CSDS 341 - Final Project

Database Project Team 10
Group Members: Selah Dean, Stephen Hogeman, Adriana Kamor

Table of Contents:

Section 1: Access Our Application

Database Backup File: collegeswimmingDB.bak
Virtual Machine: csdswinlab018
    Username: student
    Password: J4GDmq6qfeB4wJgL
See “User Manual” for the description of how to run the application
Zip File: CollegeSwimmingDBCode.zip
TA Meeting: May 2, 2023 at 2:00pm
Username and Password to connect in JDBC:
    Username: dbuser
    Password: scsd431134dscs

Section 2: Description of the Database

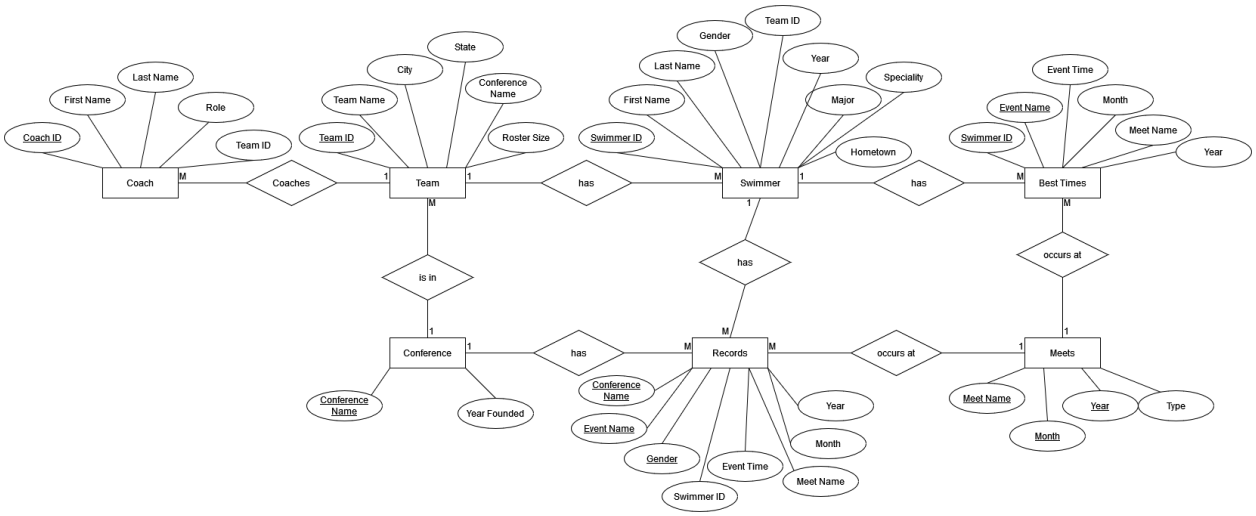
This project is a database that stores data of college swimmers. This database provides a way to store all college swimmers with their best times and other relevant information such as the team they swim on and the coach of a team. The information for different swim meets and records for each conference will also be stored in the database. The explanation for each table is as follows:

- **Swimmer**
    - The Swimmer table stores the individuals who swim(compete) at the collegiate level. Each swimmer has their own unique ID as well as their demographic information. There can only be one instance per person for the Swimmer table. Team ID is a foreign key that references the Team table. This records what team the swimmer competes for. Many swimmers can belong to one team.
- **Coach**
    - The Coach table stores the coaches who coach each registered team in the Team table. There can be many coaches for one team, but there cannot be a single coach coaching multiple teams. Each coach has their own unique ID as well as their first and last name. Each coach will also have a role which denotes whether they are a head or assistant coach.
- **Team**
    - The Team table stores each team that competes at the collegiate level. The name, location, conference, and roster size are recorded in this table. Each team has its own unique ID. The conference name references the Conference table, where the collegiate level teams are broken down into conferences(groups that compete within each other) based on speed.
- **Conference**
    - The Conference table stores the name of each conference(groups that compete within each other based on speed) and what year the conference was founded.
- **Best Times**
    - The Best Times table stores a swimmer's fastest time for each event they have done. This table references the Swimmer ID to associate the event and best time to a specific swimmer. This table stores the time in which the race was completed, what the event was, and when. This table also stores what meet the best time was achieved at, so the Meet table is referenced using the Meet name, Month, and Year which all form the key of the Meet table.
- **Meet**
    - The Meet table stores all meets(scheduled competitions between teams) performed at the collegiate level. The date, name, and type of meet are recorded in the table. The type of meet can be dual(two teams), invite(any team from any conference), conference(all teams within the same conference), or nationals(a qualifying competition for the top swimmers in each conference).

- **Records**
    - The Records table stores the top times in each event per conference per gender. This table stores what meet the record was accomplished at and what the time and event were. This table also stores information to delineate which gender and conference the record is associated with.

Section 2a: ER Diagram

The following is an ER diagram depicting the structure of the database and the relationships between each table:



The relationships between each table illustrated in this diagram are as follows:

- Each swimmer can only belong to one team, but each team has many swimmers.
- Each swimmer has many best times, but they can only have one best time per event type (represented by the event name).
- Each coach can only belong to one team, but each team can have multiple coaches.
- Each team can only be a part of one conference, but one conference can have multiple teams within it.
- Each conference has multiple records , but they can only have one record per gender per event type (represented by the event name and the gender).
- Each meet can have multiple records broken (new records being set), but each record has one meet that it has occurred at (the same record cannot occur at multiple meets at the same time).
- Each meet can have multiple best times (new best times being set), but each best time has one meet that it occurred at (the same best time cannot occur at multiple meets at the same time).

Section 3: Use Cases

Implemented Use Cases

Find a swimmer's rank within a team for an event (Selah)

- **Description:** This use case uses a given swimmer ID and an event name and returns the swimmer's rank in a certain event within their team.
- **User Requirements:** In order to perform this query, input from the user for swimmer ID and event name are required.
- **SQL Query:** First create a temporary table roster that stores the swimmerIDs that have the same teamID as the inputted swimmerID and the same gender as the inputted swimmerID. Then find the count plus one of the eventtime column from the inner join between besttimes and roster where the eventname is the inputted eventname and the eventtime is less than the best time of the swimmer corresponding to the inputted swimmerID and eventname. The query returns the count plus 1 since if a swimmer is top ranked the count would return 0 even though the swimmer is ranked first. This query requires the swimmer table and the best times table.
- **Code:**

```
create procedure dbo.findEventRank (@SwimmerID as int, @EventName as varchar(25))
as
begin
select count(eventtime) + 1 as teamrank
from (select swimmerID from swimmer
where teamID = (select teamID from swimmer where swimmerID = @SwimmerID)
and gender = (select gender from swimmer where swimmerID = @SwimmerID)) as roster INNER
JOIN besttimes ON roster.swimmerID = besttimes.swimmerID
 where eventname = @EventName and
 eventtime < (select eventtime from besttimes where swimmerID = @SwimmerID and
eventname = @EventName);
end;
```

- **File Needed:** SQLFindEventRank.java

Add a swimmer (Selah)

- **Description:** This use case adds a new swimmer to the database given all of the information stored for a swimmer. When the swimmer is inserted into the database, the roster size for the team they are on increments by one.
- **User Requirements:** The user provides the first name, last name, gender, team ID, year, major, speciality, and hometown of the new swimmer
- **SQL Query:** This use case requires a trigger on the swimmer table after insert. When a swimmer is inserted the team table is updated so the roster size is incremented by one on the tuple that has the same team ID as the new swimmer. The insert method for a swimmer is a stored procedure in the database and uses the values for the user input. This query requires the swimmer table and the team table.

- **Code:**

```
create trigger increaserostersize
        on swimmer
        after insert
        as
        begin
         update team
                set rostersize = rostersize + 1
                where team.teamID = (select teamID from inserted);
end;
create procedure dbo.insertSwimmer @FirstName varchar(50), @LastName varchar(50),
@Gender varchar(6), @TeamID int, @Year int, @Major varchar(50), @Speciality varchar(50),
@Hometown varchar(50)
as
begin
insert into swimmer(firstname, lastname, gender, teamID, year, major, speciality, hometown)
values(@FirstName, @LastName, @Gender, @TeamID, @Year, @Major, @Speciality,
@Hometown)
end;
```

- **File Needed:** SQLInsertSwimmer.java

Switch Teams (Selah)

- **Description:** This use case pertains to when a swimmer transfers schools and switches teams using the swimmer's ID and a new team ID. The original team that the swimmer transferred from will have their roster size decremented by 1. Then teamID for the swimmer will be updated. For the new team the swimmer has joined, the roster size will increase by 1.
- **User Requirements:** The user provides the swimmer ID and the new team ID for the swimmer.
- **SQL Query:** This use case requires a trigger on the swimmer table after update. When the team ID is updated for a swimmer in the table, the team table is also updated. The roster size of the old team ID of the swimmer is decremented by 1 and the roster size of the new team ID of the swimmer is incremented by 1. The switch teams method for a swimmer is a stored procedure in the database and uses the values for the user input. The query requires the swimmer table and the team table.
- **Code:**

```
create trigger switchteams
        on swimmer
        after update
        as
        begin
                update team
                        set rostersize = rostersize + 1
                        where team.teamID = (select teamID from inserted)
```

```
                update team
                        set rostersize = rostersize - 1
                        where team.teamID = (select teamID from deleted)
end;
create procedure dbo.swimmerTransfer @swimmerID int, @teamID int
as
begin
        update swimmer
                set teamID = @teamID
                where swimmerID = @swimmerID
end
```

- **File Needed:** SQLSwitchTeams.java

Find meets where records were broken (Stephen)

- **Description:** This use case returns meets where records were broken to the user. Given an input for the meet name and using the foreign key for the meet name, the event name and the swimmer ID of the records table is joined with the meet name, month, year, and type of meet from the meet table. If no records were broken at the meet, nothing is returned. But, if there were records broken at that meet, the relation of those records is returned.
- **User Requirements:** The user is required to give an input of the meet name in order for the query to find what records were broken at a specific meet.
- **SQL Query:**

```
with M(meetname, month, year) as (select meetname, month, year from meet where
meet.meetname = '       ' and meet.month = '        ' and meet.year = '         ')
select R.meetname, R.month, R.year, R.gender, R.eventname, R.eventtime from records as R
inner join M
on R.meetname = M.meetname and R.month = M.month and R.year = M.year;
```

- **File Needed:** SQLFindRecordByMeet.java

Add a new conference (Stephen)

- **Description:** This use case creates a simple conference that teams can be added to. From creation, a conference name and year founded is needed from the user. Later, when teams are added, their corresponding conference names will need to be updated to match their new conference.
- **User Requirements:** The user is required to give an input of the conference name and the year in which it was founded.
- **SQL Query:**

```
insert into conference(conferencename, yearfounded) values('    ', '         ')
```

- **File Needed:** SQLInsertConference.java

Update a best time and a record when broken (Adriana)

- **Description:** This use case is to check the current records and update them as needed each time a best time is updated or created. For the swimmer whose best time is being updated or inserted, the team ID is found and used to get the conference name of the swimmer. Then using the swimmer's gender, conference name, and the event name for the best time, the corresponding record is found. If the best time is faster than the record, the record is updated with the appropriate information. If not, the record remains the same. This use case is in the form of a trigger associated with updating/creating best times.
- **User Requirement:** There is no user requirement for this specific use case, as it was decided that it would be more beneficial to automatically implement upon the user updating or creating best times for a swimmer.
- **SQL Query:**

  For updating a best time:

```
create procedure dbo.updateBestTimes @SwimmerID int, @EventName varchar(25),
@EventTime float, @MeetName varchar(100), @Month varchar(25), @Year int
as
begin
update besttimes set eventname = @EventName where besttimes.swimmerID = @SwimmerID
update besttimes set eventtime = @EventTime where besttimes.swimmerID = @SwimmerID
update besttimes set meetname = @MeetName where besttimes.swimmerID = @SwimmerID
update besttimes set month = @Month where besttimes.swimmerID = @SwimmerID
update besttimes set year = @Year where besttimes.swimmerID = @SwimmerID
end;
```

  For triggering record-break check:

```
alter trigger recordbreak on besttimes
after insert, update
as
begin
if ((select eventtime from inserted) < (select eventtime from records where records.eventname =
(select eventname from inserted) and records.gender = (select gender from swimmer where
swimmer.swimmerID = (select swimmerID from inserted)) and records.conferencename = (select
conferencename from team where teamID = (select teamID from swimmer where swimmerID =
(select swimmerID from inserted)))))
        begin
        update records set records.swimmerID = (select swimmerID from inserted);
        update records set records.eventtime = (select eventtime from inserted);
        update records set records.meetname = (select meetname from inserted);
        update records set records.month = (select month from inserted);
        update records set records.year = (select year from inserted);
        End
if ((select eventtime from records where records.eventname = (select eventname from inserted)
 and records.gender = (select gender from swimmer where swimmer.swimmerID = (select
```

swimmerID from inserted))
and records.conferencename = (select conferencename from team where teamID = (select teamID from swimmer where swimmerID = (select swimmerID from inserted))))is null)
begin
insert into records(conferencename, gender, eventname, swimmerID, eventtime, meetname, month, year)
values((select conferencename from team where teamID = (select teamID from swimmer where swimmerID = (select swimmerID from inserted))),
(select gender from swimmer where swimmer.swimmerID = (select swimmerID from inserted)) ,
(select eventname from inserted), (select swimmerID from inserted), (select eventtime from inserted), (select meetname from inserted), (select month from inserted),
(select year from inserted))
End

- **File Needed:** SQLInsertBestTimes.java

Retire a swimmer (Adriana)

- **Description:** This use case is to retire a swimmer when they have either graduated or quit swimming. Retiring is different from deletion because their records can still be accessed no matter the period of time for which the swimmer has actively competed. Instead of deleting a swimmer once they have retired, their team ID is set to null and decreaserostersize is triggered which will decrease the roster size of their previous team by 1.
- **User Requirement:** The user is required to supply the swimmer ID of the swimmer who is retiring.
- **SQL Query:**
  create procedure dbo.retiredSwimmer @SwimmerID as varchar(50)
  as
  begin
  update swimmer
  set teamID = null where swimmerID = @SwimmerID
  end;
- **File Needed:** SQLRetireSwimmer.java

Potential Use Cases (These are descriptions of possible uses only, no queries)

Updating best time for a swimmer

- Given a time for an event from a meet for a swimmer, check the best time for the event name and the swimmerID. If the given time is faster than best then update the value if not nothing changes.

Adding new coach

- New tuple inserted in the Coach relationship, and the attributes created are First Name, Last Name, Role, and their corresponding Team ID, which references the Team table.

Add new conference

- Uses the Conference table to actually create the new conference, with a created year. Uses the Team table, as the value of *Conference name* is updated for the teams added to that conference

Coach switching teams

- Given a Coach ID, the tuple with corresponding Coach ID will update their corresponding Team ID

Teams switching conferences

- Given a Team ID, Conference Name is updated with corresponding new conference

Getting a list of teams in a location

- Uses the City attribute of each team to find which teams are in an input city

Finding swimmers from the same hometown

- Given a Hometown, Return swimmer tuples that meet Hometown input requirement

Find all the records broken in a certain year

- Given a year, Return all records in the Records table that have a year equivalent to the input year.

Section 4: Functional dependencies, physical database design and normalization in the database

Functional Dependencies and Normalization

For every table the only functional dependency is the primary key or a composite key that includes the primary key. For any other attribute in the table it cannot uniquely identify any other singular attribute. Each attribute or combination of attributes that doesn't include the primary key, results in duplicate tuples therefore is not a functional dependency.

Physical Database Design

We have structured the physical database into 7 tables. Each table has a primary key where the Coach, Swimmer, and Team tables have unique numerical IDs. The other tables have composite primary keys. In Microsoft SQL Server, a clustered index is created for each primary key for each table.

Section 5: User Manual

To access this database, make sure all files are downloaded from the ZIP file listed above. In order to structure the database and command line interface, JDBC, Microsoft SQL Server, and the Windows command line are required. Files should be set up by separate use cases using Java along with a SQL procedure file for each use case. The exception to this format is for use cases that are considered triggers, as all triggers can reside in the same SQL file. This project utilizes 7 tables (swimmer, coach, team, conference, besttimes, records, and meet) to fully account for all events, people, and categorizations for college swimming. These tables should be initialized in a "create" table SQL file.

Each of the listed implemented use cases have JDBC code to allow for user input. Each use case is in a separate java file with the name SQL[use case name].java. In order to run the query, you must pull up the windows command line and navigate to the folder where the java file resides. Then compile the file using

the javac command and then run using the java command. Once the file is run, it will ask for the user input and then execute the query.

Section 6: Reflection

Through this project, we have learned the behind the scenes of creating a database and all of the different steps that need to be followed. First we had to come up with a basic idea for a database. From there we brainstormed the information that we wanted to have stored in our database. Then we had to divide up the information into tables and created an ER Diagram to lay out the relationships between the tables. Once we created the ER Diagram we were able to start making the tables in SQL and decide on different use cases that could be done with the stored data. We had to figure out the best way to implement these use cases, some of which required triggers, and then we also created stored procedures for these cases. The last step of implementing the use cases was using JDBC to allow for input from a user.

Throughout this process we have learned the intricacies of a database and the amount of work required to implement a database. We have also realized the databases are ever evolving and were able to come up with more use cases or different information that we want to store as we went along. The database we have created is still a work in progress since different information could be added to increase the scope of the database. This also taught us how important it is to build a scalable database that can incorporate more tables and values while still maintaining its structure, clarity, and efficiency. In the future, we would look to implement a more fleshed out user interface and improve the user experience, along with incorporating more attributes to describe more categorizations or open up our database to record high school swimming.