Bases de données.

Relations ou tables.

L'exemple qui nous servira de guide est celui d'une unique table regroupant des informations sur des films. Pour chaque film, on connaît

- son titre
- son année de réalisation
- le nom du réalisateur
- le genre (Drame, Western,...)
- le pays de production (USA, FR,...)

On regroupe ces renseignements dans un objet que l'on appelle *table* ou *relation*. On donne un nom à la table ainsi qu'à chaque colonne. Le début de la table Film pourrait être représenté ainsi

titre	annee	realisateur	genre	pays
Vertigo	1958	Hitchcock	Drame	USA
Alien	1979	Scott	Science-fiction	USA
Kagemusha	1980	Kurozawa	Guerre	JP

Les lignes et les colonnes n'ont pas le même rôle. Par exemple, les colonnes sont nommées (on parle d'attributs ou champs mais pas les lignes.
On dira que la table Film possède cinq *attributs* ou *champs* dont l'ordre n'a pas d'importance car ils possèdent un nom (deux attributs, ou colonnes, ne peuvent avoir le même nom).
Chaque ligne est considéré comme un *tuple* ou *uplet*. Là encore, l'ordre des lignes est aléatoire (il n'y a pas de numérotation) et il est interdit d'avoir deux tuples identiques dans une table. De plus, les tuples ont tous le même type, c'est à dire qu'un attribut (ou champ) donné contient toujours une valeur de même type.
Finalement, une table est la donnée d'un nombre fini de tuples ayant le même type, chaque coordonnée du tuple étant nommé par un nom d'attribut. Le schéma d'une table est décrit par les noms des attributs et leurs types :

$$\text{Table=(attribut}_1 : \text{type}_1,\ldots,\text{attribut}_p : \text{type}_p)$$

Les types permis sont limités. Les plus courants sont les suivants (on omet ici le type DATE ou TIME).

- CHAR(n) : chaîne de caractère de longueur fixée n
- VARCHAR(n) : chaîne de caractère de longueur variable $\leqslant n$
- INT : nombre entier
- FLOAT : nombre flottant (réel informatique)
- DECIMAL(n, d) : nombre décimal de n chiffres dont d après la virgule
- BOOLEAN : valeur booléenne à 3 valeurs (TRUE, FALSE, UNKNOWN)

On peut ainsi créer une table Film (vide) par la commande

```
CREATE TABLE film(titre VARCHAR(50), annee INT, realisateur VARCHAR(20),
                  genre VARCHAR(10), pays VARCHAR(4))
```

et remplir celle-ci par des commandes du type

```
INSERT INTO film(titre, annee, realisateur, genre, pays) VALUES
            ('toto',2014,'moi','comedie','FR')

INSERT INTO film(titre, realisateur, annee) VALUES ('titi','toi',2013)
```

Il existe aussi des commandes pour modifier une table en ajoutant un attribut ou en supprimant un tuple mais tout ceci est géré par l'interface graphique de notre SGBD.

Premières recherches : fonction SELECT.

On suppose désormais qu'une table a été créée et on veut "l'interroger" et obtenir des réponses à des questions du type suivant.

- Qui est le réalisateur de "Titanic" ? Notez qu'il peut ici y avoir plusieurs réponses.
- Quels sont les titres des films réalisés par Hitchcock entre 1950 et 1964 ?
- Quels sont les couples titre,réalisateur des films tournés en France en 2010 ?

L'instruction SELECT permet d'interroger une base de données. La syntaxe élémentaire est

```
SELECT attributs
    FROM table
    WHERE conditions
```

qui retourne les uplets d'attributs d'une table vérifiant un jeu de conditions. Les instructions suivantes permettent de répondre aux trois questions précédentes.

```
SELECT realisateur
     FROM film
     WHERE titre='Titanic'
SELECT titre
     FROM film
     WHERE (annee>=1950 AND annee<=1964)
SELECT titre,realisateur
     WHERE (pays='FR' AND annee=2010)
```

Remarque : la commande `SELECT` *fait une sélection (choix de tuples) et une projection (choix de certains attributs.*
Il est possible d'ordonner le résultat selon un attribut (croissant ou décroissant). La commande

```
SELECT realisateur
     FROM film
     WHERE pays='FR'
     ORDER BY annee DESC,realisateur ASC
```

cherche le nom des réalisateurs de films français et ordonne les résultats par année décroissante de réalisation et, en cas d'égalité, par ordre croissant du nom de réalisateur.
On peut enfin éliminer les doublons dans le résultat obtenu.

```
SELECT DISTINCT realisateur
     FROM film
```

On notera qu'il existe des caractères "joker" dont voici un exemple. La commande

```
SELECT *
     FROM film
     WHERE titre LIKE 'L%e'
```

recherche tous les attributs (symboles *) des tuples dont le titre commence par L et se termine par e.

Opérations ensemblistes.

SQL permet également de regrouper les résultats de plusieurs requêtes à l'aides opération de réunion (UNION), d'intersection (INTERSECT) et de différence (EXCEPT). On peut ainsi écrire

```
requete 1 UNION requete 2
```

Pour ces trois opérations, les schémas des requêtes doivent être compatibles (même nombre d'attributs, même types). Ces opérations sont effectuées sur des ensembles et les doublons sont éliminés (il faut ajouter le mot clef ALL après l'opérateur pour éviter l'élimination des doublons).
Voici une requête donnant dans l'ordre croissant les années où un film français a été réalisé mais pas de film japonais.

```
SELECT annee
     FROM film
     WHERE pays='FR'
     EXCEPT SELECT annee
          FROM film
          WHERE pays='JP'
     ORDER BY annee
```

Fonctions d'agrégation.

On dispose de "fonctions d'agrégation" qui calculent un résultat sur un groupe d'entrées. Le programme propose
- AVG pour le calcul de moyenne
- COUNT pour le nombre de tuples (il existe aussi COUNT DISTINCT)
- MAX et MIN pour une valeur minimum ou maximum
- SUM pour un calcul de somme

On peut ainsi calculer le nombre de films réalisés entre 1960 et 1969 :

```
SELECT COUNT(*)
     FROM film
     WHERE annee BETWEEN 1960 AND 1969
```

mais aussi le nombre de réalisateurs ayant tourné entre ces années

```
SELECT COUNT(DISTINCT realisateur)
     FROM film
     WHERE annee BETWEEN 1960 AND 1969
```

L'agrégation peut se faire "par groupes". Dans la commande suivante, on veut obtenir le nombre de films réalisés par chaque réalisateur et donc obtenir une table avec deux attributs : le nom du réalisateur et le nombre de films réalisés. Le compte du nombre de films doit se faire pour chaque réalisateur. On écrit alors

```
SELECT realisateur,COUNT(*)
     FROM film
     GROUP BY realisateur
```

Il est possible de donner un nom au nouvel attribut créé en utilisant le renommage et le mot clef AS.

```
SELECT realisateur,COUNT(*) as nfilm
     FROM film
     GROUP BY realisateur
     ORDER BY nfilm
```

On peut même ne faire apparaître dans le résultat que les groupes qui vérifient une condition. Par exemple, on se débarasse des réalisateurs n'ayant tourné qu'un seul film.

```
SELECT realisateur,COUNT(*) as nfilm
     FROM film
     GROUP BY realisateur HAVING nfilm>1
     ORDER BY nfilm
```

Requêtes imbriquées

Il est possible d'imbriquer des requêtes c'est à dire d'utiliser le résultat d'une requête (qui est une table) pour en lancer une autre. Pour chercher le nombre moyen de films tournés par un réalisateur, on commence par obtenir le nombre de films par réalisateur puis on recherche la moyenne de ces nombres.

```
SELECT AVG(nfilm)
     FROM (SELECT COUNT(*) AS nfilm FROM film GROUP BY realisateur)
```

On notera la possible utilisation de la commande `EXISTS` : `EXISTS relation` est un booléen qui vaut vrai si la relation est non vide (typiquement, `relation` est le résultat d'une requête et on veut savoir si celle-ci a au moins une solution). Si on veut connaître les réalisateur ayant réalisé au moins un film durant la même année que Hitchcock, on peut écrire

```
SELECT realisateur
     FROM film AS F1
     WHERE EXISTS(SELECT * FROM film AS F2
               WHERE F2.realisateur='Hitchcock' AND F1.annee=F2.annee)
```

On notera ici la table `film` est utilisée deux fois. Pour pouvoir différencier les tuples de la requête et de la sous-requête, on utilise un pseudonyme (renommage de table) grce à la commande `AS`.
Cela peut se compliquer autant que l'on veut…Imaginons que l'on cherche les réalisateurs ayant tourné le plus de film. Il faut réfléchir à une stratégie ! Tout d'abord, on sait obtenir la table donnant le nombre de films par réalisateur (voir plus haut)

```
SELECT realisateur,COUNT(*) as nfilm
     FROM film
     GROUP BY realisateur
```

Dans cette table, il faut chercher le maximum pour l'attribut `nfils` et ne selectionner que les réalisateurs des tuples atteignant ce maximum. On va donc obtenir une syntaxe du type

```
SELECT realisateur
     FROM (SELECT realisateur,COUNT(*) as nfilm FROM film GROUP BY realisateur)
          WHERE nfilm=...
```

Les ... doivent être égal au maximum des nombres de film. On sait obtenir ce maximum par la commande

```
SELECT MAX(nfilm)
     FROM (SELECT COUNT(*) as nfilm FROM film GROUP BY realisateur)
```

Il est cependant délicat d'utiliser plusieurs fois le même nom d'attribut `nfilm`. Dans la commande précédente, on le remplace donc, par exemple, par `N`. On obtient

```
SELECT realisateur
     FROM (SELECT realisateur,COUNT(*) as nfilm FROM film GROUP BY realisateur)
     WHERE nfilm=(SELECT MAX(N) FROM (SELECT COUNT(*) as N FROM film GROUP BY realisateur))
```

De l'utilité de tables multiples.

Comme on l'a évoqué plus haut, une manière sommaire de regrouper les données serait d'utiliser une unique table. Ceci n'est cependant pas souhaitable. Imaginons que nous voulions disposer d'une grande banque de données sur le cinéma. Pour chaque film, on veut connaître le titre, l'année de sortie, le réalisateur, les acteurs principaux ainsi que des renseignements sur ceux-ci (année de naissance par exemple). Chaque réalisateur ou acteur ayant plus d'un film à son actif, un même renseignement sera alors stocké de multiples fois. De plus, quand on va ajouter un tuple lors de la sortie d'un d'un film, on risque de faire des erreurs en orthographiant différemment le nom d'un réalisateur déjà connu. Les choses se compliquent encore si on veut utiliser ces renseignements en liaison avec un site donnant les films à l'affiche (les renseignements sur les films n'ont, eux, pas à être modifiés). Pour toute ces raisons, on choisit de travailler avec plusieurs tables en mettant ces tables en relation. Dans l'exemple précédent, on pourrait imaginer gérer tout d'abord les trois tables suivantes.

- Une table `actrice` donnant le nom et les renseignements sur les actrices (nom, prénom, année de naissance).
- Une table `acteur` donnant le nom et les renseignements sur les acteurs (nom, prénom, année de naissance).
- Une table `realisateur` donnant le nom et les renseignements sur les réalisateurs (nom, prénom, année de naissance).

La table `film` donnant les renseignements sur les films donnera sans doute le titre du film, son année de sortie, un résumé. Elle devra aussi indiquer quels tuples des trois tables précédentes correspondent à l'actrice et à l'acteur principaux et au réalisateur. Or, les tuples d'une table ne sont pas numérotés et il faut donc trouver une méthode pour repérer sans ambigité un tuple.

Le principe de base est l'utilisation d'une *clé primaire*. Il s'agit d'un ensemble d'attributs d'une table qui détermine au plus un tuple de la table. En d'autre termes, deux tuples de la table ne peuvent avoir la même valeur pour tous les attributs de la clé primaire. Si on dispose d'une clé primaire, on peut repérer chaque tuple en donnant la valeur des attributs de la clé pour ce tuple.

Le cas le plus simple est celui où on trouve une clef primaire utilisant un unique attribut. Dans les exemples précédents, ce n'est pas possible (deux acteurs peuvent avoir le même nom par exemple ; on peut même envisager qu'il existe deux actrices ayant mêmes nom et prénom et nées la même année, il n'y a donc tout simplement pas de clé primaire). On ajoute ainsi souvent un attribut dont l'objet est justement de jouer le rôle de clé primaire. Il est ainsi possible de définir un attribut qui prend comme valeur un entier incrémenté à chaque création d'un tuple pour assurer qu'on obtient bien une clé primaire. On écrit ainsi

```
CREATE TABLE acteur(
      id INTEGER AUTO_INCREMENT PRIMARY KEY,
      nom VARCHAR(20),
      prenom VARCHAR(10),
      annaiss INTEGER(4))

CREATE TABLE actrice(
      id INTEGER AUTO_INCREMENT PRIMARY KEY,
      nom VARCHAR(20),
      prenom VARCHAR(10),
```

```
      annaiss INTEGER(4))

CREATE TABLE realisateur(
      id INTEGER AUTO_INCREMENT PRIMARY KEY,
      nom VARCHAR(20),
      prenom VARCHAR(10),
      annaiss INTEGER(4))
```

Remarques.

- *Avec sqliteman, les choses se présentent différement. Lors de la création de la table avec l'interface graphique, on choisit un "type" clef primaire.*
- *Dans la présentation du schéma d'un table, on souligne les attributs correspondant à une clef primaire.*

Quand on définit la table des films, on utilise des attributs faisant référence à des clefs primaires. On parle de clefs étrangères. On peut ainsi écrire

```
CREATE TABLE film(
      id INTEGER AUTO_INCREMENT PRIMARY KEY,
      titre VARCHAR(50),
      annee INTEGER(4),
      resume VARCHAR(240),
      idacmas INTEGER REFERENCES acteur(id),
      idacfem INTEGER REFERENCES actrice(id),
      idreal INTEGER REFERENCES realisateur(id))
```

Indiquer à quelle clef primaire est reliée la clef étrangère permet au SGBD de s'assurer lors de la création d'un tuple d'une table possédant une clef étrangère que la valeur de celle-ci correspond bien à une valeur utilisée de la clé primaire. On peut de même, lors de la création de la table, indiquer que lors de la suppression d'un tuple d'une table avec clé primaire il faut aussi supprimer les tuples correspondant dans les tables ayant une clef étrangère associée.
Remarque : avec sqliteman, on ne se préoccupera pas de ces points. Notre objectif est essentiellement d'utiliser des tables existantes et pas de les créer.

Requêtes à tables multiples.

On peut utiliser les outils de requête pour faire des recherches croisées dans nos différentes tables. Pour savoir quels réalisateurs ont tourné un film contenant le mot "facteur" on peut ainsi écrire

```
SELECT nom,prenom
     FROM realisateur
     WHERE EXISTS (SELECT idreal FROM film
              WHERE titre LIKE '%facteur%' AND idreal=realisateur.id)
```

Notez que dans la seconde ligne, il faut préciser que l'attribut `id` se rapporte à la table `realisateur` (puisque la la table `film` contient aussi un attribut `id`).
Il existe des moyens spécifiques pour "coupler" les renseignements de plusieurs tables. L'outil mathématique "naturel" est le produit cartésien. A partir de deux relations R_1 (avec n_1 attributs) et R_2 (avec n_2 attributs), on construit la relation $R_1 \times R_2$ (qui aura $n_1 + n_2$ attributs). Le produit cartésien des relations acteur et actrice (qui décrit donc des couples…) a le schéma suivant

id	nom	prenom	annaiss	id	nom	prenom	annaiss

Dans le langage SQL, le produit cartésien de deux relations s'obtient en mettant les relations concernées, séparées par des virgules, juste après le mot FROM (on peut aussi utiliser le mot clef CROSS JOIN)

```
SELECT * FROM acteur,actrice
SELECT * FROM acteur CROSS JOIN actrice
```

Comme dans l'exemple précédent, on ne peut utiliser un attribut comme `id`, qui apparaît deux fois, sans préciser à quelle relation il se réfère. Pour obtenir tous les couples nom d'acteur, nom d'actrice ayant la même date de naissance, on écrit donc

```
SELECT acteur.nom,actrice.nom FROM acteur,actrice where acteur.annaiss=actrice.annaiss
```

Dans le cas de tables reliées par des couples clé primaire-clé étrangère, il est bien évident qu'un produit cartésien n'est pas adapté. Il convient d'effectuer une *jointure*, c'est à dire de relier les attributs qui se correspondent. Pour savoir quels réalisateurs ont tourné un film contenant le mot "facteur" on peut former la jointure des tables film et realisateur et chercher dans le résultat les noms voulus.

```
SELECT nom,prenom
     FROM film JOIN realisateur ON idreal=realisateur.id
     WHERE titre LIKE '%facteur%'
```

On peut joindre plusieurs relations et la condition (après le ON) est générale et ne porte pas seulement sur une égalité clef primaire-clef étrangère. On peut aussi faire une jointure interne en joignant une relation avec elle même après renommage. La commande suivante permet de trouver les acteurs homonymes.

```
SELECT A1.nom, A1.prenom,A2.nom,A2.prenom
     FROM acteur AS A1 JOIN acteur AS A2 ON (A1.nom=A2.nom AND A1.id<A2.id)
```

La condition `A1.id<A2.id` permet ici d'éviter les doublons et de considérer qu'un acteur est son propre homonyme.

Compléments : facultatif

Un peu de théorie : le modèle relationnel.

Avec ce qui précède et une pratique raisonnable, vous devriez être capable de faire des recherches assez complexes dans les bases de données déjà construites. Avant d'écrire une requête, il faut cependant prendre le temps de la réflexion pour dégager une stratégie générale de recherche.
On va ici reprendre l'ensemble des outils introduits avec le langage SQL en utilisant un formalisme issu de la logique.
Un *schéma relationnel* est un ensemble de n-uplets de la forme $S = (A_1, \ldots, A_n)$ où les A_i, appelés attributs, sont deux à deux distincts et possède un *domaine* de valeurs.
Exemple : le schéma relationnel de la table `acteur`Êest acteur(id,nom,prenom,annaiss) où le domaine de l'attribut `nom` est l'ensemble des chaînes de caractères.
Une *relation* ou *table* $R(S)$ sur un schéma relationnel $S = (A_1, \ldots, A_n)$ est un ensemble de n-uplets (éléments du produit cartésien des domaines des A_i).
Si deux relations $R_1(S)$ et $R_2(S)$ ont le même schéma, on peut leur appliquer les opérations ensemblistes d'union, d'intersection ou de différence. On note $R_1(S) \cup R_2(S)$, $R_1(S) \cap R_2(S)$ et $R_1(S) - R_2(S)$.
Soit $R(S)$ une relation de schéma S et $X \subset S$. On appelle *projection*Êde R sur X la relation notée $\pi_X(R)$ obtenue à partir de S en ne gardant que les attributs dans X.
Soit $R(S)$ une relation de schéma S, $A \in S$ (un attribut) et a un élément du domaine de A. On note $\sigma_{A=a}(R)$ la relation sur S obtenue en ne conservant que les tuples de R dont la valeur de l'attribut A est a. σ est l'opérateur de sélection simple. On peut aussi l'utiliser pour comparer des attributs de même domaine (et écrire, par exemple, $\sigma_{A=B}(R)$).
Soit R une relation de schéma $(A_1, \ldots, A_n)$. On note $\rho_{A_i \leftarrow B_i}(R)$ la relation dont le schéma est $(A_1, \ldots, A_{i-1}, B_i, A_{i+1}, \ldots, A_n)$ et qui possède les mêmes valeurs que R : c'est l'opérateur de renommage. On se permet un renommage multiple en écrivant, par exemple $\rho_{A_1,A_2 \leftarrow B_1,B_2}(R)$.
De même que vous avez appris à traduire en langage SQL des requêtes formulées en français, de même vous êtes invités à essayer de le faire en "langage relationnel", c'est à dire en utilisant les opérateurs précédents.
Exemples : on travaille avec la relation `film` dont le schéma relationnel est

$$\texttt{film(id,titre,annee,resume,idacmas,idacfem,idereal)}$$

- La requête "quels sont les titres des films tournés dans les années 60 ?" a pour réponse les éléments de

$$\pi_{\texttt{titre}}\left(\sigma_{\texttt{annee} \in [1960,1969]}(\texttt{film})\right)$$

- La requête "quels sont les réalisateurs ayant tourné un film dans les années 60 et aussi dans les années 70 ?" a pour réponse les éléments de $R_1 \cap R_2$ où

$$R_1 = \pi_{\texttt{idreal}}\left(\sigma_{\texttt{annee} \in [1960,1969]}(\texttt{film})\right)$$

$$R_2 = \pi_{\texttt{idreal}}\left(\sigma_{\texttt{annee} \in [1970,1979]}(\texttt{film})\right)$$

- Les éléments de

$$\pi_{t1}\left(\sigma_{\texttt{a1} \neq \texttt{a2}}\left(\rho_{\texttt{annee,titre} \leftarrow a1,t1}(\texttt{film}) \times \rho_{\texttt{annee,titre} \leftarrow a2,t2}(\texttt{film})\right)\right)$$

  sont des titres de films ayant été réalisés (pas forcément avec la même équipe) au moins deux années différentes.

Pour être complets, il nous manque les fonctions d'agrégation. Soit $R(S)$ une relation, $A \in S$ et f une fonction d'agrégation (comptage, maximum, minimum, somme ou moyenne). On note $\gamma_{f(A)}(R)$ le résultat de l'application de la fonction f à la relation R pour l'attribut A. Comme les fonctions d'agrégation, sont souvent utilisées avec un regroupement, on utilise aussi une notation plus complète du type ${}_{groupe}\gamma_{f(A)}(R)$.
Exemples.

- Pour obtenir l'année moyenne de production des films, on écrit

$$\gamma_{AVG(\texttt{annee})}(\texttt{film})$$

- Pour obtenir le nombre total de films de chaque réalisateur, on peut écrire

$${}_{\texttt{idreal}}\gamma_{COUNT(*)}\left(\texttt{film}\right)$$

- La requête
  `SELECT idreal,COUNT(*) as nfilm FROM film GROUP BY idreal HAVING nfilm>1`
  peut se traduire en algèbre relationnelle par

$$\sigma_{\texttt{nfilm}>1}\left(\rho_{\texttt{nfilm} \leftarrow \texttt{COUNT}(*)}\left({}_{\texttt{idreal}}\gamma_{\texttt{COUNT}(*)}\right)\right)$$

On peut enfin noter que le symbole $\bowtie$ désigne la jointure simple $R_1 \bowtie_{A_1=A_2} R_2$ correspondant à R_1 `JOIN` R_2 `ON` $A_1 = A_2$ et pouvant aussi s'écrire $\sigma_{A_1=A_2}(R_1 \times R_2)$.

Complément : la division cartésienne.

En algèbre relationnelle, on dispose aussi d'une *division cartésienne* qui est une sorte d'inverse du produit cartésien. Cette opération est introduite pour des raisons théoriques que nous n'expliciterons pas mais est, en pratique, souvent absente des langages de requête comme SQL.
Soient R, T deux relations telles que tous les attributs de T sont attributs de R (le schéma de T est un "sous-ensemble" de celui de R). Notons $C_1, \ldots, C_p$ les attributs de T et $A_1, \ldots, A_n$ les autres attributs de R. La division de R par T est la relation D de schéma $(A_1, \ldots, A_n)$ qui est composées des valeurs $(v_1, \ldots, v_n)$ telles que pour TOUT choix $(s_1, \ldots, s_p)$ de valeurs de S on ait $(v_1, \ldots, v_n, s_1, \ldots, s_p) \in R$. On a alors $D \times S = R$.
Exemple : prenons le cas simple où $n = p = 1$ avec les relations suivantes

eleve	prof
Toto	Dupont
Lili	Dupont
Toto	Durand
Max	Durand
Lili	Durand

eleve
Toto
Lili
Max

La division de la première table par la seconde donne la table à un attribut `prof` contenant l'unique élément Durand car c'est l'unique professeur ayant tous les élèves…

SGBDR, le modèle client serveur.

L'utilisation d'une base de données telle qu'on l'a vue, c'est à dire par un utilisateur effectuant des requêtes auprès d'un gestionnaire de bases de données s'appelle une *architecture clien-serveur*.
L'utilisateur est le client qui effectue des demandes auprès du serveur, lequel centralise les informations. Plusieurs clients peuvent simultanément accéder au serveur mais celui-ci doit veiller à l'ordre dans lequel sont traitées les demandes (il ne faut pas qu'une table soit modifiée au cours d'une requête d'un client).
L'architecture client-serveur est rarement mise en oeuvre telle quelle. Dans l'utilisation quotidienne des bases de données, les utilisateurs n'ont pas un accès direct à la base mais transitent par une application. On parle d'*architecture trois tiers* :

- le tiers utilisateur
- le tiers applicatif
- le tiers base de données

Dans tous les cas, l'utilisateur n'a pas à se préoccuper de la façon dont le SGBD va s'arranger pour extraire les informations de la base de données. Il doit juste se contenter de fournir une requête non ambigu. C'est le SGBD qui choisit la stratégie de requête en optimisant la demande faite par l'utilisateur (une même requête peut s'exprimer de multiples façons).