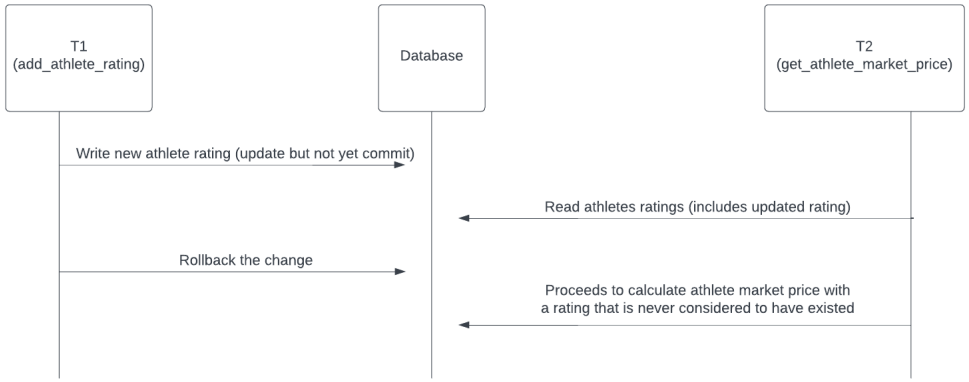
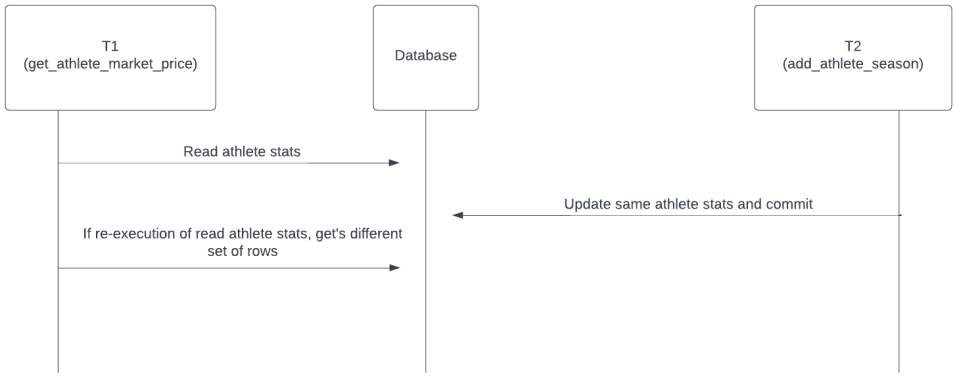
For the top 3 most complex interactions of transactions in your database, write up how without any concurrency control what phenomenon they would encounter as described in class: https://observablehq.com/@calpoly-pierce/isolation-levelsLinks to an external site.. Make a sequence diagram for each case. Lastly, what will you do to ensure isolation of your transactions and why is that the appropriate case. If you believe your transactions don't have any such cases, your transactions aren't complex enough for our purpose (or you aren't understanding what issues are occurring). Note, this can be both how a particular transaction definition interacts with other transaction definitions, but also how a transaction definition interacts with other concurrent instances of itself.3 complex transactions

**<u>Concurrency concerns for the top three most complex transactions:</u>**

1. Rate athlete (get_athlete_market_price())
    - Basic Rate Athlete flow:
        - Read athlete's stats
        - Read athlete's ratings
        - Calculate prediction based on the above
    - Vulnerable read phenomenon:
        - Dirty Read: If multiple transactions are reading the same athlete's market price simultaneously, and one transaction modifies the athlete's data (e.g., updates the stats or ratings) before committing the changes, another transaction may read the modified data before it is finalized. This can lead to inconsistent or incorrect market price calculation.



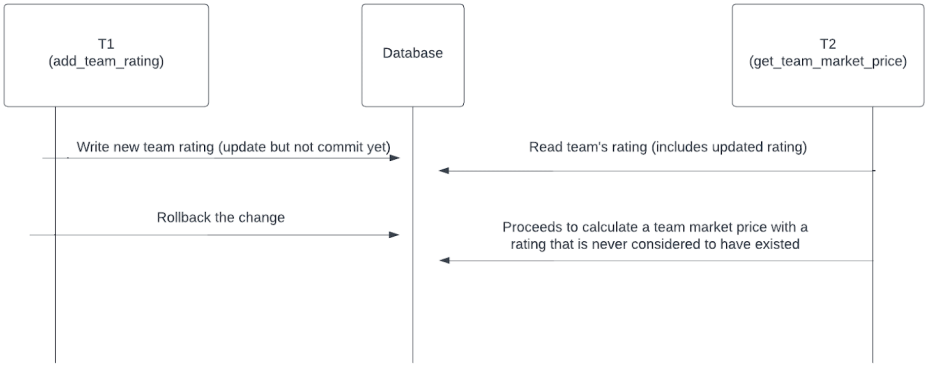

        - Phantom Read: If multiple transactions are retrieving the athlete's predictions and ratings simultaneously, and another transaction inserts or deletes data that matches the conditions used for retrieval, the result set may change between the two queries. (diagram below also applies to non-repeatable read)



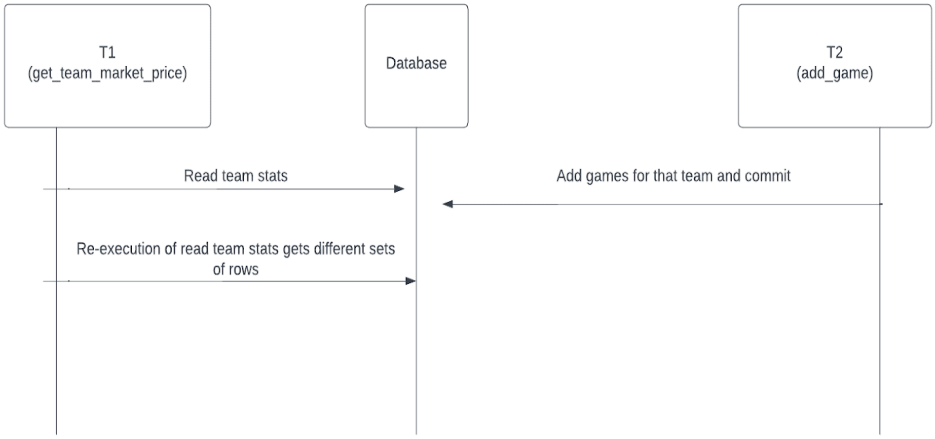

- Notes:
    - Not vulnerable to
        - Lost update
        - Read skew
        - Write skew
    - Because the rate athlete endpoint is meant to output values which reflect changes in an athlete's stats and/or ratings, the non-repeatable read and phantom read cases are not necessarily undesirable. The only issue above which should be avoided is the dirty read, which would do calculations based on data that is not supposed to exist.

2. Rate team (get_team_market_price())
    - Basic Rate Team flow:
        - Read team's stats
        - Read team's ratings
        - Calculate prediction based on the above
    - Vulnerable read phenomena:
        - Dirty Read: If multiple transactions are reading the same team's market price simultaneously, and one transaction modifies the team's data (e.g., updates the stats or ratings) before committing the changes, another transaction may read the modified data before it is finalized. This can lead

to inconsistent or incorrect market price calculation.



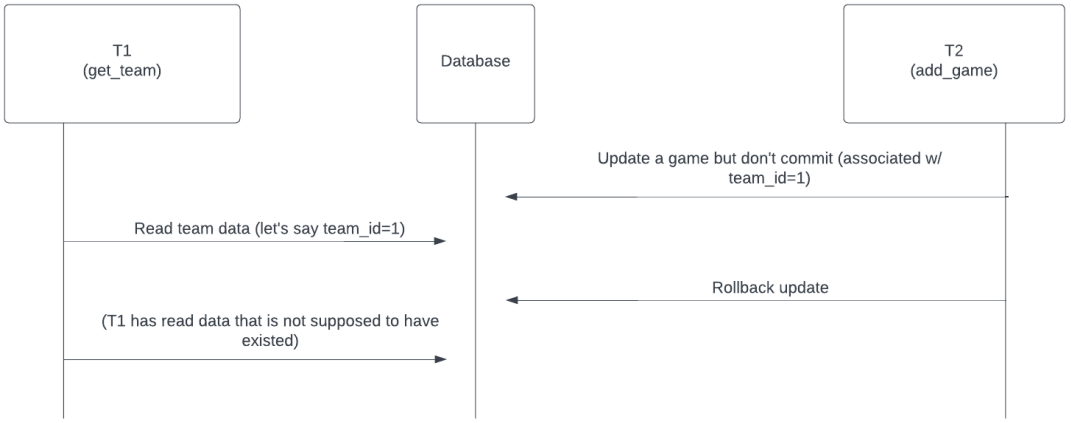

- Phantom Read: If multiple transactions are retrieving the team's predictions and ratings simultaneously, and another transaction inserts or deletes data that matches the conditions used for retrieval, the result set may change between the two queries. (diagram below also applies to non-repeatable read)



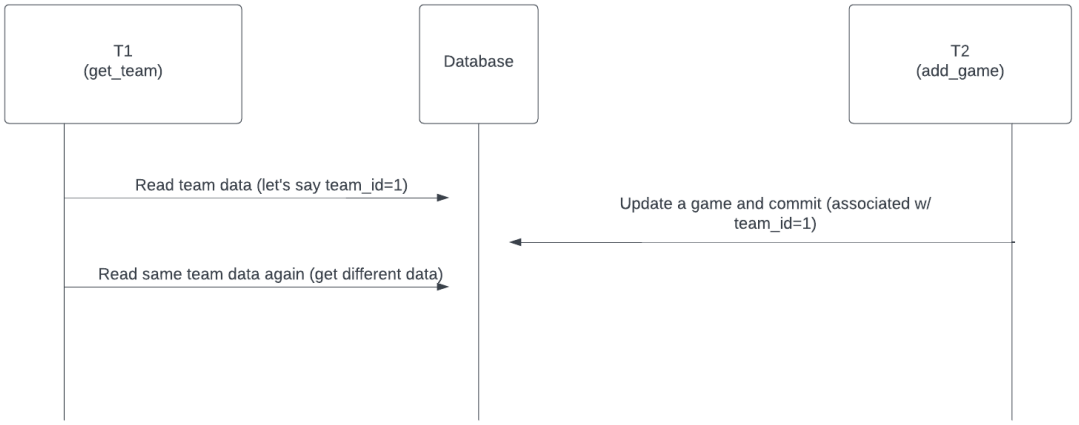

- Notes:
  - Not vulnerable to
    - Lost update
    - Read skew
    - Write skew
  - Because the rate team endpoint is meant to output values which reflect changes in an team's stats and/or ratings, the non-repeatable read and phantom read cases are not necessarily undesirable. The only issue above which should be avoided is the dirty read, which would do calculations based on data that is not supposed to exist.

3. Get Team

- Basic Get Team flow:
    - select the team
    - select all games for the team
    - do calculations to return team-year stats
- Complications: can add to games
- Vulnerable to:
    - Dirty Read




    - Non-repeatable read (and phantom read)




- Not vulnerable to:
    - Lost update
    - Read skew
    - Write skew

**<u>Chosen concurrency control and why:</u>**

We chose to use optimistic concurrency control, specifically snapshot isolation. This is for several reasons.

First, because it was set with one flag in our database connection, it is the most simple to implement. This simplicity is good for our code readability and functionality. If we were to have implemented optimistic concurrency control with timestamp data and manual checks in the code, there would be more code required in all the endpoints, and more room for bugs.

Also since we are using Postgres as our database, snapshot isolation makes more sense as postgres depends more on snapshot isolation to guarantee isolation rather than something like locks under pessimistic concurrency control, which is something databases such as MySQL depend more on to guarantee isolation.

Another reason is because this form eliminates the risk of read phenomena like dirty read, non-repeatable read, and phantom read, which are the main concerns for our code described above.