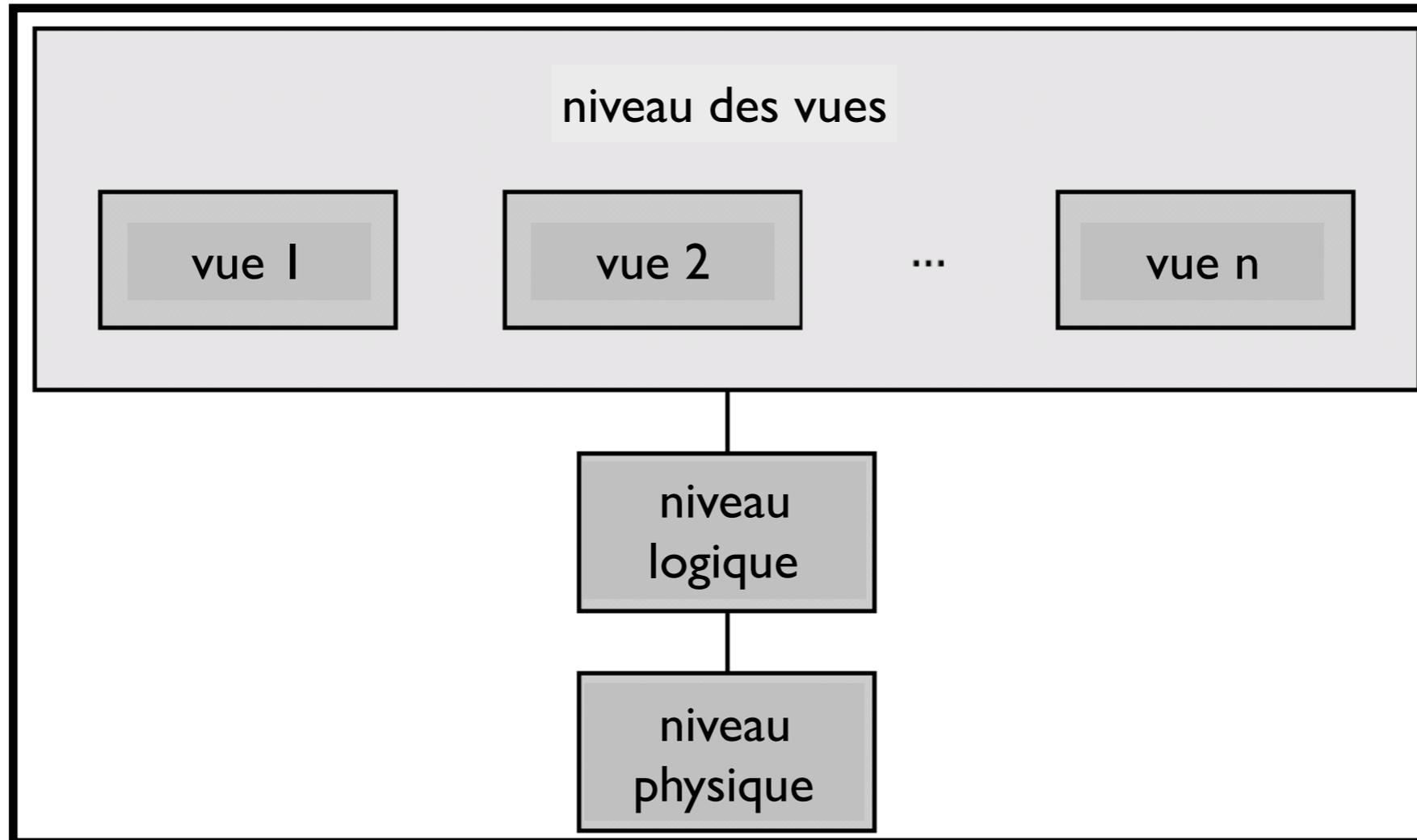


Vues, tables temporaires et requêtes récursives

Bases de données

Amélie Gheerbrant
IRIF, Université Paris Diderot
amelie@irif.fr

Rappel: architecture de base d'un SGBD



- **Les vues** constituent un mécanisme pour spécialiser une bases de données ; elles sont utilisées également pour créer des tables temporaires virtuelles

Vues

- Parfois il n'est pas souhaitable que tous les utilisateurs voient le modèle logique entier (i.e, toutes les vraies relations stockées dans la BD)
- **Exemple** : quelqu'un a besoin d'accéder au numéros de tous les prêts d'une banque, mais pas le droit de voir les montants de ces prêts. Cette personne devrait uniquement voir une relation décrite, en SQL, par :

```
SELECT nom_client, num_prêt  
FROM Client C, Credit Cr  
WHERE C.id_client = Cr.id_client ;
```

- Une **vue** fournit un mécanisme pour **cacher ou restructurer** les données accessibles à certains utilisateurs.
- Une relation qui n'était pas dans le schéma de la base de données, mais est rendue visible à un utilisateur en tant que "relation virtuelle" est appelé une **vue**.

Le schéma relationnel d'une banque

- Agence (nom_agence, ville_agence, capital)
- Client (id_client, nom_client)
- Compte (num_compte, nom_agence , montant)
- Prêt (numero_prêt, nom_agence, montant)
- Propriétaire (id_client, num_compte)
- Credit (id_client, id_prêt)

Définition de vue

- Une vue est définie en utilisant la commande **CREATE VIEW** :

CREATE VIEW *Nom_Vue* **AS** *< requête >*

où *Nom_Vue* est le nom de la vue et *<requête>* est toute requête SQL légale.

- Une liste de nom d'attributs (*att₁, ..., att_n*) après *Nom_Vue* est optionnelle
- Une fois la vue définie, *Nom_Vue* peut être utilisé dans les requêtes à la place d'une table
- Seulement une forme restreinte de mise à jour peut être appliquée à une vue (cf. plus tard)
- Définir une vue n'est pas la même chose que créer une nouvelle relation à partir du résultat de l'évaluation de *<requête>* : **le contenu d'une vue n'est pas matérialisé dans une nouvelle table, mais il est réévalué à chaque nouvel usage de la vue, il change donc automatiquement quand la base de données est modifiée**

Exemples

- Création d'une table qui fournit les agences et leurs clients

```
CREATE TABLE Tous_les_clients AS  
(SELECT nom_agence, id_client  
FROM Proprietaire P, Compte C  
WHERE P.num_compte = C.num_compte)  
UNION  
(SELECT nom_agence, id_client  
FROM Credit C, Pret P  
WHERE C.num_pret = P.num_pret);
```

- Si un nouveau client de l'agence 'Paris 13' est inséré dans la base via Propriétaire et Compte, il ne sera pas retourné par:

```
SELECT id_client  
FROM Tous_les_clients  
WHERE nom_agence = 'Paris 13';
```

Exemples

- Utiliser plutôt une vue qui fournit les agences et leurs clients

```
CREATE VIEW Tous_les_clients AS  
(SELECT nom_agence, id_client  
FROM Proprietaire P, Compte C  
WHERE P.num_compte = C.num_compte)  
UNION  
(SELECT nom_agence, id_client  
FROM Credit C, Pret P  
WHERE C.num_pret = P.num_pret);
```

- Si un nouveau client de l'agence 'Paris 13' est inséré dans la base via Propriétaire et Compte, il sera retourné par:

```
SELECT id_client  
FROM Tous_les_clients  
WHERE nom_agence = 'Paris 13';
```

Les vues peuvent simplifier des requêtes complexes

Exemple

Trouver les réalisateurs ayant généré le plus de recettes :

```
CREATE VIEW RealRecet (real, recettes) AS  
SELECT réalisateur, SUM(recette)  
FROM Film F, Finance Fi  
WHERE F.titre=Fi.titre  
GROUP BY réalisateur ;
```

```
SELECT réalisateur, recettes  
FROM RealRecet  
WHERE recettes = (SELECT MAX(recettes)  
FROM RealRecet);
```

Les vues peuvent simplifier des requêtes complexes

Formulation alternative (sans vue)

Trouver les réalisateurs ayant généré le plus de recettes :

```
SELECT réalisateur, SUM(recette)
FROM Film F, Finance Fi
WHERE F.titre=Fi.titre
GROUP BY réalisateur
HAVING SUM(recette) >= ALL(SELECT SUM(recette),
                           FROM Film, Finance
                           WHERE F.titre=Fi.titre
                           GROUP BY réalisateur) ;
```

Un exemple un peu plus compliqué

- Schéma :
 - ▶ Film(titre, année, réalisateur, pays, classement, genre, budget, producteur)
 - ▶ Distinctions(titre, année, prix, résultat) (résultat ∈ {gagné, nominé})

Requête : Pour chaque décennie à partir de 1950-59, calculer le pourcentage des prix gagnés par des films US.

On crée d'abord une vue pour stocker tous les films postérieurs à 1949 ayant obtenu un prix, avec leur décennie, titre, année et pays :

```
CREATE VIEW DécennieFilm AS
SELECT (CAST(année/10) AS INTEGER) AS décennie, titre, année, pays
FROM Film
WHERE (titre, année) in
(SELECT titre, année
FROM Distinctions
WHERE résultat='gagné' AND année >= 1950) ;
```

Un exemple un peu plus compliqué

- ▶ Maintenant qu'on dispose de la vue DécennieFilm(décennie, titre, année, pays), on écrit :

```
SELECT décennie, (US.No * 100)/Tous.No
FROM (SELECT décennie, COUNT(titre,année) as No
FROM DécennieFilm WHERE pays='US' GROUP by décennie) as US,
     (SELECT décennie, COUNT(titre, année) as No FROM DécennieFilm
      GROUPE BY décennie) as Tous
WHERE Tous.décennie=US.décennie
      UNION
SELECT décennie, 0
FROM ( (SELECT décennie FROM DécennieFilm) EXCEPT
      (SELECT décennie FROM DécennieFilm WHERE pays='US')) as foo ;
```

Attention : si aucun film US n'a gagné de récompense pour une décennie n, alors il n'y a **pas de** tuple (n, 0) dans US (d'où la table temporaire foo...).

Implémentation efficace des vues

- **Vues matérialisées:**

Créer physiquement et maintenir une table contenant la résultat de l'évaluation de la vue

- **assomption:** d'autres requêtes interrogeront la même vue plus tard
- **problématiques** : maintenir la correspondance entre les tables de la BD et la vue quand la BD est mise à jour
- **stratégies:** mise à jour incrémentale de la vue
 - ▶ syntaxe PostgreSQL :

```
CREATE MATERIALIZED VIEW vue AS SELECT ... ;
```

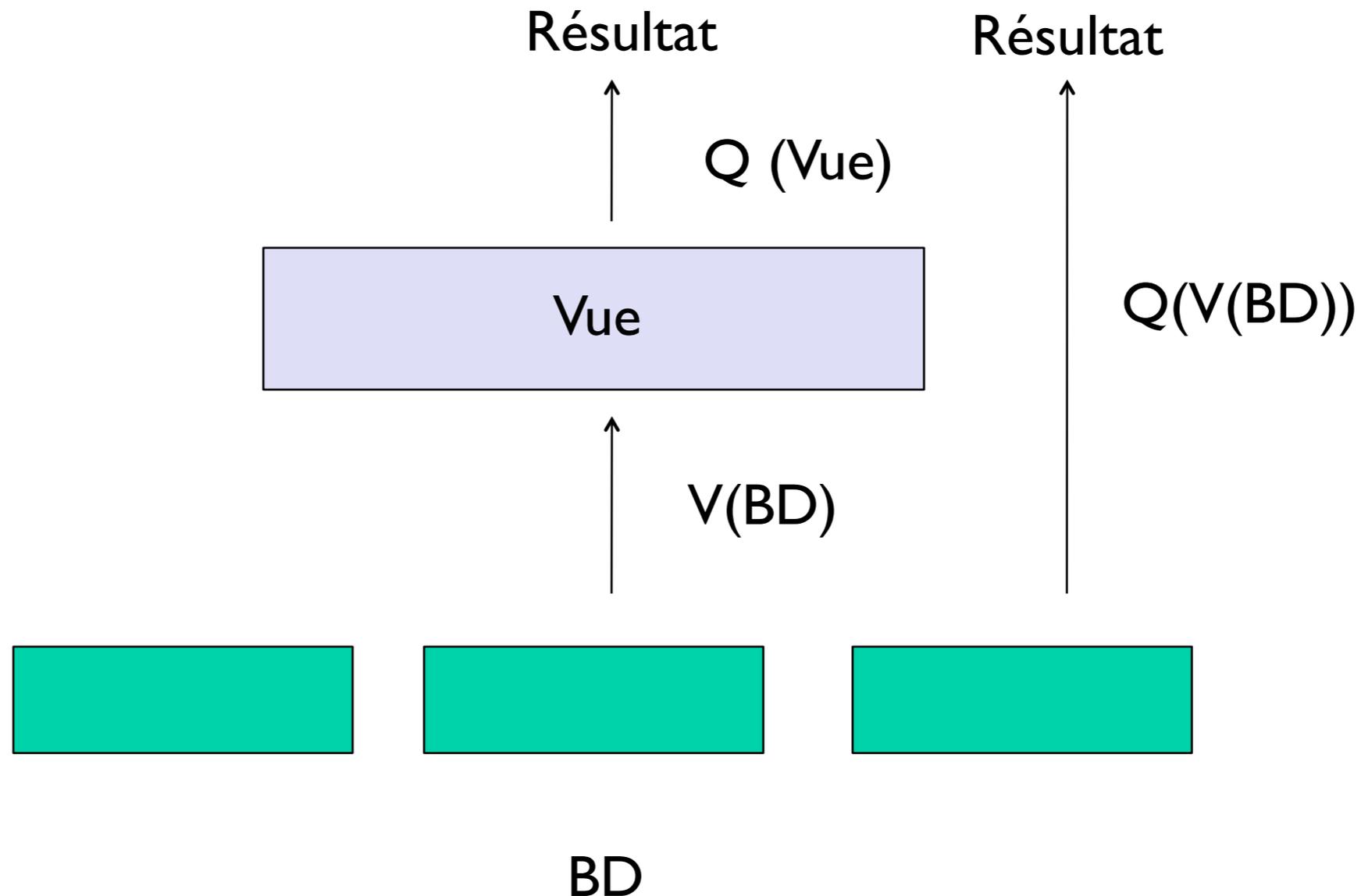
- **Attention** : la maintenance automatique de vue n'est **pas implémentée** ! On peut maintenir manuellement une vue avec :

```
REFRESH MATERIALIZED VIEW vue ;
```

Implémentation efficace des vues

- **Vues virtuelles:**
 - ▶ jamais créés physiquement
 - ▶ les requêtes qui utilisent les vues sont d'abord reformulées comme des requêtes sur les tables de base (en remplaçant chaque vue par sa définition - **déploiement de vues**)
 - ▶ **inconvenient** : inefficace pour des vues définies par des requêtes complexes (spécialement si d'autres requêtes doivent être appliquées sur les mêmes vues en un bref intervalle de temps)
 - ▶ **avantage** : pas besoin de maintenir la correspondance avec les tables de la BD

Déploiement de vues



- $V(BD)$: vue V de la BD
- $Q(Vue)$: résultat de la requête Q sur la vue V
- $Q(V(BD))$: résultat de la requête Q sur la vue V de la BD

Un exemple de déploiement de vues

BD:

Patient	pid hospital docid	Medecin	docid docnom
---------	--------------------	---------	--------------

Vue

(les médecins de “Bichat”):

```
CREATE VIEW BichatDoc AS  
SELECT M.* FROM Medecin M, Patient P  
WHERE P.hospital = 'Bichat' AND p1.docid =  
d1.docid ;
```

Vue

(les patients de Bichat):

```
CREATE VIEW BichatPatient as  
select p2.* FROM Patient p2  
WHERE hospital = 'Bichat';
```

Requête Bichat
(avec vue):

```
SELECT p.pid, d.docnom  
FROM Bichatpatient p, BichatDoc d  
WHERE p.docid = d.docid ;
```

Résultat d'une requête obtenu après déploiement des vues

requête
utilisant
des vues

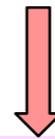
```
select p.pid, d.docnom  
from BichatPatient p, BichatDoc d  
where p.docid = d.docid
```

vue1

```
create view BichatDoc as  
select d1.* from Docteur d1, Patient p1  
where p1.hopital = 'Bichat' and p1.docid = d1.docid
```

Vue2

```
create view BichatPatient as  
select p2.* from Patient p2  
where p2.hopital = 'Bichat'
```



résultat du
déploiement
des vues

```
select p.pid, d.docname  
from Patient p, Docteur d, Patient p1  
where p.docid = d.docid and p.hopital = 'Bichat'  
and p1.hopital = 'Bichat' and p1.docid = d.docid
```

Mise à jour de vues

- Soit une vue des données de la relation Pret cachant l'attribut montant

```
create view branche_pret as  
    select branche_nom, pret_num  
    from pret
```

- Ajoutons maintenant un nouveau tuple dans branche_pret

```
insert into branch_loan  
values ('Paris_13', 'L-307',)
```

Cette insertion mène à l'ajout d'un tuple

```
('Paris_13', 'L-307', NULL )
```

dans la relation Pret

Mise à jour de vues

- Le standard SQL n'autorise les mises à jour qu'à travers des vues très simples (sélection, projection, renommage) :
 - ▶ **pas de jointure** (une seule relation dans le FROM)
 - ▶ pas d'opération ensembliste (union, intersection, différence)
 - ▶ pas d'agrégation (ni même de de group-by)
 - ▶ pas de requête récursive
- Ca ne veut pas dire qu'une mise à jour n'est possible *en théorie* dans aucun autre cas, mais ce n'est pas implémenté

Exemple

```
create view Felli-titres as  
select titre from films where realisateur = 'Fellini';
```

- suppression d'un titre T dans la vue → suppression de tous les tuples ayant pour titre T dans Film
- insertion d'un titre T dans la vue → insertion de (T, 'Fellini', NULL) dans Film
- mise à jour de "Strada" vers la "La Strada" dans la vue

→

```
UPDATE Film
```

```
SET titre = 'La Strada'
```

```
where realisateur = 'Fellini' and titre = 'Strada';
```

Exemple

```
create view Same as
select s1.cinema, s2.cinema
from seance s1, seance s2
where s1.titre = s2.titre
and s1.cinema < s2.cinema ;
```

*Same contient les paires de cinémas
qui passent les mêmes films*

- On souhaite insérer ('Le Champo', 'Odeon') dans Same
- Problème : impossible d'associer à cette insertion une mise à jour de la table Film.. en effet le titre commun est UNKNOWN et la valeur de s1.titre = s2.titre est donc également UNKNOWN !
- Exactement le même problème avec UPDATE et DELETE
- **De telles mises à jour de vues sont donc prohibées**

Exemple

```
CREATE VIEW Total AS  
SELECT realisateur, SUM(duree) AS total  
FROM Film  
GROUP BY realisateur ;
```

*Total contient pour chaque réalisateur
la durée totale de sa filmographie*

- On souhaite insérer ('Vigo', 162) dans Total
- Problème : impossible d'associer à cette insertion une mise à jour de la table Film.. insérer un tuple ? plusieurs ? avec quelle valeur de duree ?
- Exactement le même problème avec UPDATE et DELETE
- **De telles mises à jour de vues sont donc prohibées**

Usage des vues

- **Indépendance logique** : une application peut accéder à des vues, sans avoir besoin de connaître l'organisation effective des données dans la base (qui peut changer de manière transparente, en redéfinissant les vues)
- **Contrôle d'accès** : des droits d'accès différents peuvent être donnés aux tables de base et à des vues, pour qu'un utilisateur ou application donnée n'ait accès qu'à un ensemble restreint du contenu de la base
- **Intégration de données** : des vues peuvent être définies pour regrouper les données de plusieurs sources ayant des schémas relationnels différents
- **Optimisation** : des vues matérialisées peuvent être déniées pour des requêtes fréquentes, pour éviter d'avoir à les ré-évaluer à chaque fois

Vues et tables temporaires

- Nous avons rencontré plusieurs requêtes SQL imbriquées assez complexes
- Les vues constituent un moyen de simplifier ces requêtes, mais créer une vue enrichit le schéma de la BD et ce n'est pas toujours opportun
- En fait, les résultats intermédiaires peuvent être stockés soit dans des « vues », soit dans des « tables intermédiaires » (ou vues éphémères)

Une syntaxe alternative : la clause with

```
WITH Ciredu (ci, real, dur) AS  
(SELECT S.cinema, F.realisateur, F.duree  
FROM Film F, Séance S  
WHERE S.titre=F.titre)
```

```
SELECT ci  
FROM Ciredu  
WHERE dur > 120 AND real LIKE 'K%' ;
```

Syntaxe propre à Postgres
(e.g., n'existe pas dans MySQL)

- **Attention** : Ciredu est une **table temporaire**, et **non pas une vue**
- Différence entre table temporaire et vue :
 - ▶ une fois créée, une vue est utilisable dans toutes les requêtes **SELECT** (sa définition augmente le schéma de la BD)
 - ▶ une table temporaire est une façon de nommer une requête dans une autre requête, elle est utilisable uniquement dans la requête qui la définit (sa définition n'augmente pas le schéma)

Une syntaxe alternative : la clause with

- A partir de la durée maximale des films réalisés par chaque réalisateur, calculer la moyenne de ces valeur maximales :

```
WITH Duree-max(real, max_duree) AS
(SELECT realisateur, MAX(duree)
FROM Film
GROUP BY realisateur)

SELECT AVG(max_duree)
FROM Duree-max ;
```

Clauses WITH et sous-requête dans le FROM

- Alternative à la sous-requête dans le FROM d'il y a deux semaines :

```
SELECT AVG(maximum)
FROM
(SELECT MAX(duree) as maximum
FROM Film
GROUP BY realisateur) AS foo ;
```

Une syntaxe alternative : la clause with

- Il est possible de définir plusieurs tables temporaires dans WITH :

```
WITH Table1 (attribut_1, ..., attribut_n)
```

```
AS (SELECT ...)
```

```
Table2 (attribut'_1, ..., attribut'_m)
```

```
AS (SELECT ...)
```

```
...
```

```
SELECT ... ;
```

Une syntaxe alternative : la clause with

- Il est possible de définir plusieurs tables temporaires dans WITH :

WITH **US** (décennie, numéro)

AS (SELECT décennie, COUNT(titre, année)

FROM DécennieFilm WHERE pays='US' GROUP by décennie)

Tous (décennie, No)

AS (SELECT décennie, COUNT(titre, année) FROM DécennieFilm

GROUPE BY décennie)

SELECT décennie, (**US**.No * 100)/**Tous**.No

FROM **US**, **Tous**

WHERE **Tous**.décennie=**US**.décennie

UNION

SELECT décennie, 0

FROM ((SELECT décennie FROM DécennieFilm) EXCEPT

(SELECT décennie FROM DécennieFilm WHERE pays='US')) as foo ;

Une syntaxe alternative : la clause with

WITH DécennieFilm

```
AS (SELECT (CAST(année/10) AS INTEGER) AS décennie, titre,  
année, pays
```

```
FROM Film WHERE (titre, année) in (SELECT titre, année FROM  
Distinctions
```

```
WHERE résultat='gagné' AND année >= 1950))
```

US (décennie, numéro)

```
AS (SELECT décennie, COUNT(titre,année)
```

```
FROM DécennieFilm WHERE pays='US' GROUP by décennie)
```

Tous (décennie, No)

```
AS (SELECT décennie, COUNT(titre, année) FROM DécennieFilm
```

```
GROUPE BY décennie)
```

```
SELECT décennie, (US.No*100)/Tous.No
```

```
FROM US, Tous
```

```
WHERE Tous.décennie=US.décennie
```

```
UNION
```

```
SELECT décennie, 0
```

```
FROM ( (SELECT décennie FROM DécennieFilm) EXCEPT
```

```
(SELECT décennie FROM DécennieFilm WHERE pays='US'))  
as foo ;
```

La vue DécennieFilm peut également être éliminée en utilisant la même stratégie

Vues et tables temporaires : usage

- Trois façons de procéder :
 - ▶ Sous requête dans le FROM : à utiliser plutôt si la table temporaire ne sert qu'une fois dans la requête
 - ▶ WITH : à utiliser plutôt si la table temporaire ne sert que dans une seule requête (même plusieurs fois)
 - ▶ Vue : à utiliser plutôt si la table temporaire sert dans plusieurs requêtes

Les limites de SQL : requêtes d'accessibilité

Vols	ville_aller	ville_arrivée
	SCL	CDG
	AMS	CDG
	CDG	EDI
	CDG	AMS
	AMS	EDI

- Trouver toutes les paires de villes (A, B) t.q. on peut aller de A à B en au plus une escale :

```
SELECT V1.ville_départ, V2.ville_arrivée
FROM Vols V1, Vols V2
WHERE V1.ville_arrivée = V2.ville_départ

UNION

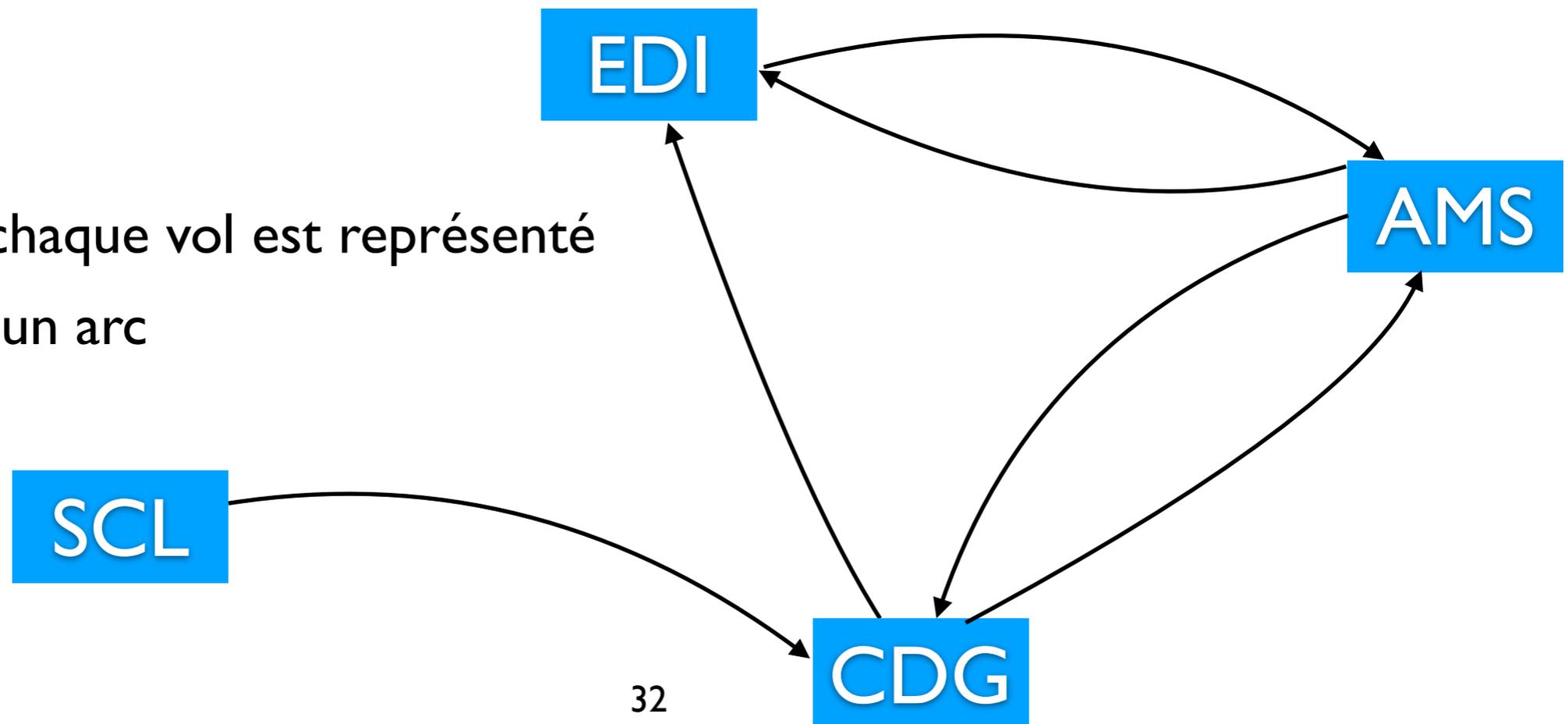
SELECT * FROM Vols ;
```

Requêtes d'accessibilité

- On commence par représenter la table sous forme de graphe :

Vols	ville_aller	ville_arrivée
SCL	CDG	CDG
AMS	CDG	CDG
CDG	EDI	EDI
CDG	AMS	AMS
AMS	EDI	EDI

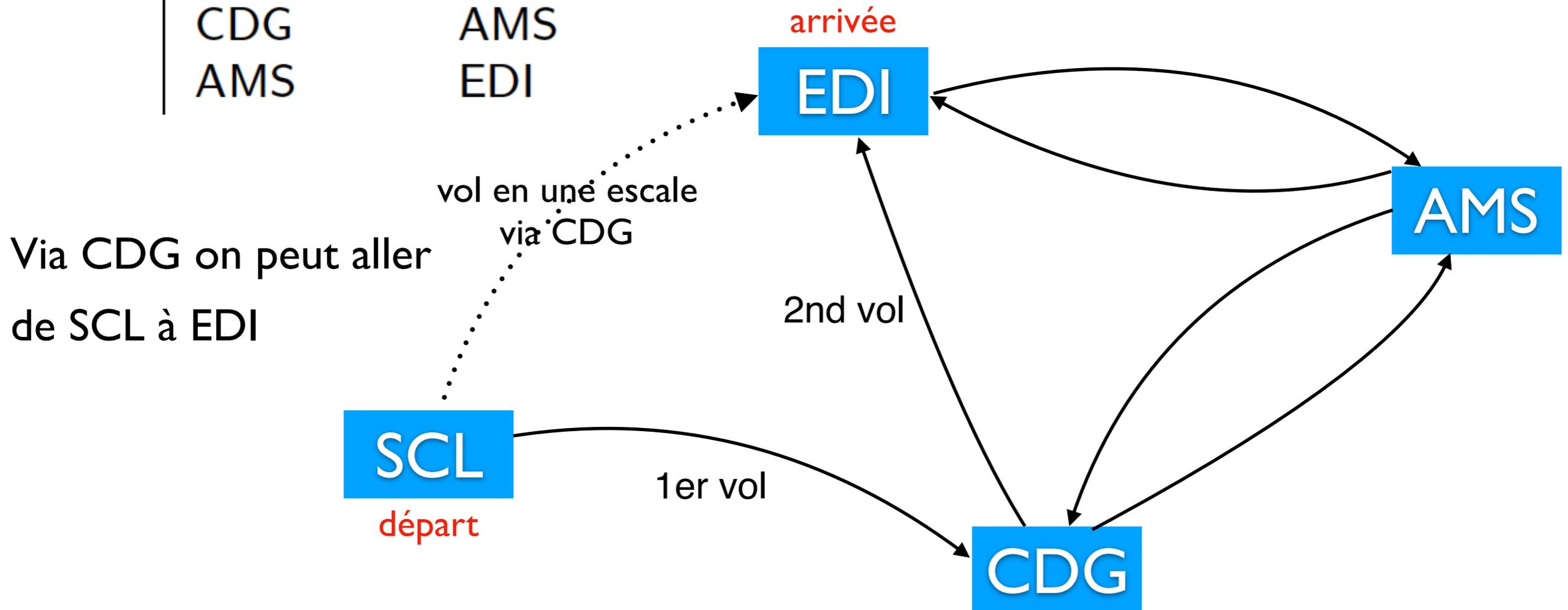
Ici, chaque vol est représenté par un arc



Requêtes d'accessibilité

- Les vols en au plus une escale :

Vols	ville_aller	ville_arrivée
	SCL	CDG
	AMS	CDG
	CDG	EDI
	CDG	AMS
	AMS	EDI

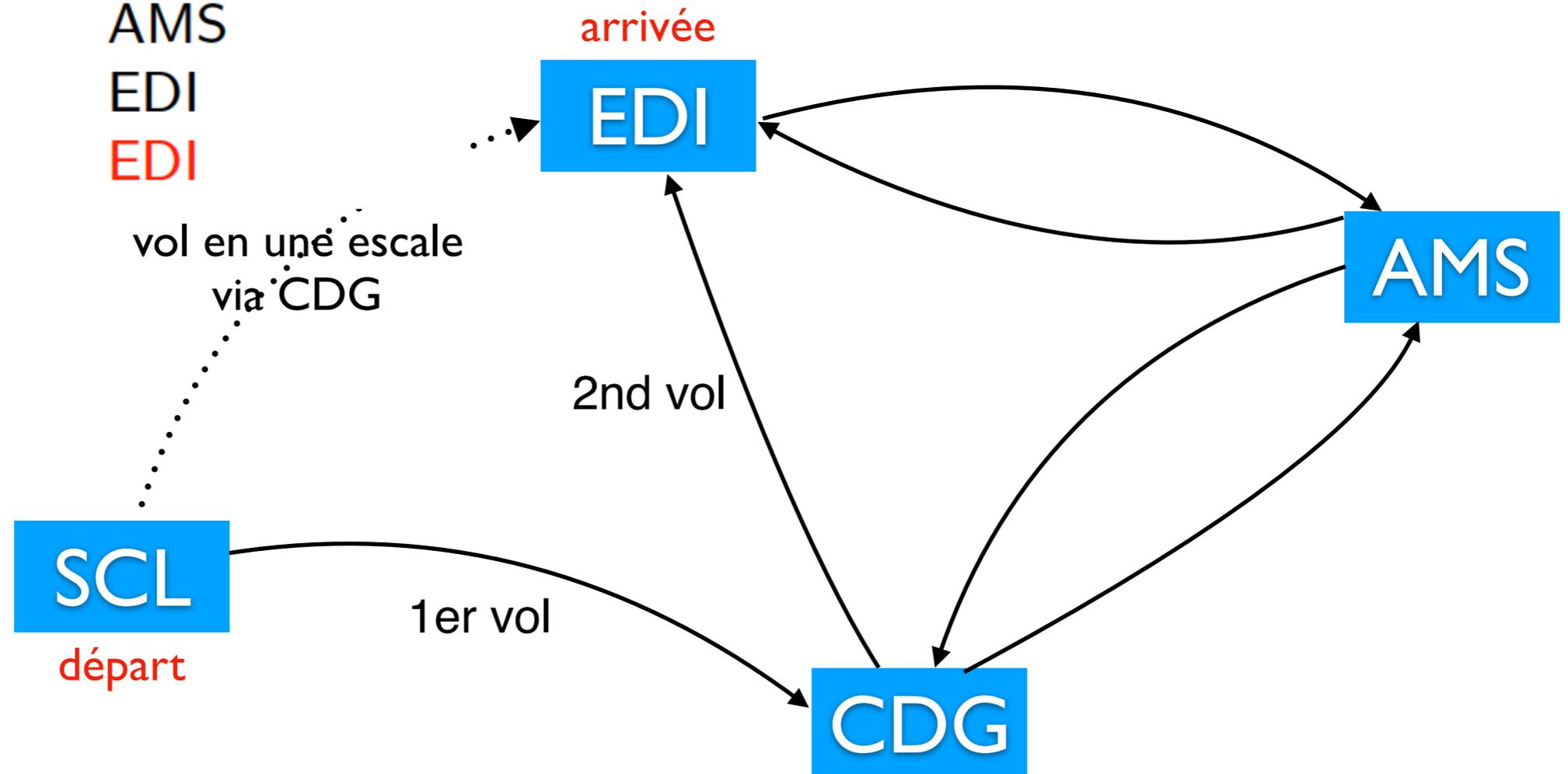


Requêtes d'accessibilité

- Les vols en au plus une escale :

Vols	ville_aller	ville_arrivée
	SCL	CDG
	AMS	CDG
	CDG	EDI
	CDG	AMS
	AMS	EDI
	SCL	EDI

On ajoute donc (SCL, EDI) à notre table

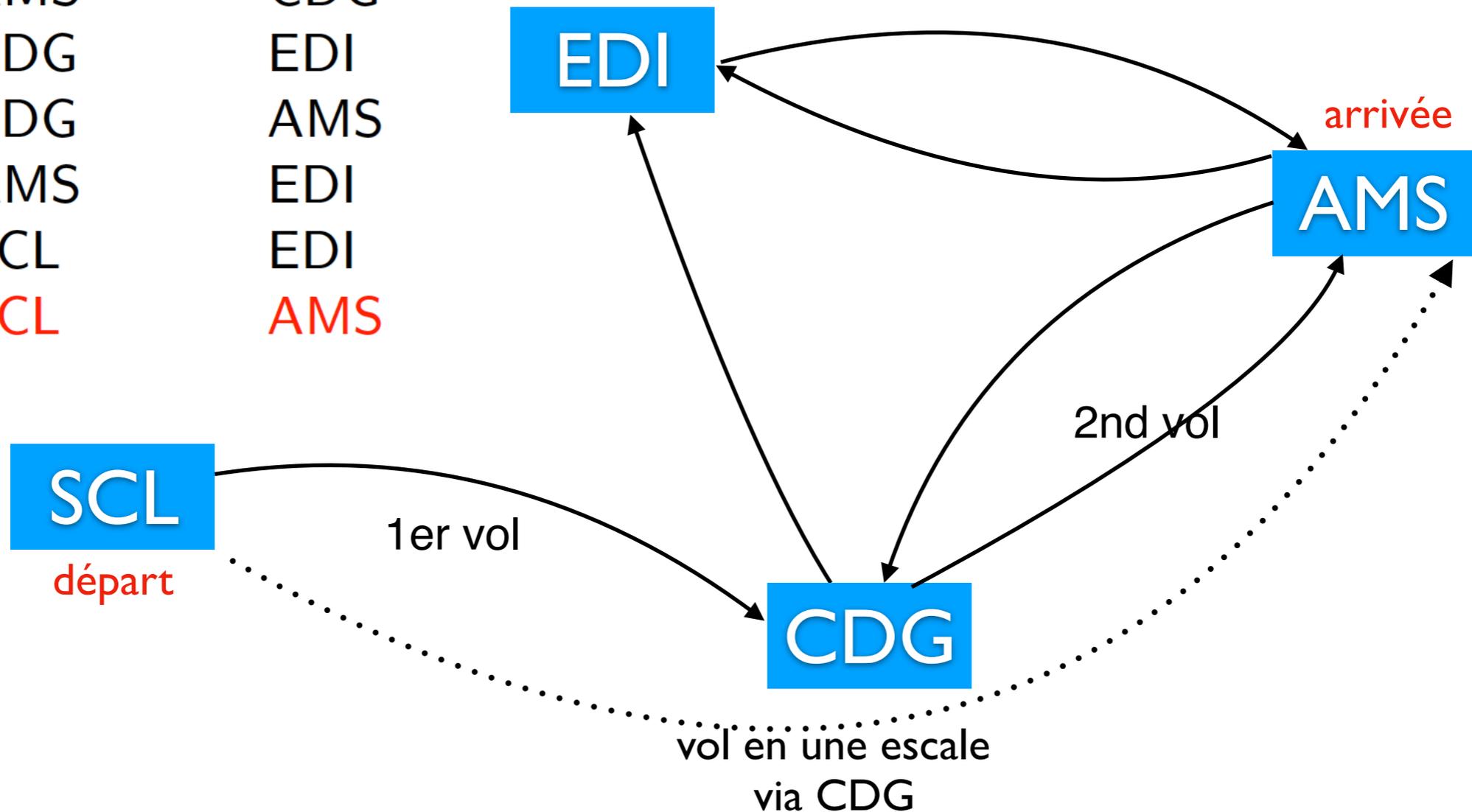


Requêtes d'accessibilité

- Les vols en au plus une escale :

Vols	ville_aller	ville_arrivée
	SCL	CDG
	AMS	CDG
	CDG	EDI
	CDG	AMS
	AMS	EDI
	SCL	EDI
	SCL	AMS

Idem pour (SCL,AMS)

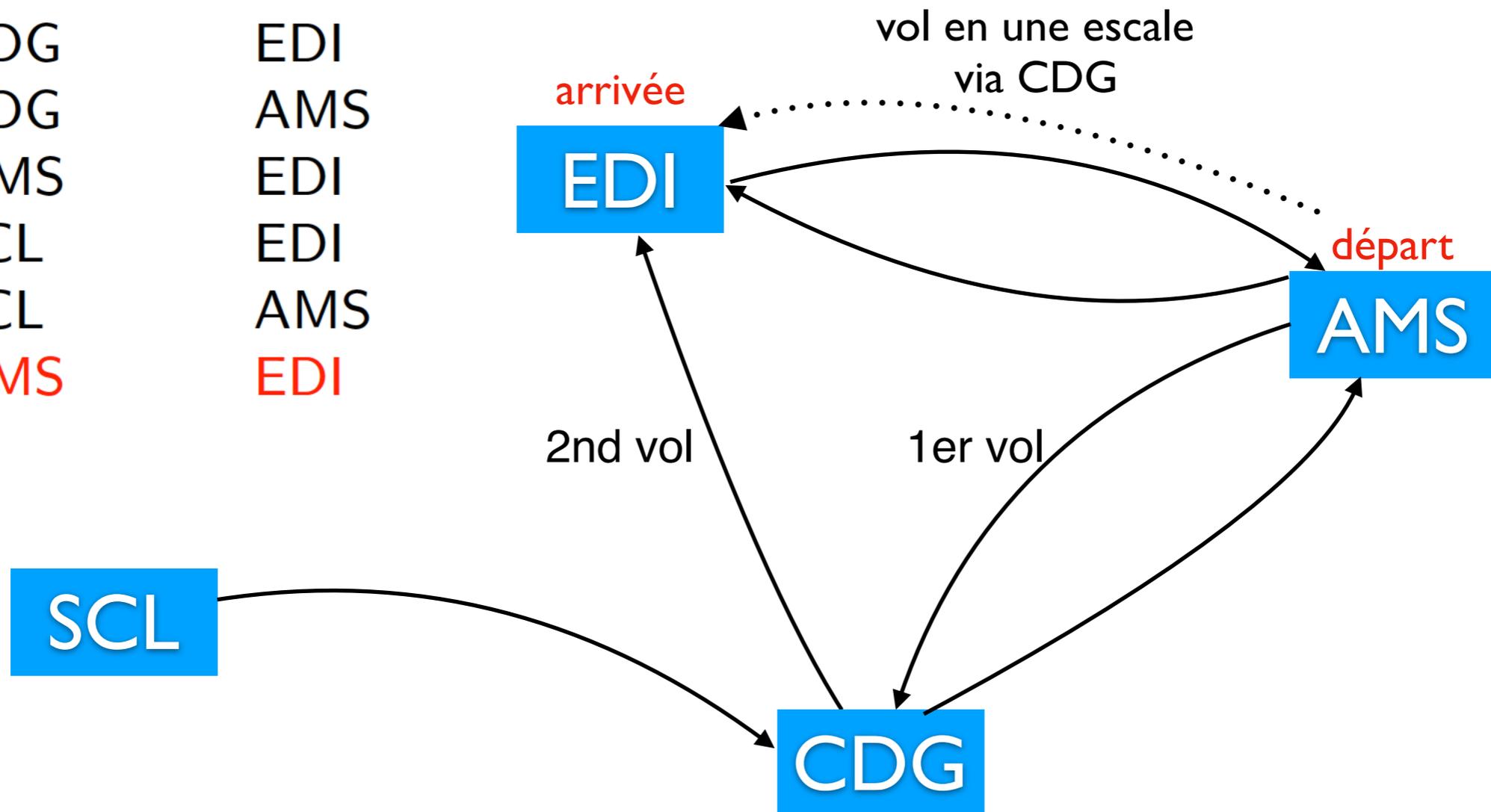


Requêtes d'accessibilité

- Les vols en au plus une escale :

Vols	ville_aller	ville_arrivée
	SCL	CDG
	AMS	CDG
	CDG	EDI
	CDG	AMS
	AMS	EDI
	SCL	EDI
	SCL	AMS
	AMS	EDI

Idem pour (AMS, EDI)

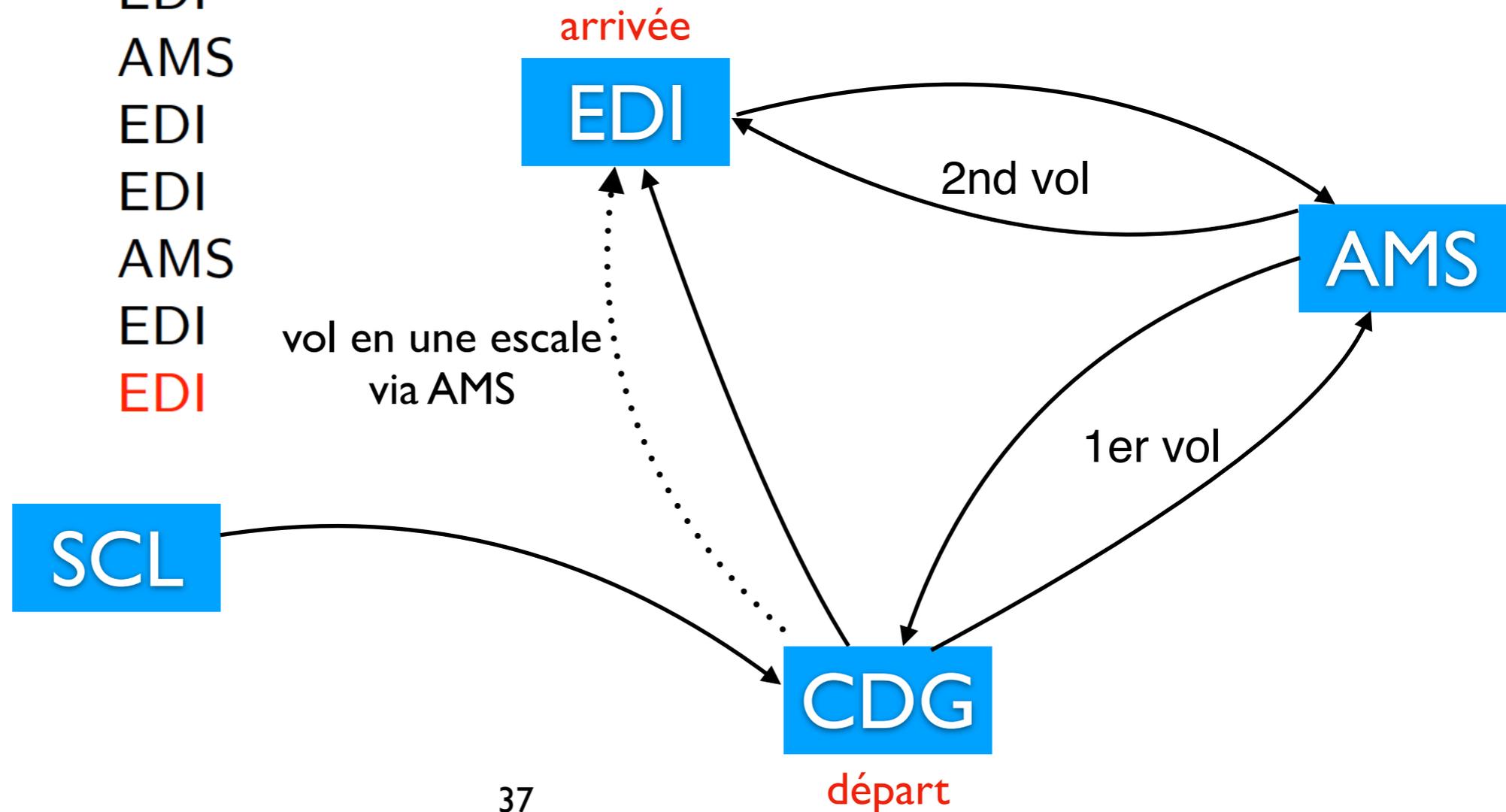


Requêtes d'accessibilité

- Les vols en au plus une escale :

Vols	ville_aller	ville_arrivée
	SCL	CDG
	AMS	CDG
	CDG	EDI
	CDG	AMS
	AMS	EDI
	SCL	EDI
	SCL	AMS
	AMS	EDI
	CDG	EDI

Idem pour (CDG, EDI)



Requêtes d'accessibilité

- On a UNION (et non UNION ALL), on élimine donc les doublons :

Vols	ville_aller	ville_arrivée
	SCL	CDG
	AMS	CDG
	CDG	EDI
	CDG	AMS
	AMS	EDI
	SCL	EDI
	SCL	AMS
	AMS	EDI
	CDG	EDI

```
SELECT V1.ville_départ, V2.ville_arrivée
FROM Vols V1, Vols V2
WHERE V1.ville_arrivée = V2.ville_départ

UNION

SELECT * FROM Vols ;
```

Requêtes d'accessibilité

- On a UNION (et non UNION ALL), on élimine donc les doublons :

Vols	ville_ aller	ville_ arrivée
	SCL	CDG
	AMS	CDG
	CDG	EDI
	CDG	AMS
	AMS	EDI
	SCL	EDI
	SCL	AMS

```
SELECT V1.ville_départ, V2.ville_arrivée
FROM Vols V1, Vols V2
WHERE V1.ville_arrivée = V2.ville_départ

UNION

SELECT * FROM Vols ;
```

Requêtes d'accessibilité

- Trouver toutes les paires de villes (A, B) t.q. on peut aller de A à B en au plus deux escales :

```
SELECT V1.ville_départ, V3ville_arrivée  
FROM Vols V1, Vols V2, Vols V3  
WHERE V1.ville_arrivée = V2.ville_départ  
AND V2.ville_arrivée=V3.ville_départ
```

UNION

```
SELECT V1.ville_départ, V2.ville_arrivée  
FROM Vols V1, Vols V2  
WHERE V1.ville_arrivée = V2.ville_départ
```

UNION

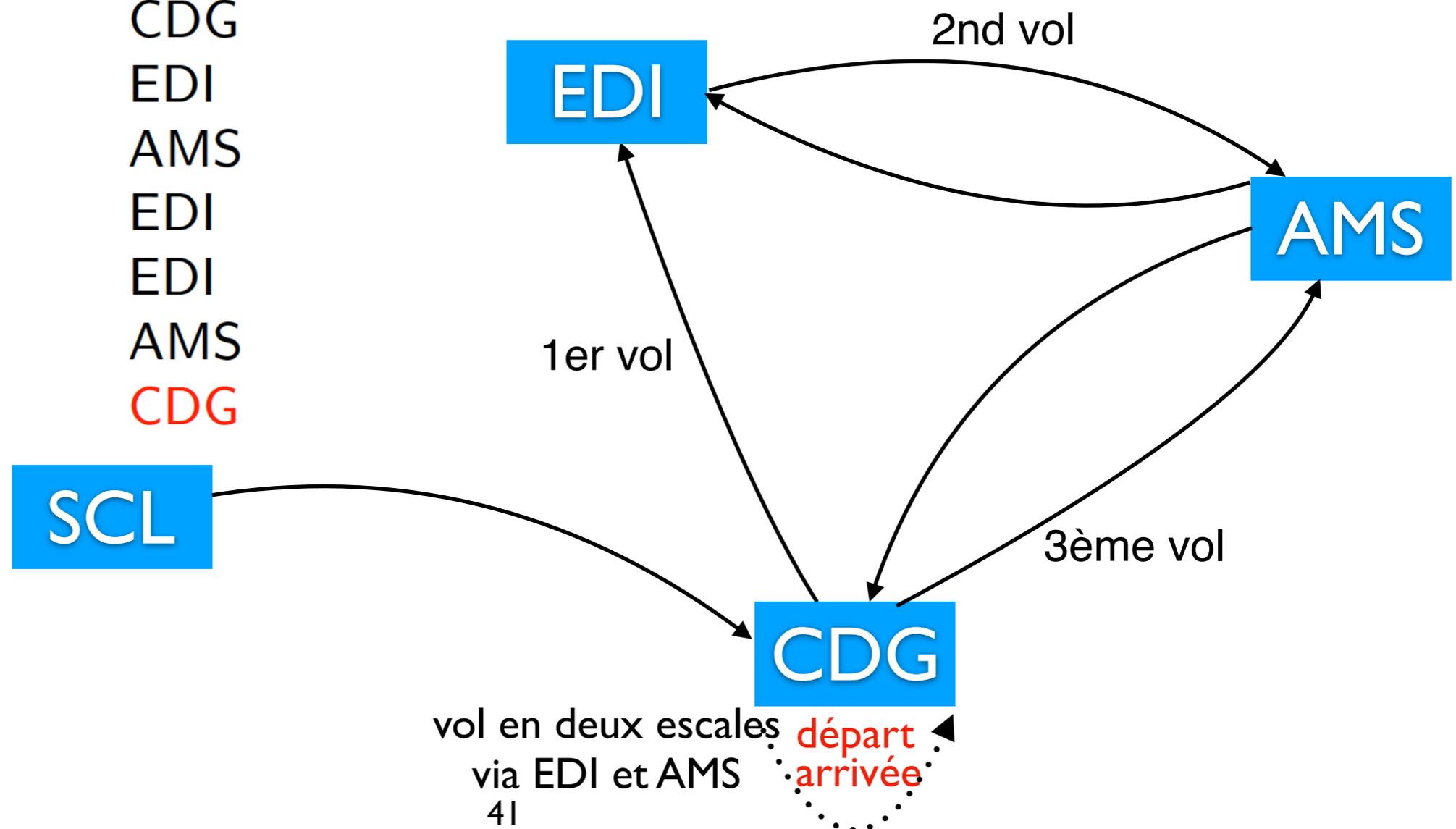
```
SELECT * FROM Vols ;
```

Requêtes d'accessibilité

- Les vols en au plus deux escales :

Vols	ville_aller	ville_arrivée
SCL	CDG	CDG
AMS	CDG	CDG
CDG	EDI	EDI
CDG	AMS	AMS
AMS	EDI	EDI
SCL	EDI	EDI
SCL	AMS	AMS
CDG	CDG	CDG

CDG à CDG via EDI et AMS

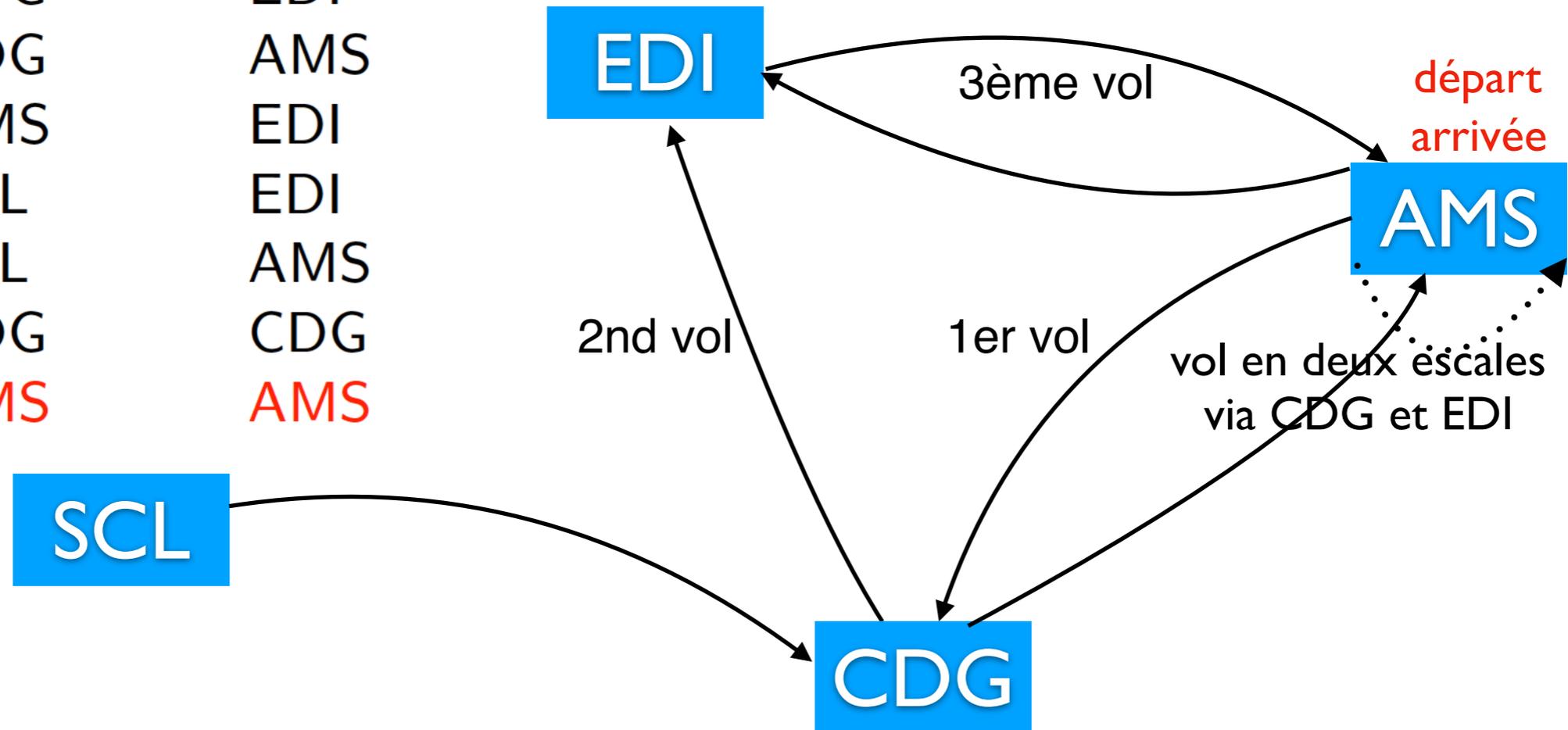


Requêtes d'accessibilité

- Les vols en au plus deux escales :

Vols	ville_aller	ville_arrivée
	SCL	CDG
	AMS	CDG
	CDG	EDI
	CDG	AMS
	AMS	EDI
	SCL	EDI
	SCL	AMS
	CDG	CDG
	AMS	AMS

AMS à AMS via CDG et EDI

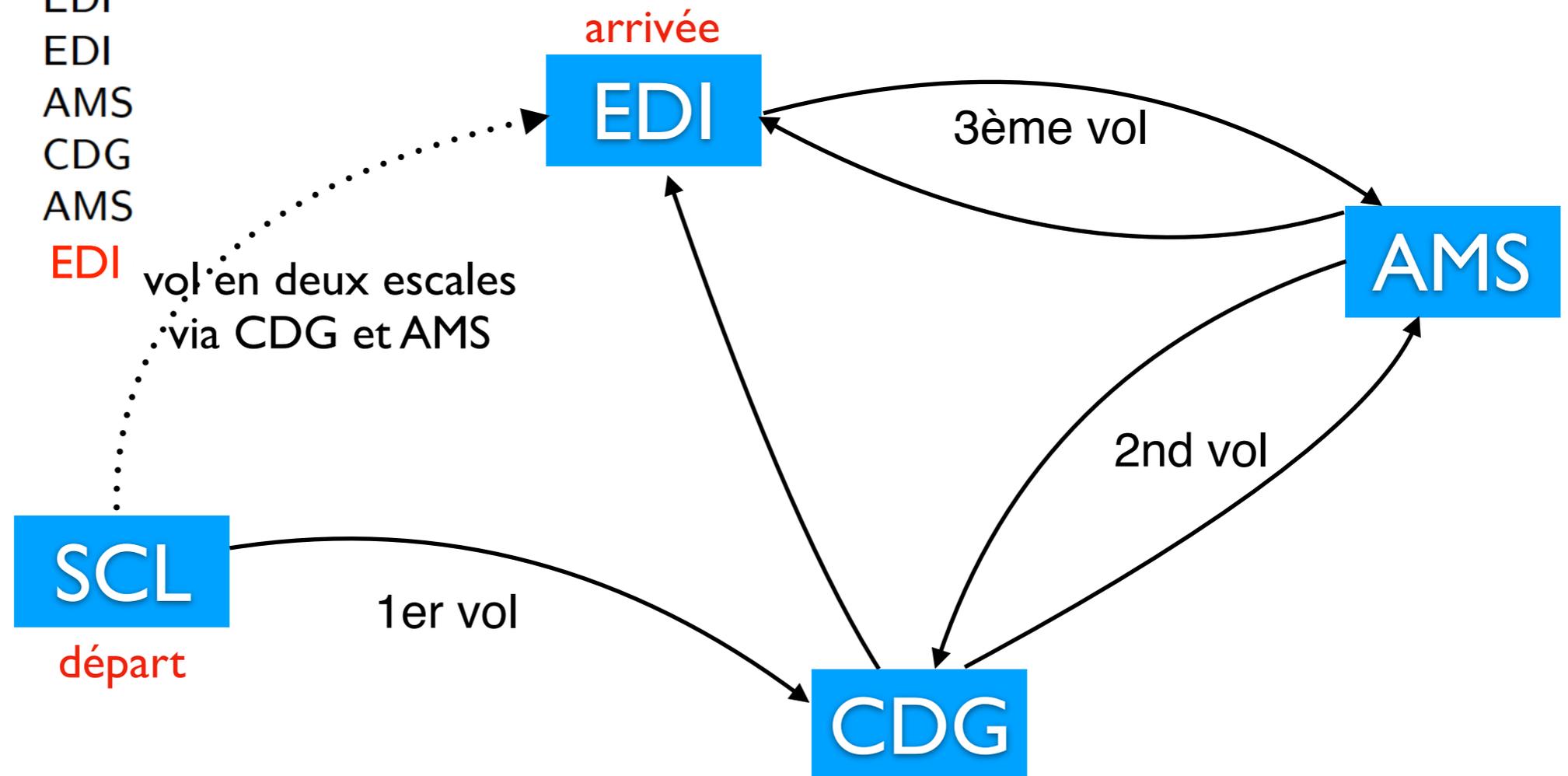


Requêtes d'accessibilité

- Les vols en au plus deux escales :

Vols	ville_aller	ville_arrivée
	SCL	CDG
	AMS	CDG
	CDG	EDI
	CDG	AMS
	AMS	EDI
	SCL	EDI
	SCL	AMS
	CDG	CDG
	AMS	AMS
	SCL	EDI

SCL à EDI via CDG et AMS



Requêtes d'accessibilité

- Les vols en au plus deux escales :

Vols	ville_aller	ville_arrivée
	SCL	CDG
	AMS	CDG
	CDG	EDI
	CDG	AMS
	AMS	EDI
	SCL	EDI
	SCL	AMS
	CDG	CDG
	AMS	AMS
	SCL	EDI

- Là encore on a UNION (et non UNION ALL), on élimine donc les doublons

Requêtes d'accessibilité

- Les vols en au plus deux escales :

Vols	ville_aller	ville_arrivée
	SCL	CDG
	AMS	CDG
	CDG	EDI
	CDG	AMS
	AMS	EDI
	SCL	EDI
	SCL	AMS
	CDG	CDG
	AMS	AMS

- Là encore on a UNION (et non UNION ALL), on élimine donc les doublons

Requêtes d'accessibilité

- Pour tout nombre k fixé, on peut écrire en SQL la requête
 - ▶ Trouver toutes les paires de villes (A, B) t.q. on peut aller de A à B en au plus k escales
- Mais qu'en est-il de la requête d'accessibilité générale :
 - ▶ Trouver toutes les paires de villes (A, B) t.q. on peut aller de A à B
- Les premières versions de SQL ne pouvaient pas l'exprimer.
- Depuis SQL3 (1999 : ISO/IEC 9075), c'est devenu possible.
- Règle générale : éviter la négation dans les requêtes récursives (critères très compliqués si on veut l'utiliser)

Requêtes d'accessibilité

- Syntaxe de PostgreSQL :

```
WITH RECURSIVE Access(ville_départ, ville_arrivée) AS  
(  
    SELECT * FROM Vol  
  
    UNION  
  
    SELECT V.ville_départ, A.ville_arrivée  
    FROM Vol V, Access A  
    WHERE V.ville_arrivée = A.ville_départ  
)  
SELECT * FROM Access ;
```
- Forme générale de la requête du AS :
 - ▶ Un terme non récursif (ici `SELECT * FROM Vol`)
 - ▶ UNION ou UNION ALL
 - ▶ Un **terme récursif** pouvant contenir une référence au résultat de la requête elle même

Requêtes d'accessibilité

- Syntaxe de PostgreSQL :

```
WITH RECURSIVE R(attribut_1, ..., attribut_n) AS
```

```
(
```

```
    requête de base (sans R)
```

```
UNION
```

```
    requête récursive (avec R)
```

```
)
```

```
Requête avec R (et possiblement d'autres tables) ;
```

Evaluation récursive de requête

- On reformule la requête sous forme de règles :
 - ▶ $\text{Access}(x,y) :- \text{Vols}(x,y)$ (se lit : si $x, y \in \text{Vols}$, alors $x, y \in \text{Access}$)
 - ▶ $\text{Access}(x,y) :- \text{Vols}(x,z), \text{Access}(z,y)$ (se lit : si $x, z \in \text{Vols}$ et $z, y \in \text{Access}$, alors $x, y \in \text{Access}$)
- La 2ème règle est **récursive** : la table Access fait référence à elle même
- Évaluation :
 - ▶ Étape 0 : initialisation $\text{Access} = \emptyset$;
 - ▶ Étape $i + 1$: on calcule
 - ▶ $\text{Access}_{i+1}(x,y) :- \text{Vols}(x,y)$
 - ▶ $\text{Access}_{i+1}(x,y) :- \text{Vols}(x,z), \text{Access}_i(z,y)$
 - ▶ Condition d'arrêt : si $\text{Access}_{i+1} = \text{Access}_i$, alors Access_i est la réponse à la requête

Attention : l'algorithme naïf de tout à l'heure avec UNION n'était donc pas tout à fait correct (les doublons ne sont pas éliminés, ils ne sont jamais ajoutés)

Evaluation récursive de requête

- Exemple :

- ▶ Supposons que Vol ne contienne que :

(SCL, CDG), (CDG,AMS),(AMS, EDI)

Étape 0 : Access = \emptyset ;

Étape 1 : Access = {(SCL, CDG), (CDG,AMS), (AMS, EDI)}

Étape 2 :

Access = {(SCL, CDG), (CDG,AMS), (AMS, EDI),(SCL,AMS),(CDG,EDI)}

Étape 3 : Access = {(SCL, CDG), (CDG,AMS), (AMS, EDI),
(SCL,AMS),(CDG,EDI),(SCL,EDI)}

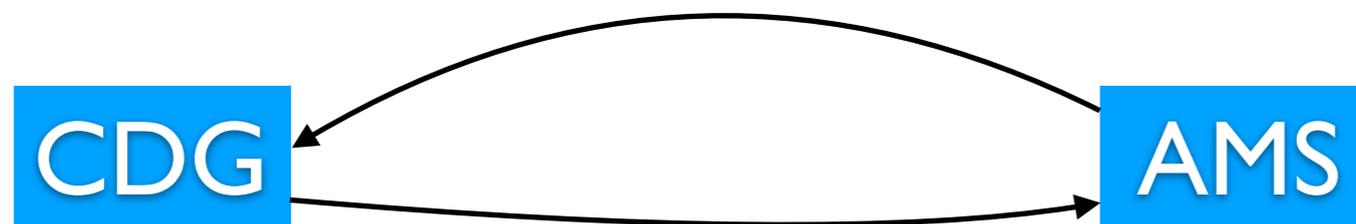
Étape 4 : on ne peut plus rien obtenir en utilisant les règles, le résultat est donc :

{(SCL, CDG), (CDG,AMS), (AMS, EDI),(SCL,AMS),(CDG,EDI),(SCL,EDI)}

Requêtes d'accessibilité

- Attention à UNION ALL :

```
WITH RECURSIVE Access(ville_départ, ville_arrivée) AS  
(  
    SELECT * FROM Vol  
UNION ALL  
    SELECT V.ville_départ, A.ville_arrivée  
    FROM Vol V, Access A  
    WHERE V.ville_arrivée = A.ville_départ  
)  
SELECT * FROM Access ;
```



Requêtes d'accessibilité

- Attention à UNION ALL :

```
WITH RECURSIVE Access(ville_départ, ville_arrivée) AS
```

```
(
```

```
  SELECT * FROM Vol
```

```
  UNION ALL
```

```
  SELECT V.ville_départ, A.ville_arrivée
```

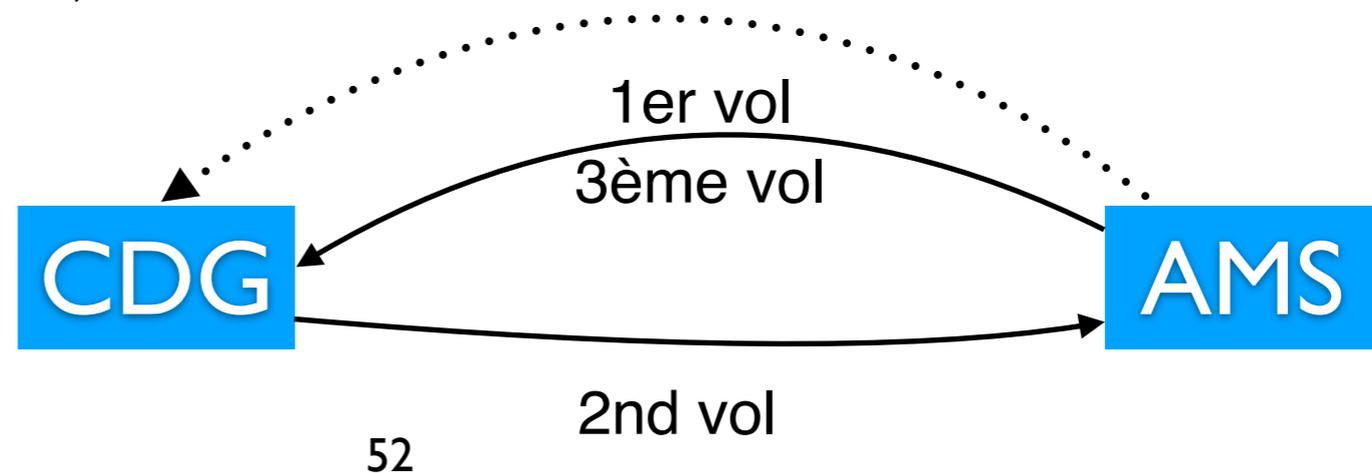
```
  FROM Vol V, Access A
```

```
  WHERE V.ville_arrivée = A.ville_départ
```

```
)
```

```
SELECT * FROM Access ;
```

vol en deux escales via CDG et AMS

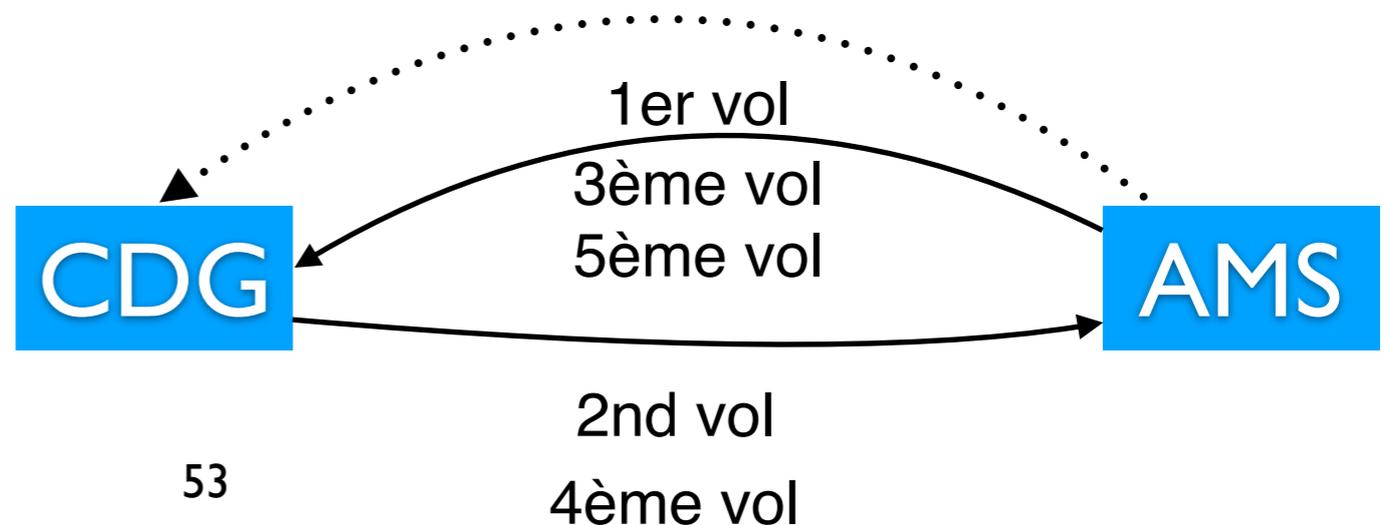


Requêtes d'accessibilité

- Attention à UNION ALL :

```
WITH RECURSIVE Access(ville_départ, ville_arrivée) AS  
(  
    SELECT * FROM Vol  
  
    UNION ALL  
  
    SELECT V.ville_départ, A.ville_arrivée  
  
    FROM Vol V, Access A  
  
    WHERE V.ville_arrivée = A.ville_départ  
)  
SELECT * FROM Access ;
```

vol en quatre escales via CDG, AMS, CDG et AMS



Requêtes d'accessibilité

- Attention à UNION ALL :

```
WITH RECURSIVE Access(ville_départ, ville_arrivée) AS
```

```
(
```

```
  SELECT * FROM Vol
```

```
  UNION ALL
```

```
  SELECT V.ville_départ, A.ville_arrivée
```

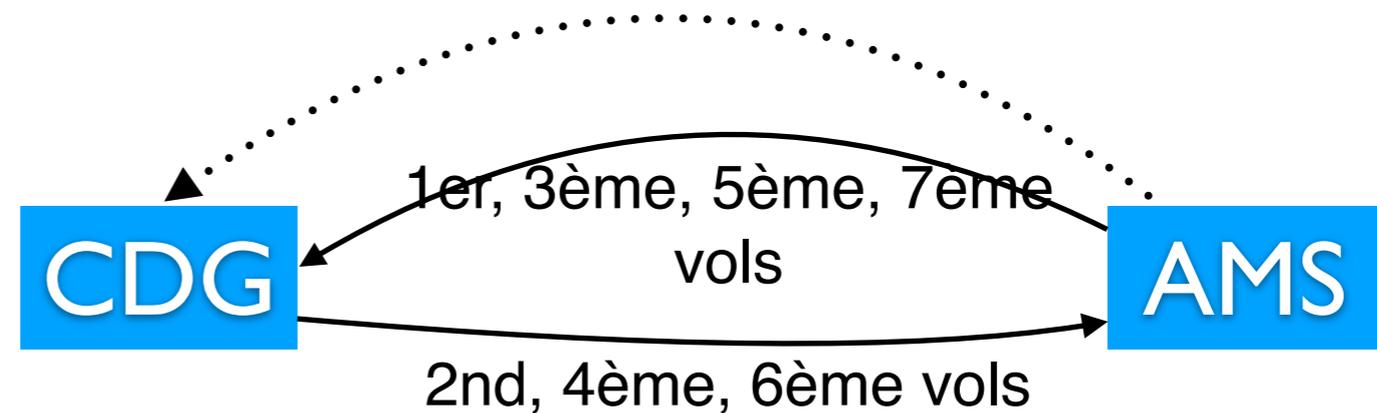
```
  FROM Vol V, Access A
```

```
  WHERE V.ville_arrivée = A.ville_départ
```

```
)
```

```
SELECT * FROM Access ;
```

vol en six escales via CDG, AMS, CDG, AMS etc



Requêtes d'accessibilité

- Les requêtes ne sont plus "sûres" et peuvent avoir des résultats infinis. Par précaution on peut utiliser LIMIT dans la requête

```
WITH RECURSIVE Access(ville_départ, ville_arrivée) AS
(
    SELECT * FROM Vol
UNION ALL
    SELECT V.ville_départ, A.ville_arrivée
    FROM Vol V, Access A
    WHERE V.ville_arrivée = A.ville_départ
)
SELECT * FROM Access
LIMIT 1000 ;
```

- Ici la réponse sera limitée aux 1000 premiers résultats

Requêtes d'accessibilité

- Trouver toutes les sous-parties directes et indirectes d'un produit, si seules les inclusions immédiates sont données par une table :

```
WITH RECURSIVE parties_incluses(sous_partie, partie, quantite) AS
```

```
(
```

```
SELECT sous_partie, partie, quantite FROM parties
```

```
WHERE partie = 'produit X'
```

```
UNION ALL
```

```
SELECT p.sous_partie, p.partie, p.quantite
```

```
FROM parties_incluses pr, parties p
```

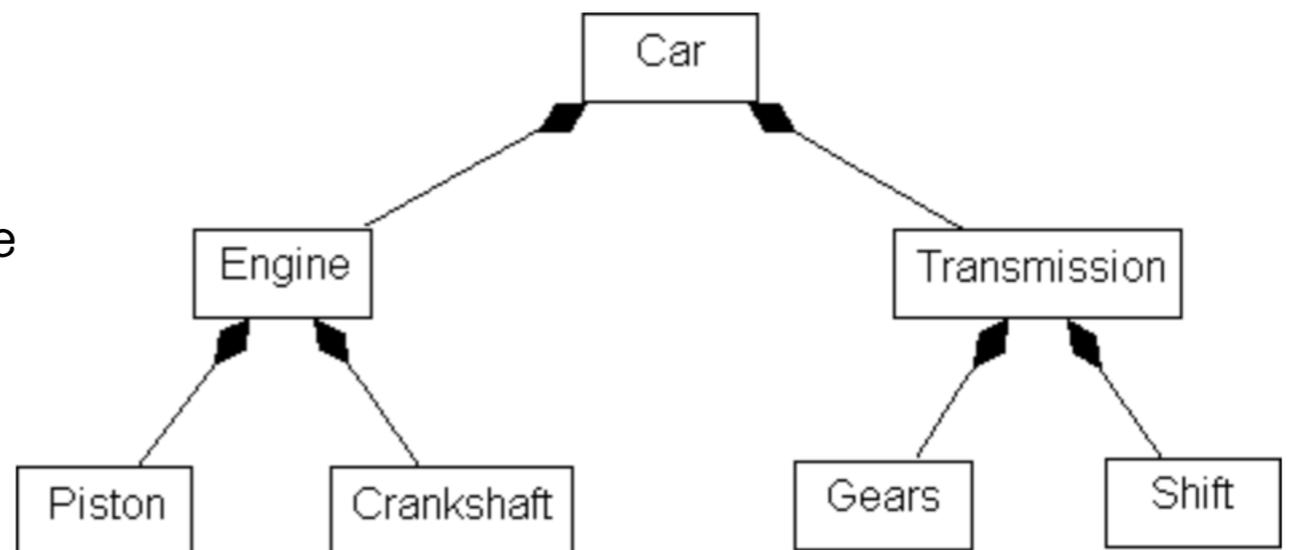
```
WHERE p.partie = pr.sous_partie
```

```
)
```

```
SELECT sous_partie, SUM(quantite) as quantite_totale
```

```
FROM parties_incluses
```

```
GROUP BY sous_partie ;
```



pour faire la somme des quantités, conserver les doublons avec UNION ALL (si par exemple la même quantité de vis est nécessaire pour chacune des 4 roues)

- Ici les données sont hiérarchiques (i.e., forment un arbre : pas de cycle), on ne bouclera donc pas indéfiniment.

Requêtes d'accessibilité

- Trouver tous les prérequis d'un cours :

```
WITH RECURSIVE c_prereq(id_cours, id_prereq) AS
(
    SELECT id_cours, id_prereq FROM prereq
UNION
    SELECT prereq.prereq_id, c_prereq.id_cours
    FROM prereq, c_prereq
    WHERE prereq.id_cours = c_prereq.id_prereq
)
SELECT *
FROM c_prereq ;
```

Requêtes d'accessibilité

- Calculer la somme des n premiers entiers :

```
WITH RECURSIVE t(n) AS
(
  VALUES (1)
  UNION
  SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

- Type d'expressivité typique des langages de programmation (ce que n'est pas SQL) : la récursion fait ici considérablement gagner en expressivité.

Récurtivité et négation

- Un exemple de récursion problématique :

```
WITH RECURSIVE R(A) AS
(
  SELECT S.A FROM S
    WHERE S.A NOT IN
      (SELECT R.A FROM R)
)
SELECT * FROM R ;
```

- **Attention** : cette syntaxe est incorrecte en SQL :

ERROR: recursive query "r" does not have the form
non-recursive-term UNION [ALL] recursive-term

Récurtivité et négation

- Avec une syntaxe « un peu moins » incorrecte :

```
WITH RECURSIVE R(A) AS
```

```
(
```

```
  SELECT S.A FROM S WHERE S.A <> S.A
```

```
UNION
```

```
  SELECT S.A FROM S
```

```
  WHERE S.A NOT IN
```

```
    (SELECT R.A FROM R)
```

```
)
```

```
SELECT *
```

```
FROM R;
```

- Encore une erreur :

```
ERROR: ERROR: recursive reference to query "r" must not appear within a subquery
```

Remarque : ici l'ajout de `SELECT S.A FROM S WHERE S.A <> S.A` a été réalisé

dans l'unique but de mieux respecter la syntaxe, mais le résultat de la requête

étant vide, la sémantique est la même

Récurtivité et négation

- Ok, Postgres a probablement de bonnes raisons de ne pas aimer
- Que se passe-t-il si on essaie de comprendre la sémantique de cette requête ?

```
WITH RECURSIVE R(A) AS
(
  SELECT S.A FROM S
  WHERE S.A NOT IN
  (SELECT R.A FROM R)
)
SELECT * FROM R ;
```

- On reformule d'abord la requête sous forme de règles :

$$R(x) : - S(x), \neg R(x) \quad \text{i.e.} \quad x \in R \text{ si } x \in S \text{ et } x \notin R$$

Récurtivité et négation

- On reformule d'abord la requête sous forme de règles :

$R(x) : \neg S(x), \neg R(x)$ i.e. $x \in R$ si $x \in S$ et $x \notin R$

- On considère $S = \{1,2\}$

```
WITH RECURSIVE R(A) AS
```

```
(
```

```
  SELECT S.A FROM S
```

```
  WHERE S.A NOT IN
```

```
    (SELECT R.A FROM R)
```

```
)
```

```
SELECT * FROM R;
```

- Etape 0 : \emptyset

Récurtivité et négation

- On reformule d'abord la requête sous forme de règles :

$R(x) : \neg S(x), \neg R(x)$ i.e. $x \in R$ si $x \in S$ et $x \notin R$

- On considère $S = \{1,2\}$

```
WITH RECURSIVE R(A) AS
```

```
(
```

```
SELECT S.A FROM S
```

```
WHERE S.A NOT IN
```

```
(SELECT R.A FROM R)
```

```
)
```

```
SELECT * FROM R;
```

- Etape 1 : $\{1,2\}$

Récurtivité et négation

- On reformule d'abord la requête sous forme de règles :

$R(x) : \neg S(x), \neg R(x)$ i.e. $x \in R$ si $x \in S$ et $x \notin R$

- On considère $S = \{1,2\}$

```
WITH RECURSIVE R(A) AS
```

```
(
```

```
SELECT S.A FROM S
```

```
WHERE S.A NOT IN
```

```
(SELECT R.A FROM R)
```

```
)
```

```
SELECT * FROM R;
```

- Etape 3 : \emptyset

Récurtivité et négation

- On reformule d'abord la requête sous forme de règles :

$R(x) : \neg S(x), \neg R(x)$ i.e. $x \in R$ si $x \in S$ et $x \notin R$

- On considère $S = \{1,2\}$

```
WITH RECURSIVE R(A) AS
```

```
(
```

```
  SELECT S.A FROM S
```

```
  WHERE S.A NOT IN
```

```
    (SELECT R.A FROM R)
```

```
)
```

```
SELECT * FROM R;
```

- Etape 4 : $\{1,2\}$ etc...
- Problème : le calcul ne terminera jamais...

Récurtivité et négation : morale

- Morale : ne pas utiliser NOT IN, EXCEPT et NOT EXISTS dans la partie récursive des requêtes d'accessibilité
- Le standard SQL3 a un ensemble de règles très compliquées spécifiant comment on pourrait les utiliser correctement
- Mais en fait l'implémentation du WITH RECURSIVE dans Postgres ne permet même pas d'utiliser le terme récursif dans une sous requête !
- Différence entre "Le" standard et ses implémentations (e.g., pas de requêtes récursives dans MySQL et restrictions par rapport au standard dans PostgreSQL, Oracle implémente encore différemment la récursivité sous forme de « requêtes hiérarchiques » : syntaxe START WITH et CONNECT BY).
Edit : Oracle implémente maintenant également le WITH RECURSIVE.