CS 411 Project Track 1 Final Report

Not Just Another Race Timer - nJART

Links

Project Repository: https://github.com/john-judge/not-just-another-race-timer.git
Project Site: https://jmjudge2.web.illinois.edu/not-just-another-race-timer/rankings

Project Overview

- Accomplishments: This web page would help long range track events like marathons that are lower budgeted have more accurate results. The concept is when spectators uses their mobile device to select a long-distance athlete's ID while standing near that race competitor during the event. The time, user, and location are recorded, providing the athlete with fine-grained, crowd-sourced timing and a list of supporters and fans who were interested in helping time his/her race performance.
- Usefulness: For low-budget or understaffed athletic events, this application can replace expensive chip or camera timing and aid in volunteer staff recruitment or even crowd-sourcing hand-timed results. It also provides a social and supportive way of connecting spectators and athletes over their shared interest in the sport, because there is a low barrier to virtual social interaction on this platform.
- Data: 3 sources - Event Manager, Registrations, and Spectators. Event manager inputs geolocation of course map waypoints. Registration feature for users to register as athletes. Spectators report their

  1) UNIX / UTC Timestamp: Server UTC instant or calling Date.now() in the client's browser
  2) Geolocation: Browser HTML5 Geolocation
  3) Selected athlete: Category select or restricted text input
  4) Comment: Free text, to be shown in results and Rankings.

  The data we used were generated by us, the users, actually walking around recording the time and location. We created multiple accounts, events and teams to test the features and functions.

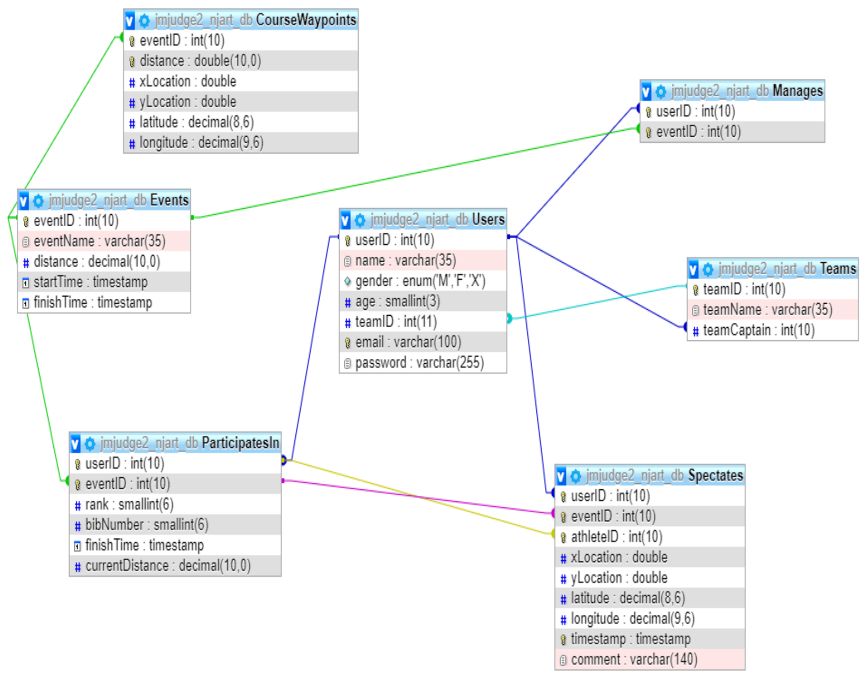

Figure 1: Database UML diagram

Project Functionality: Workflow and Dataflow

Basic Functions

1. Each event is tracked by an event ID associated with distance and start timestamp (UTC).

2. The Users table userID persists across the session and authenticates based on web-token. Database stores encrypted passwords.

3. For each event, data will be gathered via a registration form, placing user ID into the joining table Participates In which links the user ID to an event ID. This represents the roster for each event.

4. From Home, user can view registration for all participants of an event, and unregister themselves (and only themselves) from events in which they previously registered.

5. Athletes will have a foreign key team ID to table Teams which records team name and score (score is the sum of ranks of the top five team members, if the team has at least five team members. Clearly, a lower score is better).

6. The Spectated table is a relates foreign keys User ID, Athlete ID, event ID and records timestamp, geographic location, current rank*, and comment. These are entered or logically generated at entry by the spectating user.

7. Users can query and filter results at any time, including when the race is in progress, including athlete results and team results and rankings. Rankings contains the progress,

in meters, of the athlete, which is computed automatically by the before-insert trigger for `Spectates`

Advanced Functions:

1. Storing the course map and using it for integrity constraints.

   (a) The table Course Waypoints will have key {eventID, distance} and record the latitude and longitude of the location, as well as the Cartesian approximation of the location relative to the start line (see Technical Challenges section).

   (b) Any number of course waypoints may refer to a single eventID, which are ordered by distance (which is distance along the course path from the start line). This distance is accumulated by the CourseWaypoints before-insert trigger.

   (c) Events are not considered "live" unless they have at least one waypoint, have been started by the event manager, and have not been ended by the event manager. Only "live" events show up in the GUI search for events to spectate.

   (d) Geospatial constraints. CourseWaypoints must be 100 meters apart from each other. This design improves the efficiency and simplicity of the Spectates before-insert calculation by limiting the number of waypoints that can be within 100 m of a submitted spectates location.

   (e) Team rankings. Only teams with at least five members are

   (f) Entry method for course waypoints forces the manager to walk the course physically and document waypoints via gelocation. This help to encourage managers to choose responsible and safe courses for their races.

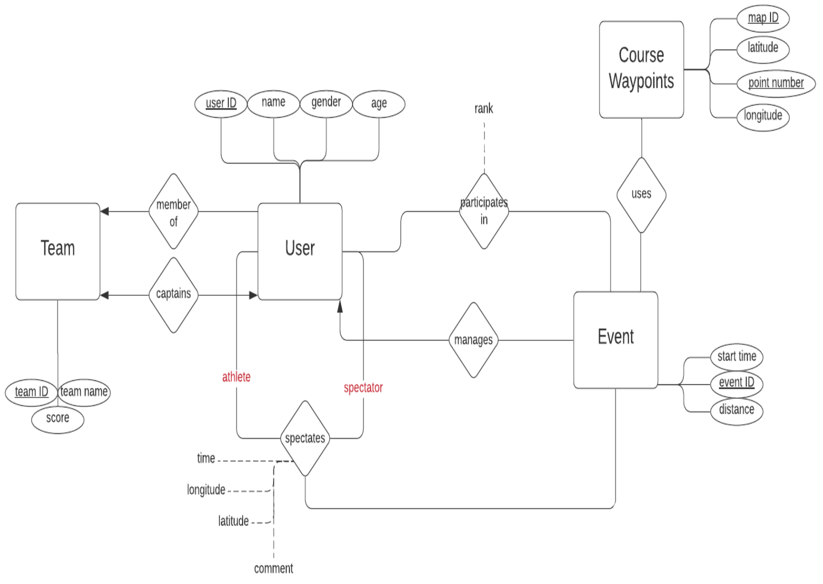

Figure 2: The original ER diagram. The attributes have been enhanced since (see the UML diagram), but this diagram serves as a useful high-level abstraction of the dataflow.

SQL code snippet

Here is an example of a query from the web server, in `app.js`:

```
var athleteFilter = "%" + req.body.athleteName + "%";

var subQueryTop5 = 'SELECT *, ' +
                   '@team_rank := IF(@curr_team = teamID, @team_rank + 1, 1) AS team_rank, ' +
                   '@curr_team := teamID ' +
                   'FROM ParticipatesIn NATURAL JOIN Users NATURAL JOIN Events ' +
                   'ORDER BY teamID, rank';



var query = "SELECT ir.teamID, t.teamName as Team, u.name as Captain, SUM(ir.team_rank) as Score, AVG(ir.finishTime) AS AverageTime FROM (" + subQueryTop5 + ") ir NATURAL JOIN (" + eligibleTeamsSize5Min +") t LEFT JOIN Users u ON u.userID = t.teamCaptain WHERE ir.team_rank <= 5 AND eventID = ? GROUP BY ir.teamID";

connection.query(query,[req.body.eventID], (err, rows) => {
    connection.release();
    if(err) { console.log(err); return; }
    resp.json(rows);
});
```

Here is an example of a SQL trigger BEFORE INSERT into the `Spectated` table, which validates our geometric constraints, and updates the user ranks and progress in the event.

```
BEGIN

DECLARE first_distance decimal(10,3);
DECLARE first_xLocation decimal(10,3);
DECLARE first_yLocation decimal(10,3);
DECLARE second_distance decimal(10,3);
DECLARE second_xLocation decimal(10,3);
DECLARE second_yLocation decimal(10,3);
DECLARE x_distance decimal(10,3);
DECLARE y_distance decimal(10,3);
DECLARE line_distance decimal(10,3);
DECLARE part decimal(10,3);
DECLARE perpendicular_x decimal(10,3);
DECLARE perpendicular_y decimal(10,3);
DECLARE distance_A decimal(10,3);
DECLARE distance_B decimal(10,3);
DECLARE cur_length decimal(10,3);
DECLARE closest_length decimal(10,3) DEFAULT 9999999.999;
DECLARE runner_distance decimal(10,3);
DECLARE finished INT DEFAULT 0;
DECLARE r INT;

DECLARE waypoints_cursor CURSOR FOR (SELECT distance, xLocation, yLocation FROM CourseWaypoints WHERE eventID = NEW.eventID ORDER BY distance);

DECLARE CONTINUE HANDLER
FOR NOT FOUND SET finished = 1;

OPEN waypoints_cursor;

FETCH NEXT FROM waypoints_cursor INTO first_distance, first_xLocation,
    first_yLocation;
FETCH NEXT FROM waypoints_cursor INTO second_distance, second_xLocation,
    second_yLocation;

WHILE finished = 0 DO

SET x_distance = second_xLocation - first_xLocation;
SET y_distance = second_yLocation - first_xLocation;
SET line_distance = x_distance * x_distance + y_distance * y_distance;
SET part = ((NEW.xLocation - first_xLocation)*x_distance+(NEW.yLocation -
    first_yLocation)*y_distance) / line_distance;
SET perpendicular_x = first_xLocation + part * x_distance;
SET perpendicular_y = first_yLocation + part * y_distance;

IF (((perpendicular_x > first_xLocation) AND (perpendicular_x >
    second_xLocation)) OR ((perpendicular_x < first_xLocation) AND (
    perpendicular_x < second_xLocation))) THEN

     SET distance_A = sqrt(POW(perpendicular_x - first_xLocation, 2) + POW(
    perpendicular_y - first_yLocation, 2));
     SET distance_B = sqrt(POW(perpendicular_x - second_xLocation, 2) + POW(
    perpendicular_y - second_yLocation, 2));
     IF (distance_A <= distance_B) THEN
       SET perpendicular_x = first_xLocation;
         SET perpendicular_y = first_yLocation;
     ELSE
       SET perpendicular_x = first_xLocation;
         SET perpendicular_y = first_yLocation;
     END IF;
END IF;

SET cur_length = sqrt(POW(perpendicular_x - NEW.xLocation, 2) + POW(
    perpendicular_y - NEW.yLocation, 2));

IF cur_length < closest_length THEN
      SET closest_length = cur_length;
      SET runner_distance = first_distance + sqrt(POW(perpendicular_x -
    first_xLocation, 2) + POW(perpendicular_y - first_yLocation, 2));
END IF;

SET first_distance = second_distance;
SET first_xLocation = second_xLocation;
SET first_yLocation = second_yLocation;

FETCH NEXT FROM waypoints_cursor INTO second_distance, second_xLocation,
    second_yLocation;

```

```
END WHILE;

CLOSE waypoints_cursor;

IF closest_length <= 100 THEN

UPDATE ParticipatesIn SET currentDistance = runner_distance WHERE userID = NEW
    .athleteID AND eventID = NEW.eventID;

SET @r = 0;
UPDATE ParticipatesIn SET rank = @r:= (@r+1) WHERE eventID = NEW.eventID ORDER
    BY currentDistance DESC;

ELSE

SIGNAL SQLSTATE '45000'
SET MESSAGE_TEXT = 'Too far from course';

END IF;

END
```

Dataflow Explanation

Explain Advanced Function

As we will describe in Technical Challenges, we managed to create a geospatial measure of spherical distances that is a robust approximation for short distances. The client code produced local displacements in meters that could be treated as reasonable Cartesian coordinates.

Even provided the approximately Cartesian `xLocation` and `yLocation`, the database trigger for insertions into `Spectates` needed to surmount additional formidable technical challenge in projecting a user's location onto a distance along the course, which comes in the form of a sequence of coordinates. This is the major purpose of the second code snippet shown in the SQL code snippet section.

To facilitate this, the primary key of `CourseWaypoints` included a `distance` attribute that not only provided ordinality to the collection of waypoints for an event, but also provided efficient accumulated progress distances for each athlete spectate entry, provided that the trigger could first localize the event to the nearest line segment of the course.

If the projected progress of the athlete is within 100 m of the total course distance, the inserted timestamp is copied into the `finishTime` of the athlete's `ParticipatesIn` table row. This attribute is not to be confused with the `endTime` of the `Events` table.[1]

Towards a similar challenge, the event manager can end an event, which populates the finish time of the event and a linearly extrapolates times for runners who aren't recorded as having finished but have completed at least half of the race. This was written in the form of an update trigger on `Events` on the `finishTime` attribute.

[1]Originally, these names were changed to coincide at the last minute, which broke all joins between Events and ParticipatesIn during our demo.

Technical Challenge

Map Projections and Haversine Distance

I wanted the backend to be able to operate on locally-Cartesian coordinates, but no map projection exists that preserves the transitivity of distances. This is due to the fact that for a surface of a sphere projected onto a finite plane, there will always be a pair of segments that are stretched at different factors.

Moreover, the approximation of flat earth breaks down widely depending on latitude. The longitudinal distances need to be adjusted based on latitude. To correct this, I adapted the Haversine distance formula to a Haversine *displacement* formula, shown below. The Haversine displacement was stored for each point relative to the start line (the first Waypoint) for the event, and stored in the databased as `xLocation` and `yLocation` in the `CourseWaypoints` and `Spectates` tables, and used for all distances and geometrical constraints computing in the DB. Meanwhile, `latitude` and `longitude` were stored and read back for map visualization only.

You may notice that Haversine displacements aren't technically Cartesian across any two points where neither is the start line. However, over short distances, the Cartesian Pythagorean theorem is a good approximation. Our correction avoids our computations from breaking down from the equator to the poles, but assumes local races. If you have a race covering a hemisphere, approximation issues may get more serious. We would need to adopt spherical coordinate integrity constraints if this were the case.

```
function haversineDisplacement(coord1, coord2) {
    // given two locations in form {lat, lng} in degrees, compute the distance
     in meters as the crow flies {x,y}, such that the distance is Haversine (
    great circle distance)
    // transitivity approximation holds well for both equator and poles

    var radian = Math.PI / 180;
    var R = 6378137; // radius of Earth in meters

    // separation of latitude is locally approx. Cartesian
    var dy = (coord2.lat - coord1.lat) * radian;
    // but longitudinal sep does NOT approximate Cartesian locally.
    var dx = (coord2.lng - coord1.lng) * radian;

    var sinLon = Math.sin(dx / 2);
    var a = Math.cos(coord2.lat * radian) * sinLon * sinLon;
    var c = Math.atan( Math.sqrt(a), Math.sqrt(1-a) );
    dx = 2 * c * c;

    return {x: dx * R, y: dy * R};
}
```

To acquire the data for this conversion, the web server and client needed to cache start line coordinates and craft efficient queries to reduce the number of database connections. Client memory and HTML form input elements were useful places to cache such values and the results of these calculations. In addition, constraints on the availability of workflow jQuery submissions needed to be imposed to avoid reaching invalid states, i.e. attempting to record a location without having the start line already cached. This was likely the main technical challenge of the client-side scripting.

The SQL trigger for update on `Events.endTime`.

```sql
BEGIN

DECLARE cur_user INT;
DECLARE user_distance decimal(20,3);
DECLARE user_finish timestamp;
DECLARE currentSeconds decimal(20,3);
DECLARE extrapolatedSeconds decimal(20,3);
DECLARE new_finishTime timestamp;
DECLARE check_finish timestamp;
DECLARE finished INT DEFAULT 0;

DECLARE participates_cursor CURSOR FOR (SELECT userID, currentDistance,
    endTime FROM ParticipatesIn WHERE eventID = NEW.eventID);

DECLARE CONTINUE HANDLER
FOR NOT FOUND SET finished = 1;

IF NEW.endTime IS NOT NULL AND OLD.endTime IS NULL THEN

OPEN participates_cursor;

FETCH NEXT FROM participates_cursor INTO cur_user, user_distance, check_finish
    ;

WHILE finished = 0 DO

IF (user_distance / NEW.distance) > .66 AND check_finish IS NOT NULL THEN

SELECT MAX(timestamp) INTO user_finish FROM Spectates WHERE athleteID =
    cur_usr AND eventID = NEW.eventID;

SET currentSeconds = UNIX_TIMESTAMP(user_finish) - UNIX_TIMESTAMP(NEW.
    startTime);

SET extrapolatedSeconds = currentSeconds * (NEW.distance / user_distance);

SET new_finishTime = TIMESTAMPADD(SECOND, extrapolatedSeconds, NEW.startTime);

UPDATE ParticipatesIn SET finishTime = new_finishTime WHERE userID = cur_user
    AND eventID = NEW.eventID;

UPDATE ParticipatesIn SET currentDistance = NEW.distance WHERE userID =
    cur_user AND eventID = NEW.eventID;

END IF;

FETCH NEXT FROM participates_cursor INTO cur_user, user_distance, check_finish
    ;

END WHILE;
```

```

CLOSE participates_cursor;

END IF;

END
```

Other Technical Challenges

The following minor technical challenges are mentioned only in passing:

1. Real-time Ranking for Eligible Teams
2. Imposing Geospatial and Temporal Constraints
3. Data formatting: Javascript has poor support for timezone to UTC conversions, and MySQL stores ISO date formats.
4. Rounding Issues from MySQL datatypes for GPS Coordinates. Decimal datatypes are suitable for the latitude and longitude due to the fixed point precision but high sensitive. Real datatypes are suitable for the more flexible but robust distance calculations

Final Division of Labor

Task	Allen Yang	John Judge	Nayun "Cloud" Dong	Tyler Lincheck
Report	X	X		
MySQL triggers		X	X	X
Database Design		X		X
Workflow Design		X		
MySQL Stored Procedures			X	X
Web Server / SQL		X		
Static HTML	X	X		
CSS styling	X			
Client scripting		X		
Data Collection	X	X	X	X
Stage 1-3		X		
Demo Video	X			