ANSWERS

A11.10 Project: the run database

Databases and big data · A level · OCR H446 1.3.2, AQA 7517 4.10.1, Eduqas A500QS 2.5 · about 75 min

BugBotLab

What this lesson is about

Design, normalise, build and fill a database of robot runs, log live runs in transactions, report with one joined query and export the report as JSON.

- The brief
- Design
- Plan
- Step 1: tables that refer to each other
- Step 2: measure a run
- Step 3: an error in SQL

Questions

5 marks in all

1. Why does the project's Run table not store each run's error? [1 mark]

☐ **A** It depends on the run's distance and the challenge's target, so storing it would be redundant and could become inconsistent

☐ **B** SQL cannot store negative numbers

☐ **C** The error is a primary key

☐ **D** JSON cannot hold it

> **Answer:** A. A derived value can be worked out in a query, so it can never disagree with the data it comes from.

2. A report query joins runs, robots, teams and challenges. How many JOIN clauses does it need? [1 mark]

> **Answer:** 3. Each table after the first needs one join: four tables, three joins.

3. What does this print? [1 mark]

```
import sqlite3
db = sqlite3.connect(":memory:")
db.execute("CREATE TABLE runs (team TEXT, cm INTEGER, target INTEGER)")
db.executemany("INSERT INTO runs VALUES (?, ?, ?)", [("Hawks", 27, 30), ("Hawks", 31, 30),
("Owls", 16, 15)])
for row in db.execute("SELECT team, COUNT(*), AVG(ABS(cm - target)) FROM runs GROUP BY team
ORDER BY team"):
    print(*row)
```

> **Answer:**
>
> ```
> Hawks 2 2.0
> Owls 1 1.0
> ```
>
> Hawks' errors are 3 and 1, a mean of 2.0; Owls has one run with an error of 1.

4. Put the stages of building the run database in order. [1 mark]

Number the lines 1 to 5 to put them in the right order.

___ `Query the tables to produce the report`

___ `Insert the data and log new runs in transactions`

___ `Normalise the design to third normal form`

___ `Create the tables, parents before children`

___ `Identify the entities, keys and relationships from the brief`

Answer:

```
Identify the entities, keys and relationships from the brief
Normalise the design to third normal form
Create the tables, parents before children
Insert the data and log new runs in transactions
Query the tables to produce the report
```

Model, normalise, define, fill, then query: each step needs the one before.

5. Ada's run is logged inside a transaction (with db:) and the program crashes halfway through the insert. What is in the database? [1 mark]

- ☐ **A** No trace of the run, because the transaction was rolled back
- ☐ **B** Half of the run's fields
- ☐ **C** The run, marked as incomplete
- ☐ **D** Nothing at all: the whole database is emptied

Answer: A. Atomicity: the incomplete transaction is undone, and everything committed before it stays.

The task: the run database

The starter gives the data as lists of tuples: `teams` holds `(team_id, team_name)`, `robots` holds `(robot_id, name, team_id)`, `challenges` holds `(code, name, target_cm)` and `earlier_runs` holds `(robot_id, code, cm)`. Ids and distances are whole numbers; names and codes are text. 1. Create the four tables of the design, with a primary key on each and the three foreign keys declared with `REFERENCES`, and switch on foreign key checking. 2. Insert the data lists. Let the database number the runs. 3. For each challenge, in the order of `challenges`, the robot (Ada, robot 1) makes one run at speed 50: `S` drives forward the target distance, `R` drives backward the target distance. Measure the straight-line distance really covered, rounded to a whole number, and insert the run inside its own transaction (`with db:`). 4. With **one** query that joins `runs`, `robots`, `teams` and `challenges`, groups by team, and uses `COUNT` and `AVG`, print one line per team in name order: `<team_name>: <n> runs, mean error <error> cm`, where the error is the mean of the absolute differences between each run's `cm` and its challenge's `target_cm`, rounded to 1 decimal place. For example `Owls: 2 runs, mean error 2.5 cm`. 5. Save the report to `report.json` with `json.dump`, as a list with one object per team, each with the keys `team` (text), `runs` (a whole number) and `mean_error` (a number). Then print `saved report.json`. Three lines in all.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

import sqlite3
import json

teams = [(1, "Hawks"), (2, "Owls")]
robots = [(1, "Ada", 1), (2, "Bolt", 1), (3, "Cog", 2)]
challenges = [("S", "sprint", 30), ("R", "reverse", 15)]
earlier_runs = [(2, "S", 27), (2, "R", 16), (3, "S", 33), (3, "R", 13)]
```

The hint students can ask for: Design first: four entities, and runs is the one with two foreign keys. The robot is Ada, robot 1. Measure each run from position() before and after, rounded to a whole cm. The error of a run is how far its distance is from the challenge's target, whichever side, and the report is one query.

A solution

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

import sqlite3
import json

teams = [(1, "Hawks"), (2, "Owls")]
robots = [(1, "Ada", 1), (2, "Bolt", 1), (3, "Cog", 2)]
challenges = [("S", "sprint", 30), ("R", "reverse", 15)]
earlier_runs = [(2, "S", 27), (2, "R", 16), (3, "S", 33), (3, "R", 13)]
db = sqlite3.connect(":memory:")
db.execute("PRAGMA foreign_keys = ON")
db.execute("CREATE TABLE teams (team_id INTEGER PRIMARY KEY, team_name TEXT NOT NULL)")
db.execute("""CREATE TABLE robots (robot_id INTEGER PRIMARY KEY, name TEXT NOT NULL,
              team_id INTEGER NOT NULL REFERENCES teams(team_id))""")
db.execute("""CREATE TABLE challenges (code TEXT PRIMARY KEY, name TEXT NOT NULL,
              target_cm INTEGER NOT NULL)""")
db.execute("""CREATE TABLE runs (run_id INTEGER PRIMARY KEY,
              robot_id INTEGER NOT NULL REFERENCES robots(robot_id),
```

```
            code TEXT NOT NULL REFERENCES challenges(code),
            cm INTEGER NOT NULL)""")
with db:
    db.executemany("INSERT INTO teams VALUES (?, ?)", teams)
    db.executemany("INSERT INTO robots VALUES (?, ?, ?)", robots)
    db.executemany("INSERT INTO challenges VALUES (?, ?, ?)", challenges)
    db.executemany("INSERT INTO runs (robot_id, code, cm) VALUES (?, ?, ?)", earlier_runs)

drives = {"S": forward, "R": backward}
for code, name, target in challenges:
    x0, y0 = position()
    drives[code](50, distance=target)
    x1, y1 = position()
    cm = round(((x1 - x0) ** 2 + (y1 - y0) ** 2) ** 0.5)
    with db:
        db.execute("INSERT INTO runs (robot_id, code, cm) VALUES (?, ?, ?)", (1, code, cm))

query = """SELECT teams.team_name, COUNT(*), ROUND(AVG(ABS(runs.cm -
challenges.target_cm)), 1)
           FROM runs
           JOIN robots ON runs.robot_id = robots.robot_id
           JOIN teams ON robots.team_id = teams.team_id
           JOIN challenges ON runs.code = challenges.code
           GROUP BY teams.team_name
           ORDER BY teams.team_name"""
report = []
for team, n, error in db.execute(query):
    print(f"{team}: {n} runs, mean error {error} cm")
    report.append({"team": team, "runs": n, "mean_error": error})

with open("report.json", "w") as f:
    json.dump(report, f, indent=2)
print("saved report.json")
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.