Bases de Données II, Mons, 5 juin 2019

NOM + PRÉNOM :

Orientation + Année :

Cet examen contient 13 questions.

Afin de lutter contre la fraude aux compteurs kilométriques, tout garagiste qui effectue une intervention sur une voiture ou une camionnette est tenu d'en transmettre le kilométrage. Ces informations sont centralisées dans un document XML :

- Chaque garage porte un identifiant unique (`gid`) et est concessionnaire pour une (et une seule) marque de voiture. Les garages sont groupés par marque. On stocke le nom (`gnom`) et la ville de chaque garage.
- Chaque voiture porte un numéro de châssis unique. On stocke la marque et le modèle de chaque voiture.
- Chaque élément `<i gid="..." km="..." date="..."/>` représente une intervention sur une voiture. Les interventions sont groupées par voiture. Pour chaque intervention, on stocke le garage qui a effectué l'intervention (`gid`), le kilométrage au moment de l'intervention (`km`) et la date de l'intervention (dans un format yymmdd). Si un garage effectue plusieurs interventions sur une même voiture pendant une même journée, seul le kilométrage au moment de la première intervention de la journée sera stocké. Cependant, si une voiture subit des interventions dans deux garages différents lors d'une même journée, chaque garage est tenu de transmettre le kilométrage.

La DTD est incluse au début du document XML de la figure 1.

Pour les questions 1 à 4, évitez, si possible, l'usage des axes *parent* et *ancestor*. **Il n'est pas permis d'utiliser des fonctions d'agrégation, telles que count, max, min...**

Question 1 Écrivez une expression XPath qui renvoie les marques (attribut `mnom`) qui ont des concessionnaires dans plus d'une ville, dont Alost, mais qui n'ont pas de concessionnaire à Bruxelles. Pour le document de la figure 1, la réponse ne contient qu'une seule marque :

```
mnom="Toyota"
```

.../5

```
//garages/marque[garage/@ville="Alost"]
                [garage/@ville!="Alost"]
                [not(garage/@ville="Bruxelles")]/@mnom
```

Remarquez que les conditions `garage/@ville="Alost"` et `garage/@ville!="Alost"` ne sont pas contradictoires à cause de la sémantique "existentielle".

Question 2 Écrivez une expression XPath qui renvoie les garages Peugeot ayant effectué une intervention sur une voiture d'une autre marque. Pour le document de la figure 1, la réponse est comme suit :

```
<garage gid="P1" gnom="ErVeeDee" ville="Alost"/>
```

.../5

```
//garages/marque[@mnom="Peugeot"]/garage[@gid=//voiture[@marque!="Peugeot"]/i/@gid]
```

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE carpass [
<!ELEMENT carpass (garages, voitures)*>
<!ELEMENT garages (marque)*>
<!ELEMENT marque (garage)*>
<!ELEMENT garage (#PCDATA)>
<!ELEMENT voitures (voiture)*>
<!ELEMENT voiture (i)*>
<!ELEMENT i (#PCDATA)>
<!ATTLIST marque mnom CDATA #REQUIRED>
<!ATTLIST garage gid CDATA #REQUIRED>
<!ATTLIST garage gnom CDATA #REQUIRED>
<!ATTLIST garage ville CDATA #REQUIRED>
<!ATTLIST voiture chassis CDATA #REQUIRED>
<!ATTLIST voiture marque CDATA #REQUIRED>
<!ATTLIST voiture modele CDATA #REQUIRED>
<!ATTLIST i gid CDATA #REQUIRED>
<!ATTLIST i km CDATA #REQUIRED>
<!ATTLIST i date CDATA #REQUIRED>
]>
<carpass>
<garages>
   <marque mnom="Peugeot">
      <garage gid="P1" gnom="ErVeeDee" ville="Alost"/>
      <garage gid="P2" gnom="Ciac" ville="Alost"/>
   </marque>
   <marque mnom="Renault">
      <garage gid="R1" gnom="Valckenier" ville="Alost"/>
      <garage gid="R2" gnom="Meiser" ville="Bruxelles"/>
   </marque>
   <marque mnom="Toyota">
      <garage gid="T1" gnom="New Energy" ville="Alost"/>
      <garage gid="T2" gnom="Serriez" ville="Mons"/>
      <garage gid="T3" gnom="Merckx" ville="Charleroi"/>
      <garage gid="T4" gnom="Castus" ville="Charleroi"/>
   </marque>
   <marque mnom="Volkswagen">
      <garage gid="V1" gnom="Sawa" ville="Bruxelles"/>
      <garage gid="V2" gnom="Carlier Motor" ville="Mons"/>
   </marque>
</garages>
<voitures>
   <voiture chassis="222" marque="Peugeot" modele="307">
      <i gid="P1" km="11000" date="180820"/>
      <i gid="R1" km="11005" date="180820"/>
      <i gid="R1" km="11099" date="180823"/>
      <i gid="P1" km="999" date="170220"/>
      <i gid="P2" km="12050" date="180827"/>
      <i gid="T4" km="12777" date="180829"/>
   </voiture>
   <voiture chassis="333" marque="Renault" modele="Megane">
      <i gid="R1" km="22000" date="180823"/>
      <i gid="P1" km="22033" date="180823"/>
      <i gid="R2" km="22099" date="180827"/>
      <i gid="T4" km="22555" date="180829"/>
   </voiture>
</voitures>
</carpass>
```

FIGURE 1 – Les garages et leurs interventions.

```
<?xml version="1.0" encoding="utf-16"?>
<OUTPUT>
   <GARAGE gnom="ErVeeDee" ville="Alost" mnom="Peugeot">
      <VOITURE chassis="333" marque="Renault" />
   </GARAGE><GARAGE gnom="Ciac" ville="Alost" mnom="Peugeot" />
   <GARAGE gnom="Valckenier" ville="Alost" mnom="Renault">
      <VOITURE chassis="222" marque="Peugeot" />
   </GARAGE>
   <GARAGE gnom="Meiser" ville="Bruxelles" mnom="Renault" />
   <GARAGE gnom="New Energy" ville="Alost" mnom="Toyota" />
   <GARAGE gnom="Serriez" ville="Mons" mnom="Toyota" />
   <GARAGE gnom="Merckx" ville="Charleroi" mnom="Toyota" />
   <GARAGE gnom="Castus" ville="Charleroi" mnom="Toyota">
      <VOITURE chassis="222" marque="Peugeot" />
      <VOITURE chassis="333" marque="Renault" />
   </GARAGE>
   <GARAGE gnom="Sawa" ville="Bruxelles" mnom="Volkswagen" />
   <GARAGE gnom="Carlier Motor" ville="Mons" mnom="Volkswagen" />
</OUTPUT>
```

FIGURE 2 – Output du programme XSLT de la question 5.

Question 3 Écrivez une expression XPath qui renvoie les villes (attribut `ville`) avec deux (ou plusieurs) concessionnaires de marques différentes. Pour le document de la figure 1, la réponse est comme suit :

```
ville="Alost"
ville="Bruxelles"
ville="Mons"
```

Puisque tous les concessionnaires de Charleroi sont de la même marque, cette ville n'apparaît pas dans la réponse.

.../5

```
//@ville[.=preceding::marque//@ville]
```

Si l'on souhaite éliminer des doublons :

```
//@ville[.=preceding::marque//@ville][not(.=following::marque//@ville)]
```

Question 4 Écrivez une expression XPath qui renvoie les voitures (attribut `chassis`) pour lesquelles les interventions ne sont pas listées dans l'ordre chronologique. Pour le document de la figure 1, la réponse est comme suit :

```
chassis="222"
```

En fait, pour la voiture 222, l'intervention du 23/08/2018 est listée avant celle du 20/02/2017.

.../5

```
//voiture[i[@date>following-sibling::i/@date]]/@chassis
```

Il est important d'imbriquer les crochets, car `following-sibling` doit être évalué par rapport aux interventions `i` appartenant à une même voiture.

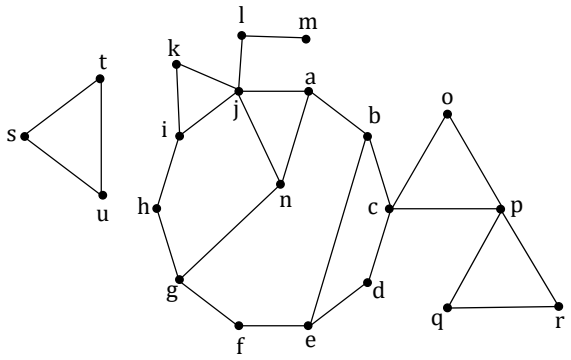


FIGURE 3 – Graphe non-orienté

Question 5 Écrivez un programme XSLT qui liste, pour chaque garage, les voitures qui ont subi une intervention au garage *et qui ne sont pas de la marque du garage*. Le format est celui de la figure 2.

L'ordre dans lequel les garages ou les voiture sont affichés n'a pas d'importance.

La position des blancs et retours à la ligne n'a pas d'importance. Le programme ne peut pas contenir les balises `xsl:for-each`, `xsl:if` ou `xsl:with-param`.

.../5

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <OUTPUT>
    <xsl:apply-templates select="//garage"/>
  </OUTPUT>
</xsl:template>

<xsl:template match="garage">
  <GARAGE>
    <xsl:copy-of select="@gnom"/>
    <xsl:copy-of select="@ville"/>
    <xsl:copy-of select="parent::marque/@mnom"/>
    <xsl:apply-templates select="//voiture[i/@gid=current()/@gid]
                                          [@marque!=current()/parent::marque/@mnom]"/>
  </GARAGE>
</xsl:template>

<xsl:template match="voiture">
  <VOITURE>
    <xsl:copy-of select="@chassis"/>
    <xsl:copy-of select="@marque"/>
  </VOITURE>
</xsl:template>

</xsl:stylesheet>
```

Question 6 Un *graphe non-orienté* est un couple $G = (V, E)$ avec V un ensemble fini de *sommets* et E un ensemble de paires $\{u, v\}$ avec $u, v \in V$ $(u \neq v)$. Chaque élément de E est appelé une *arête* du graphe.

Dans une base de données, une relation binaire $R[A, B]$ est utilisée pour stocker un graphe. Si $\{u, v\} \in E$, alors on stockera $R(u, v)$ ou $R(v, u)$ dans la base de données (pas besoin de stocker les deux). On supposera que chaque sommet fait partie d'au moins une arête. Il n'y a donc pas besoin d'avoir une relation pour V : l'ensemble V est la réponse à la requête SPJRUD $\pi_A R \cup \pi_B R$.

Un *cycle élémentaire* dans G est une séquence $\langle u_0, u_1, \ldots, u_{n-1} \rangle$ de sommets distincts tels que $n \geq 3$ et pour chaque $i \in \{0, 1, \ldots, n-1\}$, $\{u_i, u_{(i+1) \mod n}\}$ est une arête du graphe (donc $\{u_{n-1}, u_0\}$ est une arête). La *longueur* de ce cycle est de n.

Par exemple, dans le graphe de la Figure 3, $\langle a, b, c, d, e, f, g, h, i, j \rangle$ est un cycle élémentaire de longueur 10 dans G. Une *corde* d'un cycle élémentaire est une arête reliant deux sommets non-adjacents du cycle. Évidemment, un cycle de longueur 3 n'a pas de corde.

Par exemple, dans le graphe de la Figure 3, $\{b, e\}$ est la seule corde du cycle $\langle a, b, c, d, e, f, g, h, i, j \rangle$.

Le cycle élémentaire $\langle a, b, c, d, e, f, g, n, j \rangle$ a deux cordes, à savoir $\{a, n\}$ et $\{b, e\}$.

Un *trou* est un cycle élémentaire de longueur ≥ 4 qui n'a pas de corde. Le graphe de la Figure 3 contient plusieurs trous, par exemple, $\langle g, h, i, j, n \rangle$ et $\langle a, b, e, f, g, h, i, j \rangle$.

Écrivez un programme en Datalog avec $\neq$ et négation stratifiée qui renvoie $Answer(u)$ pour chaque sommet u qui appartient à un trou. Commencez votre réponse avec **une explication, en français,** de la stratégie utilisée pour trouver les sommets de la réponse.

Hint : un sommet u appartient à un trou s'il possède deux voisins non adjacents qui sont connectés par un chemin (pas nécessairement élémentaire) ne passant pas par u.

.../5

On fera appel aux prédicats IDB suivants :

- `pathWithout(X,Y,Z)` exprimera l'existence d'un chemin non-orienté entre X et Y qui n'utilise pas Z ;
- `adjacent` sera la fermeture symétrique de `E`.

Avec ces deux prédicats, le *Hint* devient facile à exprimer.

```
adjacent(X,Y) :- E(X,Y).
adjacent(X,Y) :- E(Y,X).
pathWithout(X,Y,Z) :- adjacent(X,Y), X != Z, Y != Z, adjacent(Z,U).
pathWithout(X,Y,Z) :- pathWithout(X,U,Z), E(U,Y), Y != Z.
inHole(Z) :- adjacent(X,Z), adjacent(Y,Z), X != Y, not adjacent(X,Y), pathWithout(X,Y,Z).
```

Notez que la troisième règle est *safe* grâce à l'atome `adjacent(Z,U)`. Une stratification du programme est (P_0, P_1) avec P_1 contenant la règle pour `InHole`, et P_0 contenant les autres règles.

Dans la règle pour `InHole`, on peut remplacer "`not adjacent(X,Y)`" par "`not E(X,Y), not E(Y,X)`", ce qui nous donne un programme en *semipositive Datalog*.

Plusieurs étudiants ont proposé de calculer d'abord un prédicat `PathWith(X,Y,Z)` qui est vrai s'il existe un chemin entre `X` et `Y` qui contient `Z`, puis d'ajouter une règle :

```
inHole(Z) :- adjacent(X,Z), adjacent(Y,Z), X != Y, not adjacent(X,Y),
             pathWith(X,Y,U), not pathWith(X,Y,Z).
```

Cette règle n'a peu de sens : si `adjacent(X,Z)` et `adjacent(Y,Z)` sont vrais, `pathWith(X,Y,Z)` sera aussi vrai, et donc `not pathWith(X,Y,Z)` sera faux. Notez la différence :

- `pathWithout(X,Y,Z)` exprime qu'il existe un chemin entre `X` et `Y` qui n'utilise pas `Z` ;
- `not pathWith(X,Y,Z)` exprime qu'aucun chemin entre `X` et `Y` n'utilise `Z`.

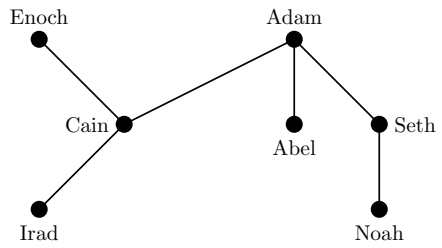


FIGURE 4 – Arbre

Question 7 Un *arbre* est un graphe non-orienté, connexe et acyclique. Comme pour la question 6, on suppose que l'on utilise une relation binaire $R[A, B]$ pour stocker un arbre. Notez : dans un arbre, chaque deux sommets sont reliés par un chemin élémentaire unique. Écrivez une requête pour le prédicat ternaire *PassePar*, tel que $PassePar(u, v, r)$ est vrai si u, v, r sont trois sommets distincts tels que le chemin entre u et v passe par r. Par exemple, pour l'arbre de la Figure 4, on a $PassePar(Cain, Noah, Adam)$, mais on n'a pas $PassePar(Cain, Noah, Enoch)$.

La requête doit être exprimée soit en algèbre relationnelle avec un opérateur de point fixe (i.e., SPJRUD+FP), soit en calcul relationnel avec un opérateur de point fixe. Le choix parmi ces deux langages est libre. Dans le calcul, vous pouvez utiliser le prédicat $\neq$ pour l'inégalité ; dans l'algèbre, vous pouvez utiliser $\sigma_{A \neq B} E$, la sélection avec inégalité.

Ajoutez une explication en français au cas où votre solution est compliquée. Notez que la requête doit renvoyer un ensemble de triples (u, v, r), mais pas le nom "*PassePar*".

.../10

Rappelons que la question 6 stipule que pour chaque fait $E(a, b)$, on aura $a \neq b$.

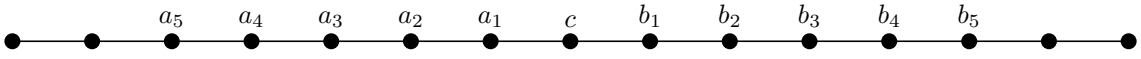
Introduisons $A(p, q)$ comme une abréviation pour $E(p, q) \vee E(q, p)$, pour toutes les variables p, q. La requête demandée est alors :

$$[\mathbf{fp}_{\Delta:u,v,r}\,((A(u,r) \wedge A(r,v) \wedge u \neq v)) \vee \exists z(\Delta(u,z,r) \wedge A(z,v) \wedge u \neq v \wedge v \neq r) \vee \Delta(v,u,r)](u,v,r)$$

Par induction, on peut vérifier que pour tout fait $\Delta(u, v, r)$ ajouté dans le calcul du point fixe, on aura $u \neq v$, $u \neq r$, $v \neq r$. Notez que pour tout fait $\Delta(u, v, r)$ ajouté, on ajoutera aussi $\Delta(v, u, r)$. Puisque la requête ne contient pas de négation $\neg$, un point fixe sera atteint à partir de $\Delta^0 = \emptyset$, pour toute base de données.

Notez que la deuxième occurrence de la condition $u \neq v$ est redondante à cause de l'acyclicité du graphe.

Pour comprendre le fonctionnement de cette requête, considérez le chemin suivant :



Regardons les tuples de la forme $(\cdot, \cdot, c)$ dans le calcul du point fixe, en commençant par $\Delta^0 = \emptyset$:

- Δ^1 contiendra (a_1, b_1, c) et (b_1, a_1, c).
- Δ^2 ajoutera (a_1, b_2, c) et (b_1, a_2, c).
- Δ^3 ajoutera (a_1, b_3, c) et (b_1, a_3, c), ainsi que (b_2, a_1, c) et (a_2, b_1, c).
- Δ^4 ajoutera (a_1, b_4, c), (b_1, a_4, c), (b_2, a_2, c), (a_2, b_2, c), ainsi que (b_3, a_1, c) et (a_3, b_1, c).
- Δ^5 ajoutera (a_1, b_5, c), (b_1, a_5, c), (b_2, a_3, c), (a_2, b_3, c), (b_3, a_2, c), (a_3, b_2, c), ainsi que (b_4, a_1, c) et (a_4, b_1, c).
- ...

Il est maintenant facile de voir que pour tous les sommets distincts d, e, c : si c est sur le chemin entre d et e, alors (d, e, c) appartient à Δ^ℓ avec ℓ le nombre d'arêtes entre d et e.

Certains étudiants ont proposé de calculer d'abord la fermeture transitive et symétrique de E (appelons-la F), puis de définir la requête $\varphi(u, v, r) \coloneqq F(u, r) \wedge F(r, v) \wedge u \neq v \wedge u \neq r \wedge v \neq r$. Notez cependant que ceci n'est pas la requête demandée : pour le chemin ci-dessus, on aura $F(a_1, c)$ et $F(c, a_2)$, et donc $\varphi(a_1, a_2, c)$; cependant $PassePar(a_1, a_2, c)$ est faux.

Question 8 Simplifiez la requête (UCQ) suivante ou expliquez pourquoi aucune simplification n'est possible :

$$\begin{cases} Answer(x,u) & \leftarrow & R(x,r,s), R(v,s,u), R(x,y,z), R(y,z,u), R(u,y,w) \\ Answer(x,u) & \leftarrow & R(x,y,z), R(x,y,u), R(y,z,u), R(u,y,w) \end{cases}$$

Détaillez les calculs.

.../10

Appelons le première règle q_1, et la deuxième q_2.

On argumentera ci-après que l'union des requêtes q_1 et q_2 est équivalente à la requête conjonctive suivante :

$$q_3 \quad : \quad Answer(x,u) \quad \leftarrow \quad R(x,y,z), R(y,z,u), R(u,y,w)$$

Soit θ la substitution telle que $\theta(r) = y$, $\theta(v) = y$, $\theta(s) = z$, et θ est l'identité sur les autres variables. On peut vérifier que :

- θ est un homomorphisme de q_1 vers q_3, donc $q_3 \sqsubseteq q_1$;
- l'identité est un homomorphisme de q_3 vers q_1, donc $q_1 \sqsubseteq q_3$;
- l'identité est un homomorphisme de q_3 vers q_1, donc $q_2 \sqsubseteq q_3$.

Puisque $q_3 \equiv q_1$ et $q_2 \sqsubseteq q_3$, la requête UCQ de l'énoncé est équivalente à q_3.

Finalement, on argumente que la requête q_3 est minimale. Une façon de prouver la minimalité de q_3 est de démontrer que l'identité est le seul homomorphisme de q_3 vers q_3. À cette fin, soit μ un homomorphisme quelconque de la requête q_3 vers elle-même. Puisque μ doit être l'identité sur les variables dans $Answer(x,u)$, on aura $\mu(x) = x$ et $\mu(u) = u$. Donc,

$$\mu(q_3) \quad : \quad Answer(x,u) \quad \leftarrow \quad R(x,\mu(y),\mu(z)), R(\mu(y),\mu(z),u), R(u,\mu(y),\mu(w))$$

Au vu des occurrences de x et u dans le *body*, on peut conclure que μ est l'identité :

- si $\mu(y) \neq y$ ou $\mu(z) \neq z$, alors le body de $\mu(q_3)$ contient un atome $R(x,\mu(y),\mu(z))$ qui n'est pas dans q_3, ce qui contredit l'hypothèse que μ est un homomorphisme ;
- si $\mu(w) \neq w$, alors le body de $\mu(q_3)$ contient un atome $R(u,\mu(y),\mu(w))$ qui n'est pas dans q_3, ce qui contredit l'hypothèse que μ est un homomorphisme.

Il est correct de conclure que $\mu(y) = y$, $\mu(z) = z$, $\mu(w) = w$. Donc, μ est l'identité. On a donc prouvé que l'identité est le seul homomorphisme de q_3 vers q_3. Il en suit que q_3 est minimal.

Question 9 Est-ce que $\{BC \to E, E \to B, D \to C, CD \to A\} \models\!\bowtie [ABC, BCD, CDE, AD]$? Détaillez les calculs qui mènent à une réponse à cette question. Si la réponse est "non", donnez un contre-exemple pour l'implication. Si la réponse est "oui", argumentez pourquoi l'implication est vraie.

.../10

La dépendance de jointure $\bowtie [ABC, BCD, CDE, AD]$ exprime la formule

$$\forall x \forall y \forall z \forall u \forall w \forall u_1 \forall w_1 \forall x_2 \forall w_2 \forall x_3 \forall y_3 \forall y_4 \forall z_4 \forall w_4 \left(\left(\begin{array}{cl} & R(x, y, z, u_1, w_1) \\ \wedge & R(x_2, y, z, u, w_2) \\ \wedge & R(x_3, y_3, z, u, w) \\ \wedge & R(x, y_4, z_4, u, w_4) \end{array} \right) \to R(x, y, z, u, w) \right),$$

qui sera abrégée comme suit :

A	B	C	D	E
x	y	z	u_1	w_1
x_2	y	z	u	w_2
x_3	y_3	z	u	w
x	y_4	z_4	u	w_4
x	y	z	u	w

L'application de $D \to C$ nous donne :

A	B	C	D	E
x	y	z	u_1	w_1
x_2	y	z	u	w_2
x_3	y_3	z	u	w
x	y_4	z	u	w_4
x	y	z	u	w

Ensuite, deux applications de $CD \to A$ nous donnent :

A	B	C	D	E
x	y	z	u_1	w_1
x	y	z	u	w_2
x	y_3	z	u	w
x	y_4	z	u	w_4
x	y	z	u	w

Ensuite, l'application de $BC \to E$ nous donne :

A	B	C	D	E
x	y	z	u_1	w_1
x	y	z	u	w_1
x	y_3	z	u	w
x	y_4	z	u	w_4
x	y	z	u	w

Aucune autre dépendance fonctionnelle ne s'applique. Le *chase* nous donne un contre-example pour l'implication logique (remplacer les variables par des constantes) :

A	B	C	D	E
a	b	c	h	i
a	b	c	d	i
a	f	c	d	e
a	g	c	d	j

Une petite réflexion suffit pour en déduire un contre-example plus petit :

A	B	C	D	E
a	b	c	d	i
a	f	c	d	e

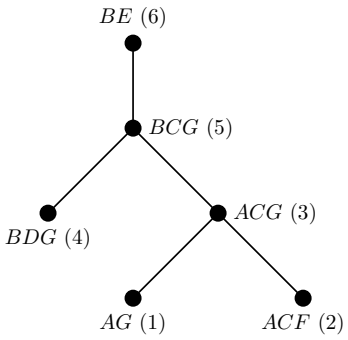
Question 10 Pour la requête

$$\pi_{CEF}(ACF \bowtie ACG \bowtie AG \bowtie BCG \bowtie BDG \bowtie BE),$$

détaillez un plan d'exécution qui garantit que, pour toute base de données, aucun résultat intermédiaire ne contiendra plus de tuples que $I \times U$, avec I le nombre de tuples dans la base de données et U le nombre de tuples dans la réponse à la requête.

.../15

Voici un arbre de jointure. Les chiffres entre parenthèses indiquent l'ordre dans lequel les oreilles ont été enlevées.



L'algorithme de Yannakakis commence par un *full reducer* :

$$\begin{array}{lcl} ACG & := & ACG \ltimes AG \\ ACG & := & ACG \ltimes ACF \\ BCG & := & BCG \ltimes ACG \\ BCG & := & BCG \ltimes BDG \\ BE & := & BE \ltimes BCG \\ BCG & := & BCG \ltimes BE \\ BDG & := & BDG \ltimes BCG \\ ACG & := & ACG \ltimes BCG \\ ACF & := & ACF \ltimes ACG \\ AG & := & AG \ltimes ACG \end{array}$$

Calculons ensuite la projection de la jointure. Notez que les feuilles AG et BDG peuvent être ignorés puisque $AG \cap CEF = \emptyset$ et $BDG \cap CEF = \emptyset$. D'abord,

$$ACG := \pi_{ACFG}(ACG \bowtie ACF).$$

À ce point $\mathsf{sort}(ACG) = ACFG$. Puis,

$$BCG := \pi_{BCFG}(BCG \bowtie ACG).$$

À ce point $\mathsf{sort}(BCG) = BCFG$. Le résultat final :

$$\pi_{CEF}(BE \bowtie BCG).$$

En conclusion, après le *full reducer*, l'algorithme de Yannakakis calcule :

$$\pi_{CEF}(BE \bowtie \pi_{BCFG}(BCG \bowtie \pi_{ACFG}(ACG \bowtie ACF))).$$

Question 11 Écrivez un programme XQuery qui effectue la même transformation que le programme XSLT de la question 5.

.../5

```
<OUTPUT>
{for $g in //garage
 return
 <GARAGE gnom="{$g/@gnom}" ville="{$g/@ville}" mnom="{$g/../@mnom}">
    {for $v in //voiture[@marque != $g/../@mnom][i/@gid = $g/@gid]
     return
     <VOITURE chassis="{$v/@chassis}" marque="{$v/@marque}"/>}
 </GARAGE>}
</OUTPUT>
```

```
<garages>
  <ville nom="Alost">
    <garage id="P2" marque="Peugeot">Ciac</garage>
    <garage id="P1" marque="Peugeot">ErVeeDee</garage>
    <garage id="T1" marque="Toyota">New Energy</garage>
    <garage id="R1" marque="Renault">Valckenier</garage>
  </ville>
  <ville nom="Bruxelles">
    <garage id="R2" marque="Renault">Meiser</garage>
    <garage id="V1" marque="Volkswagen">Sawa</garage>
  </ville>
  <ville nom="Charleroi">
    <garage id="T4" marque="Toyota">Castus</garage>
    <garage id="T3" marque="Toyota">Merckx</garage>
  </ville>
  <ville nom="Mons">
    <garage id="V2" marque="Volkswagen">Carlier Motor</garage>
    <garage id="T2" marque="Toyota">Serriez</garage>
  </ville>
</garages>
```

FIGURE 5 – Output du programme XQuery de la question 12.

Question 12 Écrivez un programme XQuery qui liste les garages présents dans chaque ville, sans doublons. Le format est celui de la figure 5. Les villes doivent être triées par ordre lexicographique. Les garages, au sein d'une ville, sont triés par leur nom. Les positions des blancs et retours à la ligne n'ont pas d'importance.

.../5

```
<garages>
{for $v in //@ville[not(.=preceding::*/@ville)]
 order by $v
 return
 <ville nom="{$v}">
    {for $g in //garage[@ville=$v]
     order by $g/@gnom
     return
     <garage id="{$g/@gid}" marque="{$g/../@mnom}">{data($g/@gnom)}</garage>}
 </ville>}
</garages>
```

En XQuery, les doublons peuvent aussi être retirés avec la fonction `distinct-values()`.

```
<visites>
  <voiture chassis="222">
    <ville>Alost</ville>
    <ville>Charleroi</ville>
  </voiture>
  <voiture chassis="333">
    <ville>Alost</ville>
    <ville>Bruxelles</ville>
    <ville>Charleroi</ville>
  </voiture>
</visites>
```

FIGURE 6 – Output du programme XQuery de la question 13.

Question 13 Écrivez un programme XQuery qui liste, sans doublons et pour chaque voiture, les villes où elle a subi une intervention. Le format est celui de la figure 6. Les voitures doivent être triées par leur numéro de chassis. Les villes sont triées par leur nom. Les positions des blancs et retours à la ligne n'ont pas d'importance.

.../5

```
<visites>
{for $vo in //voiture
 order by $vo/@chassis
 return
 <voiture chassis="{$vo/@chassis}">
    {for $vi in //@ville[not(.=preceding::*/@ville)]
     where $vi=//garage[@gid=$vo/i/@gid]/@ville
     order by $vi
     return
     <ville>{data($vi)}</ville>}
 </voiture>}
</visites>
```
