Examen final

Master 1 Ingénierie Mathématique

Bases de données relationnelles

Benjamin Auder

25 mars 2015

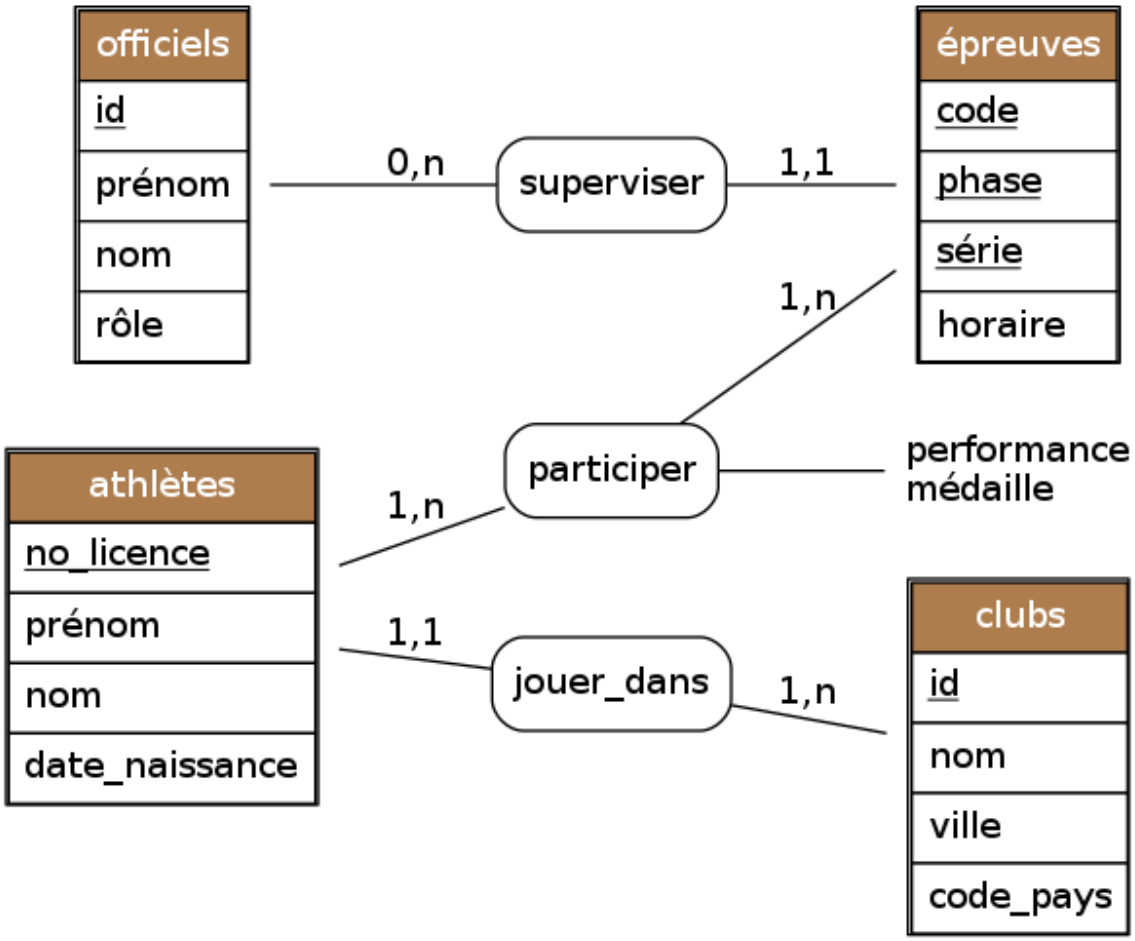
Exercice 0 [Environ 0% des points]

Tux peut-il voler ? Justifiez la réponse.

Solution :
Tux étant un manchot, en principe non car ces oiseaux ne volent pas : ils préfèrent nager. Des observations récentes remettent toutefois en cause cette affirmation.

Exercice 1 [Environ 30% des points]

Cet exercice analyse une modélisation simplifiée du déroulement d'un meeting d'athlétisme. Comme l'indique le schéma entités/associations ci-dessous, on enregistre les performances des athlètes sur des épreuves (saut en longueur, lancer de poids, 800m ...), qui sont supervisées par des "officiels". Ce dernier terme désigne toute personne impliquée dans la compétition mais n'y participant pas ; les tâches sont diverses : chronométrage, contrôle anti-dopage, vérification du bon déroulement d'une épreuve ...



Une épreuve a un code, et est divisée en phases : de "1/8$^{\text{ème}}$ de finale" à "finale". Chaque phase comporte plusieurs séries (par exemple les séries de la demi-finale du 100m). Ces trois valeurs constitue la clé de l'entité.

(a) Répondez aux questions selon les indications du schéma.
Justifiez (brièvement mais précisément).

1. Un athlète peut-il être affilié à plusieurs clubs ?
2. Peut-il y avoir des clubs sans athlètes ?
3. Peut-il y avoir des officiels qui ne supervisent pas d'épreuves ?

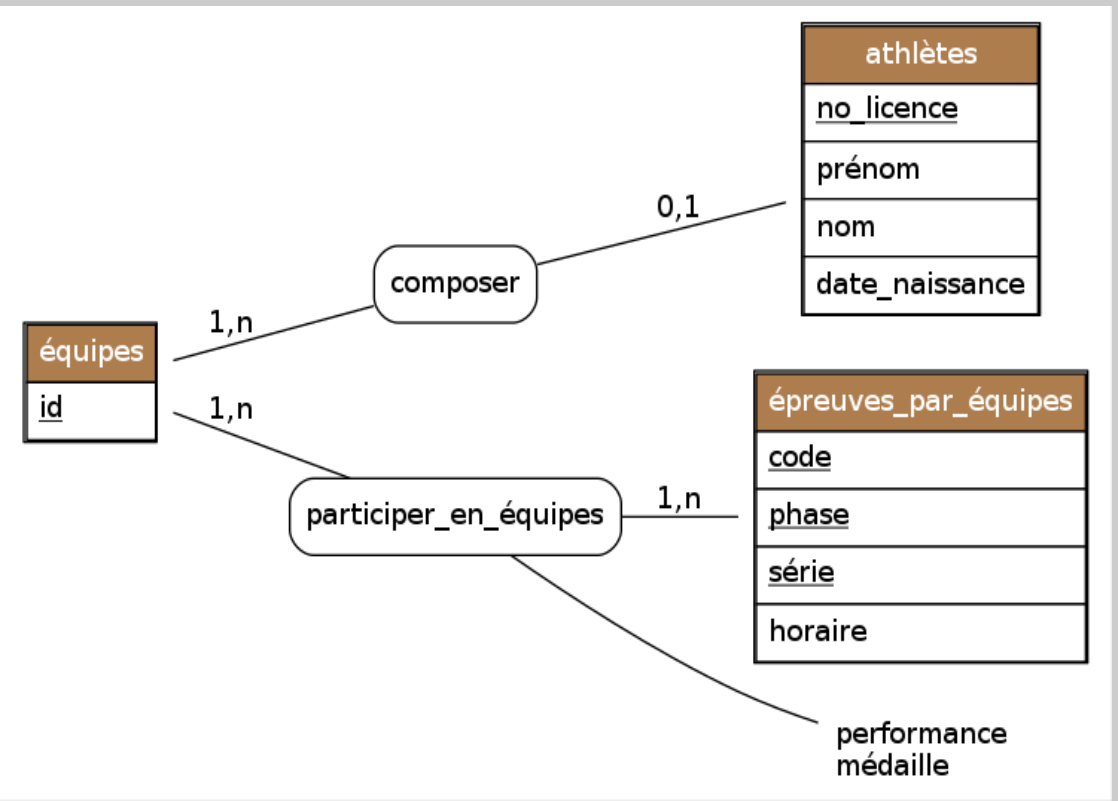
Solution :

1. Non : cardinalité maximale = 1 entre athlètes et jouer_dans.
2. Non : cardinalité minimale = 1 entre clubs et jouer_dans.
3. Oui : cardinalité minimale = 0 entre officiels et superviser.

(b) Comment étendre le schéma pour prendre en compte la participation d'équipes à certaines épreuves (le relais 4x100m par exemple) ? On ne prendra pas en compte cette extension pour les questions suivantes.

Solution :
Il faut créer une entité "équipes" qui sera en association avec *athlètes*. Ensuite le plus simple serait d'ajouter une connexion entre l'entité *équipes* et l'association *participer*, qui deviendrait alors ternaire. Il y aurait cependant une anomalie, l'association ne pouvant impliquer à la fois un athlète individuel et une équipe. On doit donc créer une nouvelle association "participer_en_équipes".

Finalement, un problème apparaît au niveau des cardinalités si l'on réutilise l'entité *épreuves* : 0,n ou 1,n vers *athlètes* et *équipes* ? Il faut alors créer une nouvelle entité "épreuves_par_équipes". En pratique cependant, on peut préférer ne conserver qu'une seule entité/table et utiliser une fonction trigger vérifiant la cohérence entre les participants et la nature d'une épreuve.



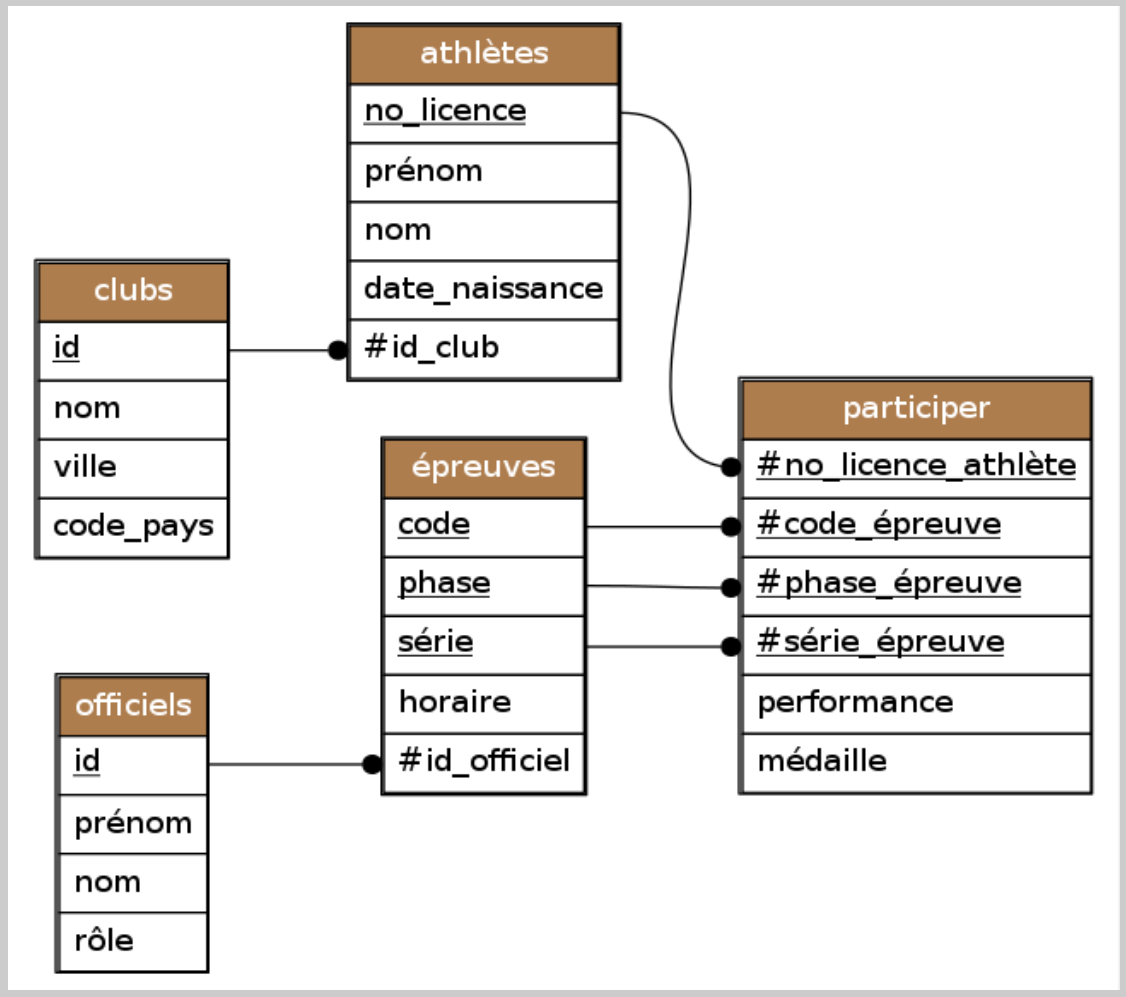
Remarque : le schéma n'assure pas que les membres d'une équipe soient du même club. Afin de pallier ce défaut on peut par exemple ajouter un trigger sur l'événement "insérer un athlète".

(c) Déterminez le schéma relationnel à partir du schéma E/A en commentant les opérations effectuées.

Solution :

Les associations avec cardinalités 1,1 d'un côté et 1,n de l'autre sont "aspirées" du côté 1,1 en tant que clés étrangères. Les attributs correspondants ne peuvent pas être NULL (il faudrait pour cela une cardinalité 0,1). L'association avec cardinalités 1,n de chaque côté devient une table, dont les attributs sont :

- les clés des entités en association (qui forment la clé de la nouvelle table),
- les attributs de l'association (performance et médaille).



(d) Déduisez-en les instructions SQL nécessaires à la création des tables. Note : afin d'alléger la réponse *vous pouvez omettre les tables* `clubs`, `officiels` *et* `athlètes`, mais les autres tables utiles doivent être indiquées.

Solution :

```
-- Note : les tailles des VARCHAR doivent être choisies
--        arbitrairement (et raisonnablement).

-- On crée d'abord les tables sans clés étrangères.
-- Ce n'est pas demandé, mais nécessaire en pratique.
CREATE TABLE clubs(
    id SERIAL PRIMARY KEY,
    -- Unicité du nom de club.
    nom VARCHAR(32) UNIQUE NOT NULL,
    ville VARCHAR(32) NOT NULL,
    code_pays CHAR(2) NOT NULL
```

```
);

CREATE TABLE officiels(
    id SERIAL PRIMARY KEY,
    prénom VARCHAR(32) NOT NULL,
    nom VARCHAR(32) NOT NULL,
    rôle VARCHAR(32)
);

-- Puis les tables comportant des clés étrangères
CREATE TABLE athlètes(
    -- La clé n'est pas de type SERIAL car le numéro
    -- de licence ne dépend pas de la compétition.
    no_licence INTEGER PRIMARY KEY,
    prénom VARCHAR(32) NOT NULL,
    nom VARCHAR(32) NOT NULL,
    date_naissance DATE NOT NULL,
    id_club INTEGER NOT NULL REFERENCES clubs(id)
);

/*
 **** À partir de là : correction de ce qui est demandé. ****
 */
-- On crée un type auxiliaire pour les phases d'épreuves.
CREATE TYPE PhaseEpreuve AS ENUM(
    '1/8', -- 1/8eme de finale ...etc
    '1/4',
    '1/2',
    'finale'
);
CREATE TABLE épreuves(
    -- On considère ici que le code d'une épreuve est universel.
    -- (Je ne sais pas si c'est le cas ; sans doute pas).
    code INTEGER,
    phase PhaseEpreuve,
    série INTEGER,
    -- La durée d'une épreuve est rarement précise.
    -- On n'enregistre donc que l'horaire de début.
    horaire TIME NOT NULL,
    id_officiel INTEGER REFERENCES officiels(id),
    PRIMARY KEY (code, phase, série)
);
```

```sql
-- Un type auxiliaire pour les médailles
CREATE TYPE Médaille AS ENUM(
    'bronze',
    'argent',
    'or'
);
-- Et enfin la table résultant de l'association "participer" :
CREATE TABLE participations(
    no_licence_athlète INTEGER REFERENCES athlètes(no_licence),
    code_épreuve INTEGER,
    phase_épreuve PhaseEpreuve,
    série_épreuve INTEGER,
    -- La performance peut être NULL si un problème survient
    -- pendant l'épreuve (blessure, abandon...)
    performance REAL,
    -- La médaille peut être NULL dans deux situations :
    -- pas de podium, ou épreuve qualificative.
    médaille Médaille,
    FOREIGN KEY (code_épreuve, phase_épreuve, série_épreuve)
        REFERENCES épreuves(code, phase, série),
    PRIMARY KEY(no_licence_athlète, code_épreuve,
                phase_épreuve, série_épreuve)
);
```

(e) Construisez une requête SQL renvoyant deux colonnes (nomClub, nbMédailles) correspondant respectivement aux noms des clubs et nombres de médailles (par club). La sortie doit être triée par valeurs de *nbMédailles* décroissantes, et exclure les clubs sans médailles.

Solution :

```sql
SELECT c.nom AS nomClub, count(*) AS nbMédailles
FROM participations AS p
    JOIN athlètes AS a
        ON a.no_licence = p.no_licence_athlète
    JOIN Clubs AS c
        ON c.id = a.id_club
WHERE p.médaille IS NOT NULL -- élimine les lignes sans médaille
GROUP BY c.nom
ORDER BY nbMédailles DESC;
```

Exercice 2 [Environ 40% des points]

Début d'une partie entre Magnus Carlsen et Levon Aronian au tournoi de Linares en 2007

Les questions de cet exercice portent sur la base de données "chess" contenant des résultats de parties d'échecs, ayant eu lieu dans des tournois. Vous n'avez pas besoin de savoir jouer pour répondre, les commentaires ci-après suffisent.

Description des tables :

```
joueurs (
    id SERIAL PRIMARY KEY,
    nom VARCHAR(32) UNIQUE NOT NULL,
    prénom VARCHAR(32) NOT NULL,
    -- Année de naissance.
    année INTEGER NOT NULL
)

-- Lors d'un tournoi chaque participant a au moins une fois
-- les blancs, et au moins une fois les noirs.
tournois (
    id SERIAL PRIMARY KEY,
    -- Souvent le nom de la ville dans laquelle le tournoi a lieu.
    nom VARCHAR(32) NOT NULL,
    année INTEGER NOT NULL,
    UNIQUE (nom, année)
)

parties (
    id_tournoi INTEGER REFERENCES tournois(id),
    -- Un tournoi est divisé en rondes. Pendant une ronde,
```

```
    -- chaque participant joue au plus une partie.
    ronde INTEGER,
    -- "blancs" et "noirs" désignent les identifiants des joueurs
    -- s'affrontant (respectivement avec les pièces blanches et noires).
    blancs INTEGER REFERENCES joueurs(id),
    noirs INTEGER REFERENCES joueurs(id),
    -- "elo_b" (resp. "elo_n") ci-dessous désigne le classement
    -- du joueur des blancs (resp. des noirs).
    elo_b INTEGER NOT NULL,
    elo_n INTEGER NOT NULL,
    -- Le résultat est un type énuméré à trois valeurs :
    --   1-0 = les blancs gagnent,
    --   0-1 = les noirs gagnent,
    --   1/2 = match nul.
    résultat ChessResult NOT NULL,
    PRIMARY KEY (id_tournoi, ronde, blancs, noirs)
)
```

Rappel : il y a deux possibilités de connexion à la base pour tester vos réponses.

1. ssh login@192.168.31.236 puis psql -d chess, ou
2. naviguer à l'adresse `http://192.168.31.236` depuis la salle d'examen.

Les réponses doivent être rendues au format électronique.

(a) Tournois (nom, année) dont le nom n'apparaît qu'une seule fois dans la table.

Solution :

```
-- Les clusters filtrés après le GROUP BY n'ont qu'un seul élément,
-- donc une seule année. Hélas PostgreSQL ne s'en rend pas compte.
-- Il faut donc explicitement utiliser une fonction d'agrégat sur
-- l'"ensemble" des années dans chaque groupe.

SELECT nom, max(année) AS année
FROM tournois
GROUP BY nom
HAVING count(*) = 1;
```

(b) Liste des participants (prénom, nom) au tournoi de Dortmund en 2005.

Solution :

```
-- Requête en trois temps :
--    1) recherche de l'identifiant ID du tournoi,
--    2) recherche des parties où id_tournoi = ID,
```

```
--    3) recherche des joueurs dont l'identifiant apparaît
--       dans la liste de ces parties.
-- Note : on peut aussi effectuer une jointure.

SELECT prénom, nom
FROM joueurs
WHERE id IN
    (SELECT blancs
    FROM parties
    WHERE id_tournoi IN
        (SELECT id
        FROM tournois
        WHERE nom='Dortmund' AND année=2005));
```

(**c**) Pourcentages de gains blancs, parties nulles et gains noirs sur l'ensemble des parties.

Solution :

```
WITH total AS
    (SELECT count(*) AS nbParties
    FROM parties)
SELECT résultat, round(100*count(*)::numeric / nbParties)
                   || '%' AS pourcentage
-- Le produit cartésien ici ne coûte rien car la table intermédiaire
-- "total" est réduite à un singleton.
FROM parties, total
-- nbParties doit apparaître dans le GROUP BY pour pouvoir être
-- utilisé dans le SELECT. C'est toutefois un peu artificiel.
GROUP BY résultat, nbParties;
```

Autre possibilité, esquissée dans une des copies :

```
-- Quelques redondances dans l'écriture, mais on y gagne
-- assez nettement en lisibilité, et évite une sous-requête.
SELECT round(sum(CASE WHEN résultat = '1-0' THEN 100.0 ELSE 0.0 END)
        / count(*)) || '%' AS gains_blancs,
       round(sum(CASE WHEN résultat = '1/2' THEN 100.0 ELSE 0.0 END)
        / count(*)) || '%' AS parties_nulles,
       round(sum(CASE WHEN résultat = '0-1' THEN 100.0 ELSE 0.0 END)
        / count(*)) || '%' AS gains_noirs
FROM parties;
```

(**d**) Moyenne des âges des participants aux tournois de (nom =) 'Linares' – toutes années confondues –, *relativement à la date du tournoi.*

Solution :

```
-- Attention : une moyenne directe après jointure des trois tables
-- renvoie un résultat potentiellement erroné, car les participants
-- ne jouent pas forcément toutes les rondes : ils peuvent être
-- malades ou abandonner avant la fin. De plus on peut envisager que
-- le nombre de rondes varie. C'est le cas du tournoi de Linares,
-- qui comptait par exemple 6 participants pour 10 rondes en 2001 et
-- 8 participants pour 14 rondes en 2007.
-- ***************************************************************
-- Il faut donc calculer la moyenne avec exactement une ligne par
-- joueur et par année : cela rend la requête également plus robuste
-- aux données manquantes.

SELECT avg(années.annéetournoi - années.annéejoueur)
FROM (
  SELECT j.année AS annéejoueur, joueurs_années.annéetournoi
  FROM joueurs j
  JOIN
    -- Sous-requête retournant les identifiants des joueurs
    -- avec les dates du tournoi, sans doublons.
    (SELECT DISTINCT j.id AS idjoueur, t.année AS annéetournoi
    FROM joueurs j
    JOIN parties p
      ON j.id = p.blancs
    JOIN tournois t
      ON p.id_tournoi = t.id
    WHERE t.nom = 'Linares') AS joueurs_années
  ON j.id = joueurs_années.idjoueur
) AS années;
```

(e) Meilleure performance (joueur A gagnant contre B, avec A moins fort que B) réalisée au tournoi de 'WijkaanZee' en 2001. Renvoyez les noms des joueurs, leurs classements et le résultat.

Solution :

```
-- Sous-requête renvoyant les parties
-- à performances potentielles ("pp").
WITH pp AS (
  SELECT blancs, noirs, résultat, elo_b, elo_n
  FROM parties p
  JOIN tournois t
    ON t.id = p.id_tournoi
```

```
  WHERE t.nom = 'WijkaanZee'
    AND t.année = 2001
    AND ((p.elo_b > p.elo_n AND p.résultat = '0-1')
      OR (p.elo_b < p.elo_n AND p.résultat = '1-0')))
SELECT jb.nom AS blancs, pp.elo_b,
       jn.nom AS noirs, pp.elo_n,
       pp.résultat
FROM pp
JOIN joueurs jb
  ON jb.id = pp.blancs
JOIN joueurs jn
  ON jn.id = pp.noirs
WHERE abs(pp.elo_b - pp.elo_n) =
  (select max(abs(pp.elo_b - pp.elo_n)) from pp);
```

(**f**) Nombre de points obtenus par Alexander Morozevich lors du tournoi de Sarajevo en 1999. On compte
1 pour les victoires,
1/2 pour les parties nulles,
0 pour les défaites.

Solution :

```
-- Une sous-requête pour éviter de rechercher l'identifiant du
-- joueur plusieurs fois.
WITH idJoueur AS
  (SELECT id
  FROM joueurs
  WHERE nom = 'Morozevich'),
-- Une sous-requête recherchant toutes les parties du tournoi
-- pour le joueur sélectionné.
partiesJoueur AS
  (SELECT blancs, noirs, résultat
  FROM parties
  WHERE
    id_tournoi IN
      (SELECT id FROM tournois WHERE nom='Sarajevo' AND année=1999)
    AND (blancs = any(SELECT * FROM idJoueur)
      OR noirs = any(SELECT * FROM idJoueur))),
-- Une sous-requête renvoyant une colonne de 0.5, avec autant
-- de lignes qu'il y a de parties nulles.
matchnuls AS
  (SELECT 0.5 AS points
  FROM partiesJoueur
```

```
  WHERE résultat = '1/2'),
-- Une sous-requête renvoyant une colonne de 1, avec autant
-- de lignes qu'il y a de parties gagnées.
gains AS
  (SELECT 1.0 AS points
  FROM partiesJoueur
  WHERE (blancs = any(SELECT * FROM idJoueur) AND résultat = '1-0')
    or (noirs = any(SELECT * FROM idJoueur) AND résultat = '0-1'))
-- Finalement on concatène les deux dernières tables,
-- et effectue la somme de toutes ses entrées.
SELECT sum(touslespoints.points) AS totalpoints
FROM
    (SELECT points
    FROM matchnuls
  UNION ALL
    SELECT points
    FROM gains) AS touslespoints;
```

(g) Joueurs (prénom, nom) ayant battu Garry Kasparov.

Solution :

```
  -- On peut gagner avec les blancs...
  SELECT prénom, nom
  FROM joueurs
  WHERE id IN
    (SELECT p.blancs
    FROM parties p
    JOIN joueurs j
      ON p.noirs = j.id
    WHERE p.résultat='1-0' AND j.nom='Kasparov')
UNION
  -- ...ou avec les noirs.
  SELECT prénom, nom
  FROM joueurs
  WHERE id IN
    (SELECT p.noirs
    FROM parties p
    JOIN joueurs j
      ON p.blancs = j.id
  WHERE p.résultat='0-1' AND j.nom='Kasparov');
```

(h) On définit le "nombre de Kasparov" comme suit :
0 pour Garry Kasparov lui-même,

1 pour les joueurs l'ayant battu,
2 pour les gens ayant battu quelqu'un qui a battu Kasparov (mais n'ayant pas réalisé cette performance eux-mêmes),
...etc,
$+\infty$ si aucun entier n'est valide.

Écrivez une requête qui calcule le nombre de Kasparov.

Solution :

```
-- WITH RECURSIVE est incontournable : on considère les joueurs
-- comme des noeuds dans un graphe orienté, un arc de J1 à J2
-- signifiant "J1 a battu J2"
-- ************************************************************
-- Les attributs de la requête se lisent comme suit :
--    prof = profondeur courante du graphe
--    idjoueur = identifiant d'un joueur battu au niveau nb
--    path = chemin courant [J1,J2,J3,...] signifiant
--           "J1 bat J2 bat J3 ..."
--    cycle = true si un cycle est détecté (auquel cas on n'ajoute
--            pas les joueurs correspondants)

WITH RECURSIVE nbKasparov(prof, idjoueur, path, cycle) AS (
    -- Initialisation : profondeur 0, graphe réduit au singleton
    -- du joueur de départ (Anand sur l'exemple)
    SELECT 0, id, ARRAY[id], false
    FROM joueurs
    WHERE nom = 'Anand'
  UNION (
    -- Requête récursive : les joueurs du niveau courant sont
    -- remplacés par l'ensemble de ceux qu'ils ont battus.
    -- La sous-requête suivante calcule cet ensemble, en
    -- détectant les cycles comme expliqué dans la documentation
    -- http://www.postgresql.org/docs/9.1/static/queries-with.html
    -- ************************************************************
    -- "rec" contient les paramètres du niveau courant
    WITH rec AS (select * from nbKasparov),
    perdants AS (
        SELECT rec.prof+1 as nextProf, p.noirs as nextIdjoueur,
               rec.path || p.noirs AS newPath,
               p.noirs = any(rec.path) AS newCycle
        FROM parties p
        JOIN rec
          ON rec.idjoueur = p.blancs
        WHERE résultat = '1-0'
      UNION
```

```
        SELECT rec.prof+1 as nextProf, p.blancs as nextIdjoueur,
               rec.path || p.blancs AS newPath,
               p.blancs = any(rec.path) AS newCycle
        FROM parties p
        JOIN rec
          on rec.idjoueur = p.noirs
        WHERE résultat = '0-1')
    -- On retient le résultat pour un appel récursif ultérieur
    -- seulement si Kasparov ne fait pas partie des joueurs battus
    -- (ou si l'ensemble est vide, ce qui arrive si le nombre de
    -- Kasparov est infini).
    SELECT nextProf, nextIdjoueur, newPath, newCycle
    FROM perdants
    WHERE (SELECT id FROM joueurs WHERE nom='Kasparov')
      NOT IN (SELECT nextIdjoueur FROM perdants)
 )
),
-- La requête récursive renvoie une liste de chemins dont certains
-- se termineraient par Kasparov, et éventuellement d'autres qui ne
-- sont pas valides. Il faut donc s'assurer que les identifiant du
(-- dernier niveau correspondent à des joueurs ayant battu Kasparov.
-- D'où l'intérêt de la sous-requête suivante.
winKaspy AS (
    SELECT p.blancs
    FROM parties p
    JOIN joueurs j
      ON p.noirs = j.id
    WHERE p.résultat='1-0' AND j.nom='Kasparov'
  UNION
    SELECT p.noirs
    FROM parties p
    JOIN joueurs j
      ON p.blancs = j.id
    WHERE p.résultat='0-1' AND j.nom='Kasparov'
)
-- Enfin, on choisit une ligne aléatoirement dans la table filtrée :
-- toutes les lignes sont valides.
SELECT prof+1 AS nombre_de_kasparov, path || idKasparov.id AS chemin
FROM nbKasparov, (SELECT id FROM joueurs WHERE nom='Kasparov')
                 AS idKasparov
WHERE idjoueur IN
  (SELECT * FROM winKaspy)
ORDER BY random()
```

```
LIMIT 1;
```

Note : vous aurez peut-être constaté que les classements Elo ne dépendent que du tournoi et du joueur. En effet les fluctuations de l'Elo ne sont pas prise en compte sur la durée d'un tournoi. La table *parties* est donc peu normalisée (même pas en 2NF). Cependant, les requêtes sont déjà assez complexes :-)

Exercice 3 [Environ 10% des points]

Cet exercice porte sur une petite base de données (non construite : à vous de copier-coller les instructions de création ci-dessous dans votre base), contenant des résultats d'étudiants à un examen dans plusieurs matières.

Voici les instructions de création des tables :

```
-- Les etudiants.
CREATE TABLE students(
    id INTEGER PRIMARY KEY,
    firstname VARCHAR(32) NOT NULL,
    lastname VARCHAR(32) NOT NULL
);

-- Les cours.
CREATE TABLE courses(
    code INTEGER PRIMARY KEY,
    name VARCHAR(32) UNIQUE NOT NULL
);

-- Les notes, comprises entre 0 et 20.
-- **********************************
-- Remarque : on considère que la note peut être nulle, pour distinguer
-- par exemple un étudiant autorisé à compenser la matière d'un autre
-- oblige de repasser le module.
CREATE TABLE grades(
    student_id INTEGER REFERENCES students(id),
    course_code INTEGER REFERENCES courses(code),
    grade NUMERIC CHECK (grade >= 0 AND grade <= 20),
    PRIMARY KEY (student_id, course_code)
);
```

Quelques instructions possibles pour insérer des données :

```
INSERT INTO students VALUES
    (1, 'Maria', 'McConnell'),(2, 'Bernard', 'Young'),
```

```
    (3, 'Anthony', 'McCann'),(4, 'Lauren', 'Hill');

INSERT INTO courses VALUES
    (1, 'History'),(2, 'Mathematics'),
    (3, 'Chinese'),(4, 'Chemistry');

INSERT INTO grades VALUES
    -- Je laisse quelques lignes non remplies,
    -- pour tester le trigger de la premiere question.
    (1, 1, 10.0),(1, 2, 15.0),
    (2, 2, 12.0),(2, 3, 7.0),
    (3, 3, 18.0),(3, 4, 9.0),
    (4, 4, 13.0),(4, 1, 14.0);
```

(J'utilise des termes anglais pour éviter les accents, qui passent mal lors du copier-coller depuis un document PDF.)

Les réponses doivent être rendues au format électronique.

(a) Requête SQL renvoyant deux colonnes : les notes moyennes par matière, et les noms des matières correspondantes. Sur l'exemple elle doit retourner 12/20 en histoire, 13.5 en maths, 12.5 en chinois et 11.0 en chimie.

Solution :

```
-- Inutile d'explicitement éliminer les valeurs NULL :
-- PostgreSQL calcule la moyenne sur les valeurs non NULL.
SELECT c.name AS course_name, round(avg(g.grade), 2) AS mean
FROM grades g
JOIN courses c
    ON g.course_code = c.code
GROUP BY c.code, c.name;
```

(b) On ajoute – comme indiqué ci-après – une colonne à la table *courses*, contenant les moyennes obtenues par matières.

```
ALTER TABLE courses
    ADD COLUMN mean NUMERIC;
```

Implémentez une fonction trigger nommée "computeMean" qui, à l'insertion/suppression ou mise à jour d'une note, (re)calcule la moyenne de la matière concernée. Après l'opération, la valeur moyenne reportée dans la table *courses* doit être correcte.

Solution :

```
CREATE OR REPLACE FUNCTION computeMean() RETURNS trigger AS $$
<< outerblock >>
```

```
DECLARE
    new_mean NUMERIC := NULL;
    course_code INTEGER := 0;
    line grades%ROWTYPE;
BEGIN
    -- Affectation du code de matière,
    -- en fonction du type d'opération.
    CASE TG_OP
    WHEN 'DELETE' THEN
        course_code := OLD.course_code;
        line := OLD;
    WHEN 'INSERT', 'UPDATE' THEN
        course_code := NEW.course_code;
        line := NEW;
    END CASE;
    -- Calcul de la (nouvelle) moyenne.
    SELECT avg(grade) INTO new_mean
        FROM grades g
        WHERE g.course_code = outerblock.course_code;
    -- Mise à jour de la moyenne dans la table 'courses'.
    UPDATE courses
        SET mean = new_mean
        WHERE code = outerblock.course_code;
    RETURN line;
END;
$$ language plpgsql;

CREATE TRIGGER computeMean_trigger
    -- Déclencher le trigger après l'insertion / mise à jour
    -- / suppression permet un recalcul (coûteux mais) simple.
    -- Pour une mise à jour incrémentale de complexité O(1) au
    -- lieu de O(n), il serait préférable de déclencher le trigger
    -- avant l'opération. L'écriture de la fonction serait alors
    -- nettement plus laborieuse : cf. TP 4, question C.4.
    AFTER INSERT OR UPDATE OR DELETE ON grades
    FOR EACH ROW
    EXECUTE PROCEDURE computeMean();
```

Exercice 4 [Environ 20% des points]

Cette exercice considère une table unique T contenant des données relatives à des cours de ski ou surf, dans une station de sports d'hiver. La table T est constituée de 6 attributs :

nom_moniteur	[Nm]	: le nom d'un moniteur
niveau	[N]	: le niveau d'un cours,
materiel	[M]	: le type de matériel utilisé (ski ou surf),
lieu_depart_arrivee	[L]	: lieu du cours,
heure_debut	[Hd]	: horaire de début,
heure_fin	[Hf]	: horaire de fin.

On suppose les dépendances fonctionnelles suivantes :

Nm $\rightarrow$ M
N $\rightarrow$ L
Nm, N $\rightarrow$ L, Hd
N, Hd $\rightarrow$ Hf

(a) Donnez en justifiant une clé minimale de la relation. Y en a-t-il d'autres ?
On considère cette clé pour la question suivante.

Solution :
Les attributs Nm et N ne se retrouvent jamais en partie droite d'une DF (dépendance fonctionnelle). Ils ne peuvent donc pas être déduits d'autres attributs. Ils font donc nécessairement partie d'une potentielle clé.

D'autre part, les trois premières DFs indiquent que Nm et N déterminent M, L et Hd. Finalement N et Hd déterminent Hf, donc le couple (Nm, N) constitue l'unique clé minimale de la relation.

(b) Quelles formes normales vérifie la table T ? Peut-on aller plus loin (par application des théorèmes vus à la deuxième séance) ? Décrivez les étapes nécessaires.

Solution :
On est seulement en première forme normale (1NF), car M et L ne dépendent que partiellement de la clé. En effet pour être en 2NF il faut que tous les attributs non identifiants dépendent pleinement de la clé.

On peut faire mieux, car une relation peut toujours être décomposée jusqu'en 3^{eme} forme normale au moins. Cependant on ne peut pas préserver toutes les dépendances, n'étant pas dans les conditions d'application de la décomposition de Jorma Rissanen. À défaut, appliquons le théorème de Heath avec X = {Nm, N}, Y = {L, Hd} et Z = {M, Hf} : on obtient d'une part (disons T_1)

N $\rightarrow$ L
Nm, N $\rightarrow$ Hd (Nm,N $\rightarrow$ L est redondante, déduite de N $\rightarrow$ L)

Et, disons T_2 :

Nm $\rightarrow$ M
Nm, N $\rightarrow$ Hf (par transitivité + augmentation...)

Les attributs L et M ne dépendent encore que partiellement des clés. Il faut poursuivre : dans T_1 en prenant $X_1 = \{\text{N}\}$, $Y_1 = \{\text{L}\}$, $Z_1 = \{\text{Nm, Hd}\}$, et dans T_2 en choisissant $X_2 = \{\text{Nm}\}$, $Y_2 = \{\text{M}\}$, $Z_2 = \{\text{N, Hf}\}$. On obtient au final :

$\underline{\text{N}}$	$\rightarrow$	L	$[T_{1,1}]$
$\underline{\text{Nm, N}}$	$\rightarrow$	Hd	$[T_{1,2}]$
$\underline{\text{Nm}}$	$\rightarrow$	M	$[T_{2,1}]$
$\underline{\text{Nm, N}}$	$\rightarrow$	Hf	$[T_{2,2}]$

On est arrivé en 2NF, et en fait même en BCNF (vérification triviale).

(En pratique on garderait très probablement Nm, N, Hd et Hf dans la même table.)

(c) Lors de l'insertion d'un cours, il faut s'assurer qu'il est compatible avec les autres donnés par le moniteur. En particulier les plages horaires ne peuvent pas se chevaucher (les cours ont lieu tous les jours de la semaine). On considère que :

— deux cours peuvent s'enchaîner sans pause s'ils partent du même endroit,
— mais doivent être espacés d'au moins 30min dans le cas contraire.

L'objectif de cette question est de vérifier ces contraintes, qui devront être ajoutés directement dans la table (pas besoin d'un trigger).

Voici l'instruction de création de la table T :

```
CREATE TABLE cours_ski(
    nom_moniteur VARCHAR(32),
    niveau INTEGER,
    PRIMARY KEY(nom_moniteur, niveau),
    materiel VARCHAR(8) NOT NULL
    CONSTRAINT checkMateriel CHECK
        (materiel = any(ARRAY['ski', 'surf'])),
    lieu_depart_arrivee VARCHAR(32) NOT NULL,
    heure_debut TIME NOT NULL,
    heure_fin TIME NOT NULL,
    -- Un cours doit durer au moins 30 minutes.
    CONSTRAINT checkTime CHECK (heure_fin - heure_debut > '0:30')
);
```

Et quelques instructions d'insertion de données possibles, pour les tests :

```
INSERT INTO cours_ski VALUES(
  'Jade', 1, 'surf', 'pistes debutants', '11:00', '12:00');
INSERT INTO cours_ski VALUES(
  'Milla', 2, 'ski', 'telesiege chamois', '10:00', '12:00');
INSERT INTO cours_ski VALUES(
  'Andrew', 1, 'ski', 'pistes debutants', '13:00', '14:00');
```

```
INSERT INTO cours_ski VALUES(
  'Jade', 2, 'surf', 'teleski dromadaire', '8:45', '10:45'); -- echec
INSERT INTO cours_ski VALUES(
  'Jade', 1, 'surf', 'pistes debutants', '10:00', '11:00'); -- OK
INSERT INTO cours_ski VALUES(
  'Milla', 3, 'ski', 'telesiege chamois', '9:30', '11:00'); -- echec
INSERT INTO cours_ski VALUES(
  'Andrew', 2, 'ski', 'teleski dromadaire', '14:45', '16:45'); -- OK
```

Rappel : la réponse à cette question **doit être rendue au format électronique**.

Solution :

On commence par implémenter une fonction booléenne prenant en argument un nom de moniteur, un lieu de départ/arrivée ainsi que des horaires de début et fin. Cette fonction retourne "true" en cas d'insertion autorisée, et "false" sinon.

```
-- Fonction auxiliaire qui vérifie si un temps t est à l'extérieur
-- de l'intervalle heure_debut - margin, heure_fin + margin.
-- Ce n'est pas nécessaire, mais rend la fonction checkNonOverlap()
-- plus lisible.
CREATE OR REPLACE FUNCTION checkOverlap_time(t TIME,
    heure_debut TIME, heure_fin TIME, margin INTERVAL)
        RETURNS boolean AS $$
BEGIN
  RETURN t BETWEEN heure_debut - margin AND heure_fin + margin;
END;
$$ language plpgsql;

-- La fonction utilisée comme contrainte.
CREATE OR REPLACE FUNCTION checkNonOverlap(nom_moniteur VARCHAR(32),
    niveau INTEGER, lieu_depart_arrivee VARCHAR(32),
    heure_debut TIME, heure_fin TIME) RETURNS boolean AS $$
DECLARE
  o cours_ski%ROWTYPE;
BEGIN
  FOR o IN
    SELECT *
    FROM cours_ski c
    WHERE c.nom_moniteur = checkNonOverlap.nom_moniteur
      -- On ignore la ligne identique à l'argument.
      AND c.niveau != checkNonOverlap.niveau
  LOOP
    IF o.lieu_depart_arrivee = checkNonOverlap.lieu_depart_arrivee
```

```
      -- Test de chevauchement avec un autre cours donné par
      -- le même moniteur et se déroulant dans le même lieu.
      AND (checkOverlap_time(o.heure_debut,
          checkNonOverlap.heure_debut, checkNonOverlap.heure_fin,
          '0')
        OR checkOverlap_time(o.heure_fin,
          checkNonOverlap.heure_debut, checkNonOverlap.heure_fin,
          '0'))
      THEN RETURN false;
    END IF;
    IF o.lieu_depart_arrivee <> checkNonOverlap.lieu_depart_arrivee
      -- Test de chevauchement avec un autre cours donné par
      -- le même moniteur démarrant dans un autre endroit.
      AND (checkOverlap_time(o.heure_debut,
          checkNonOverlap.heure_debut, checkNonOverlap.heure_fin,
          '30 minutes')
        OR checkOverlap_time(o.heure_fin,
          checkNonOverlap.heure_debut, checkNonOverlap.heure_fin,
          '30 minutes'))
      THEN RETURN false;
    END IF;
        END LOOP;
    RETURN true;
END;
$$ LANGUAGE plpgsql;
```

Ensuite, il suffit de modifier la table en ajoutant la contrainte personnalisée.

```
ALTER TABLE cours_ski
  ADD CONSTRAINT check_non_overlap CHECK
    (checkNonOverlap(nom_moniteur, niveau, lieu_depart_arrivee,
                     heure_debut, heure_fin));
```