Bases de données relationnelles

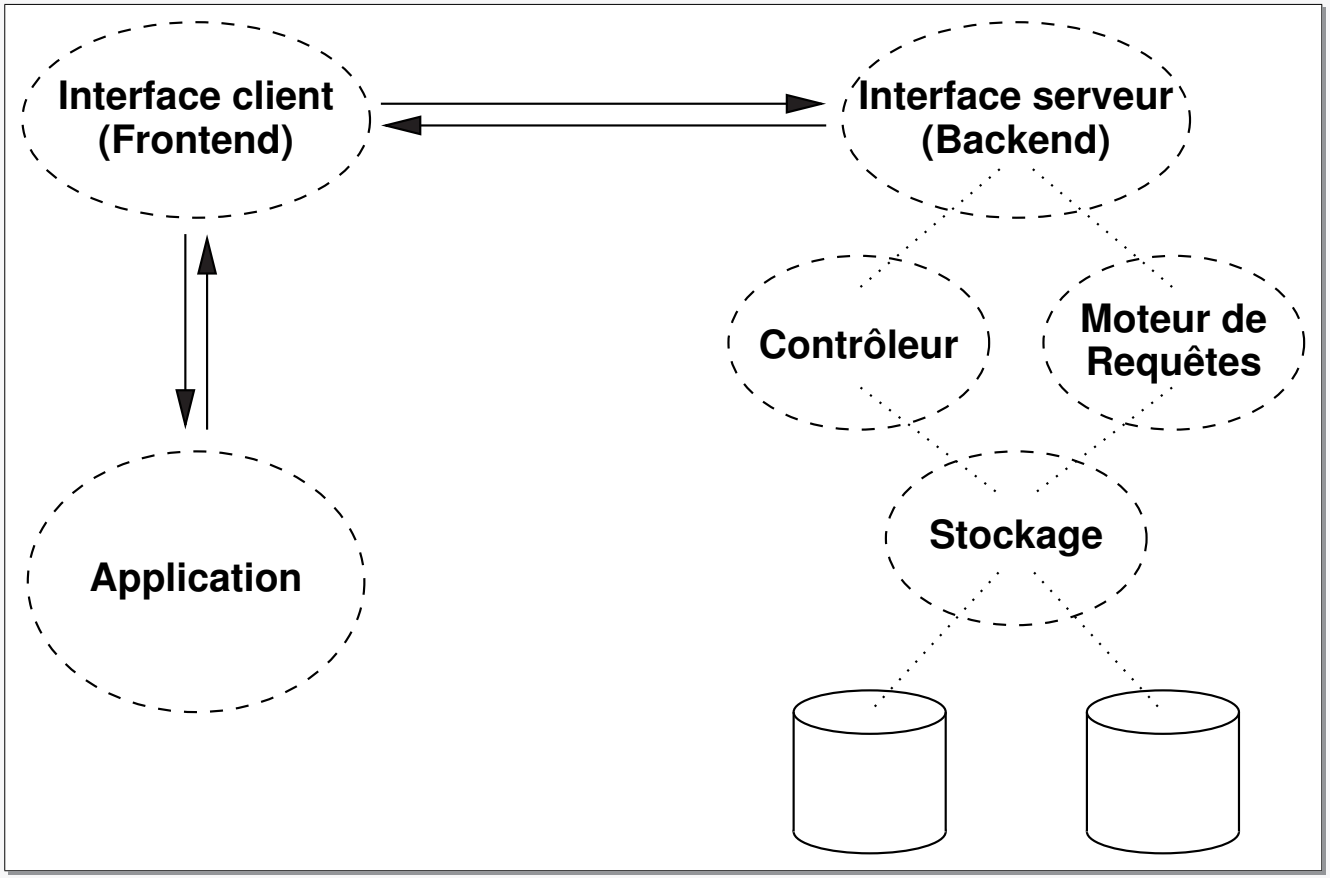
M2-Pro O.S.A.E. 2008-2009 – J.F. Rabasse

- ✘ Architecture
- ✘ Modèles
- ✘ Algèbre relationnelle
- ✘ Le langage SQL
- ✘ Extensions SQL

Architecture

Un SGBD (*Système de Gestion de Bases de Données*) est un environnement logiciel (administration, exploitation) et matériel (supports de stockage) destiné à conserver de l'information et à l'exploiter à partir de requêtes.

Les SGBD sont organisés selon un modèle client-serveur :



Architecture

Le serveur de données proprement dit est constitué de différents sous-systèmes :

- un gestionnaire de stockage assurant les accès aux supports, l'indexation, et diverses fonctions d'optimisation, cache mémoire, etc.
- un moteur de requêtes destiné au traitement des extractions de données et à l'organisation des accès. Le plus souvent, la description des requêtes passe par l'utilisation d'un langage symbolique, e.g. SQL.
- un module de contrôle gérant les accès concurrents, les protections, les utilisateurs, les droits.

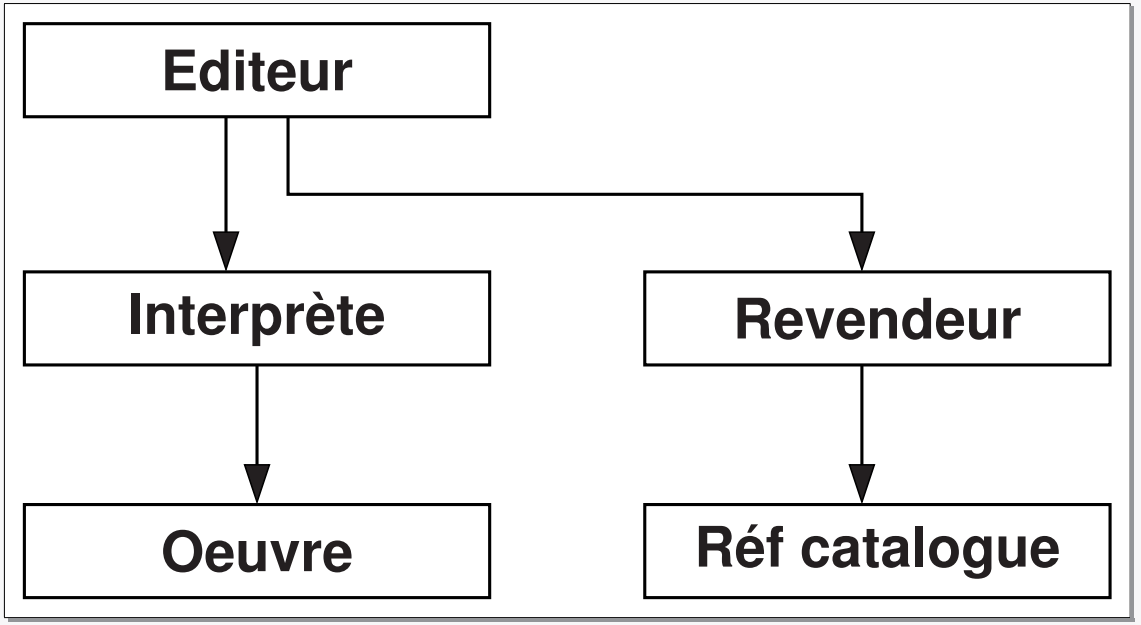
L'interface client comporte :

- Un programme *interface* pour accéder au serveur, suffisant pour de petites applications.
- Une librairie traditionnelle (en C), destinée au développement d'applications plus consistantes.
- Parfois une librairie interfacée avec un langage de script, surtout pour des applications *web*. Typiquement Perl ou PHP.

Différents modèles d'organisation des données dans une base existent (ou ont existé).

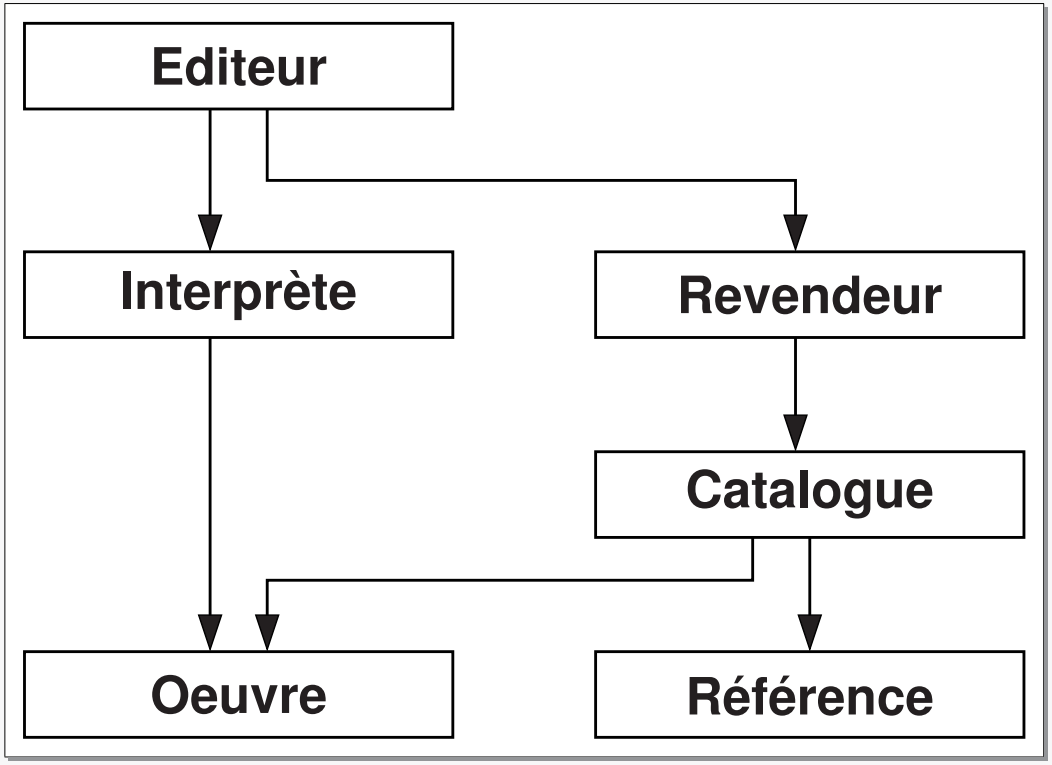
Modèle hiérarchique

Dans un modèle hiérarchique, l'information est stockée par *tables*, organisées selon un système de dépendances de type *parent/enfant*. Par exemple, un éditeur de musique travaille contractuellement avec des interprètes et dispose d'un réseau de distributeurs de ses titres.



Modèle réseau

Le modèle réseau est une amélioration du modèle hiérarchique et permet des interconnexions *horizontales* et des partages de tables d'information.



Modèle relationnel

L'inconvénient majeur des modèles précédents est que l'information de dépendance (tel interprète joue telle oeuvre, telle oeuvre a telle référence au catalogue, etc.) est partie intégrante du système d'information (index et pointeurs).

De fait, la conception initiale de la base doit prendre en compte l'exploitation future et il devient très difficile après coup de modifier les systèmes de requêtes.

Enfin, l'information étant structurée autour d'une application, elle devient faiblement utilisable dans un autre contexte (e.g. une médiathèque de prêt de disques).

Dans le modèle relationnel (ou modèle de CODD) l'information *brute* est stockée en tables indépendantes : table des interprètes, table des oeuvres, etc.

La mise en relation de l'information s'effectue en créant de nouvelles tables dédiées. Par exemple, la relation *"X interprète l'oeuvre Y"* peut se matérialiser par une table de doublets *référence interprète* + *référence oeuvre*.

Avantages

La construction des tables *relations* peut se faire indépendamment des tables *données*, y compris après coup. On peut donc construire un système d'informations sans préjuger de l'exploitation future.

Inconvénients

La mise en oeuvre de requêtes complexes nécessite des traitements croisés entre plusieurs tables. Le traitement général est lourd et fait appel à des algorithmes spécialisés (algèbre relationnel, ou algèbre de CODD).

Aujourd'hui, la puissance des machines modernes permet de supporter la lourdeur de traitement des SGBDR; l'avantage - énorme - de pouvoir dissocier la construction de l'information de son exploitation ultérieure fait que le modèle relationnel s'est généralisé.

Terminologie

- ✘ **L'algèbre relationnel a formalisé un certain nombre de termes, *relation*, *attribut*, *tuple*, qui ont un équivalent dans la terminologie usuelle, *table*, *champ*, *enregistrement*.**

  Il existe différents types de relations entre tables :

  - **Relation 1:1. Connexion biunivoque, par exemple *toute oeuvre, au catalogue, a un et un seul numéro de référence*.**
  - **Relation 1:N. Il peut exister plusieurs *tuples* ayant un même *attribut*. Par exemple *un même compositeur peut avoir composé plusieurs oeuvres*.**
  - **Relation N:M. Il peut exister plusieurs *tuples* ayant des *attributs* communs. Par exemple *un même interprète peut jouer plusieurs oeuvres, et une oeuvre donnée peut avoir plusieurs interprètes*.**

- ➤ **Lors de la conception d'une base de données, il est important d'identifier ces ordres de relations. En particulier, une relation 1:1 peut conduire à une fusion de tables, une relation 1:N à un attribut de référence, etc.**

Algèbre de CODD

Références

- Edgar F. Codd: *A Relational Model of Data for Large Shared Data Banks*, CACM 13(6): 377-387 (1970), reprint CACM 26(1): 64-69 (1983).
- Edgar F. Codd: *The Relational Model for Database Management*, version 2, Addison-Wesley 1990.

Principes

- Données regroupées par ensembles d'éléments atomiques ou composites (*t-uples*).

$$\mathcal{P} : \mathcal{E}(x_i), i = 1, N$$
$$\mathcal{Q} : \mathcal{E}(y_j), j = 1, M$$

- Relations entre données implantées par *traces* (ensembles d'éléments composites, au minimum des *2-uples*).

$$\mathcal{R} \mapsto \mathcal{E}(x_i, y_j), x_i \in \mathcal{P}, y_j \in \mathcal{Q}$$
$$\vdash \mathcal{R}(x_i, y_j)$$

Ordres des relations

- **Relations 1:1 (fusion d'éléments possible en t-uples)**

$$\mathcal{R}(x_i, y_j) \Rightarrow \nexists x_k, k \neq i \vdash \mathcal{R}(x_k, y_j)$$
$$\nexists y_k, k \neq j \vdash \mathcal{R}(x_i, y_k)$$

- **Relations N:1 (référencement possible entre ensembles)**

$$\mathcal{R}(x_i, y_j) \Rightarrow \exists x_k, k \neq i \vdash \mathcal{R}(x_k, y_j)$$
$$\nexists y_k, k \neq j \vdash \mathcal{R}(x_i, y_k)$$

- **Relations N:M (cas général, trace séparée)**

$$\mathcal{R}(x_i, y_j) \Rightarrow \exists x_k, k \neq i \vdash \mathcal{R}(x_k, y_j)$$
$$\exists y_k, k \neq j \vdash \mathcal{R}(x_i, y_k)$$

Épilogue

Avec les SGBD relationnels on dispose donc d'outils logiciels puissants, capables de manipuler des ensembles (au sens algébrique) de données.

Par rapport aux SGBD classiques (avant 1980) on a :

- **Dissociation du schéma de données et des relations entre données. On saura donc réintroduire après-coup de la sémantique dans une base en exploitation sans devoir tout reconstruire.**
- **Homogénéité des implémentations internes des données et des relations. Les primitives opératoires ensemblistes sont les mêmes.**
- **Réversibilité complète des relations. On saura décrire les relations entre données sans préjuger du type d'utilisation ultérieure.**

Donc, au final, un environnement très souple dans lequel la description du monde des données est, à peu près, indépendante des utilisations possibles.

Le prix à payer est un traitement lourd qui nécessite des machines puissantes (depuis 1990) et des algorithmes internes sophistiqués (*requests planners* exploitant des analyses statistiques).

Principaux logiciels

DB2	**Logiciel commercial (IBM). Puissant et très complet (IBM quoi), surtout destiné à des mainframes.**
Oracle	**Logiciel commercial (Oracle Corp.). Le *best seller*, disponible sur mainframes, mini, micro-ordinateurs.**
Postgres	**Logiciel libre (Berkeley). Puissant et complet, disponible dans tout le monde Unix, interface d'administration rustique (à la Unix).**
MySQL	**Logiciel libre[1]. Très répandu dans le petit monde web, simple à utiliser pour de petites applications.**
SQL-Server	**Logiciel commercial (Microsoft Corp.). Rien à en dire si ce n'est que c'est du logiciel Microsoft, disponible uniquement sur plateformes MS-Windows.**

(1) En passe de devenir payant.

Un “Server Query Language”

Comme son nom l’indique, SQL a été initialement créé pour servir de langage de requête. En fait, il a été étendu pour assurer toutes les opérations de maintenance et d’exploitation d’une base de données, requêtes mais aussi créations, mises à jour, etc.

SQL fait l’objet de deux standards, SQL92 puis SQL99.

Construction de tables

On veut construire une base de données pour gérer un module d’enseignement, par exemple (au hasard !) un Master.

Tout naturellement, la première table est une description des étudiant(e)s : nom, âge, université d’origine, adresse e-mail, etc. Pour une base de données, on essaie de choisir des informations stables; ainsi, une année de naissance est plus stable qu’un âge.

Création d'une table

```
CREATE TABLE etudiants (
   nom        text,
   dnaiss     int4,
   origine    text,
   email      text);
```

Population d'une table

```
INSERT INTO etudiants
    VALUES('Alice', 1982, 'P6', 'alice@obspm.fr');
INSERT INTO etudiants
    VALUES('Toto', 1983, 'P11', 'toto@obspm.fr');
```

NB : implicitement, on donne pour chaque *tuple* une liste de valeurs dans l'ordre des attributs. Mais on peut modifier cet ordre à condition de le spécifier :

```
INSERT INTO etudiants (nom, email, origine, dnaiss)
    VALUES('Alice', 'alice@obspm.fr', 'P6', 1982);
```

Sélection

- L'opération de base de l'algèbre relationnel est la sélection, i.e. une extraction de *tuples* avec ou sans critères.

Extraction de toute une table

```
SELECT * FROM etudiants;
  nom   | dnaiss | origine |      email
--------+--------+---------+-----------------
 Alice  |   1982 | P6      | alice@obspm.fr
 Toto   |   1983 | P11     | toto@obspm.fr
(2 rows)
```

Recherche des étudiant(e)s venant de Paris VI

```
SELECT * FROM etudiants
    WHERE origine = 'P6';
  nom   | dnaiss | origine |      email
--------+--------+---------+-----------------
 Alice  |   1982 | P6      | alice@obspm.fr
(1 row)
```

Étudiant(e)s de Paris VI ET de plus de 30 ans

```
SELECT * FROM etudiants
    WHERE origine = 'P6' AND dnaiss < (2008 - 30);
 nom | dnaiss | origine | email
-----+--------+---------+-------
(0 rows)
```

etc.

Projection

La projection est une vue partielle d'une table :

```
SELECT nom, email FROM etudiants;
  nom   |      email
--------+-----------------
 Alice  | alice@obspm.fr
 Toto   | toto@obspm.fr
(2 rows)
```

Dans une projection, on peut redéfinir les noms des attributs :

```
SELECT nom AS "Nom", email AS "Adresse Mail"
    FROM etudiants;
  Nom   |  Adresse Mail
--------+-----------------
 Alice  | alice@obspm.fr
 Toto   | toto@obspm.fr
(2 rows)
```

Présentation

Enfin, on peut également spécifier un ordre de présentation des résultats de requêtes :

```
SELECT nom AS "Nom", email AS "Adresse Mail"
    FROM etudiants ORDER BY dnaiss DESC;
  Nom   |  Adresse Mail
-------+----------------
 Toto   | toto@obspm.fr
 Alice  | alice@obspm.fr
(2 rows)
```

Identifiants et clés

Les attributs texte, dans les systèmes d'informations, présentent un certain nombre de défauts.

Ajoutons à la base précédente une table de modules de cours :

```
CREATE TABLE cours (
   intitule     text,
   intervenant  text);
```

et peuplons la :

```
INSERT INTO cours
    VALUES('Traitement du Signal', 'Abergel');
INSERT INTO cours
    VALUES('Bases de Donnees', 'Rabasse');
INSERT INTO cours
    VALUES('Programmation', 'Rabasse');
```

etc.

Identifiants et clés

Construisons une relation *"étudiant(e) X suit le cours Y"*. Dans le modèle relationnel, une telle connexion n'est autre qu'une nouvelle table, la *trace* de la relation :

```
CREATE TABLE cours_suivis (
   etudiant  text,
   intitule  text);
```

que l'on peuplera avec des couples de données :

```
INSERT INTO cours_suivis
    VALUES('Alice', 'Traitement du Signal');
INSERT INTO cours_suivis
    VALUES('Alice', 'Programmation');
INSERT INTO cours_suivis
    VALUES('Alice', 'Base de Donnees');
INSERT INTO cours_suivis
    VALUES('Toto', 'Traitement du Signal');
INSERT INTO cours_suivis
    VALUES('Toto', 'Programmation');
```

(Il semble que Toto ait "séché" le cours de bases de données !)

Identifiants et clés

Une telle approche conduit à une duplication importante d'informations. De plus, de l'information texte est rarement non ambiguë, ou peut se prêter à de multiples interprétations. (Par exemple, l'attribut *origine* de la table *etudiants* pourrait être saisi de multiples façons : P6, Paris VI, Pierre et Marie Curie, etc.)

Il est d'usage d'utiliser des identifiants numériques, plus compacts, non ambigus. De plus, les gestionnaires de bases de données permettent des contrôles de non collision. Ainsi, en attribuant un code numérique aux étudiants on peut assurer l'unicité :

```
CREATE TABLE etudiants (
   code       int4 UNIQUE,
   nom        text,
   dnaiss     int4,
   origine    text,
   email      text);

INSERT INTO etudiants
    VALUES(1, 'Alice', 1982, 'P6', 'alice@obspm.fr');
INSERT INTO etudiants
    VALUES(2, 'Toto', 1983, 'P11', 'toto@obspm.fr');
```

Identifiants et clés

Si, par erreur, on effectue une inscription avec un code déjà utilisé, l'insertion échouera. Un identifiant unique est appelé une *clé primaire*.

Même chose pour les cours :

```
CREATE TABLE cours (
   code          int4 UNIQUE,
   intitule      text,
   intervenant   text);
```

```
INSERT INTO cours
    VALUES(1, 'Traitement du Signal', 'Abergel');
INSERT INTO cours
    VALUES(2, 'Bases de Donnees', 'Rabasse');
```

etc.

La relation *cours suivis* devient purement numérique :

```
CREATE TABLE cours_suivis (
   code_etudiant   int4,
   code_cours      int4);
```

Produit cartésien

En algèbre relationnel, le *produit cartésien* de deux ou plusieurs tables est la fusion de toutes les combinaisons possibles de *tuples*.

Par exemple :

```
SELECT * from cours, etudiants;
 code |        intitule        | intervenant | code |  nom ..
------+------------------------+-------------+------+-----
    1 | Traitement du Signal | Abergel     |    1 | Alice |
    2 | Bases de Donnees     | Rabasse     |    1 | Alice |
    3 | Programmation        | Rabasse     |    1 | Alice |
    1 | Traitement du Signal | Abergel     |    2 | Toto  |
    2 | Bases de Donnees     | Rabasse     |    2 | Toto  |
    3 | Programmation        | Rabasse     |    2 | Toto  |
(6 rows)
```

Produit cartésien

On peut projeter un produit cartésien :

```
SELECT intitule, email FROM cours, etudiants;
       intitule        |     email
-----------------------+---------------
 Traitement du Signal  | alice@obspm.fr
 Bases de Donnees      | alice@obspm.fr
 Programmation         | alice@obspm.fr
 Traitement du Signal  | toto@obspm.fr
 Bases de Donnees      | toto@obspm.fr
 Programmation         | toto@obspm.fr
(6 rows)
```

Produit cartésien

Lorsqu'une ambiguité de nom existe, par exemple le *code* cours ou le *code* étudiant, on précise les tables :

```
SELECT cours.code, etudiants.code, email
   FROM cours, etudiants;
 code | code |      email
------+------+-----------------
    1 |    1 | alice@obspm.fr
    2 |    1 | alice@obspm.fr
    3 |    1 | alice@obspm.fr
    1 |    2 | toto@obspm.fr
    2 |    2 | toto@obspm.fr
    3 |    2 | toto@obspm.fr
(6 rows)
```

On peut également renommer les tables :

```
SELECT C.code, E.code, email
   FROM cours AS C, etudiants AS E;
```

Jointure

Toutes les combinaisons possibles de cours et d'étudiant(e)s de l'exemple précédent n'ont pas grand sens et n'ont surtout aucun intérêt.

La relation `cours_suivis`, elle, contient de l'information intéressante mais faiblement exploitable sous sa forme numérique :

```
SELECT * FROM cours_suivis;
 code_etudiant | code_cours
---------------+------------
             1 |          1
             1 |          2
             1 |          3
             2 |          1
             2 |          3
(5 rows)
```

Une jointure est un produit cartésien dont on ne conserve que les combinaisons ayant certains attributs communs.

Jointure

Par exemple :

```
SELECT E.nom, C.intitule
   FROM cours_suivis, etudiants AS E, cours AS C
   WHERE code_etudiant = E.code AND code_cours = C.code;
  nom   |        intitule
-------+-----------------------
 Alice | Traitement du Signal
 Toto  | Traitement du Signal
 Alice | Bases de Donnees
 Alice | Programmation
 Toto  | Programmation
(5 rows)
```

On a réalisé une jointure des trois tables, avec les codes étudiant et cours comme *attributs de jointure*, suivie d'une projection.

Jointure

La jointure peut se combiner avec des attributs de sélection. Par exemple, liste des cours auxquels est inscrite Alice :

```
SELECT intitule AS "Inscriptions Alice"
   FROM cours_suivis, etudiants AS E, cours AS C
   WHERE code_etudiant = E.code AND code_cours = C.code
         AND nom = 'Alice';
  Inscriptions Alice
----------------------
 Traitement du Signal
 Bases de Donnees
 Programmation
(3 rows)
```

Opérateurs ensemblistes

(Exemples extraits d'une base documentaire.)

```
SELECT theme, label FROM booktheme_table;
  1 | Astrophysique generale
  2 | Galaxies
  3 | Physique
  4 | Rayonnement / Spectroscopie
  5 | Observation / Instrumentation
  6 | Analyse numerique
  7 | Programmation
 ...
 19 | Statistiques / Probabilites
(18 rows)

SELECT theme, bcode FROM themeindex_table;
  6 |     3
  6 |    23
  6 |    24
  1 |    11
  2 |    11
```

Opérateurs ensemblistes

Recherche des titres indexés par *Analyse numérique*.

```
SELECT title FROM themeindex_table T, book_table B
WHERE T.bcode = B.bcode
AND T.theme = 6;
 Analyse mathematique et calcul numerique (vol. 4)
 Analyse numerique et calcul numerique
 Analyse numerique et calcul numerique (vol. 1)
 Applied Numerical Analysis, 5th edition
 Astrophysical processes in upper main sequence stars
 Calcul Infinitesimal
 Computational techniques for Fluid Dynamics (vol. 2)
 Computer Simulation using particles
 ...
 Numerical Recipes : Example Book (FORTRAN)
 Numerical Recipes in FORTRAN, 2nd edition
 Numerical analysis of spectral methods
```

Opérateurs ensemblistes

Recherche des titres indexés par *Analyse numérique* ET *Programmation*.

```
SELECT title FROM themeindex_table T, book_table B
WHERE T.bcode = B.bcode
AND T.theme = 6
INTERSECT
SELECT title FROM themeindex_table T, book_table B
WHERE T.bcode = B.bcode
AND T.theme = 7;
 Computational techniques for Fluid Dynamics (vol. 2)
 Computer Simulation using particles
 ...
 Numerical Recipes : Example Book (FORTRAN)
 Numerical Recipes in FORTRAN, 2nd edition
```

- SQL implémente :

INTERSECT	$\mathcal{S}_1 \cap \mathcal{S}_2$
UNION	$\mathcal{S}_1 \cup \mathcal{S}_2$
EXCEPT	$\mathcal{S}_1 \cap \neg\mathcal{S}_2$

Extensions SQL

- Certaines implémentations logicielles proposent des extensions au standard, sous forme de type et opérateurs spéciaux.

  Leur utilisation rend l'application non portable vers d'autres SGBD.

  Exemple tiré d'une base de radiosources.

```
CREATE TABLE field_table (
    ion     text,
    field   box,
    ...
CREATE TABLE psc_table (
    g_lon   float4,
    g_lat   float4,
    flux1   float4,
    ...
SELECT flux1 FROM psc_table P, field_table F
WHERE F.ion = '13600107'
AND point(P.g_lon, P.g_lat) @ F.field;
```

Fonctions backend

- Certaines implémentations logicielles permettent de définir, au niveau du serveur, des fonctions utilisables dans des requêtes.

Fonctions en langage SQL

(Exemple tiré d'une base documentaire.)

```
CREATE FUNCTION numaut(int4)
    RETURNS int4
    AS 'SELECT sum(1) FROM bookwriter_table WHERE bcode = $1'
    LANGUAGE 'SQL';
SELECT numaut(bcode), title FROM book_table;
 2 | Mathematical techniques, 2nd edition
 1 | Statistique appliquée à l'exploitation des mesures, T1
 3 | Methods in computational physics : Quantum Mechanics
 1 | The theory of homogeneous turbulence
 2 | Self-Organization in Nonequilibrium Systems
 4 | Numerical Recipes in FORTRAN, 2nd edition
 2 | Handbook of Mathematical Functions
 ...
```

Fonctions en langage natif C (logiciel Postgres)

Certains applicatifs (surtout dans les domaines scientifiques) peuvent nécessiter des écritures de requêtes comportant des calculs plus ou moins complexes.

Par exemple une base de données comporte un catalogue de sources données avec leur position équatorial :

```
CREATE TABLE sources_table (
    sname   text,     -- Nom de la source
    ra2000  float8,   -- Ascension droite
    de2000  float8,   -- Declinaison
    ...
```

Une interface utilisateur (e.g. Web) permet de saisir une position équatoriale, `A0`, `D0` et une distance angulaire `R` et l'on veut toutes les sources se trouvant au maximum à `R` de la position donnée.

Le calcul de distance angulaire, en trigonométrie sphérique, ne peut se faire avec SQL alors qu'une telle fonction s'écrit facilement en C, en utilisant l'interface de programmation PostgreSQL.

Fonctions backend

```
float8* arcdist(float8* pAs, float8* pDs,
    float8* pAc, float8* pDc)
{
    float8 A, a, b, cosa;
    float8* result = (float8*)palloc(sizeof(float8));
    A = *pAs - *pAc;
    a = *pDs;
    b = *pDc;
    cosa = sin(a) * sin(b) + cos(a) * cos(b) * cos(A);
    if( cosa < 0.99 ) *result = acos(cosa);
    else {
        float8 D = a - b;
        float8 cosd = cos((a + b) / 2);
        *result = sqrt(A * A * cosd * cosd + D * D);
    }
    return result;
}
```

Cette fonction, compilée sous forme d'une librairie partageable sera ensuite "greffée" dans le serveur PostgreSQL :

```
CREATE FUNCTION arcdist(float8, float8, float8, float8)
```

Fonctions backend

```
    RETURNS float8 AS '/var/libpq/arcdist.so' LANGUAGE 'C';
```

Une application disposant des paramètres de requête `A0`, `D0`, `R`, pourra l'utiliser :

```
SELECT sname, ra2000, de2000, ... FROM sources_table
    WHERE arcdist(ra2000, de2000, A0, D0) < R;
```