Chess Mate

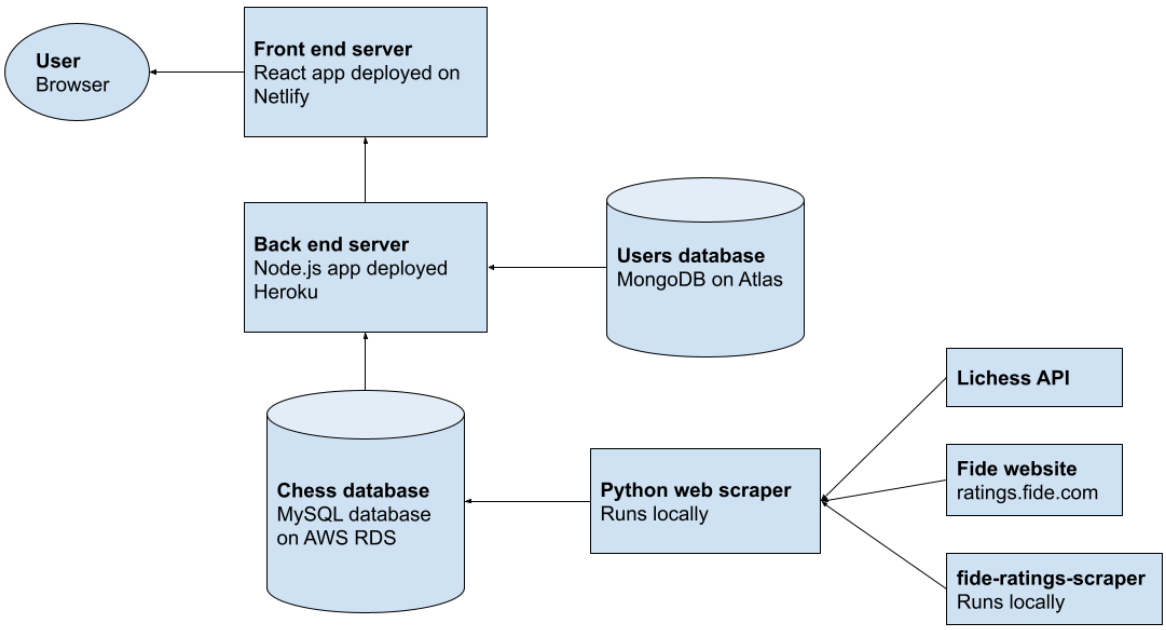
CIS 450 Final Project - Group 13
Constance Wang
Serena Jiao
Henry Chen
Aydan Gooneratne

Introduction

Lichess.org is an online platform where users can register an anonymous account and play games against others. The International Chess Federation, known as **FIDE** (Fédération Internationale des Échecs), is an organization under which chess players can play over the board games in rated tournaments.

Many Fide chess players have Lichess accounts where they play for fun or to practice their skills. We gathered data from both platforms and created an app that could help the user guess which Fide player a specific Lichess account belonged to, and provide general statistics and trivia that incorporated both platforms' data that helped users get a better understanding of the differences between the two platforms. This could help chess players prepare for real-life opponents by following their games on Lichess.

Architecture



Technologies used

Frontend:

- React
- Bootstrap - styling
- Typescript - improve code quality
- Chart.js - data visualization
- Netlify - deployment

Backend:

- Node.js, Express
- Heroku - deployment

Database:

- MySQL on AWS RDS - storing chess game and player data

- MongoDB on Atlas - storing user credentials

Scraping:

- Python, including various utility libraries
- Puppeteer for web-scraping with a headless browser

Data

We compiled games, players, and rating history for both Lichess and over the board (FIDE) chess.

Data Sources

Lichess

Lichess data was relatively clean and easy to work with since it originated from an online database.

We started off downloading 90 million Lichess games from March 2022 from the lichess.org open database. Chess games use a standard, human readable Portable Game Notation (PGN) format. We used python-chess to parse the PGNs into the LichessGame schema and extracted the Lichess user ids of black and white players.

This enabled us to construct a large list of Lichess users, since there wasn't one available from any API endpoint. We then used the Lichess API's users endpoint to scrape user data and parse them into the LichessUser schema format. We fetched rating histories for each user in a separate pass from the rating history endpoint, parsed them, and inserted them into the LichessHistory table.

Fide

Fide data was much harder to obtain. Similar to Lichess, we used PGN games as our entrypoint to scraping the player data. We used three open-source PGN databases of tournament games: CanBase, Ozbase, and Caissabase. Caissabase is the largest database at 5.6 million games, which was large enough for the project. However, Caissabase only collects top-level grandmaster games. In order to find more potential matches in rating histories with Lichess players, who are overwhelmingly amateurs, we included CanBase and Ozbase, which have more amateur-level games.

We sampled games from each of these sources and extracted the player names. The official Fide website has no public API for obtaining data. As a result, we used Puppeteer to programmatically search the Fide website for the Fide id of the given player. Then, we sent the FIDE IDs to fide-ratings-scraper, an open-source project we ran locally to obtain rating history and player profile information, given the Fide id. Finally, we parsed and inserted Fide games, ids, and rating history into the database.

Overlap

Because they both describe the same game—chess—Fide and Lichess games have many overlapping fields to join on such as black/white players, game type (classical, blitz, rapid), ECO code (code corresponding to a chess game's opening moves), etc. The same is true for Fide and Lichess players, who both have nationality, ratings, titles, rating histories, etc.

Database

Cleaning and Entity Resolution

We took the following into account to ensure the quality and standardization of our data across the overlapping attributes of Lichess and over the board chess data sources:

- Lichess users with disabled accounts were ignored since we could not fetch data for them anyway.
- Filtered out Fide games played before 1990.
- For simple overlapping attributes, we picked the more compact format (usually Lichess's format) as the standard. For example, Lichess used 2-letter country codes like "CA", while Fide data had "Canada". We used python's country_converter to parse the Fide data. Title, rating, and ECO codes were handled similarly.
- Lichess had daily rating history points, while Fide had monthly data - we chose with monthly data to keep the size of our dataset manageable and sampled the Lichess users' history for each month.
- We used SQL enums where possible to keep the data clean and compact, like game type, which was either "classical", "blitz", "rapid", or "bullet".

- We did not parse all the games from either of our game sources, since the database would've become too large and expensive to query, but tried to sample games from a variety of sources.

Schema

LichessPlayers (id, username, bio, country, location, firstName, lastName, fideRating, uscfRating, ecfRating, createdAt, seenAt, title, bulletRating, bulletNumGames, bulletIsProvisional, blitzRating, blitzNumGames, blitzIsProvisional, rapidRating, rapidNumGames, rapidIsProvisional, classicalRating, classicalNumGames, classicalIsProvisional, totalPlayTime, isVerified)

LichessHistory (lichessId, month, year, type, rating), lichessId foreign key references LichessPlayers(id)

LichessGames (id, date, time, timeControl, eco, whiteId, whiteRating, blackId, blackRating, result, pgn), whiteId foreign key references LichessPlayers(id), blackId foreign key references LichessPlayers(id)

FidePlayers (id, firstName, lastName, country, birthYear, sex, title, classicalRating, rapidRating, blitzRating, worldRankAllPlayers, worldRankActivePlayers, nationalRankAllPlayers, nationalRankActivePlayers)

FideHistory (fideId, month, year, type, rating), fideId foreign key references FidePlayers(id)

FideGames (id, date, time, timeControl, eco, whiteId, whiteRating, blackId, blackRating, result, pgn), whiteId foreign key references FideIdLookup(playerId), blackId foreign key references FideIdLookup(playerId)

FideIdLookup (playerId, fideId), fideId foreign key references FidePlayers(id)

Fide Id Resolution

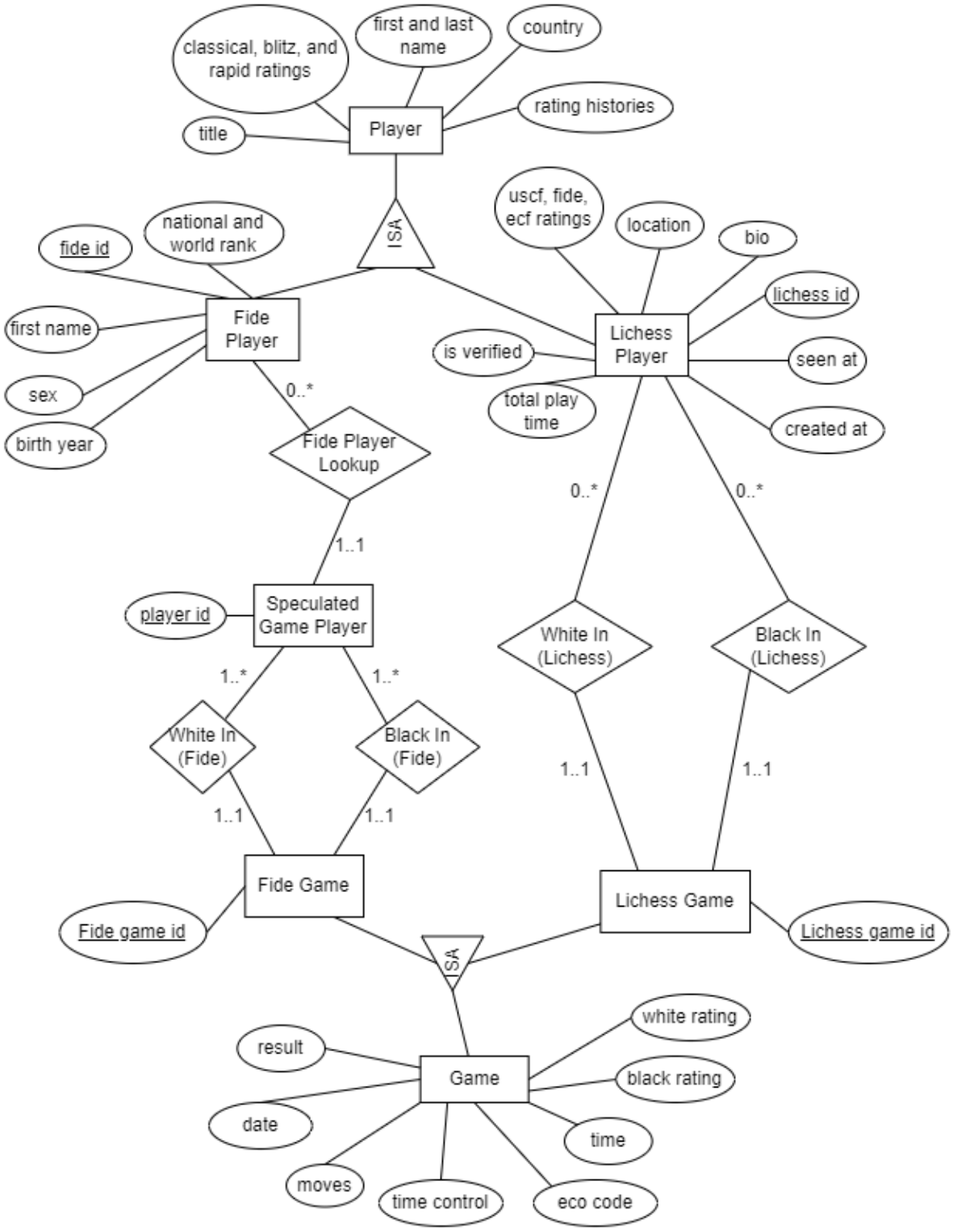
Because over the board PGN games do not contain the Fide ids of the black and white players, we could not directly use Fide ids in the FideGames table as foreign keys - we just cannot tell which Fide user played the games.

Our solution to model the relationship between Fide users and games was to use a separate player id for the black and white players of the games. Then, we store a "guess" of which Fide player the player id actually is in the FideIdLookup table. We obtain this guess by scraping the Fide ratings website to lookup a Fide player's id using the name. It's not a guarantee that the returned result is correct, because it is possible for two Fide players to have the same name; we are just taking the first of a list of results.

Our earlier idea was to make first name and last name the key of Fide players: while that is BCNF, that would disallow players of the same name.

The FideIdLookup table gives us the flexibility to correct incorrect guesses. Another idea was to put fide ids directly as the black/white player columns; however, this does not enable us to separate out games we know to be played by the same person, but for whom we are not sure of the fide ID. With more time, we planned to extend our app to allow users to upload more tournament games to the database - in this case, if we may know the players with the same name in the uploaded set of games are the same, but may still not know their fide ID and need to correct that later.

Entity-Relationship Diagram



Normal Forms

Table	Functional Dependencies
LichessPlayers	id → all other fields in table

LichessHistory	lichessId, year, month, type → rating
LichessGames	id → all other fields in table except pgn*
FidePlayers	id → all other fields in table
FideHistory	fideId, year, month, type → rating
FideGames	id → all other fields in table except pgn*
FideIdLookup	playerId → fideId

We see that all the (non-trivial) functional dependencies have the left side as a superkey for the respective table. Therefore, our database is in Boyce-Codd Normal Form (BCNF).

*Technically, there is one field, the pgn field, containing the PGN file from which the game is parsed that functionally determines the other fields in FideGames and LichessGames, but this field would be prohibitively large to use as a key. It is a file we only query to replay the game on our game board UI, not actually part of our relational model.

Statistics

Table	Row count
LichessGames	178,029
LichessPlayers	220,987
LichessHistory	4,932,774
FideGames	95,249
FideIdLookup	192,506
FidePlayers	1,890
FideHistory	364,768

Web App Description

Our web app has the following pages:

Home: /

Entrypoint to application. The user begins by searching for the lichess id that they want more information about, eg “-tristan-”. This takes them to the players page. If the user is logged in, also display their list of liked players.

Players: /players

Query params: q
Show information about the profile of the lichess player that matches the query parameter q. Provide a list of users with similar rating histories to the player q, along with information used to calculate the similarity score, and links to their rating history comparison pags. Also, show a link to the list of games played by player q.

Rating History Comparison: /history

Query params: fideId, lichessId
Show an interactive line chart to compare rating histories between a lichess player and a fide player with the ids given by the query params. This lets users see for themselves whether the two players’ ratings peaked/declined at about the same times.

Player Games: /games/:username

A page containing a list of lichess games played by a player with the specified lichess username. Also provides a list of players who played similar openings in their games to the specified lichess username.

Game View: /game/:game_id

A page for viewing a chess game in PGN format. Provides an overview about the game as well as an interactive board to replay the game on.

Login: /login

Log into an existing account or create a new account with Chess Mate in order to save a list of liked players and search them easily on the homepage.

Top Games: /topGames

For providing general information about chess games. Submit a rating threshold and view games between countries with top players.

Amateur Games: /poorGames

For providing general information about chess games. Submit a rating threshold and view players that have low elo and their playtime.

API Specification

Route	Params	Method	Notes	Response*
`/users/signup`	`username: string` `password: string`	POST	Adds a user to the database.	
`/users/login`	`username: string` `password: string`	POST	Logs a user in.	
`/users/liked/:loggedUser`		GET	Get the liked players of a logged in user.	
`/users/liked/:loggedUser/:username`		PUT	Like a player.	
`/users/liked/:loggedUser/:username`		DELE…	Remove like from a player.	
`/player/lichess/:id`		GET	Get a player by their lichess id.	`LichessPlayers object`
`/player/fide/:id`		GET	Get a player by their fideid.	`FidePlayers object`
`/history/lichess/:id`		GET	Get player history by their lichess id.	`LichessHistory object`
`/history/fide/:id`		GET	Get player history by their fide id.	`FideHistory object`

`/sortFidePlayers/:country`		GET	List of fide players for a country sorted by rank	`firstName: string`
				`lastName: string`
				`worldRankAllPlayers: number`

`/getGames/:id`		GET	Get all games for a lichess player by id	`id: string,`
				`date: YYYY.MM.DD`
				`timeControl: string`
				`whiteId: string`
				`blackId: string`

`/getGame/:gameid`		GET	Get a game's data for a lichess game id	`Time: string`
				`date: YYYY.MM.DD`
				`timeControl: string`
				`whiteId: string`
				`blackId: string`
				`whiteRating: number`
				`blackRating: number`
				`pgn: string (chess pgn format)`

`/playerRatings`		GET	Returns lichess ids and corresponding player ratings for various game formats	`id: string`
				`bulletRating: number`
				`blitzRating: number`
				`rapidRating: number`
				`classicalRating: number`

`/sortTitle/:title`		GET	Given a title (e.g. GM) find all players with that title sorted by classical rating	`id: string`
				`username: string`
				`firstName: string`
				`lastName: string`

Route	Params	Method	Notes	Response*
				`country: string` `classicalRating: number`

`/sortByTypeRating/:type`		GET	Given a type (e.g. blitz) find the top 100 players that play that type sorted by rating.	`id: string`
				`username: string`
				`country: string`
				`<type>Rating: string`

`/getRatingsInYear/:year/:type`		GET	Find all fide players and their classical ratings in the year.	`id: string`
				`year: number`
				`rating: number`
				`type: string`

`/getFidePlayerInfo/:id`		GET	Get all information about a fide player given their id	`FidePlayers object`
`/players`	`p: string`	GET	Get all lichess players who have the string p appear in their name at all.	`LichessPlayer object`
`/topGamesCountryPlayers/:elo`		GET	Get a list of all users in all the countries for all players who have played games where both players have an elo above the specified elo	`LichessPlayers object`
`/poorPlayers/:elo`		GET	Find some poor level games (sum of elo < 2400) and determine all players' playtime. Find all the unverified players with an elo less than the specified one and return them sorted by elo	`LichessPlayers object`
`/getSimilarPlayersOpenings/:id`		GET	Get a list of lichess player ids for players who play similar openings to player with id id.	`username: string`

`/lichesstofidehistory`	`lichessId: string`
	`lichessType: string`
	`fideType: string`
	`limit: number`
		GET	Get a list of 10 fide players with a similar rating history to the provided lichess user, for the given lichess rating type (blitz, bullet, classical, rapid) and given Fide rating type. Score is calculated as the average difference between ratings in the same month/year, taking into account the general 400 point offset between Lichess and Fide ratings. Variance and number of matching points used for the calculation are also returned, along with the players' name to make the result easier for the user to remember.	`lichessId: string`
				`score: number`
				`variance: number`
				`numPoints: number`
				`firstName: string`
				`lastName:string`
				`rating: number`

`/fidetolichesshistory`	`fideId: string`
	`lichessType: string`
	`fideType: string`
		GET	Similar to the above, but from fide to lichess.	`lichessId: string`
				`score: number`
				`variance: number`
				`numPoints: number`

*all response objects are returned in the following json format: `{ results: [<objects>] }`

Queries

Select Fide players with similar rating history to a given Lichess player and output their id, names, and similarity score, as well as information about how the score was calculated.

This query was used to generate the list of users with similar rating histories in the Players page.

```
WITH MergedHistory AS (
    SELECT lichessId, L.month AS month, L.year AS year, L.rating AS lichessRating, F.rating AS fideRating, fideId
    FROM (SELECT * FROM LichessHistory WHERE lichessId = "${lichessId}" AND type = "${lichessType}") L
             JOIN (SELECT * FROM FideHistory WHERE type = "${fideType}") F
                  ON L.year = F.year AND L.month = F.month
)
SELECT id, score, variance, numPoints, firstName, lastName, classicalRating AS rating
FROM (
         SELECT fideId                                                AS id,
                ABS(AVG(fideRating - lichessRating) + ${threshold}) AS score,
                VARIANCE(fideRating - lichessRating + ${threshold}) AS variance,
                COUNT(*)                                             AS numPoints
         FROM MergedHistory
         GROUP BY fideId
     ) Scores
         NATURAL JOIN (
    SELECT id, firstName, lastName, classicalRating
```

```
    FROM FidePlayers
) Players
ORDER BY score, numPoints DESC, variance DESC
LIMIT ${limit || 100}
```

Given an `id`, find all openings that player `id` has played. Then, find all the players that play an opening that `id` plays. Take into account whether both players have played the opening as White or Black.

This query is used in the Player Games page to help find players who play similar openings.

```
WITH black_games AS (
          SELECT eco
          FROM LichessGames
          WHERE blackId = '${id}'
      ),
      white_games AS (
          SELECT eco
          FROM LichessGames
          WHERE whiteId = '${id}'
      ),
      combine AS (
          SELECT eco
          FROM white_games
          UNION
          SELECT eco
          FROM black_games
      ),
      other_users_black AS (
          SELECT username
          FROM LichessPlayers lp JOIN LichessGames LG on lp.id = LG.blackId
          WHERE LG.eco IN (SELECT * FROM combine)
      ),
      other_users_white AS (
          SELECT username
          FROM LichessPlayers lp JOIN LichessGames LG on lp.id = LG.whiteId
          WHERE LG.eco IN (SELECT * FROM combine)
      ),
      combine_others AS (
          SELECT username
          FROM other_users_black
          UNION
          SELECT username
          FROM other_users_white
      )
      SELECT *
      FROM combine_others;
```

Find some amateur level games (sum of elo is < 2400), and determine all the players' playtime. Find all the unverified players with elo less than `elo` (with similar playtime). Sort the users by playtime, elo.

This query was used in the Top Games page.

```
WITH poor_games AS (
          SELECT whiteId, blackId
          FROM LichessGames
          WHERE (whiteRating + blackRating) <= 2400
        ),
        Playtime_poor_games_white AS (
          SELECT totalPlayTime, fideRating, lp.id
          FROM poor_games pg JOIN LichessPlayers lp ON pg.whiteId = lp.id
        ),
        Playtime_poor_games_black AS (
          SELECT totalPlayTime, fideRating, lp.id
          FROM poor_games pg JOIN LichessPlayers lp ON pg.blackId = lp.id
        ),
        Combine AS (
          SELECT *
          FROM Playtime_poor_games_white
          UNION
          SELECT *
          FROM Playtime_poor_games_black
        )
        SELECT *
        FROM LichessPlayers lp JOIN Combine c ON c.id = lp.id
```

```
        WHERE lp.isVerified = 0 AND lp.fideRating < ${elo}
        ORDER BY lp.totalPlayTime, lp.fideRating;
```

Find all top level games (where the sum of both players' elo is >= param elo). Find all the players in these games, and get all the players' countries. Then find all the users in these countries. Sort the users by country and elo.

This query was used in the Amateur Games page.

```
WITH top_games AS (
        SELECT whiteId, blackId
        FROM LichessGames
        WHERE (whiteRating + blackRating) >= ${elo}
      ),
      Countries_top_games_white AS (
        SELECT country
        FROM top_games tg JOIN LichessPlayers lp ON tg.whiteId = lp.id
      ),
      Countries_top_games_black AS (
        SELECT country
        FROM top_games tg JOIN LichessPlayers lp ON tg.blackId = lp.id
      ),
      Countries AS (
        SELECT *
        FROM Countries_top_games_white
        UNION
        SELECT *
        FROM Countries_top_games_black
      )
      SELECT *
      FROM LichessPlayers lp
      WHERE lp.country IN (SELECT * FROM Countries)
      ORDER BY lp.fideRating DESC;
```

Performance Evaluation

Query	Original	Optimized	With Caching
1. Get 100 users with similar rating history	36.3 s	30.9 s	.007 s
2. Find all top level games' players → Return users from these players' countries	73.6 s	67.5 s	.823 s
3. Find poor level games and players' playtime → Return unverified players with similar playtime	62.5 s	58.8 s	.011 s
4. Get all players that play an opening 'id' plays.	58.6 s	55.1 s	.304 s

First, we made sure that our relations were efficiently created with all attributes being necessary and unrepeated, as well as creating the schema in BCNF.

For query optimization, we utilized four strategies: creating indexes, connection pooling, restructuring queries, and caching. 6 indexes were created to address the needs of our complex queries, which utilize information across all 7 tables.

```
CREATE INDEX NameIndex ON FidePlayers (lastName, firstName);
CREATE INDEX lichess_rating ON LichessHistory (rating);
CREATE INDEX fide_rating ON FideHistory (rating);
CREATE INDEX lichess_username ON LichessPlayers (username);
CREATE INDEX lichess_game ON LichessGames (eco);
CREATE INDEX fide_game ON FideGames (eco);
```

Next, we implemented connection pooling in order to ensure that a new database connection was not being made each time the data is asked for; rather, the same connection is used again, allowing faster access to the data and other components.

Moreover, we structured our queries through moving projections down, narrowing selections, rearranging joins, and employing views. For example, in complex query 2 (finding top level games and then users), we simplified the top_games view through changing two nested “select” clauses into one. Furthermore, a few unnecessary attributes that were being projected were removed, keeping only the ones used in further parts of the query. Through these steps, the query time decreased by around 6.1 seconds. Similarly, other queries were optimized around 3-6 seconds through the restructuring techniques mentioned above.

We hoped to optimize our complex queries more, wanting them to take around 1 second. However, given the size and complexity of our relations, we realized that it would be difficult to optimize the queries more with the current techniques presented. Thus, we looked into caching and created maps to store the (key, value) pairs of information for our queries. After implementing caching, complex query 4 (get players that play an opening another player plays) took less than 1 second after searching up the same player id for the second time (id is the key). Similarly, other queries took less than 1 second after looking up the same information previously asked for.

Through these techniques, we were able to optimize our queries and better the response times for users.

Technical Challenges

Finding and cleaning the data was quite complicated. The Lichess API rate limiting bottlenecked our ability to fetch users. To deal with rate-limiting, we batched user requests in groups of 300 (the maximum batch size), checked in the DB for users we already added to avoid including them in the API request and thus using up our rate limit, and ran our scraper on a homemade distributed environment, aka. multiple group members’ laptops. On the Fide side, we had to make optimizations with Puppeteer, eg. refreshing the headless browser connection before it timed out. We even found a bug with the open source fide-ratings-scraper tool we used and made an issue on github for it, which unfortunately went unaddressed, so we just had to code around it.

Extra Credit

For extra credit, we implemented three tasks:

1. Used NoSQL in addition to SQL.

We used MongoDB to store documents about users. Users can register accounts and login/logout. Their usernames and passwords are stored on MongoDB as strings. Users can also “favorite” chess players by first searching the username of a chess player and clicking the favorite button. The users’ liked players are represented as an array in MongoDB. An example is shown below:

2. Deployment

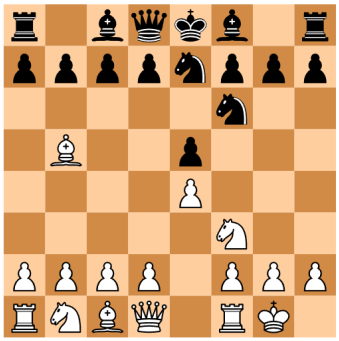
We used Netlify (https://www.netlify.com/) to host our frontend web user interface.

3. Anything cool you guys might have missed

We implemented our own UI for our chessboard since the most commonly used library for chess board visualization was deprecated. The user can see all the moves in a particular game by hitting the next and previous buttons. An example is shown below:

-Tristan- (White 1474) vs. Castor1990 (Black 1411)

Opening: Ruy Lopez: Berlin Defense
Result: 1-0
Termination: Normal
Time Control: 300+0



Reset < >

Appendix I. Sample PGN Data

[Event "London-B"]
[Site "London ENG"]
[Date "1862.??.??"]
[Round "?"]
[White "Steinitz, Wilhelm"]
[Black "Wilson, J."]
[Result "1-0"]
[ECO "C39"]

1. e4 e5 2. f4 exf4 3. Nf3 g5 4. h4 g4 5. Ne5 Nf6 6. Bc4 d5 7. exd5 Bd6 8. d4 Nh5 9. Bb5+ Kf8 10. O-O Qxh4 11. Bxf4 g3 12. Bh6+ Kg8 13. Rf3 Qh2+ 14. Kf1 Qh1+ 15. Ke2 Qxg2+ 16. Kd3 f6 17. Bc4 fxe5 18. Rf8+ Bxf8 19. d6+ Be6 20. Bxe6# 1-0