IronTrack Training Database

CSDS 341 Final Project Report

Camden Larson

December 8, 2025

Final Project Report: IronTrack Training Database

1. Application Background

- IronTrack aims to allow athletes and coaches to track workouts, gear utilization, and more for triathletes. It provides a central dashboard with weekly distance and duration metrics as well as workout logging functionality. It also lets athletes track their gear usage by linking different shoes, bikes, etc. to workouts.
- Motivation: I wanted to provide users with centralized logging of swim, bike, and run workouts in one visualization, with the ability to alter time periods and workout type as well as see gear usage over time.
- Existing fitness applications like Strava, TrainingPeaks, Garmin Connect, etc. hold a lot of this data but sometimes make it difficult for users to see segmented information and track their progress on different sports.
- Unique design goals of IronTrack:
  - Athlete-centered full data ownership.
  - Flexible multi-sport schema with unit conversion views.
  - Gear usage tracking and automatic wear estimation.
  - Fine-grained workout metadata for analysis and trend exploration.
- Example use cases:
  - Weekly training load summaries.
  - Gear mileage tracking for running shoes and bikes.
  - Comparing performance across locations and conditions.
  - Swim, bike, and run discipline-specific analysis.

2. Data Description and Constraints

The IronTrack database is designed to model multi-sport endurance training data with an emphasis on accuracy, extensibility, and analytical flexibility. The schema captures user accounts, workout metadata, sport-specific performance metrics, and the relationship between workouts and physical gear. All data is stored in canonical units to ensure consistency, and multiple integrity constraints are enforced to guarantee correctness.

- **Major Entities**
  - **Users**: Represents athletes using the system. Each user has a unique username and email address. The table stores basic account metadata, including password hashes and account creation timestamps.
  - **Workout_Types**: Defines the three supported disciplines: *swim*, *bike*, and *run*. This table allows the schema to remain extensible (e.g., new workout types could be added later) while ensuring that all workouts reference a valid, enumerated sport.
  - **Locations**: Stores information about where workouts occur. The `location_type` attribute captures the environment of the workout, restricted to valid categories such as *pool*, *fresh_water*, *road*, *trail*, and others. Optional city and state fields support more detailed geographic analysis.
  - **Gear**: Represents physical equipment used by athletes, such as running shoes, bikes, wetsuits, and goggles. Gear records are associated with the user who owns them, and they may optionally track brand, model, purchase date, and retirement status. The table supports analysis of gear usage and wear over time.
  - **Workouts**: The central fact table capturing individual training sessions. Each workout references its user, workout type, and (optionally) location. To support precise and consistent metrics, all quantitative measurements are stored in canonical units: meters for distance, seconds for time, meters for elevation gain, kilocalories for energy expenditure, and beats per minute for heart rate. Additional fields store cadence (steps/min or rpm), average power (for cycling), subjective effort level, and free-form notes.
  - **Workout_Gear**: A linking table enabling a many-to-many relationship between workouts and pieces of gear. This allows a workout to involve multiple gear items (e.g., cycling shoes + bike) and allows a piece of gear to accumulate distance across many workouts. The primary key is a composite of `workout_id` and `gear_id`.
- **Canonical Units for Standardization**

  All raw metrics are stored in standardized SI-based units to avoid inconsistencies across workout types:

  - Distance: meters (`distance_m`)

  - Duration: seconds (`duration_seconds`)
  - Elevation gain: meters (`elevation_gain_m`)
  - Heart rate: beats per minute (`avg_heart_rate_bpm`)
  - Energy expenditure: kilocalories (`calories_kcal`)

  This design allows accurate cross-discipline comparison and simplifies unit conversions inside derived views.

- **Sport-Specific Derived Views**

  Because swimming, cycling, and running use different common units and pace metrics, the system defines three read-only views:

  - **run_workouts**: Converts distance to miles and computes pace in seconds per mile.
  - **bike_workouts**: Converts distance to miles and computes speed in miles per hour.
  - **swim_workouts**: Converts distance to yards and computes pace per 100 yards.

  These views preserve the normalized structure of the database while providing athletes with sport-appropriate analytics.

- **Integrity Constraints**

  The database enforces several classes of constraints to maintain correctness and semantic accuracy:

  - **Primary Keys**: Every table defines a unique primary key (`user_id`, `workout_id`, `gear_id`, etc.), ensuring that all tuples can be uniquely identified.
  - **Foreign Keys with Cascading Deletes**:
    - Deleting a user automatically deletes their gear and workouts.
    - Deleting a workout automatically deletes its associated `Workout_Gear` records.

    This preserves referential integrity and prevents orphaned records.
  - **Check Constraints**: These protect against invalid or physically impossible values:
    - `duration_seconds > 0`
    - `distance_m` $\geq$ `0`, `elevation_gain_m` $\geq$ `0`
    - `avg_heart_rate_bpm BETWEEN 30 AND 250`
    - `effort_level BETWEEN 1 AND 10`
    - Valid gear types (e.g., *shoe*, *bike*, *wetsuit*, etc.)
    - Valid location types (e.g., *pool*, *trail*, *indoor*, etc.)
  - **Unique Constraints**:
    - `username` and `email` must be unique across all users.

  * Workout type names (e.g., “run”) must be unique to avoid redundancy.
- **Enumerated Categories**: Several attributes enforce restricted sets of allowed values through `CHECK` constraints:
  * `location_type`: pool, fresh_water, salt_water, road, trail, track, indoor, other.
  * `gear_type`: shoe, bike, wetsuit, goggles, other.

  These restrictions ensure compatibility with the application logic and promote data consistency.

3. ER Diagram

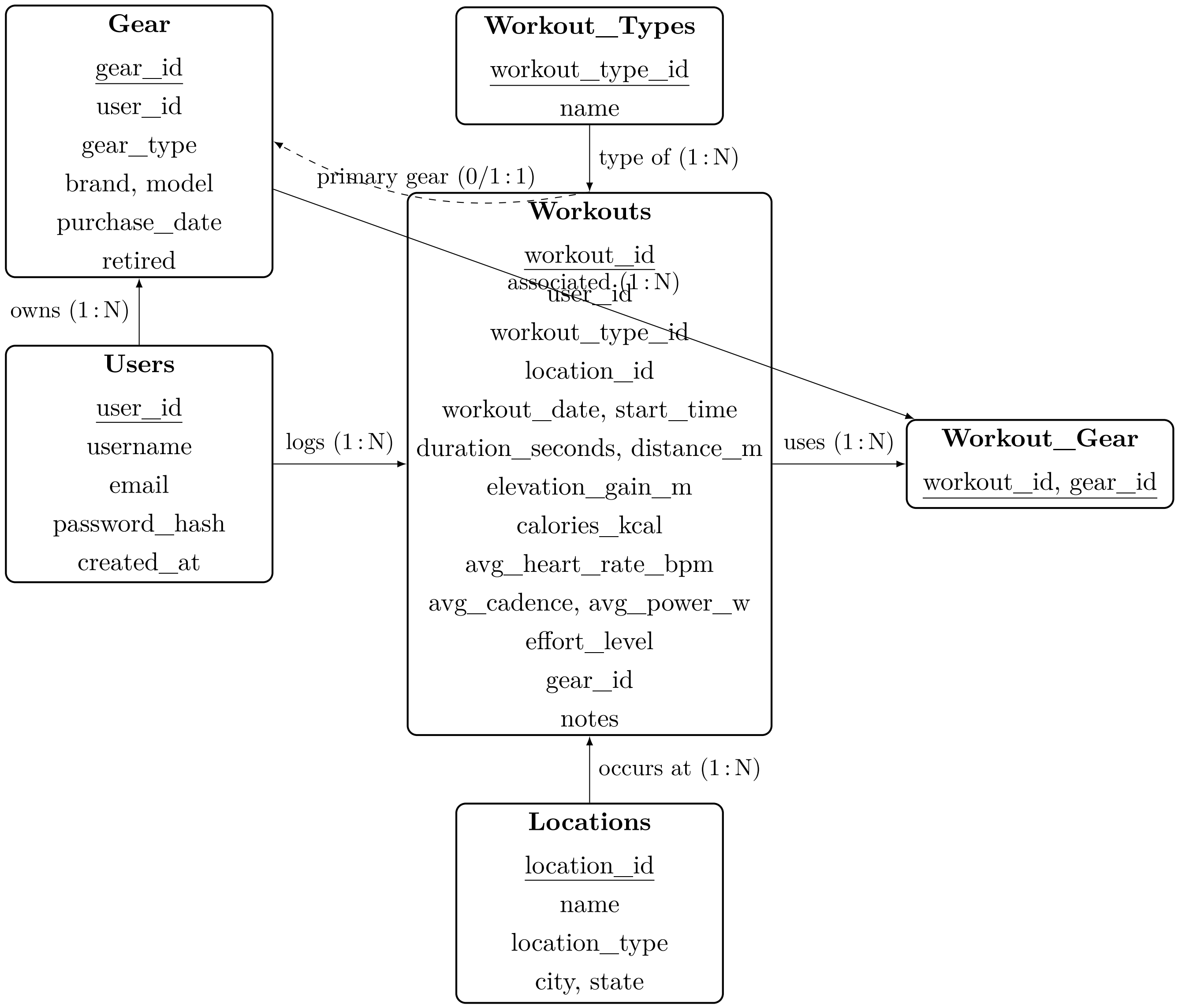


Figure 1: ER-style schema diagram for the IronTrack training database.

4. Functional Dependencies

This section outlines the functional dependencies related to the IronTrack database schema as well as the semantics behind the FDs and the candidate keys.

4.1 Functional Dependencies by Relation

Users Each user is uniquely identified by `user_id`. Since both `username` and `email` are also declared `UNIQUE`, they are alternative identifiers for the same tuple.

$$\begin{aligned}\text{Users:}\quad & user\_id \to username, email, password\_hash, created\_at\\ & username \to user\_id, email, password\_hash, created\_at\\ & email \to user\_id, username, password\_hash, created\_at\end{aligned}$$

Workout_Types Each workout type has a surrogate key and a unique name:

$$\begin{aligned}\text{Workout\_Types:}\quad & workout\_type\_id \to name\\ & name \to workout\_type\_id\end{aligned}$$

Locations The surrogate key `location_id` determines all other attributes of a location:

$$\text{Locations:}\quad location\_id \to name, location\_type, city, state$$

(We do not assume any additional FDs such as $(name, city, state) \to location\_id$, since in practice different facilities may share names or cities.)

Gear Each piece of gear is uniquely identified by `gear_id`:

$$\text{Gear:}\quad gear\_id \to user\_id, gear\_type, brand, model, purchase\_date, retired$$

Workouts Each workout tuple is uniquely identified by `workout_id`, which determines all non-key attributes:

$$\begin{aligned}\text{Workouts:}\quad workout\_id \to\ & user\_id, workout\_type\_id, location\_id,\\ & workout\_date, start\_time,\\ & duration\_seconds, distance\_m, elevation\_gain\_m,\\ & calories\_kcal, avg\_heart\_rate\_bpm,\\ & avg\_cadence, avg\_power\_w, effort\_level,\\ & gear\_id, notes\end{aligned}$$

We do not assume that any combination of non-key attributes (e.g., $(user\_id, workout\_date, start\_time)$) uniquely determines a workout, since the same user can log multiple sessions on the same day and time granularity is not unique.

Workout_Gear The join table `Workout_Gear` only contains the composite key; there are no additional non-key attributes. The only non-trivial FD is that the composite key determines itself:

$$\text{Workout\_Gear:} \quad (workout\_id, gear\_id) \rightarrow workout\_id, gear\_id$$

Equivalently, there are no proper FDs with a strict superset of the key on the left-hand side or any non-key attributes on the right-hand side.

4.2 Semantics Behind the FDs

The FDs above follow directly from the intended meaning of the data:

- In **Users**, a person cannot have two different accounts with the same username or email, so these attributes function as natural keys in addition to the surrogate `user_id`.
- In **Workout_Types**, each sport ("swim", "bike", "run") is represented exactly once, and its numeric identifier is simply a surrogate key.
- In **Locations**, `location_id` is the only guaranteed unique identifier; multiple gyms or pools may share similar names or be in the same city.
- In **Gear**, `gear_id` uniquely identifies each item, while attributes such as `brand` and `model` are descriptive and not guaranteed to be unique.
- In **Workouts**, `workout_id` is the unique identifier for each logged session; the same user may perform multiple workouts with identical metrics, so no natural composite key is assumed.
- In **Workout_Gear**, the combination $(workout\_id, gear\_id)$ uniquely represents a single association between a workout and a piece of gear, and there are no further attributes to introduce additional dependencies.

4.3 Candidate Keys

From these dependencies we derive the candidate keys for each relation:

- **Users**: $\{user\_id\}$, $\{username\}$, and $\{email\}$ are all candidate keys.
- **Workout_Types**: $\{workout\_type\_id\}$ and $\{name\}$ are candidate keys.
- **Locations**: $\{location\_id\}$ is the only candidate key.
- **Gear**: $\{gear\_id\}$ is the only candidate key.

- **Workouts**: $\{workout\_id\}$ is the only candidate key.
- **Workout_Gear**: $\{workout\_id, gear\_id\}$ is the composite candidate key.

5. Final Database Schema (3NF)

In this section we summarize the final relational schema of the IronTrack database and argue that it satisfies Third Normal Form (3NF). We focus on the base tables; the sport-specific views (`run_workouts`, `bike_workouts`, `swim_workouts`, `gear_distance`) are derived and do not introduce additional redundancy in the stored data.

5.1 Relational Schema

* Underlines attributes are primary key attributes.

$$\text{Users}(\underline{user\_id}, username, email, password\_hash, created\_at)$$

$$\text{Workout\_Types}(\underline{workout\_type\_id}, name)$$

$$\text{Locations}(\underline{location\_id}, name, location\_type, city, state)$$

$$\text{Gear}(\underline{gear\_id}, user\_id, gear\_type, brand, model, purchase\_date, retired)$$

$$\text{Workouts}(\underline{workout\_id}, user\_id, workout\_type\_id, location\_id, workout\_date, start\_time, duration\_seconds, d$$

$$\text{Workout\_Gear}(\underline{workout\_id,\ gear\_id})$$

All foreign keys are as specified in the SQL schema:

- `Gear.user_id` → `Users.user_id`
- `Workouts.user_id` → `Users.user_id`
- `Workouts.workout_type_id` → `Workout_Types.workout_type_id`
- `Workouts.location_id` → `Locations.location_id`
- `Workouts.gear_id` → `Gear.gear_id` (optional primary gear)
- `Workout_Gear.workout_id` → `Workouts.workout_id`
- `Workout_Gear.gear_id` → `Gear.gear_id`

5.2 3NF (and BCNF) Justification

The 3NF condition for a relation R with a set of functional dependencies F means for every non-trivial FD $X \to A$ that holds in R, either

1. X is a superkey of R, or
2. A is a prime attribute (part of some candidate key of R).

Using the functional dependencies from Section 4, we argue relation by relation:

- **Users**:

  $$user\_id \to username, email, password\_hash, created\_at$$

  and, by uniqueness constraints,

  $$username \to user\_id, email, password\_hash, created\_at$$

  $$email \to user\_id, username, password\_hash, created\_at.$$

  Thus `user_id`, `username`, and `email` are all candidate keys. In each non-trivial FD, the left-hand side is a candidate key, so every FD has a superkey determinant. Therefore `Users` is in BCNF, and hence in 3NF.

- **Workout_Types**:

  $$workout\_type\_id \to name, \qquad name \to workout\_type\_id.$$

  Both `workout_type_id` and `name` are candidate keys. Again, every FD has a superkey on the left-hand side, so this relation is in BCNF (and therefore in 3NF).

- **Locations**:

  $$location\_id \to name, location\_type, city, state.$$

  The only non-trivial FD uses the primary key `location_id` as determinant. We do not assume any additional FDs such as $(name, city, state) \to location\_id$, so `location_id` is the only candidate key. Hence `Locations` is in BCNF.

- **Gear**:

  $$gear\_id \to user\_id, gear\_type, brand, model, purchase\_date, retired.$$

  Because `gear_id` is the primary key and no other non-trivial FDs exist, the only determinant is a key. Thus `Gear` is in BCNF.

- **Workouts**:

  $$workout\_id \to (user\_id, workout\_type\_id, location\_id, workout\_date, start\_time, duration\_seconds, dista$$

We do not assume any FD where a proper subset of attributes (such as $(user\_id, workout\_date)$) determines the rest of the tuple, since the same user may log multiple workouts per day with overlapping times and similar metrics. Therefore `workout_id` is the only candidate key and is the determinant in every non-trivial FD. Hence `Workouts` is in BCNF.

- **Workout_Gear**: The only non-trivial FD is

$$(workout\_id, gear\_id) \rightarrow workout\_id, gear\_id,$$

where $(workout\_id, gear\_id)$ is the composite primary key. There are no additional attributes and no other FDs, so the relation trivially satisfies BCNF.

Since in each base relation every non-trivial functional dependency has a superkey on the left-hand side, all relations are in BCNF, and therefore satisfy 3NF as well.

5.3 BCNF and Foreign Keys

In many practical database designs, full BCNF is not enforced because decomposing to BCNF can:

- introduce additional joins on common query paths, and
- separate attributes that are naturally queried together.

Foreign key constraints themselves do not violate BCNF, because they relate keys across different relations rather than creating additional intra-relation functional dependencies.

In the design, we deliberately separated concepts such as locations and workout types into their own tables, with `Workouts` referencing them via foreign keys. This avoids typical BCNF violations due to transitive dependencies (e.g., storing `city` and `state` directly in `Workouts` would make `workout_id` determine `location_id`, which in turn determines `city` and `state`). By isolating such attributes in their own relations, we achieve a schema that is both normalized (BCNF/3NF) and still efficient for typical query workloads.

6. Example Queries (SQL and Relational Algebra)

In this section we present several representative analytical queries supported by the IronTrack schema. For each query, we give both an SQL formulation and a corresponding relational algebra (RA) expression using extended RA with grouping/aggregation.

Q1: Total Mileage for Each Running Shoe

Goal: For each piece of gear of type `shoe`, compute the total distance (in miles) accumulated across all workouts where it was used.

SQL.

```
SELECT
    g.gear_id,
    g.brand,
    g.model,
    SUM(w.distance_m) / 1609.34 AS total_miles
FROM Gear g
JOIN Workout_Gear wg
    ON wg.gear_id = g.gear_id
JOIN Workouts w
    ON w.workout_id = wg.workout_id
WHERE g.gear_type = 'shoe'
GROUP BY g.gear_id, g.brand, g.model
ORDER BY total_miles DESC;
```

Relational Algebra (extended). Let G = Gear, WG = Workout_Gear, W = Workouts.

$$\begin{aligned}
R_1 &= \sigma_{\text{gear\_type='shoe'}}(G) \\
R_2 &= R_1 \bowtie_{R_1.\text{gear\_id}=WG.\text{gear\_id}} WG \\
R_3 &= R_2 \bowtie_{WG.\text{workout\_id}=W.\text{workout\_id}} W \\
\text{Result} &= \gamma_{\text{gear\_id},\, brand,\, model;\ \text{total\_miles:=SUM}(distance\_m/1609.34)}(R_3)
\end{aligned}$$

Q2: Average Run Pace by Location Type

Goal: For all run workouts, compute the average pace (seconds per mile) grouped by `location_type` (e.g., `road`, `trail`, `track`).

SQL.

```
SELECT
    l.location_type,
    AVG(w.duration_seconds / (w.distance_m / 1609.34)) AS avg_pace_sec_per_mile
FROM Workouts w
JOIN Workout_Types t
    ON w.workout_type_id = t.workout_type_id
JOIN Locations l
    ON w.location_id = l.location_id
WHERE t.name = 'run'
  AND w.distance_m > 0
GROUP BY l.location_type
ORDER BY l.location_type;
```

Relational Algebra (extended). Let W = Workouts, T = Workout_Types, L = Locations.

$$\begin{aligned} R_1 &= W \bowtie_{W.\text{workout\_type\_id}=T.\text{workout\_type\_id}} T \\ R_2 &= R_1 \bowtie_{R_1.\text{location\_id}=L.\text{location\_id}} L \\ R_3 &= \sigma_{T.\text{name}=\text{'run'} \wedge W.\text{distance\_m}>0}(R_2) \\ \text{Result} &= \gamma_{\text{location\_type; avg\_pace:=AVG}\left(\text{duration\_seconds/(distance\_m/1609.34)}\right)}(R_3) \end{aligned}$$

Q3: Weekly Training Volume by Discipline

Goal: For each week and each discipline (swim/bike/run), compute the total training volume (in kilometers).

SQL.

```
SELECT
    date_trunc('week', w.workout_date) AS week_start,
    t.name AS discipline,
    SUM(w.distance_m) / 1000.0 AS total_km
FROM Workouts w
JOIN Workout_Types t
    ON w.workout_type_id = t.workout_type_id
GROUP BY week_start, discipline
ORDER BY week_start, discipline;
```

Relational Algebra (extended). Let W = Workouts, T = Workout_Types. We use an extended RA function WEEK(·) to extract the week (e.g., the Monday of that ISO week).

$$\begin{aligned} R_1 &= W \bowtie_{W.\text{workout\_type\_id}=T.\text{workout\_type\_id}} T \\ \text{Result} &= \gamma_{\text{WEEK(workout\_date)}, T.\text{name; total\_km:=SUM(distance\_m/1000.0)}}(R_1) \end{aligned}$$

Q4: Workouts Using a Specific Gear Item

Goal: Given a specific gear item (identified by `:gear_id`), list all workouts that used this gear, along with date, type, and distance.

SQL.

```
SELECT
    w.workout_id,
    t.name AS workout_type,
    w.workout_date,
    w.start_time,
    w.distance_m
FROM Workout_Gear wg
JOIN Workouts w
    ON wg.workout_id = w.workout_id
JOIN Workout_Types t
    ON w.workout_type_id = t.workout_type_id
WHERE wg.gear_id = :gear_id
ORDER BY w.workout_date DESC, w.start_time DESC;
```

Relational Algebra (extended). Let WG = Workout_Gear, W = Workouts, T = Workout_Types, and let a specific gear identifier be g_0.

$$\begin{aligned}
R_1 &= \sigma_{\text{gear\_id}=g_0}(WG) \\
R_2 &= R_1 \bowtie_{R_1.\text{workout\_id}=W.\text{workout\_id}} W \\
R_3 &= R_2 \bowtie_{W.\text{workout\_type\_id}=T.\text{workout\_type\_id}} T \\
\text{Result} &= \pi_{\text{workout\_id},\ T.\text{name},\ \text{workout\_date},\ \text{start\_time},\ \text{distance\_m}}(R_3)
\end{aligned}$$

Q5: Average Bike Power and Speed by User

Goal: For each user, compute average power (W) and average speed (mph) across all bike workouts.

SQL. Using the base tables (rather than the `bike_workouts` view) for clarity:

```
SELECT
    u.user_id,
    u.username,
    AVG(w.avg_power_w) AS avg_power_w,
    AVG((w.distance_m / 1609.34) / (w.duration_seconds / 3600.0)) AS avg_speed_mph
FROM Workouts w
JOIN Workout_Types t
    ON w.workout_type_id = t.workout_type_id
JOIN Users u
    ON w.user_id = u.user_id
WHERE t.name = 'bike'
  AND w.duration_seconds > 0
  AND w.distance_m > 0
GROUP BY u.user_id, u.username
ORDER BY avg_speed_mph DESC;
```

Relational Algebra (extended). Let W = Workouts, T = Workout_Types, U = Users.

$R_1 = W \bowtie_{W.\text{workout\_type\_id}=T.\text{workout\_type\_id}} T$

$R_2 = R_1 \bowtie_{W.\text{user\_id}=U.\text{user\_id}} U$

$R_3 = \sigma_{T.\text{name}=\text{'bike'} \wedge W.\text{duration\_seconds}>0 \wedge W.\text{distance\_m}>0}(R_2)$

Result =

$\gamma_{U.\text{user\_id},\, U.\text{username};\ \text{avg\_power\_w:=AVG(avg\_power\_w), avg\_speed\_mph:=AVG}}\big((\text{distance\_m}/1609.34)/(\text{duration\_seconds}/3600.0)\big)$

Q6: Fast Swim Workouts (Pace Threshold)

Goal: List all swim workouts where the athlete held a pace faster than 90 seconds per 100 yards, including username and computed pace.

SQL. Using the sport-specific `swim_workouts` view:

```
SELECT
    u.username,
    sw.workout_id,
    sw.workout_date,
    sw.duration_seconds,
    sw.distance_yards,
    sw.pace_seconds_per_100yd
FROM swim_workouts sw
JOIN Users u
    ON sw.user_id = u.user_id
WHERE sw.pace_seconds_per_100yd < 90  -- faster than 1:30 / 100 yd
ORDER BY sw.pace_seconds_per_100yd ASC;
```

Relational Algebra (extended). We can express the same logic in RA using base relations. Let W = Workouts, T = Workout_Types, U = Users. We derive the swim subset and compute pace in the selection predicate.

$R_1 = W \bowtie_{W.\text{workout\_type\_id}=T.\text{workout\_type\_id}} T$

$R_2 = R_1 \bowtie_{W.\text{user\_id}=U.\text{user\_id}} U$

$R_3 = \sigma_{T.\text{name}=\text{'swim'}}(R_2)$

Result =

$\pi_{U.\text{username},\, W.\text{workout\_id},\, W.\text{workout\_date},\, W.\text{duration\_seconds},\, W.\text{distance\_m},\, \text{pace\_sec\_per\_100yd}}\big(\sigma_{\text{pace\_sec\_per\_100yd}<90}(R_3)$

where, in an extended RA sense, we define

$$\text{pace\_sec\_per\_100yd} := \text{duration\_seconds}/\big((\text{distance\_m}/0.9144)/100\big),$$

i.e., seconds divided by the number of 100-yard segments.

7. Implementation Details

The IronTrack system is implemented as a lightweight Python application with both a web-based UI and a command-line interface, backed by a PostgreSQL database. This section summarizes the implementation environment, technology stack, and how the main components interact.

- **Operating system and environment.**
  - Development OS: Windows.
  - Python environment: Python 3.11 in a virtual environment.
  - All scripts are intended to be run from the project root (e.g., `python iron`$_m$*an/app/cli.py*).
- **DBMS: PostgreSQL.**
  - The database is implemented in PostgreSQL.
  - The full schema is defined in `iron`$_m$*an/db/schema.sql, including* :
    - Base tables: `Users`, `Workout_Types`, `Locations`, `Gear`, `Workouts`, `Workout_Gear`.
    - Derived sport-specific views: `run_workouts`, `bike_workouts`, `swim_workouts`.
    - Aggregation view: `gear_distance` for summarizing gear usage.
- Schema application is done via:

  ```
  psql -f iron_man/db/schema.sql
  ```

  after creating the database and user.

Programming environment and main components.

- **Language:** Python.
- **Database access:**
  - PostgreSQL connections are created via `iron_man/app/db.py`.
  - The helper function `app.db.get_connection` centralizes connection parameters (host, port, DB name, user, password).
  - A standard PostgreSQL driver (e.g., `psycopg2`) is used under the hood for executing SQL.
- **Data access layer:** `iron_man/app/queries.py`
  - Encapsulates all SQL used by the app and CLI.

- Example functions:
  - `insert_workout`: inserts a new workout into `Workouts`.
  - `insert_gear`: adds a new gear item for a user.
  - `attach_gear_to_workout`: populates `Workout_Gear`.
  - `get_weekly_volume_by_sport`: queries aggregated training volume by week and discipline.
  - `get_total_distance_per_gear`: queries the `gear_distance` view.
  - `fetch_workouts`: returns recent workouts for display in the UI/CLI.
- Centralizing queries in one module improves maintainability and keeps the UI logic free of raw SQL strings.

- **Streamlit web UI:** `iron_man/app/app.py`
  - Implements a simple dashboard and forms for:
    - Viewing recent workouts.
    - Adding new workouts and gear.
    - Visualizing weekly training volume.
  - The main entry point is `app.app.main`, which:
    - Establishes a database connection using `get_connection`.
    - Calls page renderer functions such as `render_dashboard`, `render_view_workouts`, and `render_add_workout`.
  - Sport-specific views (e.g., `run_workouts`, `bike_workouts`) are used by the UI to display discipline-appropriate units (miles, mph, yard pace).
- **Command-line interface (CLI):** `iron_man/app/cli.py`
  - Provides a lightweight alternative to the web UI for quick interactions.
  - The main entry point is `app.cli.main`.
  - Typical commands include:
    - Listing recent workouts for a user.
    - Adding a workout from the terminal.
    - Inspecting gear usage (via `get_total_distance_per_gear`).
  - The CLI relies on the same query functions in `queries.py`, ensuring that both UI and CLI share a consistent data layer.
- **Seed and demo scripts.**
  - `iron_man/app/seed_demo.py`:
    - Function `seed_demo_user_and_workout` creates a demo user and a single workout.
    - Useful for quickly verifying that the schema and app wiring are correct.
  - `iron_man/app/populate_workouts.py`:

        * Functions such as `repopulate_trending_workouts` and `create_demo_gear` generate a season's worth of synthetic data.
        * This enables testing of queries, dashboards, and plots without manual data entry.
    - `iron_man/app/test_query.py`:
        * Provides small helpers (e.g., `get_recent_workouts`) for ad-hoc testing and debugging of SQL queries.

Setup and execution (summary).

- *Configure database connection:*
    - Create a PostgreSQL user and database.
    - Update credentials (host, port, user, password, DB name) in `iron_man/app/db.py`.
- *Apply schema:*
    - Run `psql -f iron_man/db/schema.sql` to create tables, constraints, and views.
- *Seed demo data:*
    - Run `python iron_man/app/seed_demo.py` to insert a demo user and workout.
    - Optionally run `python iron_man/app/populate_workouts.py` to generate more extensive training data.
- *Run the UI:*
    - Start the Streamlit app with:

```
streamlit run iron_man/app/app.py
```

- *Run the CLI:*
    - Invoke the command-line interface with:

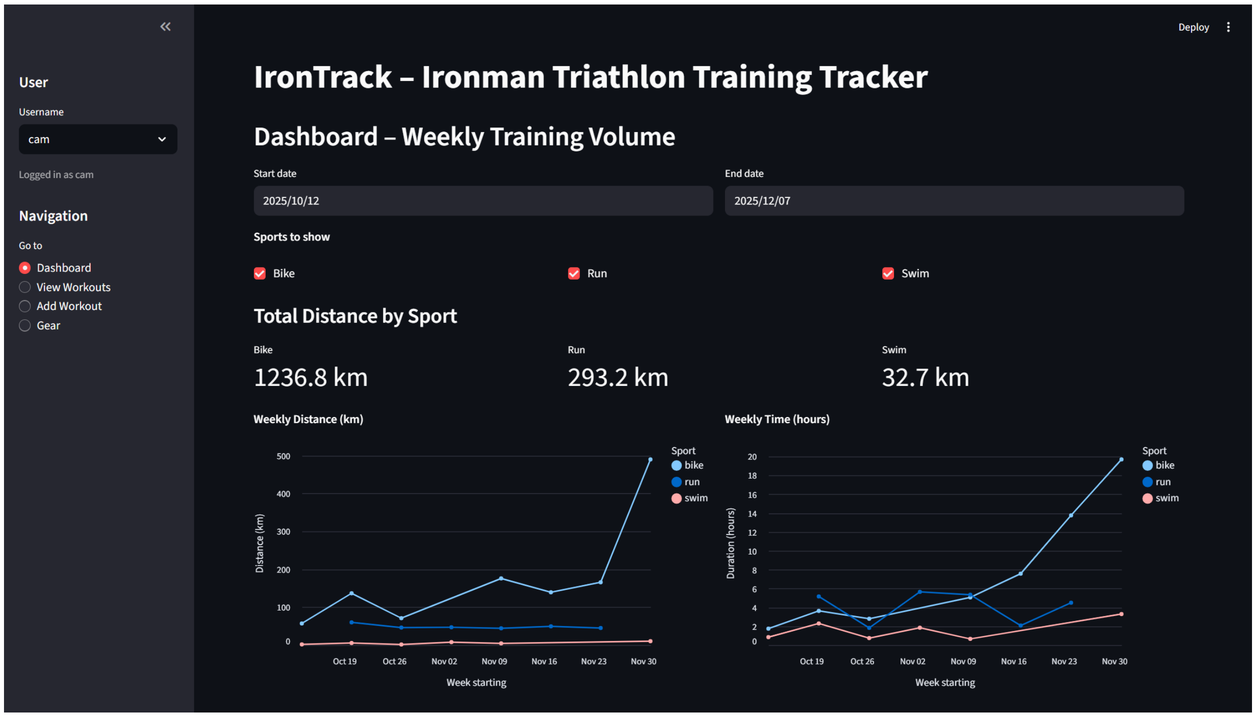
```
python iron_man/app/cli.py
```

8. Team Member Roles and Contributions

- Camden Larson performed:
    - Schema design
    - ER modeling
    - View design and unit conversion logic
    - Query design (SQL and RA)
    - CLI implementation
    - UI development
    - Testing and validation
    - Documentation and writing

9. Reflections: Lessons Learned

- Insights gained beyond the course material:
  - Practical schema evolution and normalization.
  - Real-world enforcement of constraints.
  - Design of usability views (sport-specific analytics).
  - Mapping relational models to application layers.
  - Managing multi-entity interactions (gear usage, locations, types).
- There were some issues with getting the streamlit UI to refresh data when workouts were added.
- Creating the sport-specific views from the workouts table brought some difficulties in making those views show different metrics.
- Iterating on the UI to make it presentable took a lot of time.
- For future database-backed applications I'd make sure my schema was entirely finished before doing anything else or make my code more modular to allow for seamless addition of new tables/columns, it was annoying to go back and change core functionality once I added new columns in some tables.



IronTrack – Ironman Triathlon Training Tracker

Dashboard – Weekly Training Volume

Start date: 2025/10/12

End date: 2025/12/07

Sports to show

✓ Bike ✓ Run ✓ Swim

Total Distance by Sport

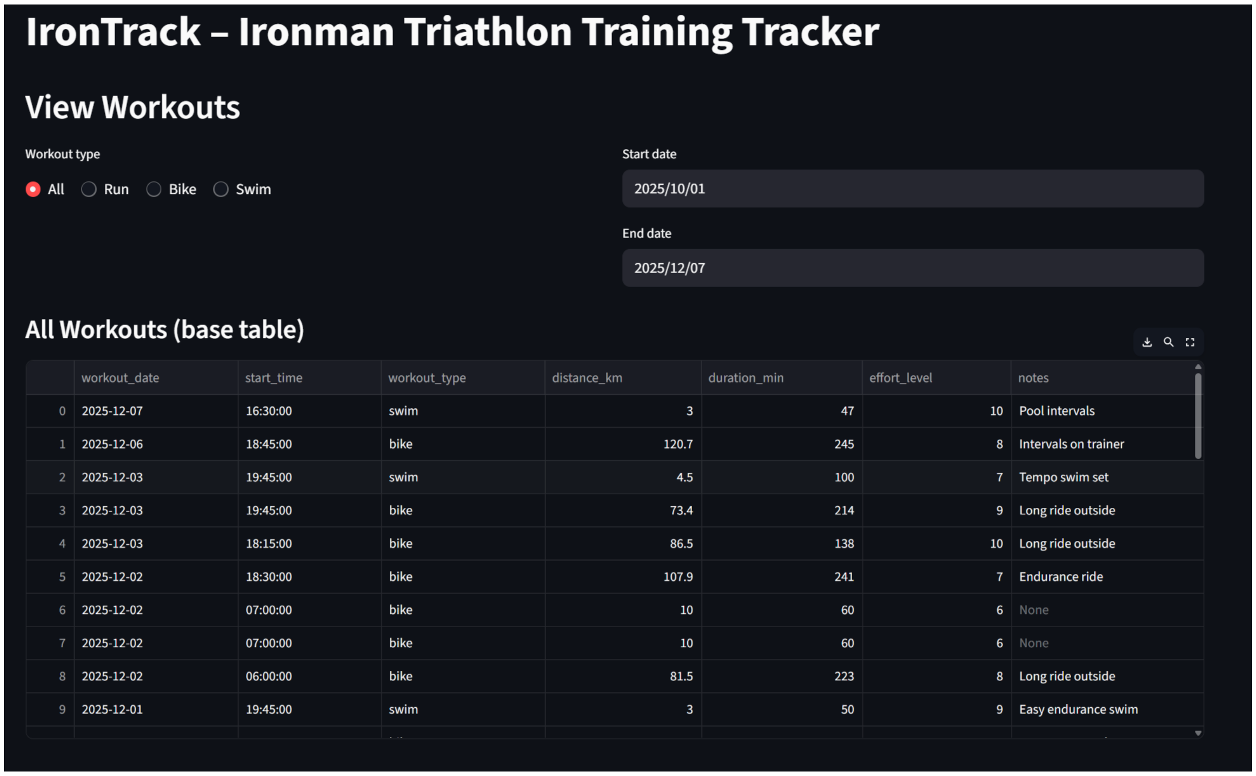
Bike	Run	Swim
1236.8 km	293.2 km	32.7 km

Weekly Distance (km)

Weekly Time (hours)

Figure 2: Central dashboard

UI images



IronTrack – Ironman Triathlon Training Tracker

View Workouts

Workout type

[✓ All] Run Bike Swim

Start date

2025/10/01

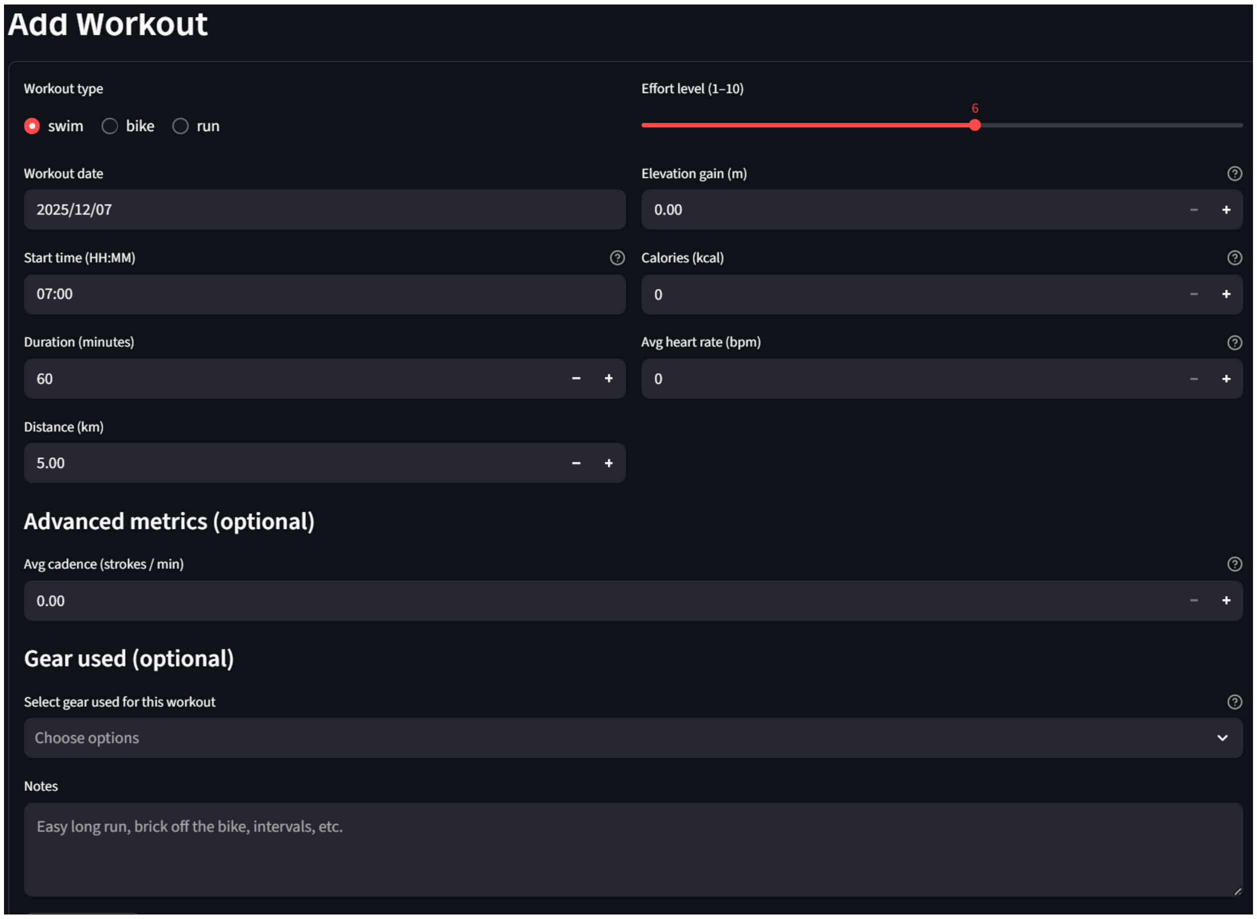
End date

2025/12/07

All Workouts (base table)

	workout_date	start_time	workout_type	distance_km	duration_min	effort_level	notes
0	2025-12-07	16:30:00	swim	3	47	10	Pool intervals
1	2025-12-06	18:45:00	bike	120.7	245	8	Intervals on trainer
2	2025-12-03	19:45:00	swim	4.5	100	7	Tempo swim set
3	2025-12-03	19:45:00	bike	73.4	214	9	Long ride outside
4	2025-12-03	18:15:00	bike	86.5	138	10	Long ride outside
5	2025-12-02	18:30:00	bike	107.9	241	7	Endurance ride
6	2025-12-02	07:00:00	bike	10	60	6	None
7	2025-12-02	07:00:00	bike	10	60	6	None
8	2025-12-02	06:00:00	bike	81.5	223	8	Long ride outside
9	2025-12-01	19:45:00	swim	3	50	9	Easy endurance swim

Figure 3: View Workouts



Add Workout

Workout type

[✓ swim] bike run

Effort level (1–10)

6

Workout date

2025/12/07

Elevation gain (m)

0.00

Start time (HH:MM)

07:00

Calories (kcal)

0

Duration (minutes)

60

Avg heart rate (bpm)

0

Distance (km)

5.00

Advanced metrics (optional)

Avg cadence (strokes / min)

0.00

Gear used (optional)

Select gear used for this workout

Choose options

Notes

Easy long run, brick off the bike, intervals, etc.

Figure 4: Add Workouts