# Web Data Management Development Project

Xiaoxu Gao

Alexander Overvoorde

Tom Peeters

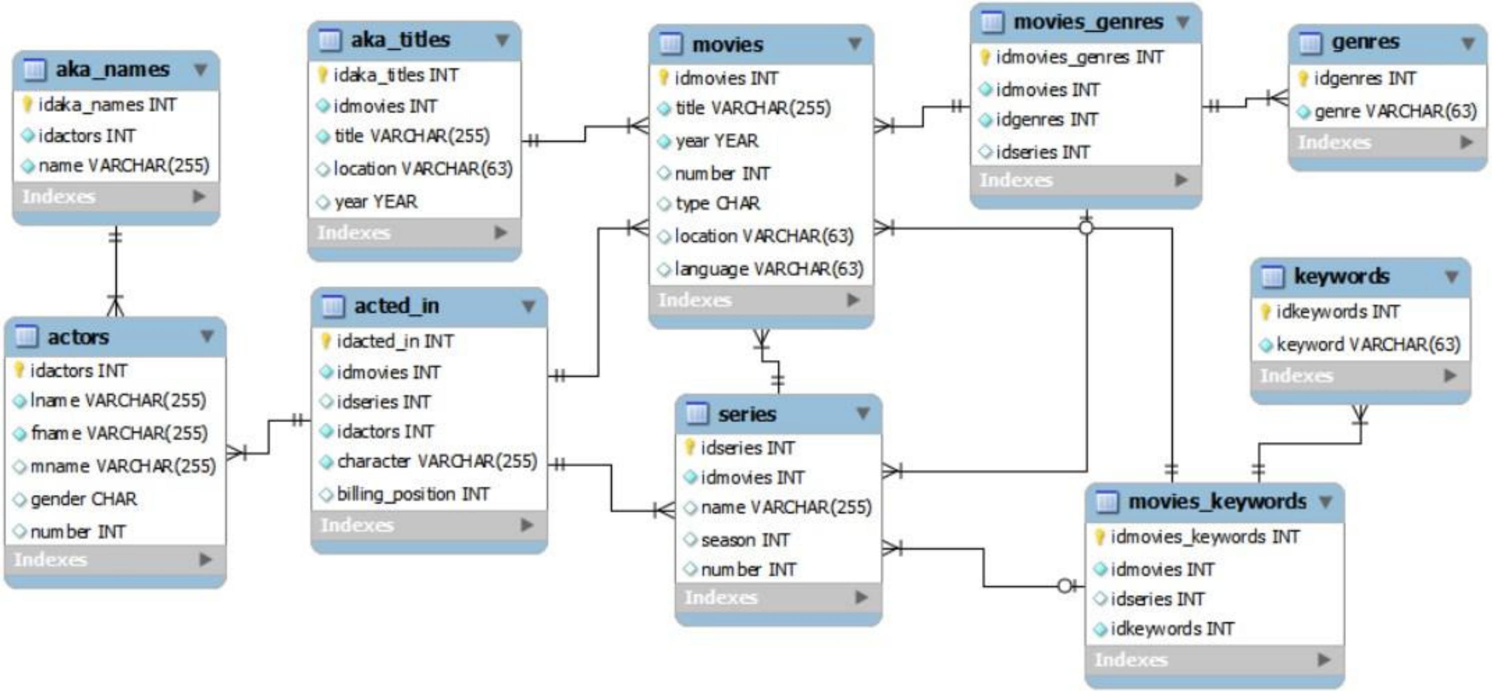
Piotr Tekieli

TU Delft

# Choice of technologies

- PostgreSQL
- MongoDB (document-based)
- Neo4j (graph-based)

# PostgreSQL - Schema



# PostgreSQL - Query design 1/5

- For each SC, we worked around a different “central point” :
- SC1 - Movies
- SC2-3 - Actors
- SC4-5 - Genres
- Further connections with a related information were obtained with JOINs.

# PostgreSQL - Query design 2/5

- Each JOIN is written as separate SQL query, therefore, to get <Full Movie Info> we need to send 5 queries to the DB.

Example : (based on movie id) *Get Actor Info + Character Played + His Billing Position*

SELECT **DISTINCT** a.idactors, a.fname, a.lname, a.gender, ai.character, ai.billing_position

FROM actors a

**JOIN** acted_in ai **ON** a.idactors = ai.idactors

**JOIN** movies m **ON** m.idmovies = ai.idmovies

WHERE m.idmovies = <someid> **ORDER BY** ai.billing_position;

TU Delft

# PostgreSQL - Query design 3/5

- We used built-in functions (i.e. COUNT) to create statistics for genres and actors.
- WHERE clause was used for specifying time frames.

<u>Example : *Recover genres and count number of titles associated with them within a specific time frame*</u>

SELECT DISTINCT g.idgenres, g.genre, **COUNT(DISTINCT m.idmovies)** AS number

FROM genres g **JOIN** movies_genres mg **ON** mg.idgenres=g.idgenres

**JOIN** movies m **ON** mg.idmovies = m.idmovies

**WHERE m.year >=** <startyear> **AND m.year <=** <endyear>

GROUP BY g.idgenres ORDER BY <sortby>

# PostgreSQL - Query design 4/5

- Restricting to movies in case of partial matches was realized by :

1. Looking for a number of results returned by querying the DB with a specific title
2. Based on that information, deciding whether to access returned ID (if only a single row was returned) or performing more detailed scan (if more results were obtained) to get all titles consisted of the entered string (i.e. by applying % operators).

In case of “The Sopranos” only a SINGLE row was returned, however the exact phrase “Star Wars” is associated with more than one position in DB

TU Delft

# PostgreSQL - Query design 5/5

Pseudocode (how it looks in practice):

```
JDBC.PerformQuery("SELECT COUNT(DISTINCT m.idmovies) AS number FROM movies m WHERE m.title ILIKE '" + title + "';");

if (JDBC.getResultSet().getInt("number") == 1)
JDBC.PerformQuery("SELECT DISTINCT m.idmovies FROM movies m WHERE m.title ILIKE '" + title + "';");

else
JDBC.PerformQuery("SELECT m.idmovies FROM movies m WHERE m.title ILIKE '%" + title + "%' AND m.idmovies NOT IN (SELECT DISTINCT m.idmovies FROM movies m JOIN series s ON m.idmovies = s.idmovies WHERE m.title ILIKE '%" + title + "%') AND m.year >= " + syear + " AND m.year <= " + eyear + " ORDER BY m." + sort + ";");
```

TU Delft

# PostgreSQL - Advantages

- Flexibility in acquiring any information from the DB (with a proper statement)
- Intuitive SQL statements (to an extent)

TU Delft

# PostgreSQL - Limitations/issues

- Impedance mismatch = usually it’s difficult to map java objects with relations and tables.
- More complex SQL statements can be tricky to create and “difficult to tune”.
- For related (processable) information multiple JOIN queries need to be submitted.

# PostgreSQL - Lessons learned

- Linking together OOP language and RD schema can sometimes be difficult and may require a large amount of testing to prepare working solution.
- SQL statements with JOINs and their output can be messy (in terms of the structure).
- For most queries with JOINs it’s good to use DISTINCT clause which usually affects its performance.

# MongoDB - Schema

- Two possible approaches:
  - Copy PostgreSQL schema and perform JOINs client side
  - Denormalization
- We chose a hybrid form
- Limit denormalization to small data like keywords and genres
- Duplicating movie and actor details too inefficient

```
{
        "_id" : ObjectId("574cb18d3233b8399595ff3d"),
        "idmovies" : 213978,
        "title" : "Matrix",
        "year" : 1993,
        "number" : "",
        "type" : 3,
        "location" : "",
        "language" : "",
        "genres" : ["action", "drama", "fantasy", "thriller"]
}
```

# MongoDB - Schema

- Base schema built by importing CSV export from PostgreSQL
- Post-processed with Python script to denormalize fields like genres and keywords based on query needs
- Indexes added to support aggregation queries for statistics operations

# MongoDB - Query design

- General steps
  - Build search query to get base results (e.g. matching movie or actor ids)
  - Build bulk data query to get the details of all of the matched objects by id (simulate INNER JOIN)
- Statistics SC3 and SC5 can be implemented natively using MongoDB aggregate queries
- Add appropriate indexes to accommodate queries afterwards

# MongoDB - Query design

- Example: find movies that a certain actor has played in and list their co-stars
- Steps:
  - Search query to find movie ids in the acted_in collection
  - Bulk query to find details in the movies collection using $in operator and list of ids from previous query
  - Bulk query to find ids of actors starring in those movies using $in query
  - Bulk query to retrieve actor details (name, gender) using all unique actor ids from previous query, again using $in
  - Combine results client side in Java
- Fixing this would require denormalizing acted_in = massive storage requirements

# MongoDB - Query design

- Client side joins through bulk queries required for nearly endpoint
- Only statistics queries (SC3 & SC5) can rely on a single query
- Required a lot of performance tuning and optimization of server -> database communication
- Queries could not be accommodated by more schema changes without significant storage increase

TU Delft

# MongoDB - Advantages

- Document model maps nicely to Java objects and arrays
- Scaling is better supported than with PostgreSQL
- May be more suited to variable amounts of data available about movies and actors

# MongoDB - Limitations/issues

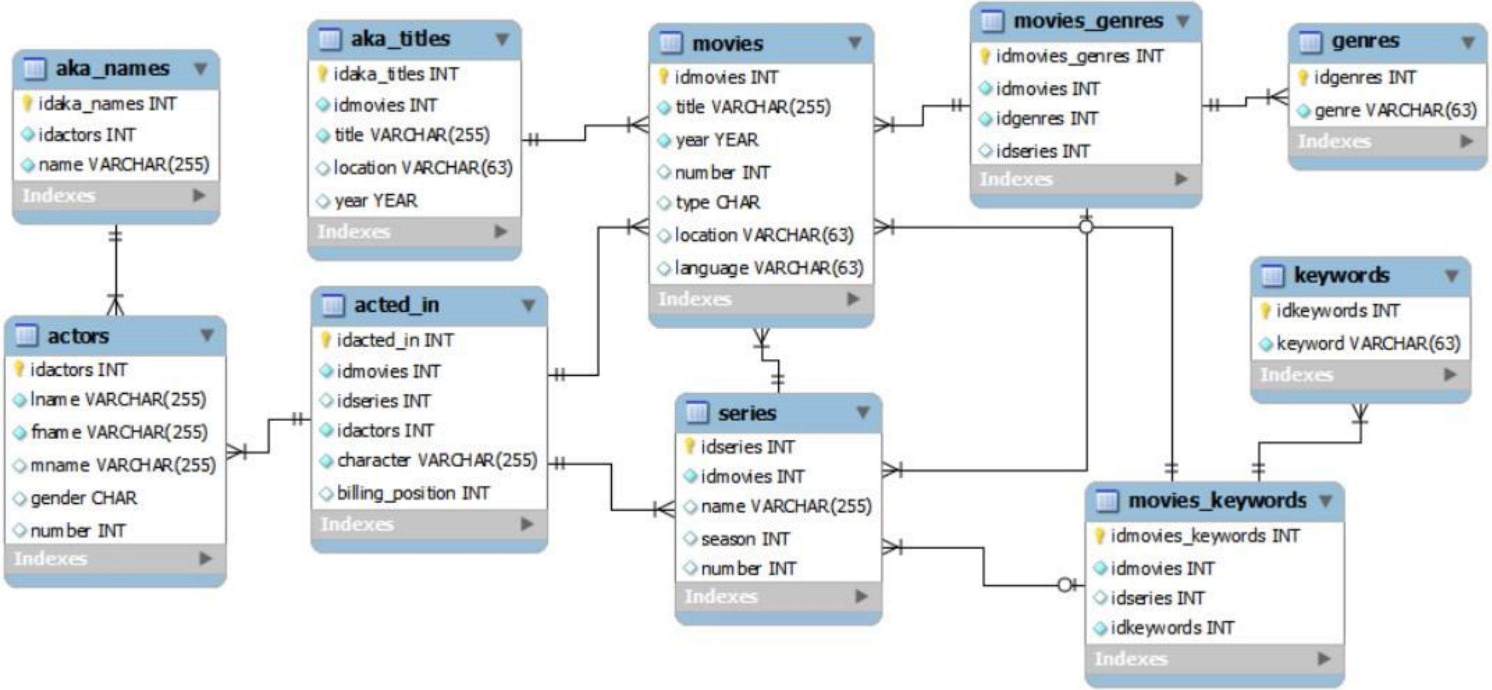
- PostgreSQL offers better support for hybrid approach through JSON columns
- Loose typing and optional fields in documents requires client side validation
- Impedance mismatch: MongoDB is optimized for heavy writes, not flexible reads as required by this application
- Relational database with replication much more fitting
- Increased storage requirements because of metadata

# MongoDB - Lessons learned

- Less powerful queries impose constraints on data format
- Requires client side code to handle data validation and joins
- Shines when data can be denormalized well, but this is not one of those applications

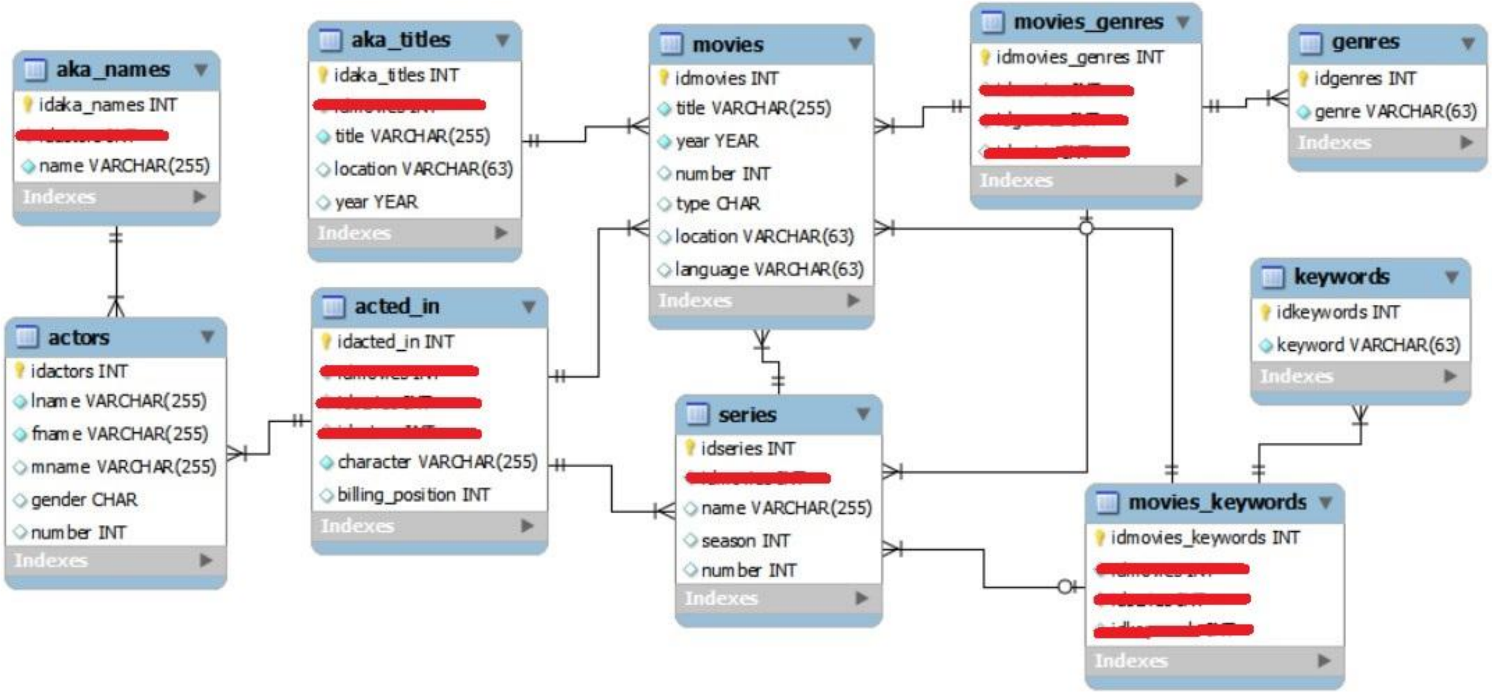
TU Delft

# Neo4j - Schema

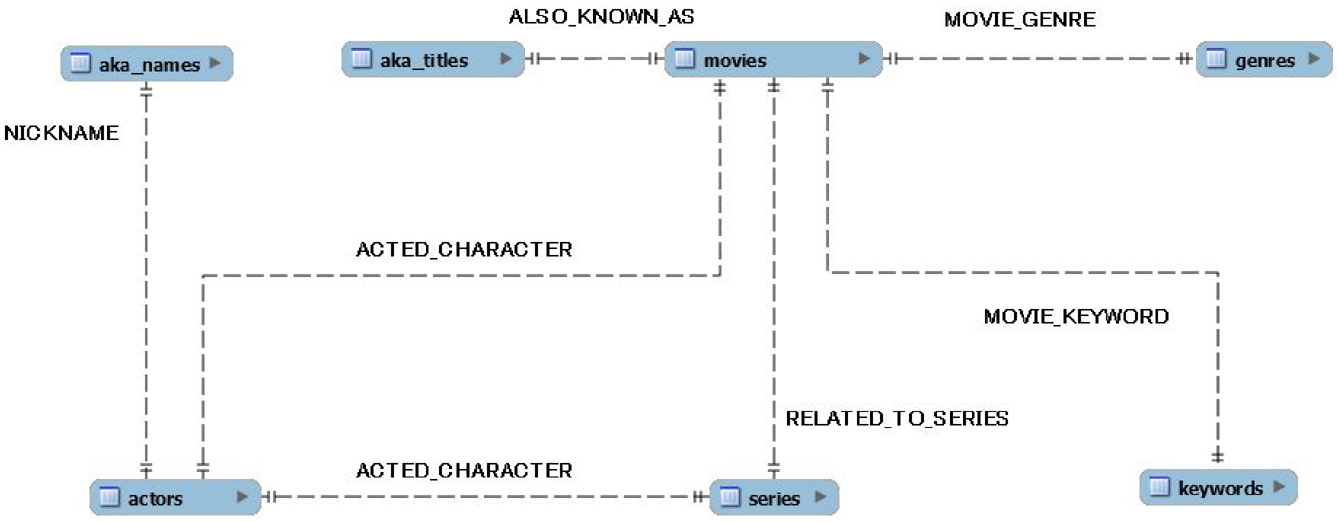



TU Delft

# Neo4j - Schema



# Neo4j - Schema



# Neo4j - Database import

## Step 1 - Extracting PostgreSQL data

```
COPY acted_in TO 'C:\acted_in.csv' WITH (FORMAT CSV, HEADER);
COPY actors TO 'C:\actors.csv' WITH (FORMAT CSV, HEADER);
COPY aka_names TO 'C:\aka_names.csv' WITH (FORMAT CSV, HEADER);
COPY ...
```

# Neo4j - Database import

## Step 2 - Importing data

```
USING PERIODIC COMMIT 1000 LOAD CSV WITH HEADERS FROM "file:
///C:\\actors.csv" AS row CREATE (:actors{
idactors:toInt(row.idactors),
lname:row.lname,
fname:row.fname,
mname:row.mname,
gender:toInt(row.gender),
number:toInt(row.number)
});
```

TUDelft

# Neo4j - Database import

## Step 3 - Creating indices

```
CREATE CONSTRAINT ON (t:actors) ASSERT t.idactors IS UNIQUE;
CREATE CONSTRAINT ON (t:aka_names) ASSERT t.idaka_names IS
UNIQUE;
CREATE CONSTRAINT ON (t:aka_titles) ASSERT t.idaka_titles IS
UNIQUE;
CREATE CONSTRAINT ON (t:genres) ASSERT t.idgenres IS UNIQUE;
CREATE CONSTRAINT ON (t:keywords) ASSERT t.idkeywords IS
UNIQUE;
CREATE CONSTRAINT ON (t:movies) ASSERT t.idmovies IS UNIQUE;
CREATE CONSTRAINT ON (t:series) ASSERT t.idseries IS UNIQUE;
```

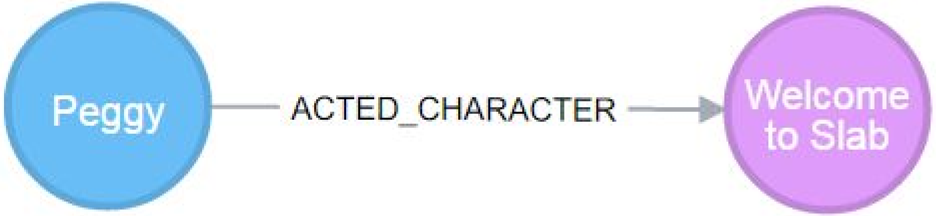
# Neo4j - Database import

## Step 4 - Creating relationships

```
USING PERIODIC COMMIT LOAD CSV WITH HEADERS FROM "file:///C:
\\acted_in.csv" AS row

MATCH (a:actors {idactors:toInt(row.idactors)})

MATCH (b:movies {idmovies:toInt(row.idmovies)})

MERGE (a)-[:ACTED_CHARACTER {

character:coalesce(row.character, "N/A"),

billing_position:coalesce(toInt(row.billing_position), 0)

}]->(b);
```

# Neo4j

| | idmovies integer | idactors integer |
|---|---|---|
| 1 | 14 | 6 |
| 2 | 14 | 6 |
| 3 | 14 | 6 |
| 4 | 14 | 6 |
| 5 | 14 | 6 |
| 6 | 14 | 6 |
| 7 | 14 | 6 |
| 8 | 14 | 6 |
| 9 | 14 | 6 |
| 10 | 14 | 6 |
| 11 | 14 | 6 |
| 12 | 14 | 6 |
| 13 | 14 | 6 |
| 14 | 14 | 6 |
| 15 | 14 | 6 |
| 16 | 14 | 6 |
| 17 | 14 | 6 |
| 18 | 14 | 6 |
| 19 | 14 | 6 |



# Neo4j - Query Design

- Functionally very similar to SQL
  - No need for **DISTINCT**
- Same pattern of controllers, joins replaced by relationships
  - **MATCH** p=(m:movies {idmovies:1})-[r:MOVIE_GENRE]->(g:genres) RETURN p;
- Specific info = 1 query
- Full info = up to 5 queries
  - Untapped potential?
- Sometimes, relationships are needed when joins are not

TU Delft

# Neo4j - Advantages

- Removal of foreign keys
- Traverses relationships quickly
- Allows exotic relationships

# Neo4j - Limitations/issues

Within the context of IMDB:

- Not enough relationships
- Relationships too simple

# Neo4j - Limitations/issues

- Not suited for API
- Lack of parameterized queries
- Method of creating relationships (CSV) inconvenient
- Lack of different data types

# Neo4j - Lessons learned

- Make sure to remove foreign keys
- Still a little immature
- Not built for supporting average use cases

# Demo

- Movie information
- Actor information
- Genre information

# Recap

•

# Questions?

- 