Project plan

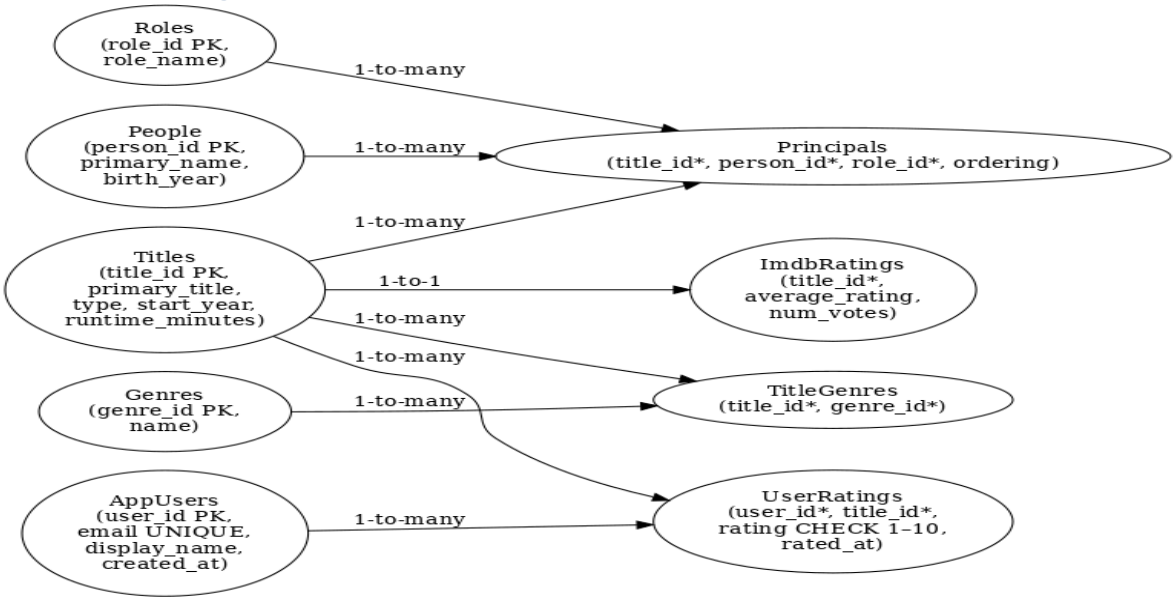
Project Plan Report: IMDb Recommendations App

Dataset Choice https://datasets.imdbws.com/

We will use the IMDb Open Datasets small subset:
- title.basics.tsv.gz (titles + year + type)
- title.ratings.tsv.gz (IMDb average rating + votes)
- title.principals.tsv.gz (who worked on a title + category)
- name.basics.tsv.gz (people + primary professions)

with optional title.akas.tsv.gz slice.

Draft ER Diagram



Preliminary Requirements / User Stories

1. Search & Browse titles by keyword, year, type, and genre.
2. View detailed title pages with cast, crew, and ratings.
3. Rate titles (1–10) and edit ratings.
4. Get recommendations:
   - Cold-start: trending/popular titles.
   - Warm-start: similarity via co-ratings.
5. Admin can upsert titles/people with integrity checks.
6. Analysts can run trend and distribution reports.

Intended UI Features

- Search/Browse Screen with filters and sortable results.
- Title Detail Screen with ratings, cast, and recommendations.
- Personalized Recommendations Screen with explanations

10 non trivial queries

```sql
USE imdb_app;

-- ============================================================
-- Q1 - Search & browse titles
--      Demonstrates: view, filtering, ORDER BY, aggregation
-- ============================================================

SELECT
    tvo.title_id,
    tvo.primary_title,
    tvo.start_year,
    tvo.title_type,
    tvo.genres,
    tvo.avg_user_rating,
    tvo.user_rating_count
FROM v_title_overview AS tvo
WHERE
    LOWER(tvo.primary_title) LIKE '%space%'      -- keyword search
    AND (tvo.start_year BETWEEN 2018 AND 2023)   -- year range filter
    AND (tvo.title_type = 'movie')               -- type filter
    AND (tvo.genres LIKE '%Sci-Fi%')             -- genre filter
ORDER BY
    tvo.avg_user_rating DESC,
    tvo.user_rating_count DESC;


-- ============================================================
-- Q2 - Title detail with cast/crew and ratings
--      Demonstrates: join of 4+ tables, LEFT JOIN
-- ============================================================

SELECT
    t.title_id,
    t.primary_title,
    t.start_year,
    t.title_type,
    g.genre_name,
    p.primary_name AS person_name,
    tpr.role_type,
    t.avg_rating,
    t.num_votes
```

```sql
FROM title t
LEFT JOIN title_genre tg        ON tg.title_id = t.title_id
LEFT JOIN genre_lookup g        ON g.genre_id = tg.genre_id
LEFT JOIN title_person_role tpr ON tpr.title_id = t.title_id
LEFT JOIN person p              ON p.person_id = tpr.person_id
WHERE t.title_id = 1
ORDER BY
    tpr.role_type,
    p.primary_name;


-- ============================================================
-- Q3 - Top genres by average rating
--      Demonstrates: aggregation, GROUP BY, HAVING
-- ============================================================

SELECT
    g.genre_name,
    AVG(ur.rating_value) AS avg_rating,
    COUNT(*)             AS num_ratings
FROM genre_lookup g
JOIN title_genre tg ON tg.genre_id = g.genre_id
JOIN user_rating ur ON ur.title_id = tg.title_id
GROUP BY
    g.genre_name
HAVING
    COUNT(*) >= 2              -- only genres with at least 2 ratings
ORDER BY
    avg_rating DESC;


-- ============================================================
-- Q4 - Users who rated a given title above a threshold
--      Demonstrates: join, filtering, ORDER BY
-- ============================================================

SELECT
    u.user_id,
    u.username,
    ur.rating_value,
    ur.rated_at
FROM app_user u
JOIN user_rating ur ON ur.user_id = u.user_id
```

```sql
WHERE ur.title_id = 1
  AND ur.rating_value >= 8
ORDER BY
    ur.rating_value DESC,
    ur.rated_at DESC;


-- ============================================================
-- Q5 - Titles with or without ratings
--      Demonstrates: LEFT OUTER JOIN, aggregation
-- ============================================================

SELECT
    t.title_id,
    t.primary_title,
    t.start_year,
    COALESCE(t.avg_rating, 0) AS avg_rating,
    COALESCE(t.num_votes, 0)  AS num_votes
FROM title t
LEFT JOIN user_rating ur ON ur.title_id = t.title_id
GROUP BY
    t.title_id,
    t.primary_title,
    t.start_year,
    t.avg_rating,
    t.num_votes
ORDER BY
    t.num_votes DESC;


-- ============================================================
-- Q6 - Titles better than global average
--      Demonstrates: scalar subquery, filtering, ORDER BY
-- ============================================================

SELECT
    t.title_id,
    t.primary_title,
    t.avg_rating,
    t.num_votes
FROM title t
WHERE
    t.avg_rating IS NOT NULL
```

```sql
    AND t.avg_rating >= (
        SELECT AVG(avg_rating)
        FROM title
        WHERE avg_rating IS NOT NULL
    )
ORDER BY
    t.avg_rating DESC,
    t.num_votes DESC;


-- ============================================================
-- Q7 - User rating history with window function (ROW_NUMBER)
--      Demonstrates: view usage, window function, PARTITION BY
-- ============================================================

SELECT
    v.user_id,
    v.username,
    v.title_id,
    v.primary_title,
    v.rating_value,
    v.rated_at,
    ROW_NUMBER() OVER (
        PARTITION BY v.user_id
        ORDER BY v.rated_at DESC
    ) AS rating_rank_recent
FROM v_user_rating_history v
WHERE v.user_id = 1;


-- ============================================================
-- Q8 - Rating percentile within each title
--      Demonstrates: window function (PERCENT_RANK)
-- ============================================================

SELECT
    ur.user_rating_id,
    ur.user_id,
    ur.title_id,
    ur.rating_value,
    PERCENT_RANK() OVER (
        PARTITION BY ur.title_id
        ORDER BY ur.rating_value
```

```sql
    ) AS rating_percentile
FROM user_rating ur
WHERE ur.title_id = 1;


-- ============================================================
-- Q9 - Use stored procedure for recommendations
--      Demonstrates: stored procedure, business logic
-- ============================================================

CALL get_recommendations_for_user(1, 5);


-- ============================================================
-- Q10 - Titles where the same person is both actor and director
--       Demonstrates: self-joins on junction table, complex join
-- ============================================================

SELECT
    t.title_id,
    t.p
```

Data init

```sql
-- 1) Create DB and select it
CREATE DATABASE IF NOT EXISTS imdb_app
  DEFAULT CHARACTER SET utf8mb4
  DEFAULT COLLATE utf8mb4_unicode_ci;

USE imdb_app;

-- 2) Tables

-- USERS
CREATE TABLE app_user (
    user_id      INT AUTO_INCREMENT PRIMARY KEY,
    username     VARCHAR(50) NOT NULL,
    email        VARCHAR(255) NOT NULL,
    created_at   TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    is_admin     TINYINT(1) NOT NULL DEFAULT 0,
    CONSTRAINT uq_app_user_username UNIQUE (username),
    CONSTRAINT uq_app_user_email UNIQUE (email),
    CONSTRAINT chk_username_not_empty CHECK (username <> '')
) ENGINE=InnoDB;

-- TITLES
CREATE TABLE title (
    title_id         INT AUTO_INCREMENT PRIMARY KEY,
    imdb_tconst      VARCHAR(12),
    primary_title    TEXT NOT NULL,
    start_year       INT,
    title_type       VARCHAR(20) NOT NULL,  -- movie, tvSeries, etc
    runtime_minutes  INT,
    is_adult         TINYINT(1) NOT NULL DEFAULT 0,
    avg_rating       DECIMAL(3,1),
    num_votes        INT NOT NULL DEFAULT 0,
    CONSTRAINT uq_title_imdb_tconst UNIQUE (imdb_tconst),
    CONSTRAINT chk_start_year CHECK (start_year IS NULL OR start_year >=
1870),
    CONSTRAINT chk_runtime CHECK (runtime_minutes IS NULL OR
runtime_minutes > 0)
) ENGINE=InnoDB;

-- PEOPLE
CREATE TABLE person (
    person_id     INT AUTO_INCREMENT PRIMARY KEY,
```

```
    imdb_nconst   VARCHAR(12),
    primary_name  TEXT NOT NULL,
    birth_year    INT,
    death_year    INT,
    CONSTRAINT uq_person_imdb_nconst UNIQUE (imdb_nconst),
    CONSTRAINT chk_birth_year CHECK (birth_year IS NULL OR birth_year >=
1850),
    CONSTRAINT chk_death_year CHECK (
        death_year IS NULL OR birth_year IS NULL OR death_year >=
birth_year
    )
) ENGINE=InnoDB;

-- LOOKUP / REFERENCE TABLE (GENRES)
CREATE TABLE genre_lookup (
    genre_id    INT AUTO_INCREMENT PRIMARY KEY,
    genre_name  VARCHAR(50) NOT NULL,
    CONSTRAINT uq_genre_name UNIQUE (genre_name)
) ENGINE=InnoDB;

-- JUNCTION: titles <-> genres  (many-to-many)
CREATE TABLE title_genre (
    title_id  INT NOT NULL,
    genre_id  INT NOT NULL,
    PRIMARY KEY (title_id, genre_id),
    CONSTRAINT fk_title_genre_title
        FOREIGN KEY (title_id) REFERENCES title(title_id)
        ON DELETE CASCADE,
    CONSTRAINT fk_title_genre_genre
        FOREIGN KEY (genre_id) REFERENCES genre_lookup(genre_id)
        ON DELETE RESTRICT
) ENGINE=InnoDB;

-- JUNCTION: titles <-> people + role
CREATE TABLE title_person_role (
    title_id    INT NOT NULL,
    person_id   INT NOT NULL,
    role_type   VARCHAR(30) NOT NULL, -- actor, director, writer, etc
    characters  TEXT,
    PRIMARY KEY (title_id, person_id, role_type),
    CONSTRAINT fk_tpr_title
        FOREIGN KEY (title_id) REFERENCES title(title_id)
        ON DELETE CASCADE,
```

```
    CONSTRAINT fk_tpr_person
        FOREIGN KEY (person_id) REFERENCES person(person_id)
        ON DELETE CASCADE
) ENGINE=InnoDB;

-- USER RATING TABLE (junction: users <-> titles)
CREATE TABLE user_rating (
    user_rating_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id        INT NOT NULL,
    title_id       INT NOT NULL,
    rating_value   INT NOT NULL,
    rated_at       TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    review_text    TEXT,
    CONSTRAINT uq_user_title UNIQUE (user_id, title_id),
    CONSTRAINT fk_rating_user
        FOREIGN KEY (user_id) REFERENCES app_user(user_id)
        ON DELETE CASCADE,
    CONSTRAINT fk_rating_title
        FOREIGN KEY (title_id) REFERENCES title(title_id)
        ON DELETE CASCADE,
    CONSTRAINT chk_rating_value CHECK (rating_value BETWEEN 1 AND 10)
) ENGINE=InnoDB;
```

2 views

Title Overview

```
CREATE OR REPLACE VIEW v_title_overview AS
SELECT
    t.title_id,
    t.primary_title,
    t.title_type,
    t.start_year,
    t.is_adult,
    GROUP_CONCAT(DISTINCT g.genre_name ORDER BY g.genre_name SEPARATOR ',
') AS genres,
    COALESCE(AVG(ur.rating_value), 0) AS avg_user_rating,
    COUNT(ur.user_rating_id) AS user_rating_count
FROM title t
LEFT JOIN title_genre tg ON tg.title_id = t.title_id
LEFT JOIN genre_lookup g ON g.genre_id = tg.genre_id
LEFT JOIN user_rating ur ON ur.title_id = t.title_id
GROUP BY
    t.title_id,
    t.primary_title,
    t.title_type,
    t.start_year,
    t.is_adult;
```

User rating history view

```
CREATE OR REPLACE VIEW v_user_rating_history AS
SELECT
    u.user_id,
    u.username,
    t.title_id,
    t.primary_title,
    ur.rating_value,
    ur.rated_at
FROM app_user u
JOIN user_rating ur ON ur.user_id = u.user_id
JOIN title t        ON t.title_id = ur.title_id;
```

Transaction/Concurrency

Transaction to roll back the score if new score is invalid

```
DELIMITER //

CREATE PROCEDURE demo_rating_transaction()
BEGIN
    DECLARE v_max_rating INT;

    START TRANSACTION;

    INSERT INTO user_rating (user_id, title_id, rating_value)
    VALUES (1, 3, 11);

    SELECT MAX(rating_value)
    INTO v_max_rating
    FROM user_rating
    WHERE user_id = 1;

    IF v_max_rating > 10 THEN
        ROLLBACK;
    ELSE
        COMMIT;
    END IF;
END //

DELIMITER ;

-- Run:
CALL demo_rating_transaction();
```

Concurrency
User A:

```
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;

SELECT * FROM user_rating WHERE title_id = 1 FOR UPDATE;
```

User B:

```
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;
```

```
UPDATE user_rating
SET rating_value = 3
WHERE title_id = 1;
```

This will BLOCK until Session A commits/rollbacks because of FOR UPDATE lock.

User A:

```
UPDATE user_rating
SET rating_value = 10
WHERE title_id = 1;

COMMIT;
```

How to init the project

1. Install Docker
2. Open VsCode
3. Create a project folder and create the below 2 files
4. docker-compose.yml

```yaml
version: '3.9'

services:
  mysql:
    image: mysql:8.0
    container_name: imdb-mysql
    restart: unless-stopped
    environment:
      MYSQL_ROOT_PASSWORD: rootpassword
      MYSQL_DATABASE: imdb_app
      MYSQL_USER: imdb_user
      MYSQL_PASSWORD: imdb_pass
    ports:
      - "3306:3306"
    volumes:
      # All .sql files in this folder run automatically on first startup
      - ./sql:/docker-entrypoint-initdb.d
      # Persistent data volume
      - imdb-mysql-data:/var/lib/mysql

volumes:
  imdb-mysql-data:
```

5. sql/01_init_imdb_app.sql

```sql
CREATE DATABASE IF NOT EXISTS imdb_app
  DEFAULT CHARACTER SET utf8mb4
  DEFAULT COLLATE utf8mb4_unicode_ci;

USE imdb_app;

-- ============================================================
-- 1. TABLES
-- ============================================================

-- USERS
CREATE TABLE app_user (
    user_id      INT AUTO_INCREMENT PRIMARY KEY,
    username     VARCHAR(50) NOT NULL,
    email        VARCHAR(255) NOT NULL,
```

```
    created_at   TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    is_admin     TINYINT(1) NOT NULL DEFAULT 0,
    CONSTRAINT uq_app_user_username UNIQUE (username),
    CONSTRAINT uq_app_user_email UNIQUE (email),
    CONSTRAINT chk_username_not_empty CHECK (username <> '')
) ENGINE=InnoDB;

-- TITLES
CREATE TABLE title (
    title_id         INT AUTO_INCREMENT PRIMARY KEY,
    imdb_tconst      VARCHAR(12),
    primary_title    TEXT NOT NULL,
    start_year       INT,
    title_type       VARCHAR(20) NOT NULL,  -- movie, tvSeries, etc
    runtime_minutes  INT,
    is_adult         TINYINT(1) NOT NULL DEFAULT 0,
    avg_rating       DECIMAL(3,1),
    num_votes        INT NOT NULL DEFAULT 0,
    CONSTRAINT uq_title_imdb_tconst UNIQUE (imdb_tconst),
    CONSTRAINT chk_start_year CHECK (start_year IS NULL OR start_year >=
1870),
    CONSTRAINT chk_runtime CHECK (runtime_minutes IS NULL OR
runtime_minutes > 0)
) ENGINE=InnoDB;

-- PEOPLE
CREATE TABLE person (
    person_id     INT AUTO_INCREMENT PRIMARY KEY,
    imdb_nconst   VARCHAR(12),
    primary_name  TEXT NOT NULL,
    birth_year    INT,
    death_year    INT,
    CONSTRAINT uq_person_imdb_nconst UNIQUE (imdb_nconst),
    CONSTRAINT chk_birth_year CHECK (birth_year IS NULL OR birth_year >=
1850),
    CONSTRAINT chk_death_year CHECK (
        death_year IS NULL OR birth_year IS NULL OR death_year >=
birth_year
    )
) ENGINE=InnoDB;

-- LOOKUP / REFERENCE TABLE (GENRES)
CREATE TABLE genre_lookup (
```

```
    genre_id    INT AUTO_INCREMENT PRIMARY KEY,
    genre_name  VARCHAR(50) NOT NULL,
    CONSTRAINT uq_genre_name UNIQUE (genre_name)
) ENGINE=InnoDB;

-- JUNCTION: titles <-> genres  (many-to-many)
CREATE TABLE title_genre (
    title_id  INT NOT NULL,
    genre_id  INT NOT NULL,
    PRIMARY KEY (title_id, genre_id),
    CONSTRAINT fk_title_genre_title
        FOREIGN KEY (title_id) REFERENCES title(title_id)
        ON DELETE CASCADE,
    CONSTRAINT fk_title_genre_genre
        FOREIGN KEY (genre_id) REFERENCES genre_lookup(genre_id)
        ON DELETE RESTRICT
) ENGINE=InnoDB;

-- JUNCTION: titles <-> people + role
CREATE TABLE title_person_role (
    title_id    INT NOT NULL,
    person_id   INT NOT NULL,
    role_type   VARCHAR(30) NOT NULL, -- actor, director, writer, etc
    characters  TEXT,
    PRIMARY KEY (title_id, person_id, role_type),
    CONSTRAINT fk_tpr_title
        FOREIGN KEY (title_id) REFERENCES title(title_id)
        ON DELETE CASCADE,
    CONSTRAINT fk_tpr_person
        FOREIGN KEY (person_id) REFERENCES person(person_id)
        ON DELETE CASCADE
) ENGINE=InnoDB;

-- USER RATING TABLE (junction: users <-> titles)
CREATE TABLE user_rating (
    user_rating_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id        INT NOT NULL,
    title_id       INT NOT NULL,
    rating_value   INT NOT NULL,
    rated_at       TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    review_text    TEXT,
    CONSTRAINT uq_user_title UNIQUE (user_id, title_id),
    CONSTRAINT fk_rating_user
```

```
        FOREIGN KEY (user_id) REFERENCES app_user(user_id)
        ON DELETE CASCADE,
    CONSTRAINT fk_rating_title
        FOREIGN KEY (title_id) REFERENCES title(title_id)
        ON DELETE CASCADE,
    CONSTRAINT chk_rating_value CHECK (rating_value BETWEEN 1 AND 10)
) ENGINE=InnoDB;

-- ============================================================
-- 2. SAMPLE DATA
-- ============================================================

INSERT INTO genre_lookup (genre_name) VALUES
('Action'), ('Drama'), ('Comedy'), ('Sci-Fi');

INSERT INTO app_user (username, email, is_admin) VALUES
('alice', 'alice@example.com', 0),
('bob',   'bob@example.com',   1),
('carol', 'carol@example.com', 0);

INSERT INTO title (imdb_tconst, primary_title, start_year, title_type,
runtime_minutes, is_adult)
VALUES
('tt0000001', 'Space Adventure', 2020, 'movie',    120, 0),
('tt0000002', 'Sad Clown',       2018, 'movie',     95, 0),
('tt0000003', 'Robot Tales',     2022, 'tvSeries',  45, 0);

INSERT INTO person (imdb_nconst, primary_name, birth_year)
VALUES
('nm0000001', 'Jane Director', 1975),
('nm0000002', 'John Actor',    1985),
('nm0000003', 'Sara Writer',   1980);

-- Simple genre mapping for demo
INSERT INTO title_genre (title_id, genre_id)
SELECT t.title_id, g.genre_id
FROM title t
JOIN genre_lookup g
  ON ( (t.imdb_tconst = 'tt0000001' AND g.genre_name = 'Sci-Fi')
    OR (t.imdb_tconst = 'tt0000002' AND g.genre_name = 'Drama')
    OR (t.imdb_tconst = 'tt0000003' AND g.genre_name = 'Sci-Fi') );

-- People roles
```

```sql
INSERT INTO title_person_role (title_id, person_id, role_type)
SELECT t.title_id, p.person_id, 'director'
FROM title t
JOIN person p ON p.primary_name = 'Jane Director';

INSERT INTO title_person_role (title_id, person_id, role_type)
SELECT t.title_id, p.person_id, 'actor'
FROM title t
JOIN person p ON p.primary_name = 'John Actor'
WHERE t.imdb_tconst IN ('tt0000001','tt0000002');

-- User ratings
INSERT INTO user_rating (user_id, title_id, rating_value) VALUES
(1, 1, 9),
(1, 2, 7),
(2, 1, 8),
(3, 3, 10);

-- ============================================================
-- 3. VIEWS
-- ============================================================

CREATE OR REPLACE VIEW v_title_overview AS
SELECT
    t.title_id,
    t.primary_title,
    t.title_type,
    t.start_year,
    t.is_adult,
    GROUP_CONCAT(DISTINCT g.genre_name ORDER BY g.genre_name SEPARATOR ',
') AS genres,
    COALESCE(AVG(ur.rating_value), 0) AS avg_user_rating,
    COUNT(ur.user_rating_id) AS user_rating_count
FROM title t
LEFT JOIN title_genre tg ON tg.title_id = t.title_id
LEFT JOIN genre_lookup g ON g.genre_id = tg.genre_id
LEFT JOIN user_rating ur ON ur.title_id = t.title_id
GROUP BY
    t.title_id,
    t.primary_title,
    t.title_type,
    t.start_year,
    t.is_adult;
```

```sql
CREATE OR REPLACE VIEW v_user_rating_history AS
SELECT
    u.user_id,
    u.username,
    t.title_id,
    t.primary_title,
    ur.rating_value,
    ur.rated_at
FROM app_user u
JOIN user_rating ur ON ur.user_id = u.user_id
JOIN title t        ON t.title_id = ur.title_id;

-- ============================================================
-- 4. STORED PROCEDURE (RECOMMENDATIONS)
-- ============================================================

DELIMITER //

CREATE PROCEDURE get_recommendations_for_user (
    IN p_user_id INT,
    IN p_limit   INT
)
BEGIN
    WITH fav_genres AS (
        SELECT DISTINCT tg.genre_id
        FROM user_rating ur
        JOIN title_genre tg ON tg.title_id = ur.title_id
        WHERE ur.user_id = p_user_id
          AND ur.rating_value >= 8
    )
    SELECT DISTINCT
        t.title_id,
        t.primary_title,
        'Matches your favorite genres' AS reason
    FROM title t
    JOIN title_genre tg ON tg.title_id = t.title_id
    WHERE tg.genre_id IN (SELECT genre_id FROM fav_genres)
      AND t.title_id NOT IN (
          SELECT title_id FROM user_rating WHERE user_id = p_user_id
      )
    ORDER BY COALESCE(t.avg_rating, 0) DESC, t.num_votes DESC
    LIMIT p_limit;
```

```
END //

DELIMITER ;

-- ============================================================
-- 5. TRIGGERS (KEEP avg_rating AND num_votes UPDATED)
-- ============================================================

DELIMITER //

CREATE TRIGGER trg_user_rating_ai
AFTER INSERT ON user_rating
FOR EACH ROW
BEGIN
    UPDATE title t
    JOIN (
        SELECT
            title_id,
            COALESCE(AVG(rating_value), 0) AS avg_rating,
            COUNT(*) AS num_votes
        FROM user_rating
        WHERE title_id = NEW.title_id
        GROUP BY title_id
    ) r ON r.title_id = t.title_id
    SET
        t.avg_rating = r.avg_rating,
        t.num_votes  = r.num_votes;
END //

CREATE TRIGGER trg_user_rating_au
AFTER UPDATE ON user_rating
FOR EACH ROW
BEGIN
    UPDATE title t
    JOIN (
        SELECT
            title_id,
            COALESCE(AVG(rating_value), 0) AS avg_rating,
            COUNT(*) AS num_votes
        FROM user_rating
        WHERE title_id = NEW.title_id
        GROUP BY title_id
    ) r ON r.title_id = t.title_id
```

```
    SET
        t.avg_rating = r.avg_rating,
        t.num_votes  = r.num_votes;
END //

CREATE TRIGGER trg_user_rating_ad
AFTER DELETE ON user_rating
FOR EACH ROW
BEGIN
    UPDATE title t
    LEFT JOIN (
        SELECT
            title_id,
            COALESCE(AVG(rating_value), 0) AS avg_rating,
            COUNT(*) AS num_votes
        FROM user_rating
        WHERE title_id = OLD.title_id
        GROUP BY title_id
    ) r ON r.title_id = t.title_id
    SET
        t.avg_rating = COALESCE(r.avg_rating, 0),
        t.num_votes  = COALESCE(r.num_votes, 0)
    WHERE t.title_id = OLD.title_id;
END //

DELIMITER ;

-- ============================================================
-- 6. INDEXES (INCLUDING COMPOSITE)
-- ============================================================

-- Composite index on (title_type, start_year)
CREATE INDEX idx_title_title_type_start_year
    ON title (title_type, start_year);

CREATE INDEX idx_user_rating_title_id
    ON user_rating (title_id);

CREATE INDEX idx_user_rating_user_id
    ON user_rating (user_id);

CREATE INDEX idx_title_genre_genre_id
    ON title_genre (genre_id);
```

Stored procedures

```sql
DELIMITER //

CREATE PROCEDURE get_recommendations_for_user (
    IN p_user_id INT,
    IN p_limit   INT
)
BEGIN
    WITH fav_genres AS (
        SELECT DISTINCT tg.genre_id
        FROM user_rating ur
        JOIN title_genre tg ON tg.title_id = ur.title_id
        WHERE ur.user_id = p_user_id
          AND ur.rating_value >= 8
    )
    SELECT DISTINCT
        t.title_id,
        t.primary_title,
        'Matches your favorite genres' AS reason
    FROM title t
    JOIN title_genre tg ON tg.title_id = t.title_id
    WHERE tg.genre_id IN (SELECT genre_id FROM fav_genres)
      AND t.title_id NOT IN (
          SELECT title_id FROM user_rating WHERE user_id = p_user_id
      )
    ORDER BY COALESCE(t.avg_rating, 0) DESC, t.num_votes DESC
    LIMIT p_limit;
END //

DELIMITER ;
```