Databases Autumn 2025

Hand-In Exercise 2

October 25, 2025

Aiysha Frutiger
Jannick Seper
Luis Tritschler

Total Points	

Task	**Points**

Aiysha Frutiger
Jannick Seper
Luis Tritschler

Task 1

Given excerpt of the relational schema plus added relations and integrety constrain befor the first bullet point.

```
====== Entity ======
CREATE TABLE Lecturer (
        LecturerID  INT PRIMARY KEY,
        FirstName   VARCHAR(255),
        LastName    VARCHAR(255),
        Title       VARCHAR(255)
);

CREATE TABLE Lecture (
        Title               VARCHAR(255) PRIMARY KEY,
        CreditPoints        INT,
        SemesterWeekHours   INT,
        LecturerID          INT,
        FOREIGN KEY (LecturerID) REFERENCES Lecturer (LecturerID)
);

CREATE TABLE Exercise (
        ExID            INT PRIMARY KEY,
        No              INT,
        Semester        VARCHAR(255),
        LectureTitle VARCHAR(255),
        FOREIGN KEY (LectureTitle) REFERENCES Lecture (Title)
);

CREATE TABLE Author (
        AuthorID    INT PRIMARY KEY,
        LastName    VARCHAR(255),
        FirstName   VARCHAR(255),
        Title       VARCHAR(255)
);

CREATE TABLE Task (
        TaskID      INT PRIMARY KEY,
        Points      INT,
        Difficulty  INT,
        Text        VARCHAR(65535),
        AuthorID    INT,
        FOREIGN KEY (AuthorID) REFERENCES Author (AuthorID)
);

====== Relationships ======
```

```
CREATE TABLE Contains (
        ExID      INT,
        TaskID    INT,
        PRIMARY KEY (ExID, TaskID),
        FOREIGN KEY (ExID)  REFERENCES Exercise (ExID),
        FOREIGN KEY (TaskID) REFERENCES Task (TaskID),
        Sequence  INT
);

CREATE TABLE Consists_of (
        SuperTaskID INT,
        SubTaskID   INT,
        PRIMARY KEY (SuperTaskID, SubTaskID),
        FOREIGN KEY (SuperTaskID) REFERENCES Task (TaskID),
        FOREIGN KEY (SubTaskID)   REFERENCES Task (TaskID),
        Sequence    INT
);

====== Integrity ======
CREATE ASSERTION contains_only_super_tasks
  CHECK ( NOT EXISTS (
    SELECT *
    FROM Contains c
    WHERE EXISTS (
      SELECT *
      FROM Consists_of co
      WHERE co.SubTaskID = c.TaskID
    )
  ) );

CREATE ASSERTION consists_of_non_recursive
  CHECK ( NOT EXISTS (
    SELECT *
    FROM Consists_of x
    WHERE EXISTS (
      SELECT *
      FROM Consists_of y
      WHERE y.SuperTaskID = x.SubTaskID
    )
  ) );
```

- *The title of the lecture has to be unique and may not be altered if any exercise is available for the lecture.*
  The first part of the point is already enforced bc the title of the lecture is a `PRIMARY KEY` and therefore must be unique. The second part can be enforced with this addition.

```
CREATE TABLE Exercise (
        ExID            INT PRIMARY KEY,
        No              INT,
        Semester        VARCHAR(255),
        LectureTitle    VARCHAR(255),
        FOREIGN KEY (LectureTitle) REFERENCES Lecture (Title)
->      ON UPDATE RESTRICT
);
```

- *For a lecture, no more than 10 credit points may be awarded.*
  This part can be enforced with this assertion.

```
CREATE ASSERTION no_more_than_10_credits
        CHECK (NOT EXISTS (
                SELECT * FROM Lecture
                WHERE CreditPoints > 10))
        NOT DEFERRABLE;
```

- *Lecturers may give multiple lectures.*
  This part is beeing enforced with this.

```
CREATE TABLE Lecture (
        Title               VARCHAR(255) PRIMARY KEY,
        CreditPoints        INTEGER,
        SemesterWeekHours   INTEGER,
        LecturerID          INTEGER,
->      FOREIGN KEY (LecturerID) REFERENCES Lecturer (LecturerID)
);
```

- *A lecture may include several exercises. An exercise always belongs to exactly one lecture.*
  The first part is already enforced via the foreign key while for the second part we have to add the `NOT NULL` so we guaratee that e exersise must be in one lecture.

```
CREATE TABLE Exercise (
        ExID            INT PRIMARY KEY,
        No              INT,
        Semester        VARCHAR(255),
->      LectureTitle    VARCHAR(255) NOT NULL,
->      FOREIGN KEY (LectureTitle) REFERENCES Lecture (Title)
        ON UPDATE RESTRICT
);
```

- *Before a new author is entered into the system, it should be checked that no other author with the same first name, last name and title is present.*
  This part is beeing enforced with this.

```
CREATE TABLE Author (
        AuthorID    INT PRIMARY KEY,
        LastName    VARCHAR(255),
        FirstName   VARCHAR(255),
        Title       VARCHAR(255),
->      UNIQUE (FirstName, LastName, Title)
);
```

Task 2 Foreign Keys

2a) **Foreign Keys**
For the relation `Lecture`, the following foreign key is defined in SQL:

```
CREATE TABLE Lecture (
  ...,
  FOREIGN KEY (fk_lecturer) REFERENCES Lecturer(pk_lecturer)
);
```

In the following, this foreign key is simulated using *(i)* an assertion and *(ii)* triggers.

(i) **Assertion (static integrity constraint)**
An `ASSERTION` ensures that no tuple in `Lecture` references a non-existent `Lecturer`. We formulate this as a double `NOT EXISTS` statement that expresses the same semantics as a foreign key.

```
CREATE ASSERTION FK_Lecture_Lecturer_OK
CHECK (
  NOT EXISTS (
    SELECT 1
    FROM Lecture L
    WHERE L.fk_lecturer IS NOT NULL
      AND NOT EXISTS (
        SELECT 1
        FROM Lecturer R
        WHERE R.pk_lecturer = L.fk_lecturer
      )
  )
)
DEFERRABLE INITIALLY IMMEDIATE;
```

This assertion guarantees that for every non-NULL value in `Lecture.fk_lecturer`, a matching `Lecturer.pk_lecturer` exists. The condition is checked globally and can be deferred to commit time using the `DEFERRABLE` clause.

(ii) **Triggers (dynamic integrity constraint)**
The same referential integrity can be dynamically enforced by triggers (E–C–A pattern):

- **Child-side:** Prevent inserting or updating a `Lecture` with a non-existent `fk_lecturer`.
- **Parent-side:** Prevent deleting or updating a `Lecturer` that is still referenced by a `Lecture`.

Child-side triggers (on `Lecture`)

```
CREATE TRIGGER Lecture_FK_Check_Ins
BEFORE INSERT ON Lecture
WHEN (NEW.fk_lecturer IS NOT NULL AND
      NOT EXISTS (SELECT 1 FROM Lecturer R
                  WHERE R.pk_lecturer = NEW.fk_lecturer))
( ROLLBACK WORK );

CREATE TRIGGER Lecture_FK_Check_Upd
BEFORE UPDATE OF fk_lecturer ON Lecture
REFERENCING NEW AS Lnew
WHEN (Lnew.fk_lecturer IS NOT NULL AND
      NOT EXISTS (SELECT 1 FROM Lecturer R
                  WHERE R.pk_lecturer = Lnew.fk_lecturer))
( ROLLBACK WORK );
```

Parent-side triggers (on `Lecturer`)

```
CREATE TRIGGER Lecturer_Ref_Block_Del
BEFORE DELETE ON Lecturer
REFERENCING OLD AS Pold
WHEN (EXISTS (SELECT 1 FROM Lecture L
              WHERE L.fk_lecturer = Pold.pk_lecturer))
( ROLLBACK WORK );

CREATE TRIGGER Lecturer_Ref_Block_Upd
BEFORE UPDATE OF pk_lecturer ON Lecturer
REFERENCING OLD AS Pold NEW AS Pnew
WHEN (EXISTS (SELECT 1 FROM Lecture L
              WHERE L.fk_lecturer = Pold.pk_lecturer))
( ROLLBACK WORK );
```

- The **child-side** triggers prevent inserting or updating a lecture referencing a non-existent lecturer.
- The **parent-side** triggers prevent deleting or changing a lecturer ID if at least one lecture still refers to it.
- The `ROLLBACK WORK` aborts the transaction on violation (`NO ACTION`-like).

2b) **Simulating `FOREIGN KEY (FkExam) REFERENCES Exam(PkExam) ON DELETE SET NULL` with triggers**

We enforce referential integrity procedurally:

- **Child-side (Student).** Block `INSERT`/`UPDATE` that sets `FkExam` to a non-existent `Exam.PkExam` (allow `NULL`).
- **Parent-side (Exam) on `DELETE`.** Before deleting an `Exam` row, set `Student.FkExam := NULL` for all referencing students (`ON DELETE SET NULL`).
- **Parent-side (Exam) on `UPDATE PkExam`.** Default is *NO ACTION*; block if referenced.

Child-side checks (INSERT/UPDATE on `Student`)

```
CREATE TRIGGER Student_FK_Check_Ins
BEFORE INSERT ON Student
WHEN ( NEW.FkExam IS NOT NULL AND
       NOT EXISTS (SELECT 1 FROM Exam E
                   WHERE E.PkExam = NEW.FkExam) )
( ROLLBACK WORK );

CREATE TRIGGER Student_FK_Check_Upd
BEFORE UPDATE OF FkExam ON Student
REFERENCING NEW AS Snew
WHEN ( Snew.FkExam IS NOT NULL AND
       NOT EXISTS (SELECT 1 FROM Exam E
                   WHERE E.PkExam = Snew.FkExam) )
( ROLLBACK WORK );
```

Parent-side action (DELETE on `Exam`) — simulate `ON DELETE SET NULL`

```
CREATE TRIGGER Exam_Delete_SetNull
BEFORE DELETE ON Exam
REFERENCING OLD AS Eold
(
  UPDATE Student
  SET FkExam = NULL
  WHERE FkExam = Eold.PkExam
);
```

Parent-side block (UPDATE of `PkExam`) — simulate `ON UPDATE NO ACTION`

```
CREATE TRIGGER Exam_Block_Update_PK
BEFORE UPDATE OF PkExam ON Exam
REFERENCING OLD AS Eold NEW AS Enew
WHEN ( EXISTS (SELECT 1 FROM Student S
               WHERE S.FkExam = Eold.PkExam) )
( ROLLBACK WORK );
```

Notes.

- Ensure `Student(FkExam)` allows `NULL`; otherwise `SET NULL` would fail.
- The `DELETE` trigger clears references before parent removal.
- Using `ROLLBACK WORK` follows the lecture's "block on violation" pattern.

Task 3

Using Assertions or Triggers because integrity constraints span multiple relations and not only one (Create Table Statement).

a) Using Assertions because constraints too general and not specific event.
e.g. no new orders when ...

U2109: each User must have SUM(c.CreditScore) >= -1000 creditpoints over all credits.

```
CREATE ASSERTION CreditLimitU2109
CHECK(NOT EXISTS(
    SELECT u.UID
    FROM User u
    JOIN Regulation r ON u.Country = r.Country
    JOIN Credit c ON u.UID = c.UID
    WHERE r.RegulationCode = 'U2109'
    GROUP BY u.UID
    HAVING SUM(c.CreditScore) < -1000))
DEFFERABLE INITIALLY DEFERRED
```

U2304: each User must have SUM(c.CreditScore) >= (-0.1 * MAX(b.BetAmount))

```
CREATE ASSERTION CreditLimitU2109
CHECK(NOT EXISTS(
    SELECT u.UID
    FROM User u
    JOIN Regulation r ON u.Country = r.Country
    JOIN Credit c ON u.UID = c.UID
    JOIN Bet b ON u.UID = b.UID
    WHERE r.RegulationCode = 'U2304'
    GROUP BY u.UID
    HAVING SUM(c.CreditScore) < (-0.1 * MAX(b.BetAmount))))
DEFFERABLE INITIALLY DEFERRED
```

b) Materialized View:

```
CREATE VIEW DailyOnlineTime
REFRESH FORCE ON COMMIT
BUILD DEFERRED AS
    SELECT u.UID, u.Name, u.Country, s.Date,
        SUM(s.Duration) AS dailyTotalTime
    FROM User u
    JOIN Regulation r ON u.Country = r.Country
```

```
    JOIN Session s ON u.UID = s.UID
    JOIN Bet b ON u.UID = b.UID
    WHERE r.RegulationCode = 'M5475'
        AND u.Age BETWEEN 14 AND 18
    GROUP BY u.Name, u.Country, s.Date;
```

c) Refresh Materliazized View on Demand:
Using additionaly a trigger to "manually" trigger the refresh of the view. First changing the view to "ON DEMAND". Second creating a trigger that refreshes the view when event happens.

```
CREATE VIEW DailyOnlineTime
REFRESH FORCE ON DEMAND
BUILD DEFERRED AS
    SELECT u.UID, u.Name, u.Country, s.Date,
        SUM(s.Duration) AS dailyTotalTime
    FROM User u
    JOIN Regulation r ON u.Country = r.Country
    JOIN Session s ON u.UID = s.UID
    JOIN Bet b ON u.UID = b.UID
    WHERE r.RegulationCode = 'M5475'
        AND u.Age BETWEEN 14 AND 18
    GROUP BY u.Name, u.Country, s.Date;


CREATE TRIGGER refreshView_onM4575_change
REFRESH FORCE ON DEMAND
AFTER INSERT OR DELETE OR UPDATE OF RegulationCode, Country
ON Regulation
REFERENCING OLD AS OldReg NEW AS NewReg
WHEN((NewReg.RegulationCode = 'M5475')
        OR (OldReg.RegulationCode = 'M5475'))
(CALL DBMS_MVIEW.REFRESH('DailyOnlineTime'));
```

Used own notation of view-refresh because no example or definition with refresh on slides.

Task 4

When we say something is **DEFERRABLE INITIALLY DEFERRED**, it means that the database will wait until the end of a transaction before checking certain rules, such as foreign key constraints. This is useful when two tables depend on each other. For example, a `Lecture` table and a `Lecturer` table. A lecture record might point to the lecturer who gives it, while the lecturer record might also depend on the lecture. If we try to insert both of these in the same transaction, the database might raise an error because when we insert the first one, the other does not yet exist.

If the rule is **NOT DEFERRABLE**, the database checks immediately, and the operation fails. If it is **DEFERRABLE INITIALLY IMMEDIATE**, we would need to manually tell the database to delay the check. However, if it is **DEFERRABLE INITIALLY DEFERRED**, the database automatically waits until the transaction ends to perform the checks. This allows both inserts to complete first, and the database only verifies the rules afterward, which helps prevent errors.