

# Bases de données

## Requêtes récursives & complexité descriptive

Pierre Senellart



25 mars 2026

## Motivation : Requêtes récursives

- Langages de requêtes considérés jusqu'à présent (algèbre relationnelle, calcul) ont un **horizon limité**
- Certaines structures de données (arbres, graphes) nécessitent des parcours **arbitrairement profonds**
- Applications **naturelles** sur certains types de données : listes de discussion, graphes de réseaux de transport, etc.
- **Ce cours** : Comment exprimer des requêtes **récursives** dans les SGBD relationnels ?
- Les SGBDR ne sont **pas toujours** adaptés pour ce type de données/requêtes, cf. SGBD XML ou SGBD graphes
- **Exemple d'application** : clôture transitive d'un graphe  $G(from, to)$

# Plan

Datalog

Récursion et négation

Récursion en SQL

Complexité descriptive

Références

# Datalog

- Langage récursif le plus simple : on ajoute de la récursion aux **requêtes conjonctives**
- Inspiré de la **programmation logique**
- Requête (ou **programme**) Datalog : ensemble de règles de production de **faits intensionnels**
- **Schéma** d'un programme Datalog : schéma relationnel classique (**schéma extensionnel**) + schéma (disjoint) des faits intensionnels (**schéma intensionnel**)
- On fixe une relation distinguée *But* du schéma intensionnel, dont l'arité est l'arité de la requête

# Syntaxe

Ensemble fini de règles  $r$  de la forme :

$$\underbrace{S(\mathbf{y})}_{\text{tête}} \leftarrow \underbrace{R_1(\mathbf{x}_1), \dots, R_n(\mathbf{x}_n)}_{\text{corps}}$$

avec :

- $S$  relation du schéma **intensionnel**
- $R_1, \dots, R_n$  **relations** du schéma intensionnel ou du schéma extensionnel
- $\mathbf{x}_1, \dots, \mathbf{x}_n, \mathbf{y}$  : tuples de **variables** (ou éventuellement de constantes), d'arité compatible avec les relations
- Chaque variable de la tête est **présente** dans le corps

## Sémantique par point fixe

- Chaque règle  $r$  d'un programme  $P$  peut être vue comme une **requête conjonctive** sur la base de données  $D$  :

$$r(D) := \{S(\nu(y)) \mid \nu \text{ t.q. } R_1(\nu(x_1)) \in D \wedge \dots \wedge R_n(\nu(x_n)) \in D\}$$

où les  $z_i$  sont les variables du corps de la règle

- Opérateur de conséquence**  $\Gamma_P$  défini par :

$$\Gamma_P(D) := D \cup \bigcup_{r \in P} r(D)$$

- On considère la **suite**  $(D_n)$  définie par :  
 $D_0 = D, D_{n+1} = \Gamma_P(D_n)$
- La sémantique de  $P$  sur  $D$  est l'ensemble des faits de la relation *But* dans  $D_\infty$ , le **point fixe** de la suite  $(D_n)$

## Exemple : Clôture transitive

$$But(x, y) \leftarrow G(x, y)$$

$$But(x, y) \leftarrow But(x, z), G(z, y)$$

## Évaluation naïve et semi-naïve

- La sémantique par point fixe suggère une **évaluation naïve** : à chaque itération, on réévalue toutes les règles sur l'ensemble complet des faits déjà produits
- Problème : on **redérive** sans cesse les mêmes faits
- Idée de l'évaluation **semi-naïve** : à l'itération  $i + 1$ , ne produire que les **nouveaux faits**, à partir des faits nouvellement obtenus à l'itération  $i$
- On note souvent :

$$D_i = \text{ensemble des faits connus à l'étape } i, \quad \Delta_i = D_i \setminus D_{i-1}$$

- On calcule alors  $\Delta_{i+1}$ , puis

$$D_{i+1} = D_i \cup \Delta_{i+1}$$

- Même **résultat final**, mais beaucoup moins de calculs redondants



## Exemple : clôture transitive en semi-naïf

$$But(x, y) \leftarrow G(x, y)$$

$$But(x, y) \leftarrow But(x, z), G(z, y)$$

- Initialisation :

$$\Delta_0 = G, \quad D_0 = G$$

- Itération :

$$\Delta_{i+1} = \pi_{from, to}(\rho_{to \rightarrow int}(\Delta_i) \bowtie \rho_{from \rightarrow int}(G)) \setminus D_i$$

- Puis :

$$D_{i+1} = D_i \cup \Delta_{i+1}$$

- On ne réutilise que les faits nouvellement dérivés

## Pourquoi l'évaluation semi-naïve est correcte

- À l'étape  $i + 1$ , tout fait nouvellement dérivé utilise au moins un fait qui était **nouveau** à l'étape  $i$
- Sinon, il aurait déjà pu être dérivé à l'étape précédente
- On peut donc remplacer l'évaluation complète des règles par une évaluation où l'un des atomes récurifs est remplacé par  $\Delta_i$
- On évite ainsi les redérivations, sans perdre aucun fait
- C'est le principe de base des moteurs Datalog et de nombreuses implémentations de requêtes récursives

## Détour : Théorème de Knaster–Tarski

### Définition

Un *treillis complet*  $L$  est un ensemble muni d'un ordre partiel dans lequel *tout sous-ensemble* admet une *borne supérieure* (élément minimum supérieur ou égal à tous les éléments de l'ensemble) et une *borne inférieure*.

### Théorème ([Knaster, 1928, Tarski et al., 1955])

Si  $L$  est un *treillis complet* et  $f : L \rightarrow L$  une application *croissante* sur  $L$ , alors les *points fixes* de  $f$  forment un *treillis complet*.

## Application de Knaster–Tarski à Datalog

- L'ensemble des parties d'un ensemble, ordonné par inclusion, est toujours un treillis complet ( $\bigcup X$  et  $\bigcap X$  sont les bornes supérieures et inférieures)
- $\Gamma_P$  est croissant pour l'inclusion des faits
- Il existe donc un plus petit point fixe de  $\Gamma_P$ , obtenu comme l'intersection de tous les points fixes
- Comme l'ensemble des faits possibles est fini (domaine actif fini), ce plus petit point fixe est obtenu après un nombre fini d'itérations à partir de la base de données initiale
- Définition alternative de l'ensemble des faits produits par un programme Datalog : l'ensemble des faits inclus dans tout modèle du programme Datalog

## Arbre de preuve d'un programme Datalog

- Inspiré par la **programmation logique**
- Pour un tuple  $But(a_1, \dots, a_n)$ , on peut produire une **preuve** de  $But$  comme un arbre de preuve :
  - La racine est  $But(a_1, \dots, a_n)$
  - Tout nœud interne est l'instantiation de la tête d'une règle, ses enfants une instantiation compatible du corps de cette même règle
  - Les feuilles sont des faits extensionnels
- Utile pour **raisonner** sur les programmes Datalog

## Datalog non récursif

### Définition

*Le graphe d'un programme Datalog est le graphe dont les nœuds sont les relations du schéma intensionnel et dans lequel il existe une arête  $(R, S)$  s'il existe une règle avec  $R$  dans le corps et  $S$  dans la tête. Un programme Datalog est *non récursif* si ce graphe est *acyclique*.*

- Même pouvoir d'expression que les unions de requêtes conjonctives
- Mais exponentiellement plus compact !

# Plan

Datalog

Réursion et négation

While en algèbre relationnelle

Logique à point fixe

Datalog avec négation

Réursion en SQL

Complexité descriptive

Références

## Opérateur While

- Algèbre : essentiellement **programmation impérative** (contrairement au calcul)
- On ajoute à l'algèbre :
  - la possibilité de définir des **variables intermédiaires** et de leur **affecter** une valeur (ne change pas le pouvoir d'expression)
  - un **opérateur while** de la forme :

**while change do**

*...affectation de valeur à une ou plusieurs variables...*

**done**

- **Sémantique** : le contenu de la boucle est exécuté **tant que** les affectations changent les variables sous-jacentes
- **Boucles infinies** possibles !



## While inflationniste vs non-inflationniste

- Deux variantes :

**Non-inflationniste** Opérateur d'affectation  $:=$ , affectation arbitraire

**inflationniste** Opérateur d'affectation  $+=$ , la variable affectée ne peut que **croître**

- Boucles infinies **impossibles avec un while inflationniste** s'il n'y a pas de possibilité de produire de nouvelle valeur (p. ex., arithmétique), parce que le domaine actif est fini
- Variante non-inflationniste **plus expressive**

## Exemple : Clôture transitive

On introduit un schéma de relation  $C$  avec attributs *from* et *to*, tout comme  $G$ .

Non-inflationniste

$C := G$

**while change do**

$C := C \cup \pi_{from,to}(\rho_{to \rightarrow int}(C) \bowtie \rho_{from \rightarrow int}(G))$

**done**

$C$

inflationniste

$C += G$

**while change do**

$C += \pi_{from,to}(\rho_{to \rightarrow int}(C) \bowtie \rho_{from \rightarrow int}(G))$

**done**

$C$

# Plan

Datalog

Récursion et négation

While en algèbre relationnelle

Logique à point fixe

Datalog avec négation

Récursion en SQL

Complexité descriptive

Références

## Point fixe non-inflationniste

- On ajoute au calcul relationnel une construction de **point fixe**
- Supposons  $\varphi(T)$  formule du calcul, mentionnant les relations du schéma ainsi qu'une **nouvelle relation  $T$** , et ayant pour variables libres  $x_1, \dots, x_n$
- Alors  $\mu_T[\varphi(T)](x_1, \dots, x_n)$  est une formule du **calcul avec point-fixe**
- Sémantique** : On considère le **point fixe** de la relation  $T$  (s'il existe)  
en **remplaçant**  $T$  à chaque étape par l'ensemble des faits de la forme  $T(x_1, \dots, x_n)$  pour  $\varphi(T)(x_1, \dots, x_n)$  satisfait (en partant de  $T = \emptyset$ )
- Équivalent** à l'algèbre avec **while** non-inflationniste !

## Point fixe inflationniste

- On ajoute au calcul relationnel une construction de **point fixe**
- Supposons  $\varphi(T)$  formule du calcul, mentionnant les relations du schéma ainsi qu'une **nouvelle relation  $T$** , et ayant pour variables libres  $x_1, \dots, x_n$
- Alors  $\mu_T^+[\varphi(T)](x_1, \dots, x_n)$  est une formule du **calcul avec point-fixe**
- **Sémantique** : On considère le **plus petit point fixe** de la relation  $T$   
en **ajoutant** à  $T$  à chaque étape l'ensemble des faits de la forme  $T(x_1, \dots, x_n)$  pour  $\varphi(T)(x_1, \dots, x_n)$  satisfait (en partant de  $T = \emptyset$ )
- **Équivalent** à l'algèbre avec **while** inflationniste !

## Exemple : Clôture transitive

Non-inflationniste

$$\{ (x, y) \mid \mu_C [G(x, y) \vee C(x, y) \vee (\exists z C(x, z) \wedge G(z, y))](x, y) \}$$

inflationniste

$$\{ (x, y) \mid \mu_C^+ [G(x, y) \vee (\exists z C(x, z) \wedge G(z, y))](x, y) \}$$

# Complexité en les données

## Théorème

*L'évaluation de  $FO+\mu^+$  est **PTIME** en complexité en les données.*

*L'évaluation de  $FO+\mu$  **sur les structures où l'itération atteint un point fixe** est faisable en espace polynomial. En général, la convergence n'est pas garantie.*

## Démonstration.

Au tableau. Facile.



## Logique du second ordre (SO)

- Extension de la logique du premier ordre dans laquelle il est possible d'utiliser des **quantificateurs sur des relations**
- Par exemple, on peut exprimer en logique du second ordre qu'un graphe  $G$  est **biparti** :

$$\begin{aligned} \exists X \exists Y \forall x \forall y [G(x, y) \rightarrow ((X(x) \wedge Y(y)) \vee (X(y) \wedge Y(x)))] \\ \wedge \neg(\exists x X(x) \wedge Y(x)) \end{aligned}$$

- Peut être aussi vu comme un **langage de requêtes**, plus puissant que le calcul relationnel



## Point fixe inflationniste et SO

- Dans certains cas, **traduction naturelle** de logique à point fixe vers logique du second-ordre
- En particulier, pour un point fixe inflationniste :

$$\{(x_1, \dots, x_n) \mid \mu_T^+[\varphi(T)](x_1, \dots, x_n)\}$$

s'exprime **en SO** :

$$\{(x_1, \dots, x_n) \mid \\ \forall T (\forall y_1 \dots \forall y_n \varphi(T)(y_1, \dots, y_n) \rightarrow T(y_1, \dots, y_n)) \rightarrow \\ T(x_1, \dots, x_n)\}$$

- Application immédiate de **Knaster-Tarski** ! Le plus petit point fixe (calculé par  $\mu^+$ ) de l'opérateur qui ajoute à  $T$  les tuples pour lesquels  $\varphi(T)$  est satisfaite est croissant, et donc l'intersection de tous ( $\forall T$ ) les modèles de  $\varphi$

## Exemple : Clôture transitive

$$\{ (x, y) \mid$$

$$\forall C \ ( \forall f \forall t \ ( G(f, t) \vee ( \exists z \ C(f, z) \wedge G(z, t) ) ) \rightarrow C(f, t) )$$

$$\rightarrow C(x, y) \}$$

# Plan

Datalog

Récursion et négation

While en algèbre relationnelle

Logique à point fixe

Datalog avec négation

Récursion en SQL

Complexité descriptive

Références

## Datalog avec négation dans le corps

- On ajoute à Datalog la possibilité d'écrire  $\neg R(x_1, \dots, x_n)$  dans le **corps** d'une règle
- $R$  peut être **extensionnel** ou **intensionnel**
- **Sémantique domaine actif** (nombre fini de faits tels que  $\neg R(x_1, \dots, x_n)$ )
- Définition par **point fixe inflationniste**, comme pour Datalog :

$$r : S(y) \leftarrow R_1(x_1), \dots, \neg R_i(x_i), \dots, R_n(x_n)$$

$$r(D) := \{S(\nu(y)) \mid \nu \text{ t.q. } R_1(\nu(x_1)) \in D \wedge \dots \wedge R_i(\nu(x_i)) \notin D \wedge \dots \wedge R_n(\nu(x_n)) \in D\}$$

- L'opérateur de conséquence reste **inflationniste**, terminaison assurée (aucun fait n'est supprimé)
- Équivalent au calcul avec point fixe **inflationniste**, à l'algèbre avec **while inflationniste**

## Exemple : Clôture transitive du complément

$$But(x, y) \leftarrow \neg G(x, y)$$

$$But(x, y) \leftarrow But(x, z), \neg G(z, y)$$

## Datalog stratifié

- Une autre manière classique d'ajouter la **négation** à Datalog consiste à imposer une **stratification**
- **Intuition** : on découpe les relations intensionnelles en **strates** successives
- Une règle peut dépendre :
  - **positivement** d'une relation de la même strate
  - **positivement ou négativement** d'une relation d'une strate **strictement inférieure**
- On interdit donc les **cycles passant par une négation**
- **Évaluation** :
  - on calcule d'abord la première strate
  - puis la suivante, en considérant les strates précédentes comme déjà fixées
  - etc.
- On obtient ainsi une sémantique **bien définie** de la négation, très utilisée en pratique

## Exemple de programme Datalog stratifié

$$Reach(x, y) \leftarrow G(x, y)$$

$$Reach(x, y) \leftarrow Reach(x, z), G(z, y)$$

$$NoReach(x, y) \leftarrow V(x), V(y), \neg Reach(x, y)$$

- Première strate : calcul de *Reach* par récursion positive
- Deuxième strate : calcul de *NoReach* par négation sur la relation déjà calculée *Reach*
- Stratifié : la négation porte sur une relation d'une strate inférieure
- Non stratifié :

$$P(x) \leftarrow \neg Q(x) \quad Q(x) \leftarrow \neg P(x)$$

car il y a un cycle négatif

# Plan

Datalog

Récursion et négation

Récursion en SQL

Complexité descriptive

Références



## Common Table Expressions (CTE)

- Seule possibilité en SQL d'introduire de la récursion (on parle aussi de **requêtes hiérarchiques**)
- **Idée** : On nomme une table, dont la définition s'utilise elle-même
- Inspiré des logiques à **point fixe inflationniste**, mais aussi des fonctions récursives dans les langages de programmation (penser au **let rec** d'OCaml)

## Syntaxe

```

WITH RECURSIVE T(x1, ..., xn) AS (
    SELECT -- cas de base
UNION
    SELECT -- cas récursif, utilisant la table T
)
SELECT * FROM T;

```

Cette forme précise, utilisant **UNION** (ou **UNION ALL**) entre cas de base et cas récursif est **obligatoire**

## Sémantique

- On évalue la requête **itérativement**, en partant de  $T = \emptyset$  et en remplaçant à chaque étape  $T$  par le résultat de l'évaluation de la définition de  $T$
- On s'arrête quand un **point fixe est atteint**
- La partie récursive doit être formulée de façon à permettre une évaluation itérative bien définie ; les SGBD imposent des restrictions syntaxiques et les cas **monotones** sont les plus naturels
- Optimisations possibles (et appliquées) permettant de ne tenir compte que des **nouveaux faits** à chaque étape (caractère inflationniste)
- Boucle infinie **possible** si on crée de nouvelles valeurs

## Exemple : Clôture transitive

```

WITH RECURSIVE C(f,t) AS (
  SELECT * FROM G
  UNION
  SELECT C.f, G.t FROM C JOIN G ON C.t=G.f
)
SELECT * FROM C;

```

## Notes sur le support historique

- CTE normalisées **tardivement** [ISO, 1999] en SQL
- Extensions historiques **non compatibles** des SGBD (**CONNECT BY** introduit par Oracle, reprises dans certains autres SGBD)
- Support aujourd'hui **raisonnablement bon** dans les principaux SGBD
- Les requêtes récursives ne sont pas une priorité des SGBD, optimisations à la traîne (dans PostgreSQL, les CTE ont longtemps été des **barrières d'optimisation**)

# Plan

Datalog

Récursion et négation

Récursion en SQL

Complexité descriptive

Références

## Rappel : Complexité descriptive

- Un langage de requêtes  $\mathcal{Q}$  **capture** une classe de complexité  $\mathcal{C}$  si :
  - Pour tout  $Q \in \mathcal{Q}$ , le problème d'évaluation de requêtes pour  $Q$  est dans  $\mathcal{C}$  (complexité en les données)
  - Pour tout problème  $P$  dans  $\mathcal{C}$ , il existe une requête  $Q \in \mathcal{Q}$  telle que l'évaluation de  $Q$  **résout exactement**  $P$  (sans réduction)!
- Si  $\mathcal{Q}$  capture  $\mathcal{C}$  et si  $\mathcal{C}$  a des problèmes complets pour  $\mathcal{C}$ , alors il existe  $Q \in \mathcal{Q}$  telle que  $Q$  est  $\mathcal{C}$ -complet, mais **la réciproque n'est pas vraie**

## Rappel : FO ne capture pas tout PTIME

### Théorème

*On ne peut pas* calculer en FO qu'une relation contenant un ordre total a un nombre pair d'éléments, ou qu'un graphe est connexe.

Non prouvé, la preuve repose sur les jeux d'Ehrenfeucht–Fraïssé (voir [Libkin, 2004]).



## Connexité, parité

### Théorème

*Le problème de connexité peut s'exprimer en  $FO+\mu^+$ .*

*Le problème de parité du nombre d'éléments (ou de tuples) dans une relation arbitraire **ne peut pas** s'exprimer en  $FO+\mu$ .*

### Démonstration.

Preuve de la première partie au tableau, facile.

Deuxième partie admise (voir [Abiteboul et al., 1995], chapitre 17).



## Parité et $FO+\mu^+$ , avec ordre

### Théorème

*$FO+\mu^+$  peut calculer si une relation contenant un ordre total a un nombre pair d'éléments.*

### Démonstration.

Au tableau, par construction explicite.



## Résultats plus généraux

Théorème (Immerman–Vardi, 1986 & 1982)

$FO + \mu^+$  capture *PTIME* sur les données avec un ordre total sur les éléments du domaine.

Théorème (Abiteboul–Vianu, 1991)

$FO + \mu$  capture *PSPACE* sur les données avec un ordre total sur les éléments du domaine.

Par conséquent,  $FO + \mu \equiv FO + \mu^+$  ssi  $PTIME = PSPACE$ .

Démonstration.

Intuition de preuve au tableau.



## Complexité en les données (bis)

### Théorème

*L'évaluation de  $FO+\mu^+$  est **PTIME-complète** en complexité en les données.*

*L'évaluation de  $FO+\mu$  est **PSPACE-complète** en complexité en les données (restreinte aux structures où il y a convergence).*

### Démonstration.

La difficulté vient de la complexité descriptive sur les structures ordonnées. □

## Au-delà : Théorème de Fagin

### Théorème (Fagin, 1973)

$\exists SO$  (fragment de  $SO$  sans quantification universelle de deuxième ordre) *capture NP*.

### Démonstration.

Résultat fondamental, difficile à prouver formellement mais assez intuitif : on utilise le quantificateur existentiel pour deviner une trace de l'exécution de la machine de Turing non-déterministe. On peut aussi utiliser le quantificateur existentiel pour deviner une relation d'ordre permettant d'ordonner les étapes de la machine de Turing et les positions de la bande. On peut ensuite exprimer en FO que les transitions sont correctes.



## Complexité descriptive et $P \neq NP$

- On a une **caractérisation de NP** :  $\exists SO$
- Si on avait une caractérisation de P... (indépendamment d'ordre sur les éléments du domaine!) par une classe  $\mathcal{Q}$
- ... on pourrait montrer  $P \neq NP$  en exhibant une requête de  $\exists SO$  non exprimable dans  $\mathcal{Q}$ !
- Mais on en est **loin**...

# Plan

Datalog

Récursion et négation

Récursion en SQL

Complexité descriptive

Références

## Références

- Datalog : chapitre 12 de [Abiteboul et al., 1995]
- While, Logiques à point fixe, Datalog à négation : chapitre 14 de [Abiteboul et al., 1995]
- Complexité descriptive : chapitre 17 de [Abiteboul et al., 1995]
- Récursion en PostgreSQL : <https://www.postgresql.org/docs/current/static/queries-with.html>



## Bibliographie I

- Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- ISO. *ISO 9075:1999: SQL*. International Standards Organization, 1999.
- Bronisław Knaster. Un théoreme sur les fonctions d'ensembles. *Ann. Soc. Polon. Math*, 6(133) :2013134, 1928.
- Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. doi : 10.1007/978-3-662-07003-1. URL <http://dx.doi.org/10.1007/978-3-662-07003-1>.
- Alfred Tarski et al. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2) : 285–309, 1955.