3I009 : Systèmes de Gestion de Bases de Données

http://www-bd.lip6.fr/wiki/doku.php?id=site:enseignement:licence:3i009:start

Description du cours

Ce module s'inscrit dans la suite du module 2I009 de L2. Après un bref rappel des concepts vus en L2, le cours présente comment l'algèbre relationnelle utilisée en interne par les SGBD pour évaluer les requêtes. Ensuite, le contrôle de la concurrence puis l'optimisation de schéma sont abordées Enfin les outils classiques de bases de données (triggers instead of et vues) viennent compléter le module.

Toutes les séances sont **TD** en première partie et **TME** en seconde partie.

Actualités et informations pratiques

Lire les instructions de connexion oracle
Lire les instructions d'utilisation de H2
Récuperer Raeval et les fichiers foofle RaevalFoofle
Notes de cours S. Gançarski 2003 (un peu ancien, quelques bugs, mais utile) polylicbd2003.pdf

Contrôle de connaissance

Le contrôle de connaissances est composé d'un examen final (60%) et d'un contrôle continu (40%) sous la forme de 4 interros (10% chacune). Les interros auront lieu pendant les séances de TD, les semaines 4, 7, 9 et 12 (voir planning) Tous les documents (déjà lus) sont autorisés à l'examen. Pour les interros, uniquement une feuille A4 recto/verso manuscrite

Bases de Données : rappels et problèmes à traiter

- **Fichiers et Bases de Données**
- Systèmes de Gestion de Bases de Données (SGBD)
- Langages et modèles de données
- Modèle Entité-Association
- Modèle Relationnel
- Calcul relationnel et SQL

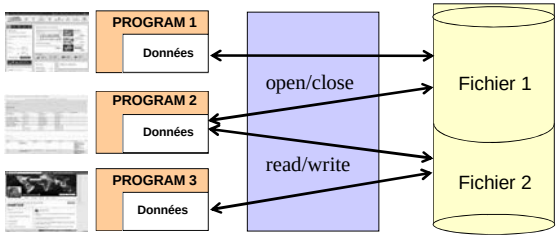
Qu'est-ce qu'une Base de Données (BD) ?

- Une **base de données** (BD) est une *collection de données structurées* sur des entités (objets, individus) et des relations dans un contexte (applicatif) particulier.
- Un **système de gestion de base de données** (SGBD) est *un (ensemble de) logiciel(s)* qui facilite la création et l'utilisation de bases de données.
- Les données sont définies, administrées et gérées en utilisant des langages fondés sur des modèles de données (2I et 3I, modèle *relationnel*).

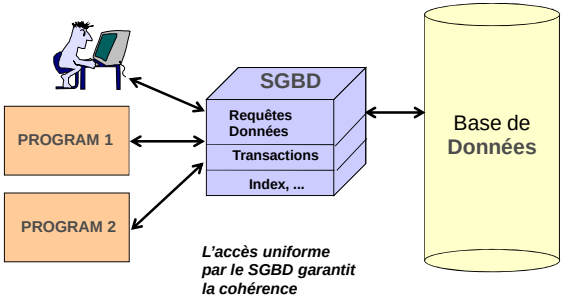
Fichier ≠ Bases de Données

Fichiers :

- opérations simples : ouvrir/fermer, lire/écrire
- différentes méthodes d'accès (séquentiel, indexé, haché, etc.)
- utilisation par plusieurs programmes difficile (format ?, concurrence)
- La gestion de données structurées dans des fichiers représente une partie importante du **coût de développement et de maintenance** d'applications.



Approche « Base de Données »



Problèmes avec les fichiers résolus avec une base de données

Fichier :

- Faible structuration des données
- Dépendance entre programmes et fichiers
- Redondance des données
- Absence de contrôle de cohérence globale des données
- Accès aux nuplets un par un, par programme

Base de Données :

- Structuration des données à travers un schéma de données
- **Indépendance entre programmes et gestion de données**
- Données partagées
- Contrôle de la cohérence logique et physique (schémas, transactions)
- Accès ensembliste et déclaratif

Objectifs du cours 3I009

- Étudier les « bases de données » du point de vue :
  - *de l'utilisateur : accès par appli ou SQL (expert)*
  - ***du développeur : définir schéma (EA,rel.) et implémenter applis***
  - *de l'administrateur :* schéma physique et performances (tuning), sécurité et fiabilité
- Comprendre les **principes** des Systèmes de Gestion de Bases de Données (SGBD) Relationnels
- Apprendre à **construire des applications** sur un SGBD
- Étudier les **mécanismes internes** des SGBD

Plan général du cours

- Rappel SGBD, modèle et langages relationnel
- Evaluation et optimisation des requêtes : du stockage disque au résultat — *Performance d'exécution*
- Transactions et tolérance aux pannes
- Contrôle de concurrence — *Cohérence d'exécution multiutilisateur*
- Dépendances fonctionnelles
- Normalisation de schémas relationnels — *Théorie de la conception*
- SQL : Triggers et vues, JDBC — *Outillage pour le développement*

Bibliographie

Livres en français (aussi disponibles à la bibliothèque MathInfo Enseignement) :

- C. Chrisment et.al., Bases de données relationnelles, Hermes - Lavoisier
- G. Gardarin. Bases de données - objet et relationnel. Eyrolles.
- J.L. Hainaut, Bases de données : concepts, utilisation et développement, Dunod
- S. Abiteboul, R. Hull, V. Vianu, Les fondements des bases de données, Vuibert

Livres en anglais :

- R. Ramakrishnan and J. Gehrke. Database Management Systems. 3e édition, McGraw Hill, 2002 - http://pages.cs.wisc.edu/~dbbook/
- H.G. Molina, J.D. Ullman, J. Widom. Database Systems: The Complete Book, Goal Series - http://infolab.stanford.edu/~ullman/dscb.html
- C.J. Date, Introduction aux bases de données, 7e édition, Vuibert
- M.T. Özsu, P. Valduriez. Principles of Distributed Database Systems. 2nd edition, Prentice Hall, 1999

Notes de cours

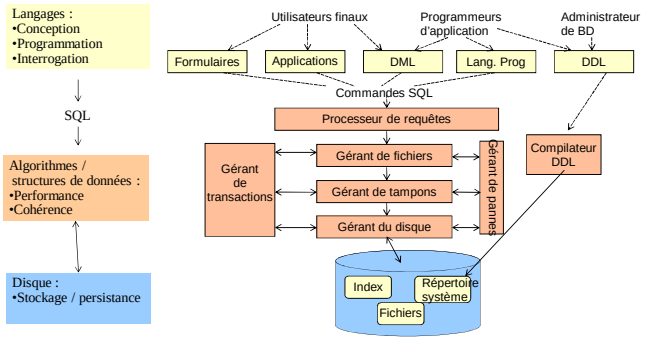
- S. Gancarski. Introduction aux bases de données. UPMC, Paris 6, janvier 2003 – lien sur le site web

Système de Gestion de Bases de Données (SGBD) Fonctions

- Représentation et structuration de l'information :
  - « Modéliser le monde réel et ses règles de fonctionnement »
  - Description de la *structure des données* : Employés(nom, age, salaire)
  - Description de *contraintes logiques sur les données* : 0 ≤ age ≤ 150, ...
  - *Vue : réorganisation (virtuelle)* de données pour des besoins spécifiques
- Gestion de l'intégrité des données :
  - Vérification des contraintes spécifiées dans le schéma
  - Exécution *transactionnelle* des requêtes (mises-à-jours)
  - Gestion de la concurrence multi-utilisateur et des pannes
- Traitement et optimisation de requêtes :
  - La performance est un problème géré par l'administrateur du SGBD et non pas par le développeur d'application (indépendance physique)

Objectif : faciliter le partage de grands volumes de données entre différents utilisateurs / applications

Architecture d'un SGBD



Langages et interfaces d'un SGBD

- **Langages de *conception*** : Entité-Association, UML
  - Utilisation : conception *haut-niveau* d'applications (données et traitements)
- **Langage base de données :** SQL, calcul relationnel, algèbre
  - langages *déclaratifs* : spécifier « *quoi* » (SQL) et non « *comment* » (Algèbre, généré par le SGBD)
  - puissance d'expression limitée par rapport à un langage de programmation comme C ou Java
  - utilisation : *définition schémas, interrogation et mises-à-jour, administration (SQL)*
- **Langages de *programmation*** : PL/SQL, Java, PHP, …
  - langages *impératifs* avec une interface SQL (ex. JDBC)
  - langages expressifs (« complet » au sens d'Alan Turing)
  - utilisation : *programmation d'applications (avec SQL pour accéder aux données)*
- **Langages de *bas niveau*:** C,…
  - Mettre en œuvre le SGBD, accès aux couches physiques, implémentation des opérateurs, ….

Langages BD (SQL)

Langage de **Définition de Données** (LDD)

- pour définir les schémas externes (vues), logiques et physiques
- les définitions sont stockées dans le répertoire système (dictionnaire)

Langage de **Manipulation de Données** (LMD)

- langage *déclaratif* pour interroger (**langage de requêtes**) et mettre à jour les données
- peut être autonome (par ex. SQL seul) ou intégré dans un langage de programmation (à travers une API comme JDBC)

Histoire des bases de données

Années 1960:

- début 1960: Charles Bachmann développe le premier SGBD, IDS, chez Honeywell
  - modèle réseau: les associations entre les données sont représentées par un graphe
- fin1960: IBM lance le SGBD IMS
  - modèle hiérarchique: les associations entre les données sont représentées par un arbre
- fin 1960: standardisation du modèle réseau Conference On DAta Systems Languages (CODASYL)

Histoire des bases de données

- 1970: Ted Codd définit le modèle relationnel au IBM San Jose Laboratory (aujourd'hui IBM Almaden)
  - 2 projets de recherche majeurs
    - INGRES, University of California, Berkeley
      - devint le produit INGRES, suivi par POSTGRES, logiciel libre, qui devint le produit ILLUSTRA, racheté par INFORMIX
    - System R, IBM San Jose Laboratory
      - devint DB2, inspira ORACLE
  - 1976: Peter Chen définit le modèle Entité-Association (Entity-Relationship)

Histoire des bases de données

Années 1980

- maturation de la technologie relationnelle
- standardisation de SQL

Années1990

- amélioration constante de la technologie relationnelle
- support de la distribution et du parallélisme
- modèle objet, ODMG, BD objets
- fin 1990 : le relationnel-objet, SQL3
- nouveaux domaines d'application: entrepôts de données et décisionnel, Web, multimédia, mobiles, etc.

Années 2000

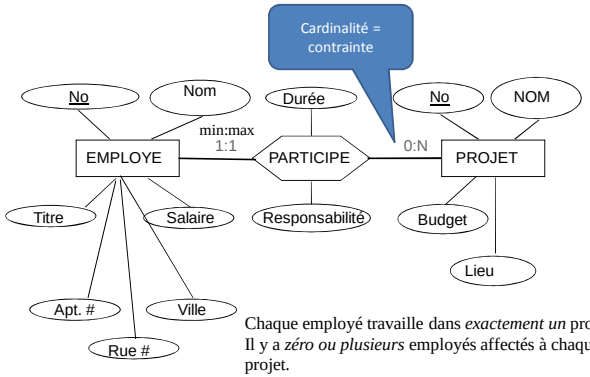
- apparition de XML et de nouvelles architectures (eg. P2P)
- NoSQL, MapReduce, RDF (Web sémantique), …

Le modèle Entité-Association (E/A)

« E/R (entity-relationship) model » en anglais

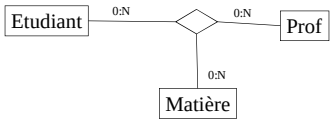
- Modèle / langage de *conception* :
  - Définition de schémas conceptuels
- Modélisation *graphique* des entités, de leurs attributs et des associations entre entités.
- Objectif :
  - détection d'erreurs de conception *avant* le développement
  - traduction « automatique » dans un modèle logique.
- Supporté par les outils CASE pour BD
- ~ partie "modélisation de données" dans UML

Schéma entité-association

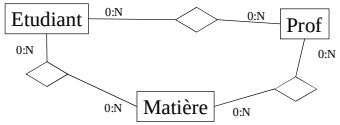


Chaque employé travaille dans *exactement un* projet. Il y a *zéro ou plusieurs* employés affectés à chaque projet.

Association ternaire



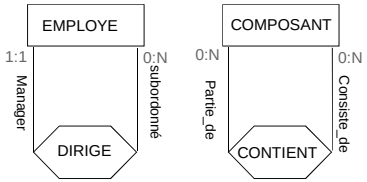
est différent de



Pourquoi ?

Association réflexive

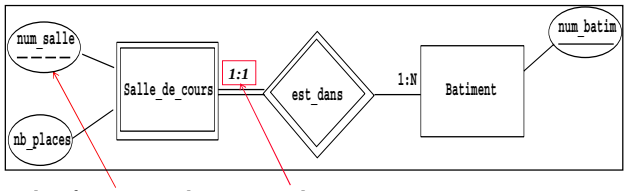
Association réflexive : une entité d'une classe C est associé à une ou plusieurs entités de la *même* classe C.



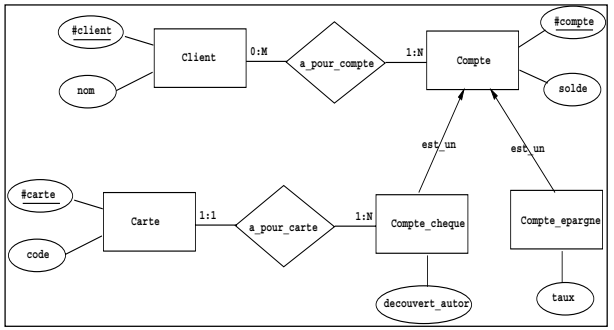
Entités fortes et faibles

Entité forte: entièrement identifiable par ses attributs

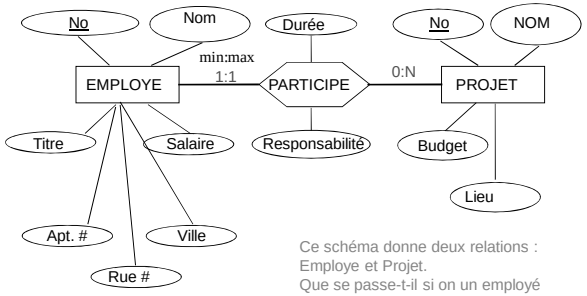
Entité faible: ne peut être identifiée que par rapport à une autre entité, dite dominante, à laquelle elle se réfère. Son identificateur est :
identificateur partiel + identificateur de l'entité dominante.



Spécialisation



Modèle relationnel et E/A



Ce schéma donne deux relations :
Employe et Projet.
Que se passe-t-il si on un employé peut être dans plusieurs projets ?
Dans aucun ?

Une base de données relationnelle

schéma

EMP

ENO	ENOM	EMPLOI
E1	J. Doe	Elect. Eng.
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

instance

PARTICIPE

ENO	PNO	RESP	DUR
E1	P1	Manager	12
E2	P1	Analyst	24
E2	P2	Analyst	6
E3	P3	Consultant	10
E3	P4	Engineer	48
E4	P2	Programmer	18
E5	P2	Manager	24
E6	P4	Manager	48
E7	P3	Engineer	36
E7	P5	Engineer	23
E8	P3	Manager	40

n-uplet

PROJ

PNO	PNOM	BUDGET
P1	Instrumentation	150000
P2	Database Develop.	135000
P3	CAD/CAM	250000
P4	Maintenance	310000
P5	CAD/CAM	500000

Monde ouvert
vs. Monde fermé

Modèle relationnel

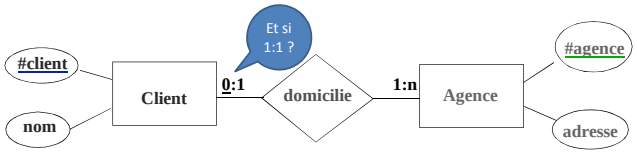
Fondements mathématiques solides :

- théorie des ensembles
- logique du premier ordre (calcul relationnel)

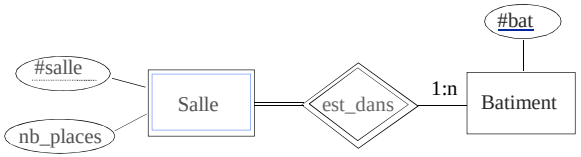
Langages de requêtes *simples, puissants et efficaces*

*Mais…*un schéma relationnel peut contenir des centaines de tables avec des milliers d'attributs.

- Problème: comment éviter des erreurs de conception?
- Deux solutions (complémentaires):
  - Génération (automatique) à partir d'un schéma E/A
  - Théorie des dépendances et normalisation (on verra plus tard)

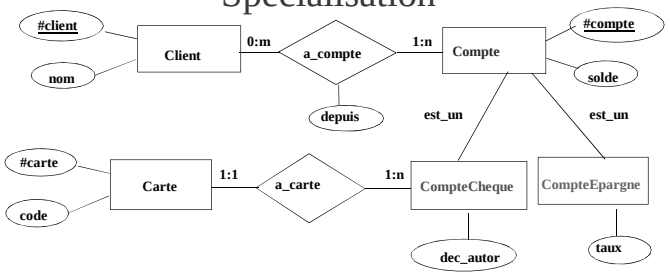


Client(#client, nom) Agence(#agence, adresse)
Domicilie(#client, #agence)



La clé de l'entité dominante fait partie de la clé de l'entité faible
Salle(#salle, #bat, nbplaces) Batiment(#bat, …)

Spécialisation



Compte(#compte, solde) CompteCh(#compte, dec-aut) CompteEp(#compte, taux)

Si Compte est un type d'entité « abstrait » (sans instances) :
CompteCh(solde, dec-aut, #compte) CompteEp(solde, taux, #compte)

Calcul relationnel de n-uplets

Une requête exprimée dans le ***calcul n-uplet*** a la forme

$$Q(x_1, x_2, \ldots, x_n) = \{ x_1.A_j,\ldots,x_2.A_k, \ldots, x_n.A_m \,/\, F(x_1, x_2, \ldots, x_n) \}$$

- F est une *formule logique*,
- $x_1, \ldots, x_n$ sont des ***variables n-uplet***,
- $x_i.A_j$ désigne l'attribut A_j d'une instance (n-uplet) de la variable x_i.

Formules logiques : syntaxe

Une formule logique F est une expression composée de

- **atomes** : R(x), T(x,y), x<3, x=y, ...
- **opérateurs booléens** :
  - $\wedge$ (conjonction)
  - $\vee$ (disjonction)
  - $\neg$ (négation)
- **quantificateurs :**
  - $\exists$ (quantificateur existentiel)
  - $\forall$ (quantificateur universels)
- virgules et parenthèses

$\Rightarrow$ F = formule de la l*ogique du premier ordre sans fonctions*

Exemples de requêtes

Emp (Eno, Ename, Title, City) **Project**(Pno, Pname, Budget, City)
Pay(Title, Salary) **Works**(Eno, Pno, Resp, Dur)

- Q(x)={ x.Ename | Emp(x) }
- Q(x)={ x.Pname, x.Budget | Project(x) }
- Q(x)={ x.Title | Emp(*x*) }
- Q(x)={ x.Ename | Emp(x) $\wedge$ x.City = 'Paris' }
- Q(x)={ x.City | Emp(x) $\vee$ Project(x) }
- Q(x)={ *x*.City | Project(*x*) $\wedge \neg\exists y$ (Emp(*y*) $\wedge$ *x*.City = *y*.City)}

- Traduction en SQL ?

Exemples de requêtes

Emp (Eno, Ename, Title, City) **Project**(Pno, Pname, Budget, City)
Pay(Title, Salary) **Works**(Eno, Pno, Resp, Dur)

- Noms de tous les employés?

  Q(x)={ x.Ename | Emp(x) }

la variable libre x est liée à tous les n-uplets de la table Emp

- Noms de tous les projets avec leurs budgets?

  Q(x)={ x.Pname, x.Budget | Project(x) } :

la variable libre x est liée à tous les n-uplets de la table Project

- Titres (d'emploi) pour lequel il y a au moins un employé?

  Q(x)={ *x*.Title | Emp(*x*) }

Exemples de requêtes

Emp (Eno, Ename, Title, City) **Project**(Pno, Pname, Budget, City)
Pay(Title, Salary) **Works**(Eno, Pno, Resp, Dur)

- Employés qui travaillent à Paris?

  Q(x)={*x.Ename* | Emp(*x*) $\wedge$ *x*.City = 'Paris' }

x est liée à tous les n-uplets t de Emp où t.City = 'Paris'

- Villes où il y a un employé ou un projet?

  Q(x)={ *x*.City | Emp(*x*) $\vee$ Project(*x*) }

x est liée à tous les n-uplets t de Emp et à tous les n-uplets t' de Project

- Villes où il y a des projets mais pas d'employés?

  Q(x)={ *x*.City | Project(*x*) $\wedge \neg\exists y$(Emp(*y*) $\wedge$ *x*.City = *y*.City) }

Exemple de requêtes

Emp (Eno, Ename, Title, City)
Pay(Title, Salary)

Project(Pno, Pname, Budget, City)
Works(Eno, Pno, Resp, Dur)

- Noms des projets de budget > 225?

Q(x)={ x.Pname | Project(x) ∧ x.Budget > 225 }

- Noms et budgets des projets où travaille l'employé E1?

Q(x)={ x.Pname, x.Budget | Project(x) ∧
∃y (Works(y) ∧ x.Pno=y.Pno ∧ y.Eno='E1') }

Sûreté des requêtes

Problème:

- La taille de Q(x)={ *x.A* | *F*(*x*) } doit être finie

- Exemple: le résultat de Q(x)={ x.A | ¬R(x) } *est infini : x est lié à tous les n-uplets qui ne sont pas dans la table R*

Sûreté:

- Une requête est sûre si, pour toute BD conforme au schéma, le résultat de la requête peut être calculé en utilisant seulement les constantes apparaissant dans la BD et la requête.
- Puisque la BD est finie, l'ensemble de ses constantes est fini de même que les constantes de la requête; donc, *le résultat de la requête est fini.*

Sûreté des requêtes (suite)

La *caractérisation syntaxique* de requêtes sûres est difficile.

Quelques conseils pour construire des requêtes sûres :

- Toujours commencer une requête par {x.A | R(x) …}
- Une quantification ∃x ou ¬∃x doit toujours être suivie d'un atome R(x) : x est bornée aux n-uplets de la table R
- Éviter d'utiliser ∀ et le remplacer par ∃ grâce à l'équivalence (∀x R(x)) ⇔ ¬∃ x ¬R(x) (on revient sur le cas d'avant)

- La syntaxe SQL garantit la formulation de requêtes sûres (il n'y a pas de ∀ générique)

Requêtes d'interrogation SQL

Structure de base d'une requête SQL simples :

```
SELECT [DISTINCT]   var_i.A_ik, …     attributs
FROM      R_i1 var_1, R_i2 var_2…     tables
WHERE     P                           prédicat/condition
```

où:

- var_i désigne la table R_{ij}
- Les variables dans la clause SELECT et dans la clause WHERE doivent être *liées* dans la clause FROM
- Le mot clé DISTINCT (optionnel) permet d'éliminer des doublons.

Simplifications :

- Si var_j n'est pas spécifiée, alors la variable s'appelle par défaut R_{ij}.
- Si une seule table/variable *var* possède l'attribut A, on peut écrire plus simplement A au lieu de *var*.A.

Exemples de requêtes

•Q(x)={ x.Ename | Emp(x) }

select Ename from Emp

•Q(x)={ x.Ename | Emp(x) ∧ x.City = 'Paris' }

select Ename from Emp where City = Paris

•Q(x)={ x.Pname, x.Budget | Project(x) ∧
∃y (Works(y) ∧ x.Pno=y.Pno ∧ y.Eno='E1') }

select x.Pname, x.Budget from Project x, Works y
where x.Pno=y.Pno and y.Eno='E1'

•Q(x)={ x.City | Emp(x) ∨ Project(x) }

(select City from Emp) union (select City from Project)

•Q(x)={ *x*.City | Project(*x*) ∧ ¬∃*y* (Emp(*y*) ∧ *x*.City = *y*.City)}

select City from Project
where not exists (select * from Emp
where Project.City = Emp.City)

Requêtes et valeurs NULL

Les valeurs d'attributs peuvent être inconnues : NULL

- une *opération* avec un attribut de valeur NULL retourne NULL
- une *comparaison* avec un attribut de valeur NULL retourne **UNKNOWN**
- UNKNOWN introduit une logique *à trois valeurs* :
- Vrai = 1, UNKNOWN = 0.5, Faux = 0
- x AND y = min(x,y), x OR y = max(x,y), not(x) = 1-x
- Attention : NULL n'est pas une constante :
  - « NAME = NULL » ou « NULL + 30 » sont incorrects.
  - Pour vérifier si la valeur d'un attribut est inconnue, on utilise IS NULL :

```
SELECT Pname
FROM Proj WHERE City IS NULL
```


Requête avec NULL

Project

Pno	Pname	Budget	City
1	Développement	NULL	Paris
2	Conception	20000	Lyon

```
SELECT Pno FROM Project
WHERE Budget = 1000
```

→ **Réponses ???**

```
SELECT P1.Pno FROM Project P1, Project P2
WHERE P1.Budget = P2.Budget AND P1.Pno <> P2.Pno
```


Insertion de tuples

INSERT INTO *table* [(*column* [, ...])]
{ VALUES ({ *expression* | DEFAULT } [, ...]) | *query* }

•en spécifiant des valeurs différentes pour tous les attributs *dans l'ordre* utilisé dans CREATE TABLE :

INSERT INTO *R* **VALUES** (value(A_1), …, value(A_n))

•en spécifiant les noms d'attributs (indépendant de l'ordre) :

INSERT INTO *R* (A_i, …, A_k) **VALUES** (value(A_i), …, value(A_k))

•insertion du résultat d'une requête (copie) :

INSERT INTO *R* <*requête_SQL*>

Suppression de tuples

DELETE FROM ***table*** **[WHERE** ***condition*** **]**

Supprimer tous les employés qui ont travaillé dans le projet P3 pendant moins de 3 mois :

DELETE FROM Emp
WHERE Eno **IN (SELECT** Eno
FROM Works
WHERE PNO='P3' **AND** Dur < 3**)**

Attention : il faut aussi effacer les n-uplets correspondants dans la table Works (cohérence des données).

Modification de tuples

UPDATE ***table***
SET ***column*** **= { *expression* | DEFAULT } [, ...]**
[WHERE ***condition*** **]**

UPDATE *R*
SET A_i=value, …, A_k=value
WHERE *P*

Commandes DDL

Création de schémas :

```
CREATE SCHEMA nom_schema AUTHORIZATION nom_utilisateur
```

Création de tables :

```
CREATE TABLE nom_table
   (Attribute_1 <Type>[DEFAULT <value>],
    Attribute_2 <Type>[DEFAULT <value>],
    …
    Attribute_n <Type>[DEFAULT <value>]
   [<Constraints>])
```


Type DATE

- Format par défaut : YYYY-MM-DD
- Fonctions :
  - SYSDATE : date / heure actuelle
  - TO_DATE('98-DEC-25:17:30','YY-MON-DD:HH24:MI')

SELECT * **FROM** my_table
WHERE datecol = TO_DATE('04-OCT-2010','DD-MON-YYYY');

- Arithmétique :
  Date +/- N jours
  SYSDATE + 1 = demain

On peut utiliser la fonction Extract (date|year|…) from expression_date

Autres commandes DDL

DROP SCHEMA *nom_schema* [,…] [CASCADE | RESTRICT]

- supprime les schémas indiqués
- CASCADE : toutes les tables du schéma
- RESTRICT : seulement les tables vides (par défaut)

DROP TABLE *nom_table* [,…] [CASCADE | RESTRICT]

- RESTRICT : supprime la table seulement si elle n'est référencée par aucune contrainte (clé étrangère) ou vue (par défaut)
- CASCADE : supprime aussi toutes les tables qui « dépendent » de *nom_table*

ALTER TABLE *nom_table* OPERATION

- *modifie* la définition de la table
- opérations:
  - Ajouter (ADD), effacer (DROP), changer (MODIFY) attributs et contraintes
  - changer propriétaire, …

SQL : Requêtes imbriquées

Requête imbriquée dans la clause WHERE d'une requête externe:

```
SELECT …
FROM   …
WHERE  [Opérande] Opérateur (SELECT …
                             FROM …
                             WHERE …)
```

Opérateurs ensemblistes :

- **(A1,…An) IN <sous-req>** : appartenance ensembliste
- **EXISTS <sous-req>** : test d'existence
- **(A1,…An) <comp> [ALL|ANY] <sous-req>** : comparaison avec quantificateur (ANY par défaut)

Expression « IN »

```
SELECT …
FROM …
WHERE (A1,…,An) [NOT] IN (SELECT B1,…,Bn
                           FROM  …
                           WHERE …)
```

Sémantique : la condition est vraie si le n-uplet désigné par $(A_1, \ldots, A_n)$ de la requête *externe* appartient (n'appartient pas) au résultat de la requête *interne*.

ALL/ANY

```
SELECT …
FROM …
WHERE (A1,…,An) θ  ALL/ANY (SELECT B1,…,Bn
                            FROM  …
                            WHERE …)
```

- On peut utiliser une comparaison $\theta \in \{=, <, <=, >, >=, <>\}$ et ALL ($\forall$) ou ANY ($\exists$): La condition est alors vraie si la comparaison est vraie pour tous les n-uplets /au moins un n-uplet de la requête interne.

Comment peut-on exprimer « IN » avec ALL/ANY?

Expression “EXISTS”

Q

```
SELECT …
FROM …
WHERE
```

[NOT] EXISTS

Q’

```
(SELECT *
 FROM …
 WHERE P)
```

- *Sémantique procédurale :*
  - pour chaque n-uplet *x* de la requête externe Q, exécuter la requête interne Q’; s'il existe au moins un n-uplet y dans le résultat de la requête interne, alors sélectionner *x*.
- *Sémantique logique :*
  - *{ x… | Q(x) ∧ [¬]∃y (Q’(y) }*
- Les deux requêtes sont généralement *corrélées* : la condition *P* dans la requête interne Q’ exprime une jointure entre les tables de Q’ et les tables de la requête externe Q.

SQL : Fonctions d'agrégation

Pour calculer une valeur numérique à partir d'une relation on applique des fonctions d'agrégation Agg(A) ou A est un nom d'attribut (ou *) et Agg est une fonction parmi :

- COUNT(A) ou COUNT(*) : nombre de valeurs ou n-uplets,
- SUM(A) : somme des valeurs,
- MAX(A) : valeur maximale,
- MIN(A) : valeur minimale,
- AVG(A) : moyenne des valeurs

dans *l'ensemble des valeurs* désignées par A

```
SELECT  Agg1(Ai), …, Agg(Aj)
FROM    R1, ..., Rm
WHERE   P
```


Requêtes de groupement : GROUP BY

Pour *partitionner* les n-uplets résultats en fonction des valeurs de certains attributs :

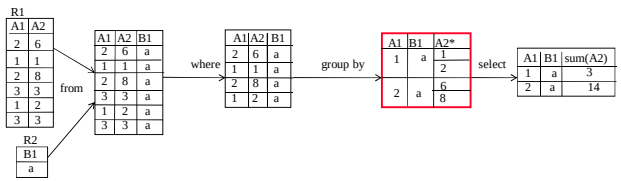
```
SELECT Ai, …, An, aggr1, aggr2, …
FROM   R1, …, Rm
WHERE  P
GROUP BY Aj …, Ak
```

Règle: **tous** les attributs projetés (A_i, …, A_n) dans la clause SELECT

- *n'apparaissent pas dans une opération d'agrégation* et
- *sont inclus* dans l'ensemble des attributs (A_j …, A_k) de la clause GROUP BY (qui peut avoir d’autres attributs en plus)

GROUP BY

```
SELECT  A1, B1, sum(A2)
FROM    R1, R2
WHERE   A1 < 3
GROUP BY A1, B1
```



Predicats sur des groupes

Pour *garder (éliminer) les groupes (partitions)* qui satisfont (ne satisfont) pas une certaine condition :

```
SELECT A1, …, An
FROM   R1, …, Rm
WHERE  P
GROUP BY Aj …, Ak
HAVING Q
```

Règle : La condition *Q* porte sur des valeurs atomiques retournées par un *opérateur d'agrégation* sur les attributs qui *n'apparaissent pas* dans le GROUP BY

Calcul relationnel et SQL

Convertir en SQL :

« Quels employés travaillent dans tous les projets »

La requête SQL correspondante est:

```
SELECT e.Eno
FROM   Emp e
WHERE NOT EXISTS
   (SELECT *
    FROM   Project p
    WHERE NOT EXISTS
      (SELECT *
       FROM   Works w
       WHERE  p.Pno=w.Pno
         AND  e.Eno = w.Eno))
```

```
{ e.Eno |
Emp(e) ∧
    ¬∃p(
        Project(p) ∧
        ¬∃w(
            Works(w) ∧
            p.Pno = w.Pno
            ∧w.Eno=e.Eno))}
```


Couplage SQL–langage de programmation (Embedded SQL)

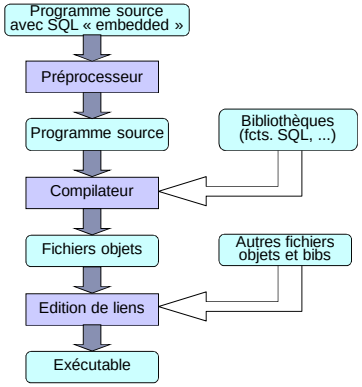
Comment accéder une BD depuis un programme ?

- SQL n'est pas suffisant pour écrire des applications (SQL n'est pas « Turing complet »)

SQL a des liaisons (bindings) pour différents langages de programmation

- C, C++, Java, PHP, etc.
- les liaisons décrivent la façon dont des applications écrites dans ces langages hôtes peuvent interagir avec un SGBD relationnel

Développement d'application



Utilisation de Embedded SQL

Interface = commandes « **EXEC SQL** »

Variables partagées entre SQL et le langage hôte (C, PHP, Java, ..) pour :

- passer des paramètres aux requêtes avant leur évaluation (par exemple nom d'une personne lu par le programme, ...)
- accéder au résultats des requêtes dans le programme (par exemple afficher le résultat)

Exemple de curseur

Pour chaque projet employant plus de 2 programmeurs, donner le numéro de projet et la durée moyenne d'affectation des programmeurs

```
…
EXEC SQL BEGIN DECLARE SECTION;
  char pno[3]; /* project number */
  real avg-dur; /* average duration */
EXEC SQL END DECLARE SECTION;
…
EXEC SQL DECLARE duration CURSOR FOR
   SELECT Pno, AVG(Dur)
   FROM   Works
   WHERE  Resp = 'Programmer'
   GROUP BY Pno
   HAVING COUNT(*) > 2;
…
EXEC SQL OPEN duration;
…
while(1) {
  EXEC SQL FETCH FROM duration INTO :pno, :avg-dur
  if(strcmp(SQLSTATE, "02000") then break
  else < print the info >
}
EXEC SQL CLOSE duration
…
```

Déclaration du curseur

Exécution de la requête

Lecture n-uplets

Fermeture curseur

Mise-à-jour en Embedded SQL

Exemple: transfert d'un montant entre deux budgets de projets

```
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
main() {
     EXEC SQL WHENEVER SQLERROR GOTO error:
     EXEC SQL CONNECT TO Company;
     EXEC SQL BEGIN DECLARE SECTION;
       int pno1, pno2; /* 2 numéros de projet */
       int amount; /* montant du transfert */
     EXEC SQL END DECLARE SECTION;
     /* Code (omis) pour lire pno1, pno2 et amount */
     EXEC SQL UPDATE Project
       SET Budget = Budget + :amount
       WHERE Pno = :pno2;
     EXEC SQL UPDATE Project
       SET Budget = Budget - :amount
       WHERE Pno = :pno1;
     EXEC SQL COMMIT;
     return(0);
 error:
     printf("update failed, sqlcode = %ld\n", SQLCODE);
     EXEC SQL ROLLBACK;
     return(-1);
 }
```