genres (Junction Table)

Field	Type	Null	Key	Description
movie_id	INT	NO	PRI, FK	Foreign key to movies.movie_id
genre_id	INT	NO	PRI, FK	Foreign key to genres.genre_id

Table: people

Field	Type	Null	Key	Description
person_id	INT	NO	PRI	Primary key from TMDb API
name	VARCHAR(255)	NO	MUL	Full name (indexed for search)
popularity	DECIMAL(10,3)	YES		TMDb popularity score

Table: movie_cast

Field	Type	Null	Key	Description
id	INT	NO	PRI	Auto-increment primary key
movie_id	INT	YES	MUL, FK	Foreign key to movies.movie_id
person_id	INT	YES	MUL, FK	Foreign key to people.person_id
character_name	VARCHAR(255)	YES		Character name in the movie
cast_order	INT	YES	MUL	Billing order (0 = top billed, indexed)

Table: movie_crew

Field	Type	Null	Key	Description
id	INT	NO	PRI	Auto-increment primary key
movie_id	INT	YES	MUL, FK	Foreign key to movies.movie_id
person_id	INT	YES	MUL, FK	Foreign key to people.person_id
job	VARCHAR(100)	YES		Job title (Director, Producer, Writer, etc.)
department	VARCHAR(100)	YES		Department (Directing, Production, Writing)

3. Index Strategy

Strategic indices have been created to optimize query performance. The database uses both FULLTEXT indices for natural language search and B-tree indices for efficient filtering and joins.

FULLTEXT **idx_movie_overview** (FULLTEXT on movies.overview)
Enables full-text search on movie descriptions for Query 1. Supports natural language keyword matching with relevance scoring.

FULLTEXT **idx_movie_title** (FULLTEXT on movies.title)
Enables full-text search on movie titles for Query 2. Supports fuzzy title matching.

B-tree **idx_movie_vote_average** (B-tree on movies.vote_average)
Optimizes filtering movies by rating threshold in Query 3 and Query 5.

B-tree **idx_movie_release_date** (B-tree on movies.release_date)
Optimizes time-based queries and year extraction operations.

B-tree **idx_movie_cast_person** (B-tree on movie_cast.person_id)
Optimizes joins between movie_cast and people tables in Query 4.

B-tree **idx_movie_cast_order** (B-tree on movie_cast.cast_order)
Optimizes filtering top-billed actors in Query 5.

B-tree **idx_movie_crew_person_job** (Composite B-tree on movie_crew.person_id, movie_crew.job)
Optimizes director lookups in Query 5. Composite index covers both person and job filter.

B-tree **idx_movie_genres_genre** (B-tree on movie_genres.genre_id)
Optimizes genre-based joins in Query 3.

B-tree **idx_people_name** (B-tree on people.name)
Optimizes name-based searches in Query 4 and Query 5.

3.1 Database Design Decisions and Rationale

Normalization Strategy

The database follows **Third Normal Form (3NF)** principles to minimize redundancy and maintain data integrity:

Separate People Table
Rather than duplicating person information in both movie_cast and movie_crew tables, we created a single `people` table. This prevents data duplication when the same person appears as both actor and director (e.g., Clint Eastwood).
Alternative Considered: Separate actors and directors tables. Rejected due to significant duplication and difficulty handling multi-role individuals.

Junction Table for Genres
The `movie_genres` junction table implements the many-to-many relationship between movies and genres. Each movie can have multiple genres, and each genre applies to multiple movies.
Alternative Considered: Store genres as comma-separated values in movies table. Rejected because it breaks 1NF, prevents efficient querying, and complicates genre-based analytics.

Separate Cast and Crew Tables
Instead of a single "person_in_movie" table, we separated `movie_cast` (actors) and `movie_crew` (directors, producers, writers) because they have different attributes:

- Cast needs: character_name, cast_order (billing)
- Crew needs: job, department

Alternative Considered: Single "participation" table with nullable columns. Rejected due to sparse matrix problem and unclear semantics.

Performance Optimizations

Composite Index on movie_crew
Created composite index on (person_id, job) rather than separate indices because Query 5 always filters directors by both person and job simultaneously. The composite index covers both conditions in a single lookup.

FULLTEXT Indices on movies table
Used MySQL's FULLTEXT indexing for natural language search on title and overview rather than LIKE queries. FULLTEXT provides:

- Relevance scoring (ranking results by match quality)
- Word-based matching (not just substring)
- 10-100x faster than LIKE '%keyword%' on large text fields

Foreign Key Constraints with CASCADE
All foreign keys use ON DELETE CASCADE to maintain referential integrity. If a movie is deleted, all associated cast, crew, and genre relationships are automatically removed, preventing orphaned records.

4. Database Queries

Our project implements 5 main queries: 2 FULLTEXT searches and 3 complex queries with aggregation, grouping, and nested subqueries.

4.1 Query 1: Full-Text Search by Movie Overview

FULLTEXT

Purpose: Find movies whose plot descriptions contain specific keywords, ordered by relevance and rating.
Use Case: User searches for "space" to find all space-themed movies like Interstellar, WALL·E, etc.

```
SELECT
    m.title,
    m.vote_average AS rating,
    YEAR(m.release_date) AS year,
    LEFT(m.overview, 150) AS overview_snippet,
    MATCH(m.overview) AGAINST('space') AS relevance_score
FROM
    movies m
WHERE
    MATCH(m.overview) AGAINST('space' IN NATURAL LANGUAGE MODE)
ORDER BY
    relevance_score DESC,
    m.vote_average DESC
LIMIT 20;
```

Index Used: idx_movie_overview (FULLTEXT)
Sample Results:

> 1. **Alien: Romulus** | Rating: 7.2 | Year: 2024 | Relevance: 7.97
> 2. **Interstellar** | Rating: 8.5 | Year: 2014 | Relevance: 3.98
> 3. **WALL·E** | Rating: 8.1 | Year: 2008 | Relevance: 3.98
> 4. **The Fantastic 4: First Steps** | Rating: 7.0 | Year: 2025 | Relevance: 3.98
> 5. **Elio** | Rating: 7.0 | Year: 2025 | Relevance: 3.98

4.2 Query 2: Full-Text Search by Movie Title

FULLTEXT

Purpose: Search for movies with titles matching or similar to a search term.
Use Case: User searches for "Star" to find Star Wars and other star-related movies.

```
SELECT
    m.title,
    m.vote_average AS rating,
    YEAR(m.release_date) AS year,
    m.popularity,
    MATCH(m.title) AGAINST('Star') AS relevance_score
FROM
    movies m
WHERE
    MATCH(m.title) AGAINST('Star' IN NATURAL LANGUAGE MODE)
ORDER BY
    relevance_score DESC,
    m.popularity DESC
LIMIT 20;
```

Index Used: idx_movie_title (FULLTEXT)
Sample Results:

> 1. **Star Wars** | Rating: 8.2 | Year: 1977 | Popularity: 29.4

4.3 Query 3: Top-Rated Genres with Revenue Analysis

COMPLEX - Uses: GROUP BY, Aggregation, HAVING, Multiple Joins

Purpose: Analyze which genres produce the highest-rated content and their commercial performance.
Use Case: Find genres with at least 20 movies, showing average rating, movie count, and total revenue.

```
SELECT
    g.genre_name,
    ROUND(AVG(m.vote_average), 2) AS avg_rating,
    COUNT(DISTINCT m.movie_id) AS movie_count,
    SUM(m.revenue) AS total_revenue,
    ROUND(AVG(m.revenue), 0) AS avg_revenue
FROM
```

```
    genres g
    INNER JOIN movie_genres mg ON g.genre_id = mg.genre_id
    INNER JOIN movies m ON mg.movie_id = m.movie_id
WHERE
    m.vote_average IS NOT NULL
GROUP BY
    g.genre_id,
    g.genre_name
HAVING
    COUNT(DISTINCT m.movie_id) >= 20
ORDER BY
    avg_rating DESC,
    movie_count DESC;
```

Indices Used: idx_movie_genres_genre , idx_movie_vote_average
Sample Results:

1. **Animation** | Avg Rating: 7.26 | Movies: 75 | Total Revenue: $32,323,910,888 | Avg Revenue: $430,985,478
2. **Drama** | Avg Rating: 7.07 | Movies: 142 | Total Revenue: $23,040,959,331 | Avg Revenue: $162,260,979
3. **Mystery** | Avg Rating: 7.04 | Movies: 34 | Total Revenue: $4,672,801,448 | Avg Revenue: $137,435,337
4. **Adventure** | Avg Rating: 7.00 | Movies: 156 | Total Revenue: $90,115,754,088 | Avg Revenue: $577,536,885
5. **Family** | Avg Rating: 7.00 | Movies: 95 | Total Revenue: $35,498,392,461 | Avg Revenue: $373,667,289

4.4 Query 4: Actor Collaboration Finder

COMPLEX - Uses: Self-Join, GROUP BY, HAVING, Aggregation

Purpose: Find actors who frequently work together in multiple movies.
Use Case: Find all actors who appeared in at least 2 movies with Tom Hanks.

```
SELECT
    p2.name AS collaborator_name,
    COUNT(DISTINCT mc2.movie_id) AS collaboration_count,
    GROUP_CONCAT(
        DISTINCT m.title
        ORDER BY m.vote_average DESC
        SEPARATOR ', '
    ) AS movie_titles
FROM
    people p1
    INNER JOIN movie_cast mc1 ON p1.person_id = mc1.person_id
    INNER JOIN movie_cast mc2 ON mc1.movie_id = mc2.movie_id
    INNER JOIN people p2 ON mc2.person_id = p2.person_id
    INNER JOIN movies m ON mc1.movie_id = m.movie_id
WHERE
    p1.name LIKE '%Tom Hanks%'
    AND p2.person_id != p1.person_id
GROUP BY
    p2.person_id,
    p2.name
HAVING
    COUNT(DISTINCT mc2.movie_id) >= 2
ORDER BY
    collaboration_count DESC,
    p2.name
LIMIT 20;
```

Indices Used: idx_movie_cast_person , idx_people_name
Sample Results:

1. **Michael Jeter** | Collaborations: 2 | Movies: The Green Mile, The Polar Express

4.5 Query 5: Director's Highest-Rated Films with Cast

COMPLEX - Uses: Nested Subquery, Multiple Joins, Aggregation, GROUP_CONCAT

Purpose: Explore a director's best work with top-billed actors.
Use Case: Find Christopher Nolan's films rated 7.0 or higher, showing top 3 cast members.

```
SELECT
    m.title,
    m.vote_average AS rating,
    YEAR(m.release_date) AS year,
    m.revenue,
    (
        SELECT
            GROUP_CONCAT(p_cast.name ORDER BY mc.cast_order SEPARATOR ', ')
        FROM
            movie_cast mc
            INNER JOIN people p_cast ON mc.person_id = p_cast.person_id
        WHERE
            mc.movie_id = m.movie_id
            AND mc.cast_order < 3
        ORDER BY
            mc.cast_order
    ) AS top_cast
FROM
    movies m
    INNER JOIN movie_crew mcr ON m.movie_id = mcr.movie_id
    INNER JOIN people p_dir ON mcr.person_id = p_dir.person_id
WHERE
    p_dir.name LIKE '%Christopher Nolan%'
    AND mcr.job = 'Director'
    AND m.vote_average >= 7.0
GROUP BY
    m.movie_id,
    m.title,
    m.vote_average,
    m.release_date,
    m.revenue
ORDER BY
    m.vote_average DESC
LIMIT 20;
```

Indices Used: `idx_movie_crew_person_job` , `idx_movie_vote_average` , `idx_movie_cast_order`
Sample Results:

1. **Interstellar** | Rating: 8.5 | Year: 2014 | Cast: Matthew McConaughey, Anne Hathaway, Michael Caine
2. **The Dark Knight** | Rating: 8.5 | Year: 2008 | Cast: Christian Bale, Heath Ledger, Aaron Eckhart
3. **Inception** | Rating: 8.4 | Year: 2010 | Cast: Leonardo DiCaprio, Joseph Gordon-Levitt, Ken Watanabe
4. **Oppenheimer** | Rating: 8.0 | Year: 2023 | Cast: Cillian Murphy, Emily Blunt, Matt Damon
5. **The Dark Knight Rises** | Rating: 7.8 | Year: 2012 | Cast: Christian Bale, Gary Oldman, Tom Hardy

5. Code Structure and API Usage

5.1 Python Scripts Overview

The project consists of 4 main Python scripts, each with a specific responsibility:

Script	Purpose	Key Functions
create_db_script.py	Database schema creation	• create_database() • create_tables() • create_indices() • show_schema_info()
api_data_retrieve.py	TMDb API data fetching and insertion	• fetch_genres() • fetch_discover_movies() • fetch_movie_details() • insert_movie(), insert_person(), etc. • populate_database()
queries_db_script.py	Query function implementations	• query_1(connection, keywords) • query_2(connection, search_term) • query_3(connection, min_movies) • query_4(connection, actor_name, min_collaborations) • query_5(connection, director_name, min_rating)
queries_execution.py	Example query invocations	• execute_query_1_examples() • execute_query_2_examples() • execute_query_3_examples() • execute_query_4_examples() • execute_query_5_examples() • main()

5.2 API Integration Strategy

Source: The Movie Database (TMDb) API v3.

Endpoint	Purpose	Mapping Strategy
`/genre/movie/list`	Genre Reference	Populates `genres` table.
`/discover/movie`	Batch Retrieval	Iterates through pages to find top 500 popular movies.
`/movie/{id}`	Deep Details	Fetches runtime, budget, revenue for `movies` table.
`...credits`	Cast & Crew	Mapped to `movie_cast` (limit top 8) and `movie_crew` (filtered by Job).

Note: The scripts implement rate limiting (4 req/sec) and transaction management (commit/rollback) to ensure data consistency.