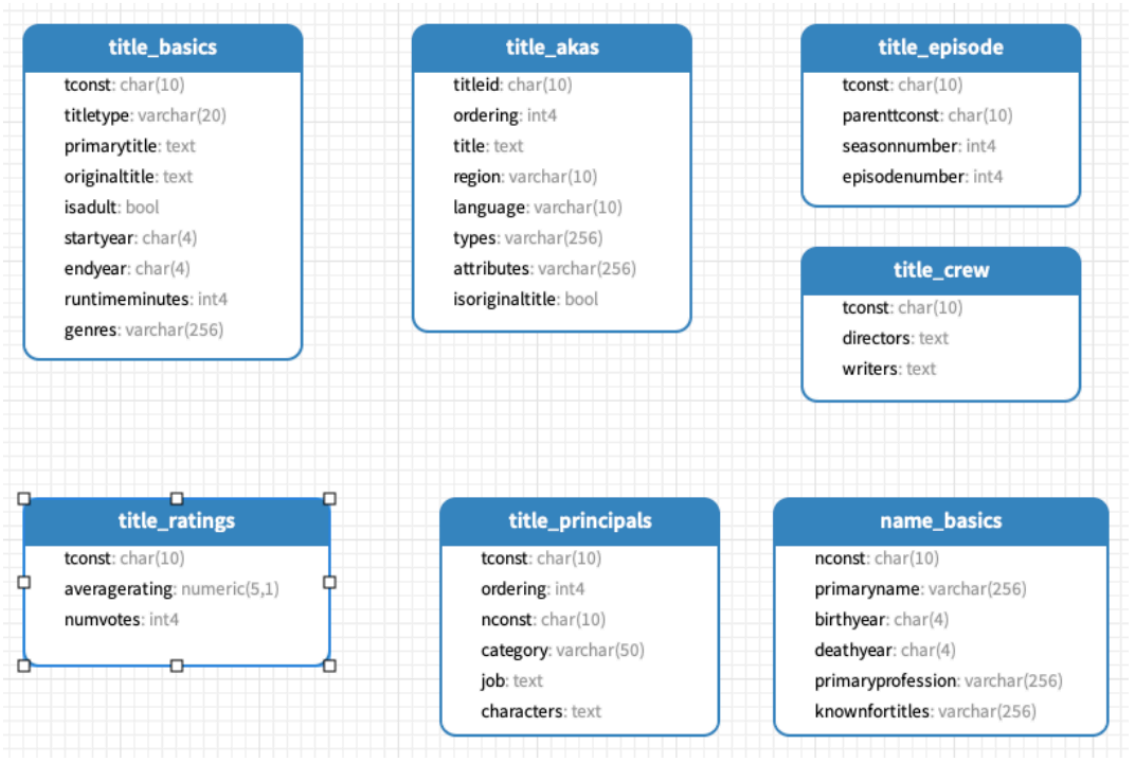
Report

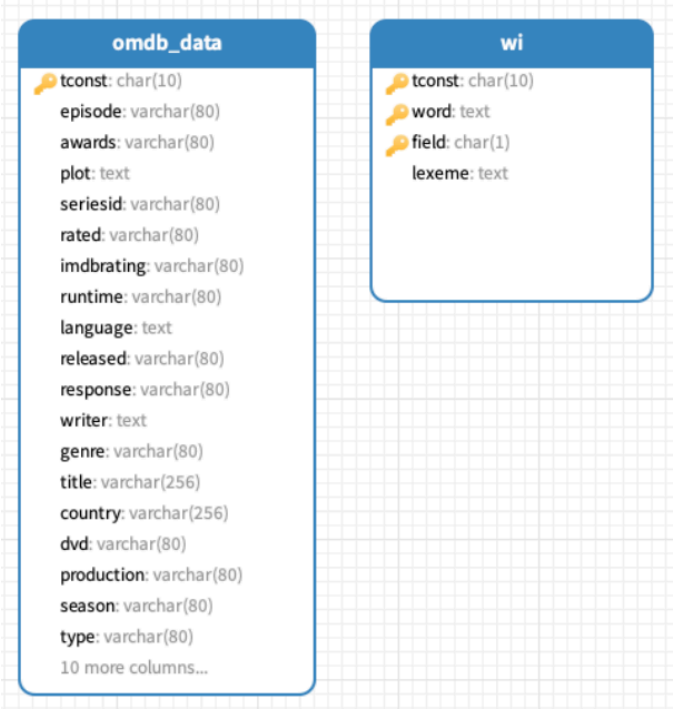
- Database Rework
  - Original Design
  - Name Changes
  - Reason for Normalisation
  - Analysing Datatype requirements
  - Key Analysis
- Design
  - Diagrams
- Implementation
  - Ideal Scenario
  - Tables
    - Profile
  - Queries
  - Functions
  - Stored Procedures
  - Views
- Data Migration
  - Steps
- Optimization
- Testing
- Deployment
  - Steps

Database Rework

Original Design



Original Design



The OMDP PART To Add

Profile (Auth) → History

We used this design due to the limitation of RUC on premise, we would normally have created a service boundary and hosted them as their own db og their own device, as user and search require different storage needs for an IMDb sized project.

1a user stories etc

Name Changes

- Better naming standard
- Clearer intend
- Better Convention

Reason for Normalisation

- Storing data in strings
- Creating Objects for reference and easy changes
- Using Enums / Objects instead of type strings?

Analysing Datatype requirements

IDs = VARCHAR(20) (tconst/nconst)

We keep IMDB IDs verbatim to satisfy frontend/ wi dependencies and avoid re-key mapping. Simpler joins and easier debugging.

Ratings = average_rating FLOAT + num_votes INT

This allows fast O(1) updates per new vote; checks 0-10 and ≥0

Key Analysis

IDs (tconst/nconst as PK, VARCHAR(20))

Kept verbatim to match frontend/wi.

Design

Purpose and scope

The app supports: (1) search in titles/plots/names, (2) rating of titles, (3) similar movie suggestions, (4) people lookups and co-players, (5) user features: bookmarks, search history, rating history.

Domain Model

Movie model: title, rating, genre, person, actor, crew, episode, also_known_as.

Framework model: profile, search_history, rating_history, bookmark

Primary keys: title.id = tconst, person.id = nconst. M:N relations via actor, crew, genre

UI Sketches

DB api?

Diagrams

- Entity Relation

- Crow Notation

Implementation

Ideal Scenario

- Service Boundaries
- Storing area of responsabilities in different databases

Tables

Profile

Queries

Functions

1-D Functionality (Implemented API)

All functions are implemented on top of our redesigned Movie Data Model and our Framework Model created and loaded by our repeatable build scripts.

B2_build_movie_db

C2_build_framework_db

1-D.1 Basic framework functionality

Goal. Support users, bookmarks, notes, and history as the foundation for later features.

Design & Implementation.

- **Users.** create_user, find_user, change_password, delete_user manage profile rows. We keep the interface minimal and deterministic (return id on create; void for updates/deletes).

- **Bookmarks.** Unified bookmark table supports *either* a title or a person per row, enforced with a CHECK constraint; functions add/remove/list both kinds.
- **Notes (extension).** Two small tables (note_title, note_person) store user-authored text; CRUD functions upsert and fetch notes with updated_at.
- **History.** Read-only accessors get_search_history(profile_id, limit) and get_rating_history(profile_id, limit) provide paginated, time-ordered results and join titles for readability.

Why this design.

The single bookmark table plus constraints avoids duplication and simplifies uniqueness rules per (user, object). Notes are an allowed framework extension.

1-D.2 Simple search: string_search(S)

Goal. Return (tconst, primarytitle) where S is a case-insensitive substring of title or plot, and record the query in the user's history.

Design & Implementation.

- Case-insensitive matching via ILIKE on title.primary_title and title.plot.
- Side-effect: insertion into search_history(profile_id, search_query) before returning results.

Outcome.

A compact function with predictable behavior and O(n) scan unless aided by optional indexes (discussed under indexing).

1-D.3 Title rating: rate(title, rating ∈ [1..10])

Goal. Update the aggregate rating while treating repeated votes by the same user consistently and logging rating history.

Design & Implementation.

- We maintain rating (title_id, average_rating, num_votes) and use an UPSERT to incorporate a new vote into a running average.

- Side-effect: rating_history(profile_id, title_id, rating) is appended for auditability and later analytics.

Consistency policy.

Our implementation treats each rate() call as a new vote; alternative “redraw” or “ignore duplicates” strategies are discussed in Section 1-E (design alternatives) but not enabled by default.

1-D.4 Structured string search: structured_string_search(title, plot, characters, person)

Goal. Case-insensitive substring search across four fields (title, plot, character names, person names), returning distinct titles and logging history.

Design & Implementation.

- Dynamic SQL builds LEFT JOIN actor a and person p only when available, keeping the query readable and efficient on our schema.
- Each non-empty parameter contributes an ILIKE predicate; results are de-duplicated and ordered.

Outcome. Flexible multi-field search that remains explainable and testable.

1-D.5 Finding names

Goal. Provide direct lookup of persons (e.g., actors) by substring.

Design & Implementation.

- find_person_by_name(search_term) returns (person_id, name) using person.primary_name ILIKE '%...%', ordered by name with an optional LIMIT.

Outcome.

Small, composable building block for UI autosuggest and for 1-D.6.

1-D.6 Finding co-players

Goal. Given an actor’s name, list the most frequent co-players with counts.

Design & Implementation.

- We resolve the target actor, collect their title_ids, then count co-occurrences in actor. The function returns (nconst, primaryname, frequency) ordered by frequency and name.

Outcome.

Simple co-occurrence analytics based on our normalized actor table; easy to wrap in a view if desired for readability.

1-D.7 Name rating

Goal. Derive a popularity score for names from the ratings of related titles, weighted by num_votes.

Design & Implementation.

- We add person_rating(person_id, weighted_rating) and compute SUM(average_rating * num_votes) / SUM(num_votes) over the actor's filmography, rounded to two decimals.
- A refresh function (calculate_actor_ratings) truncates and repopulates the table.

Outcome.

Materialized, query-friendly ratings for names used by 1-D.8.

1-D.8 Popular actors

Goal. Utilize name ratings to rank actors relevant to a context (we implement the *co-players of a given actor, ranked by popularity* option).

Design & Implementation.

- get_popular_coplayers(actor_name) finds the actor's titles, joins co-players, and orders them by person_rating.weighted_rating (NULLS LAST) then name.

Outcome.

A user-facing list that privileges widely-voted, well-rated collaborators.

1-D.9 Similar movies

Goal. Provide a principled notion of similarity and list comparable titles.

Design & Implementation.

- We implement **genre-set Jaccard similarity** between the input movie and candidates: $|G \cap|$ / $|G \cup|$, returning (sim_title_id, primary_title, jaccard_genre) limited and ordered by score.
- Choice rationale: genres are available and normalized; Jaccard is transparent, fast to compute, and easy to explain.

Outcome.

Deterministic, scalable baseline similarity suitable for UI “More like this” sections.

1-D.10 Frequent person words: person_words(person_name [, max_words])

Goal. Characterize a person by frequent words across titles they are involved in, using the provided inverted index wi.

Design & Implementation.

- Join person → actor → wi and aggregate by wi.word, filtering very short tokens.
- Return top-K words with integer frequencies.

Outcome. Lightweight descriptive tags usable in UIs (e.g., word clouds).

CITP Portfolio Project Source d...

1-D.11 Exact-match querying: exact_match_query(keywords[])

Goal. Return titles that match **all** provided keywords using wi.

Design & Implementation.

- Case-insensitive match on wi.word; GROUP BY tconst HAVING COUNT(DISTINCT lower(word)) = array_length(keywords,1).

Outcome. Deterministic boolean retrieval aligned with IR slides referenced by the assignment.

1-D.12 Best-match querying: best_match_query(keywords[])

Goal. Rank titles by the number of matched keywords (more matches → higher rank).

Design & Implementation.

- Join title with wi, count distinct matched words per title, and order by match_count DESC, primary_title.

Outcome.

Simple relevance ranking appropriate for compact result lists.

1-D.13 Word-to-words querying: word_to_words_query(keywords[])

Goal. Return a **ranked list of words** (not titles) that co-occur across all titles matching the input query.

Design & Implementation.

- Compute the set of matching titles, aggregate frequencies of all wi.word over that set, and return the top words as (word, frequency).

Outcome.

Supports query expansion and exploratory search features in the frontend.

Stored Procedures

Views

Data Migration

Steps

Optimization

1-E Improving performance by indexing
A key advantage of using a relational database is that query performance can be dynamically optimized at any stage of development and tailored to actual system usage,
particularly for the most frequent or critical queries — even after deployment. Such
performance improvements can be made by adding or adjusting indexes on database
tables, without requiring changes to other parts of the system.
1-E.1 Consider the extensions developed under Section 1-D and discuss or explain what may potentially provide significant performance improvements. Describe how your database is indexed. (Observe that this concerns database indexing, not textual inverted indexing — even though the latter may build on the
former.)

Testing

1-F Testing using the IMDB database
Demonstrate by examples that the results of Section 1-D work as intended. Write a single SQL script that activates all the written functions and procedures and, for those
that modify data, add selections to show before and after for the modifications. In this subproject you need only to proof by examples that you code is runnable. A more
elaborate approach to testing is an issue in Subproject 2. Generate an output file from
running your test script file. (See descriptions in assignment 1 and 2 on how to do this.)

Deployment

Steps