CSCI 3901: Course Project

Overview

The following project defines a system that connects family tree information with an archive of images and its metadata. It can be used by genealogists to find ancestors/descendants for an individual, find the relation between a pair of individuals as well as find corresponding pictures for those people. The system can also keep a history of marriages and divorces between individuals. The entire family tree data and media archive remains persistent through the MySQL database.

The system is implemented using Java to follow Object-Oriented Programming practices. The hierarchy of the family tree is stored in the MySQL database and represents a Directed Acyclic Graph. Algorithms such as Breadth-first Search are used to find ancestors/descendants of an individual and Lowest Common Ancestor to find a biological relationship between a pair of people.

Files and External Data

The code for the system is divided into the following files:

1. **DBConnection.java**
   Defines a static method to receive Connection object for JDBC Connection.

2. **PersonIdentity.java**
   Defines data and methods for an individual.

3. **FileIdentifier.java**
   Defines data and methods for a media file.

4. **BiologicalRelation.java**
   Defines cousinship and the level of separation between a pair of individuals.

5. **Genealogy.java**
   Defines data and methods that can be used by the user to perform operations on the family tree and media archive.

6. **GenealogyTest.java**
   Defines JUnit tests for all methods of Genealogy class.

Data Structures and their relation to each other

Classes

DBConnection

Description:
Defines a static method to receive the Connection object for JDBC Connection. All methods that interact with the database will call this method for initiating the connection.

Methods:
- public static Connection getConnection() throws SQLException
  Checks for MySQL driver and returns a JDBC Connection object based on database URL, username, and password.

PersonIdentity

Description:
Defines minimum data and methods that can uniquely identify a person in the database.

Data:
- private int personId;
  The primary key of the person in the database.

- private String name;
  The name of the person, not necessarily unique.

Methods:
- Getter methods for personId and name.

FileIdentifier

Description:
Defines minimum data and methods that can uniquely identify a file in the database.

Data:
- private int mediaId;
  The primary key of the file in the database.

- private String fileLocation;
  The location of the file.

Methods:
- Getter methods for mediaId and fileLocation.

BiologicalRelation

Description:
Defines cousinship and level of separation between a pair of individuals.

Data:

- int degreeOfCousinship;
- int levelOfSeparation;

Methods:

- Getter methods for cousinship and level of separation.

Genealogy

Description:
Defines data and methods that can be used by the user to perform operations on the family tree and media archive. The methods will take in and return objects of PersonIdentity and FileIdentifier classes.

Methods:

- All methods as defined in the assignment text.

- int addNewAttributeType(String attributeType, String tableName, Connection conn)
  Used by recordAttributes() and recordMediaAttribute() method to add a new attribute type is encountered in the Map<String, String> passed as a parameter to those methods.
  String attributeType: the name of the new attribute type.
  String tableName: Name of the table where the attribute type must be added.
  Connection conn: JDBC Connection object.

Key Algorithm, Strategy and Design Elements

The core logic of the system for generating the Family Tree Hierarchy, Media Archive, and the relationship between them is maintained in the MySQL Relational Database. This allows the system to utilize the power of SQL queries for performing various operations instead of relying more on the logic of Java code.

The following figure shows the Schema design for the database:

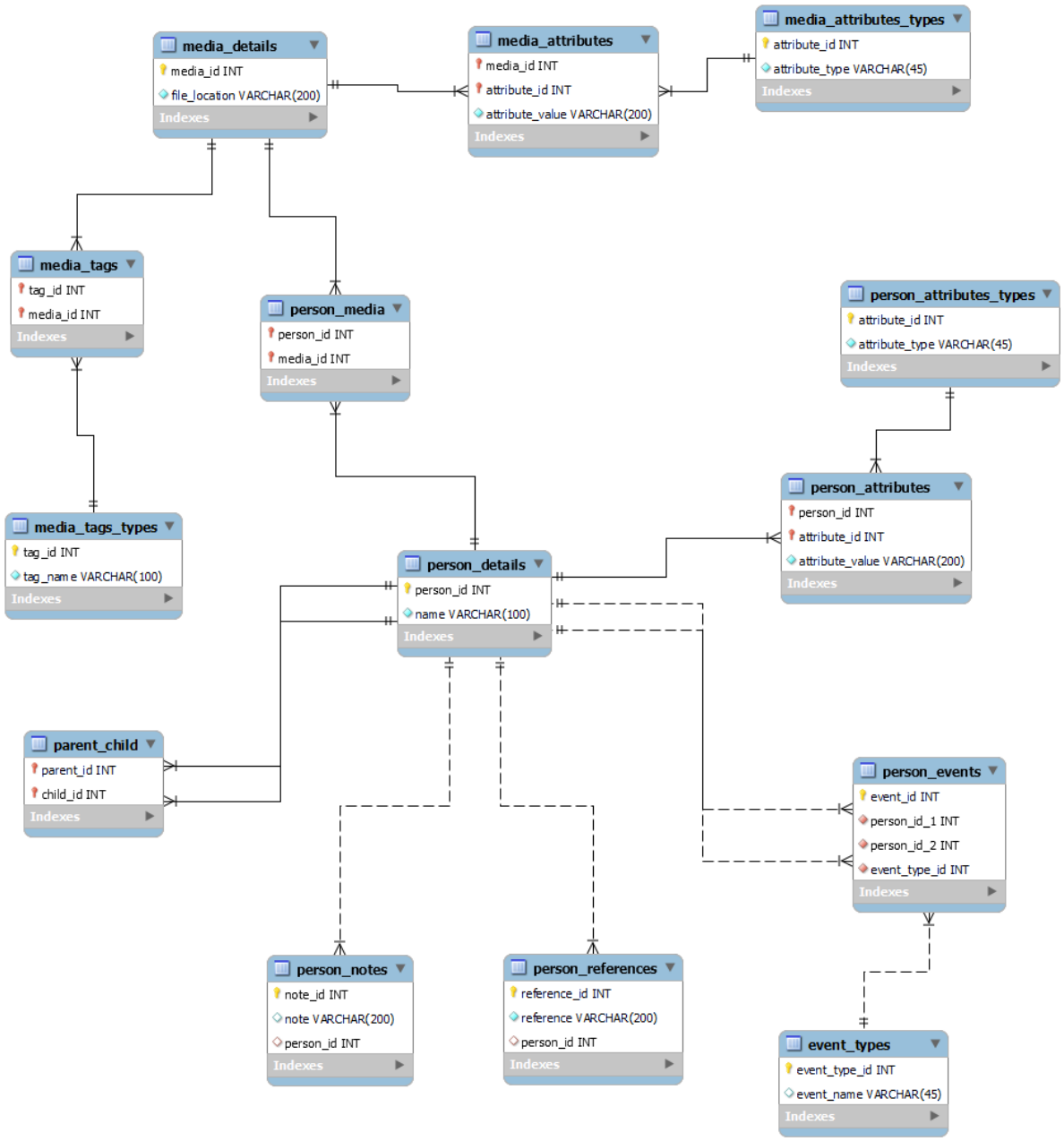


Figure 1. Schema Design for the Database

Important Relations in the Database

parent_child

Maintain family tree hierarchy:
This relation will represent the Family Tree hierarchy. **parent_id** and **child_id** are IDs of the parent and child respectively and are foreign keys to IDs in **person_details** table. A record in this relation is identified by a composite key of **parent_id** and **child_id**. It has a many to one relation with **person_details** table. A **person_id** in **person_details** table can be a parent to multiple children in **parent_child** table. Also, a **person_id** in **person_details** table can be a child to multiple parents **(at most 2)**.

The resultant hierarchy will be a **Directed Acyclic Graph (DAG)** stored in the database.

Algorithms implemented using this relation

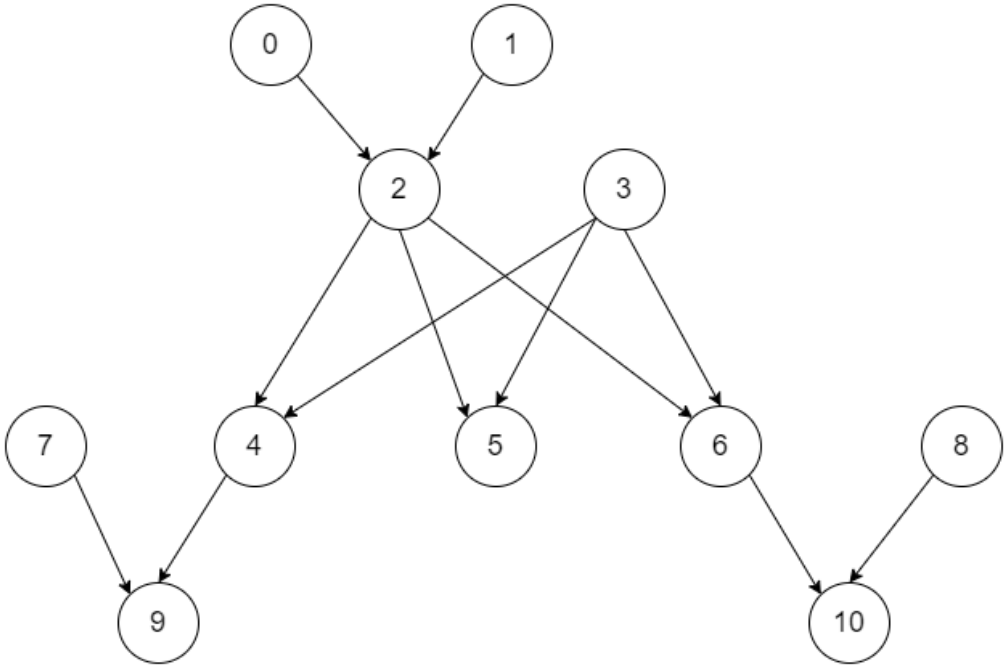
Breadth-first search:

This relationship is used by recordChild() method of Genealogy class to record a parent-child relation between 2 individuals. The system uses BFS for traversing the Directed Acyclic Graph. When given a person (node in graph), this algorithm can find the ancestors or descendants of that individual by level. The system uses a **"Recursive With" Common Table Expression** query to find the ancestors/descendants of an individual.

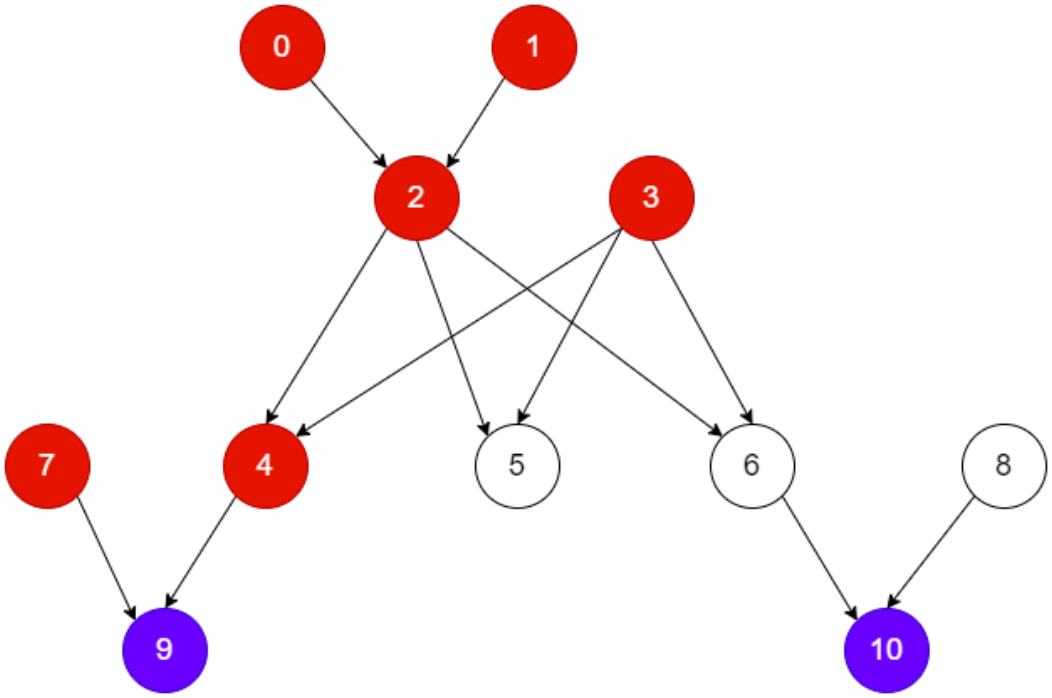
***Lowest-common ancestor in DAG to find the relation between pair of individuals* (taken and modified from Milestone 3 [page 2] of this project):**

A Family Tree is usually represented as a **Directed Acyclic Graph (DAG)**. As defined in the assignment text, to find the relationship between a pair of individuals, we must find their **Lowest Common Ancestor** first. To find this, the system will use the following approach:

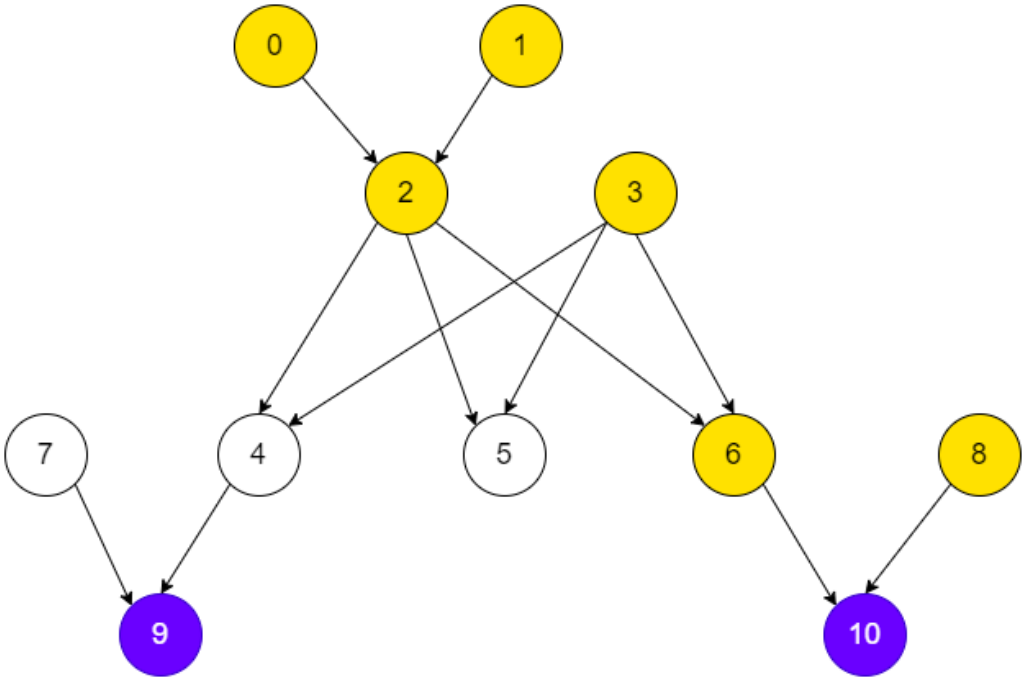
1. Consider the given graph. We want to find the Lowest Common Ancestor of Node 9 and 10.



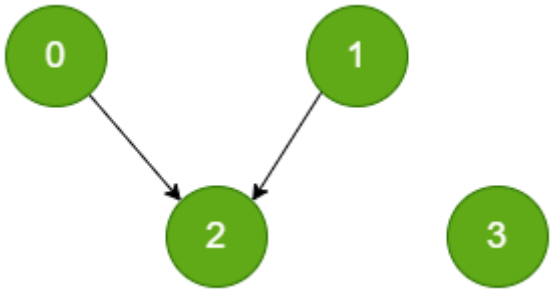
2. We will find all the parents (and their parents) till root for Node 9 using **BFS using Recursive With query**. This query will store the node itself, its ancestors, and their corresponding depths starting from this node (Node 9). These nodes are shown using **Red** colour.



3. Now, we will find all the parents of Node 10 using the same approach. They are shown in **Yellow** colour.



4. Now, the intersection of Red and Yellow will form a **sub-graph** as shown below. A **join query** will find this intersection of Nodes. A pair of individuals in a family tree can have at most 2 lowest-common ancestors at the same level/depth. So, we limit the intersection by 1 and sort the result by depth from initial nodes to get the ancestor with the least depth. This is our lowest-common ancestor. Hence, Node 2 and 3 are the LCAs of Node 9 and 10. Considering either of them will provide the same result.



5. Finally, based on the formula provided in the assignment text, the smaller of the depths of Node 9 and 10, minus 1 will give the **cousinship (2-1=1 in this case)**, and the difference of these depths will give the **level of separation (2-2=0 in this case)**. Also, from the provided nodes, if either Node is an ancestor to the other one, then the ancestor will be the **LCA. This example is tested in JUnit Class GenealogyTest in findRelation() test.**

person_media

Defines many-to-many relationship between people and media files.

Finding the corresponding images of people **(taken from Milestone 3 [page 4]):**
As per the Database Schema mentioned above, there is a **many-to-many relationship** between the people and media files. This relation is shown as a separate table, **person_media** and will store the ID of a person and the ID of their corresponding image. This way the system can fetch the image corresponding to any individual. Also, images can be filtered based on date by performing join operation with the **media_details media_attributes** table.

person_events

Record partnership and dissolution:
This relation will keep track of partnership and dissolution between a pair of individuals. The **event_id** will be the primary key to identify an event. A record will have an **event_id**, **person_id of both individuals** and an **event_type_id**. This event_type_id will be a foreign key to the event_types table. This table will define types of events that can occur between a pair of persons.

Currently, we have defined 2 types of events: **marriage** and **divorce**, but the system can accommodate other types such as **friendship** and **acquaintance**. A unique constraint is set on the **event_name** field in **event_types table** so that events are not repeated.

person_attributes and media_attributes

Record dynamic attributes **(taken from Milestone 4 [page 2]):**
Separate attribute tables were added for **person_details** and **media_details** table. Attributes will be defined in the **person_attribute_types** and **media_attributes_types** tables and identified by a key. **person_attributes** and **media_attributes** tables connect

with their respective **attribute_types** table to store attribute values. The benefit of this is that new attributes for people or media can be defined in the future as needed.

Handling Partial Dates:

Based on the above strategy for storing dynamic attributes, all attribute values are stored as VARCHAR in the database. However, for filtering data based on a date range and handling dates formats such as yyyy-MM-dd, yyyy-MM, and yyyy, the system uses MySQL's **STR_TO_DATE()** function to convert these VARCHAR dates to SQL Date when the query is fired.

Also, methods such as findMediaByTag(), findIndividualsMedia() and findMediaByLocation() take in date parameters as String. The **STR_TO_DATE()** function will handle this case as well. It will append partial dates with appropriate zeroes. For example, 2021-01 will become 2021-01-00, 2019 will become 2019-00-00. This strategy works well for filtering data by dates using >=, <= or **between** clause of MySQL.

media_tags and media_tags_types

Creating and maintaining reusable tags for media files:

A media file can have multiple useful tags through which it can be identified. Also, a tag can be assigned to various media files and new tags can be encountered in the future. Hence, the database has a **many-to-many relationship** between **media_details** table and **media_tag_types** table. This relationship is defined in the **media_tags** table through a composite key of **media_id** and **tag_id**.

The **media_tag_types** table uniquely defines all available tags. A new entry will be added whenever a new tag is added in the **tagMedia()** method. If the tag exists, then **media_tags** table will reuse the existing tag from **media_tag_types** table through **tag_id**.

Test Plans

Black-box testing:

Provided as part of Milestone 2 submitted on git repository.

White-box testing:

Defined as JUnit tests provided in src folder of git repository in file named GenealogyTest.java.

Limitations

- For filtering media files by location, the system will only consider the **"location"** type of attribute defined in the **media_attributes_types** table.
- For handling partial dates, the system appends appropriate zeroes to the date using **STR_TO_DATE()** function of MySQL. The dates themselves are stored as VARCHAR in the database.

- For recording partnerships and dissolutions, events such as **marriage and divorce** are defined in the **event_types** table. There is a unique constraint on them and the **event_type_id** is the primary key used to reference it in **person_events** table. **Marriage is assigned key of 1 and divorce is assigned 2 which will be populated on SQL file import** (provided in sql folder of git repository). This must be maintained for consistency.

References

[1] Kolesnikova, “Storing trees in RDBMS,” *Bitworks Software - custom software development company*, 24-Nov-2017. [Online]. Available: https://bitworks.software/en/2017-10-20-storing-trees-in-rdbms.html. [Accessed: 14-Nov-2021].