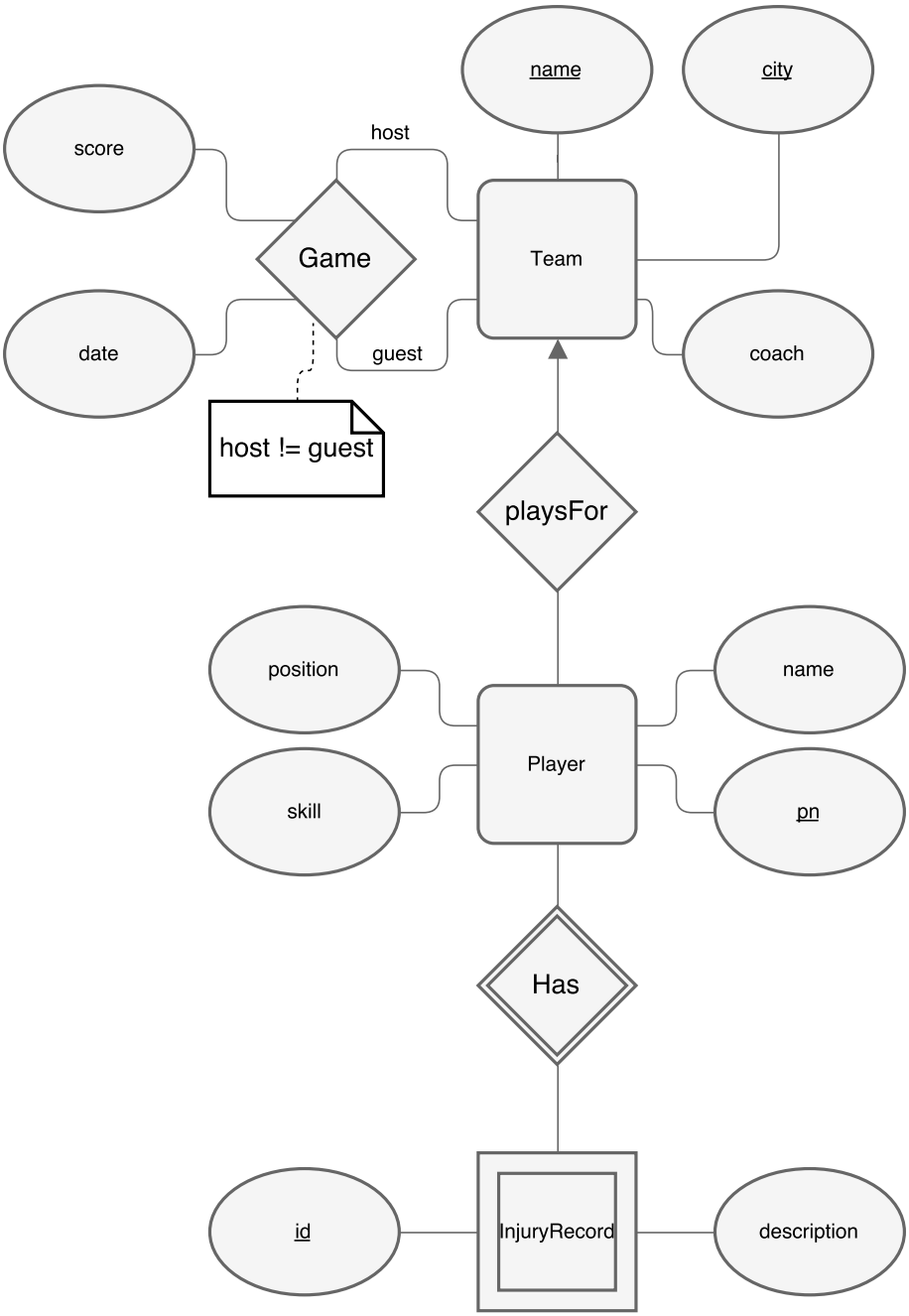
Databases Exam HT2016 re-exam Solution

Solution 1a



Solution 1b

```
Team(city, name, coach)
Player(pn, name, position, skill)
PlaysFor(pn, city, name)
    pn -> Player.pn
    (city, name) -> Team.(city, name)
InjuryRecord(pn, id, description)
    pn -> Player.pn
Game(hostcity, hostname, guestcity, guestname, date, score)
    (hostcity, hostname) -> Team.(city, name)
    (guestcity, guestname) -> Team.(city, name)
```

Solution 2a

- {A, B, C, D} does not identify all attributes.
- {A, C, D, E, G} is a superkey but not a key, since attribute G can be removed and the resulting set of attributes is a key.

Solution 2b

```
Decompose on BC -> D
{BC}+ = {BCDG}

             R1(B,C,D,G)
             R2(B,C,A,E,F)
                  B,C -> R1.(B,C)

Decompose R2 on FA -> B
{FA}+ = {FAB}
             R21(F,A,B)
             R22(F,A,C,E)
                  F,A -> R21.(F,A)

Key of R22 is FACE
```

Solution 2c

```
BC -> G
```

Left side is not a superkey of R, and G is not prime in R.

Solution 2d

```
R1(B, C, D, G)
R2(D, E, F)
R3(F, A, B)
R4(F, A, C, E)
```

Solution 3a

```
CREATE TABLE Employees(
    empId INT PRIMARY KEY,
    name TEXT NOT NULL,
    year INT NOT NULL,
    salary FLOAT NOT NULL CHECK(salary >= 0),
    entitlement INT NOT NULL DEFAULT 30 CHECK(entitlement >= 0),
    branch TEXT NOT NULL
);

CREATE TABLE ParentalLeave(
    employee INT REFERENCES Employees,
    startDay INT NOT NULL CHECK(startDay >= 0 AND startDay <= 366),
    startYear INT NOT NULL,
    endDay INT NOT NULL CHECK((endYear = startYear AND startDay <= endDay) OR
        endYear > startYear),
    endYear INT NOT NULL CHECK(endYear >= startYear)
);
```

Solution 3b

```
SELECT branch, COUNT(*)
        FROM Employees e JOIN ParentalLeave p ON e.empId = p.employee
        WHERE p.startYear <> p.endYear
        GROUP BY e.branch;
```

Solution 3c

```
SELECT branch, AVG(salary) AS avgsalary
        FROM Employees
        GROUP BY branch
        HAVING MIN(year) >= 1987
        ORDER BY avgsalary;
```

Solution 4a

$\tau_{name}(\pi_{empId,name,salary+supplement}(Employees \bowtie Offices))$

Solution 4b

$\gamma_{city,AVG(salary)\rightarrow avgSalary}(\sigma_{dept=\text{"}sales\text{"}}(Employees))$

Solution 5a

```
CREATE VIEW VoteSummary AS
    SELECT cityname, 100.0 * votecount / (SELECT SUM(votecount) FROM Votes)
       from Votes;
```

Solution 5b

```
CREATE OR REPLACE FUNCTION fixUnknownVotes() RETURNS TRIGGER AS $$
BEGIN
    IF NOT EXISTS (SELECT * FROM Votes WHERE cityname = '<not␣voted>')
    THEN
        INSERT INTO Votes(cityname, votecount) VALUES('<not␣voted>', 0);
    END IF;
    UPDATE Votes SET votecount = 1337 -
          (SELECT SUM(votecount) FROM Votes WHERE cityname <> '<not␣voted>')
        WHERE cityname = '<not␣voted>';
    RETURN NULL;
END
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER fixUnknownVotesTrigger AFTER INSERT OR UPDATE ON Votes
  FOR EACH ROW WHEN (NEW.cityname <> '<not␣voted>')
  EXECUTE PROCEDURE fixUnknownVotes();
```

Solution 6a

No, Alice does not have the correct privileges. She does not need to read the password in the Users table, she does not need INSERT privileges on UserStatus, and she does not need SELECT privileges on LogBook. In addition, Alice lacks INSERT privileges on LogBook.

The minimally required set of permissions is:

```
GRANT SELECT(id, name) ON Users TO Alice;
GRANT SELECT(id, loggedin) ON UserStatus TO Alice;
GRANT INSERT(id, timestamp, name) ON LogBook TO Alice;
```

Solution 6b

The attacker can input something like `‘ or ‘1‘=‘1`, so that the query turns into

```
SELECT     FROM UserStatus WHERE id = ‘‘ or ‘1‘=‘1‘   ;
```

(mind the closing quote).

This is an example of an SQL injection vulnerability or attack.

The vulnerability can be removed by either correctly sanitizing or escaping the data in the `userinput` variable. A better solution is to use a `PreparedStatement` with placeholder:

```
...
String query = "SELECT * FROM UserStatus WHERE id = ?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setString(1, userinput);
ResultSet rs = stmt.executeQuery();
...
```

Solution 6c

The transaction is vulnerable to “non-repeatable read” and “phantom read” interferences, because the `READ COMMITTED` transaction isolation level does not protect against them. The stronger `REPEATABLE READ` isolation level is not sufficient because it still allows phantom reads. Only the `SERIALIZABLE` isolation level is sufficient, since it protects against dirty read, non-repeatable read and phantom read.