Chapitre IV

Les bases de données relationnelles en pratique : Langages d'interrogation

Relation : ensemble ou multi-ensemble ?

Un *multi-ensemble* (*multiset*) est une collection d'éléments pour laquelle on tient compte de la multiplicité. Le multi-ensemble $\{1, 1, 2, 3\}$ est différent du multi-ensemble $\{1, 2, 3\}$.

Nous avons défini une relation comme étant un ensemble. Toutefois, dans la pratique, les systèmes de bases de données considèrent les relations comme des multi-ensembles.

Cette pratique est motivée par deux considérations :

- travailler avec des multi-ensembles accélère certaines opérations (projection, union) car il n'est pas nécessaire de rechercher les occurrences multiples de tuples ;
- pour certaines opérations (par exemple le calcul d'une moyenne), il est nécessaire de considérer des multi-ensembles plutôt que des ensembles.

L'algèbre relationnelle sur les multi-ensembles

Considérons deux relations r et s de schéma R et S et un tuple t qui apparaît m fois dans r et n fois dans s.

- Dans $r \cup s$ (supposant que $R = S$), le tuple t apparaît $m + n$ fois.
- Dans $r \cap s$ (supposant que $R = S$), le tuple t apparaît $\min(m, n)$ fois.
- Dans $r \setminus s$ (supposant que $R = S$), le tuple t apparaît $\max(0, m - n)$ fois.

Les autres opérations de l'algèbre relationnelle s'étendent aussi naturellement aux multi-ensembles.

Projection. La projection de chaque tuple est conservée, même si certains tuples deviennent identiques suite à la projection.

Sélection. Les tuples satisfaisant la condition de sélection sont conservés, y compris les occurrences multiples.

Produit cartésien. Toutes les combinaisons de tuples sont considérées avec leur multiplicité. Si un tuple t_1 apparaît m fois and une relation r et un tuple t_2 n fois dans une relation s, le tuple $t_1 t_2$ apparaît mn fois dans $r \times s$.

Joint. Même règle que pour le produit cartésien.

L'algèbre relationnelle sur les multi-ensembles : exemples

r_1 :

A	B	C
a	b	c
d	a	f
d	a	f

r_2 :

A	B	C
b	g	a
d	a	f

r_3 :

C	D	E
c	d	e
c	d	e

$r_1 \cup r_2$:

A	B	C
a	b	c
d	a	f
d	a	f
d	a	f
b	g	a

$r_1 - r_2$:

A	B	C
a	b	c
d	a	f

$r_1 \bowtie r_3$:

A	B	C	D	E
a	b	c	d	e
a	b	c	d	e

$r_1 \cap r_2$:

A	B	C
d	a	f

$\pi_{AB} r_1$:

A	B
a	b
d	a
d	a

$r_1 \times r_3$:

A	B	C	C'	D	E
a	b	c	c	d	e
a	b	c	c	d	e
d	a	f	c	d	e
d	a	f	c	d	e
d	a	f	c	d	e
d	a	f	c	d	e

Valeurs nulles

Jusqu'à présent, nous avons exigé que chaque tuple donne une valeur à chaque attribut. Toutefois, il est souvent utile de pouvoir indiquer qu'un attribut n'a pas de valeur pour l'une des raisons suivantes :

- Cette valeur n'est pas connue (ou est confidentielle) au moment où le tuple est inséré ;
- La valeur n'existe pas (par exemple le numéro de boîte dans une adresse).

Nous représenterons la valeur nulle par le symbole $\perp$ ou le mot clé `NULL`.

Il est en général possible d'éviter les valeurs nulles en décomposant les relations au maximum, mais cette approche à un impact négatif sur les performances et la lisibilité du schéma.

Une extension de l'algèbre relationnelle

Pour que l'algèbre relationnelle corresponde aux langages utilisés, il faut lui ajouter des opérateurs qui permettent de

- Convertir un multi-ensemble en ensemble ;
- Calculer des valeurs agrégées (moyenne, . . .) ;
- Trier et grouper les tuples d'une relation ;
- Réaliser des projections généralisées permettant notamment des opérations arithmétiques sur les valeurs des attributs des tuples ;
- Calculer un joint sans perdre les tuples pour lesquels il n'y a pas de tuples en concordance sur les attributs communs.

L'élimination des tuples redondants

Pour passer d'une relation traitée comme multi-ensemble à la relation correspondante sans duplication de tuples, nous introduisons un opérateur

$$\Delta(r)$$

qui élimine les occurrences multiples de tuples de la relation r.

Exemple :

r :

A	B	C
a	b	c
d	a	f
d	a	f

$\Delta(r)$:

A	B	C
a	b	c
d	a	f

Les opérateurs calculant des valeurs agrégées

Il est très fréquent de calculer la somme ou la moyenne des valeurs d'un attribut dans une relation. Pour cela on considère des opérateurs qui calculent ces fonctions (ce ne sont pas des opérateurs de l'algèbre relationnelle) :

- `SUM`, la somme des valeurs d'un attribut ;
- `AVG`, la moyenne des valeurs d'un attribut ;
- `MIN, MAX`, la valeur minimale (maximale) figurant pour un attribut dans la relation.
- `COUNT`, compte le nombre d'occurrences d'un attribut (en tenant compte des occurrences multiples) ; en fait, `COUNT` donne le nombre de tuples de la relation.

Exemple :

r :

A	B	C
1	b	c
3	a	f
3	a	f
2	b	f

$\texttt{SUM}(A) = 9$
$\texttt{AVG}(A) = 2,25$
$\texttt{MIN}(A) = 1$
$\texttt{COUNT}(A) = 4$

L'opérateur de groupement

Il est souvent nécessaire de calculer un fonction d'agrégation, non pas sur l'entièreté de la relation, mais sur des groupes de tuples de la relation.

Exemple : Considérons une relation de schéma *COURS(ID_ETUDIANT, ID_COURS, COTE)*. On peut souhaiter calculer la moyenne des cotes par cours.

On introduit l'opérateur $\gamma_L(r)$ où L est une liste d'éléments qui sont :

- Un attribut A du schéma R de r par rapport auquel le groupement est réalisé (les tuples ayant la même valeur pour l'attribut A sont placés dans le même groupe) ;
- Une opération d'agrégation $\texttt{OP}(A) \rightarrow B$ qui indique que dans chaque groupe, l'opération `OP` est appliquée à l'attribut A et le résultat affecté à l'attribut B.

Exemple : $\gamma_{ID\_COURS,\texttt{AVG}(COTE)\rightarrow M\_COURS}(r)$

Précisément, le calcul de $\gamma_L(r)$ se fait comme suit.

1. On assemble les tuples de r en groupes ayant les mêmes valeurs pour les attributs de groupement de L.
2. Pour chaque groupe, on produit un tuple comportant les valeurs des attributs de groupement pour le groupe, et une valeur pour chaque attribut correspondant à une fonction d'agrégation définie dans L.

Exemple :

r :

ID_ET	ID_COURS	COTE
1	*INFO0009*	18
3	*INFO0009*	8
3	*INFO0016*	7
2	*INFO0016*	17

$\gamma_{ID\_COURS,\texttt{AVG}(COTE)\rightarrow M\_COURS}(r)$:

ID_COURS	M_COURS
INFO0009	13
INFO0016	12

L'opérateur de tri

Par définition, les relations ne sont pas des ensembles (ou des multi-ensembles) triés. Toutefois, pour présenter le résultat d'une requête, un tri est souvent souhaitable. L'opérateur de tri est

$$\tau_L(r),$$

où L est la liste (ordonnée) des attributs par rapport auxquels on trie.

Exemple :

r :

ID_ET	ID_COURS	COTE
1	*INFO0009*	18
3	*INFO0009*	8
3	*INFO0016*	7
2	*INFO0016*	17

$\tau_{ID\_ET,COTE}(r)$:

ID_ET	ID_COURS	COTE
1	*INFO0009*	18
2	*INFO0016*	17
3	*INFO0016*	7
3	*INFO0009*	8

L'opérateur de projection étendu

Lors d'une projection , on peut souhaiter renommer un attribut, ou encore introduire un attribut dont la valeur est calculée à partir de plusieurs attributs.

Exemple : Considérons une relation de schéma

$COURS(ID\_ETUDIANT, ID\_COURS, COTE\_TP, COTE\_EX)$.

On souhaiterait produire une relation de schéma

$COURS(ID\_ETUDIANT, ID\_COURS, COTE)$

ou $COTE$ est calculé comme étant $COTE\_TP + COTE\_EX$.

L'opérateur de projection π_L est étendu en permettant aux éléments de L d'être :

1. un attribut ;
2. une contrainte de la forme $A \to B$, indiquant que l'attribut A est renommé B ;
3. une contrainte de la forme `<expression>` $\to A$, où `<expression>` est une expression arithmétique dont les éléments atomiques sont des attributs ; une telle contrainte indique que la valeur de l'expression `<expression>` est prise comme valeur pour l'attribut A dans les tuples de la relation, projetée.

Exemple : $\pi_{ID\_ET,ID\_COURS,COTE\_TP+(COTE\_EX)\to COTE}(r)$

Exemple

r

ID_ET	ID_COURS	COTE_TP	COTE_EX
1	INFO0009	9	9
3	INFO0009	5	3
3	INFO0016	6	1
2	INFO0016	10	7

$$\pi_{ID\_ET,ID\_COURS,(COTE\_TP+COTE\_EX)\rightarrow COTE}(r)$$

ID_ET	ID_COURS	COTE
1	INFO0009	18
3	INFO0009	8
3	INFO0016	7
2	INFO0016	17

L'opérateur de joint externe (outerjoin) $\overset{\circ}{\bowtie}$

Dans un joint $r \bowtie s$, les tuples de r (de s) qui ne peuvent être combinés avec aucun tuple de s (de r) sont dits *sans appui* (*dangling*). Ils n'apparaissent pas dans le résultat du joint.

On a définit une opérations qui incorpore ces tuples dans le joint en les complétant avec des valeurs nulles, il s'agit du *joint externe* (*outerjoin*).

Exemple :

r_1 :

A	B	C
a	b	c
d	a	f
d	a	f

r_2 :

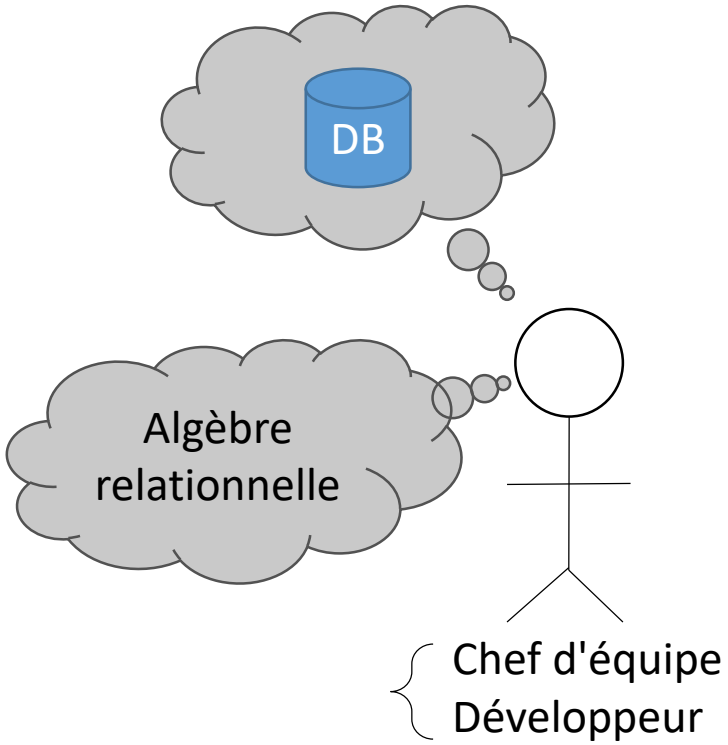
C	D	E
c	d	e
c	d	e

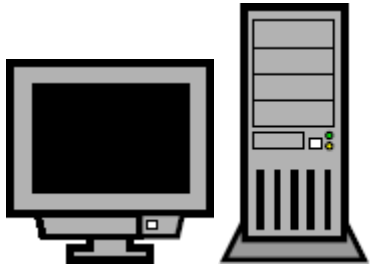
$r_1 \overset{\circ}{\bowtie} r_2$:

A	B	C	D	E
a	b	c	d	e
a	b	c	d	e
d	a	f	$\perp$	$\perp$
d	a	f	$\perp$	$\perp$

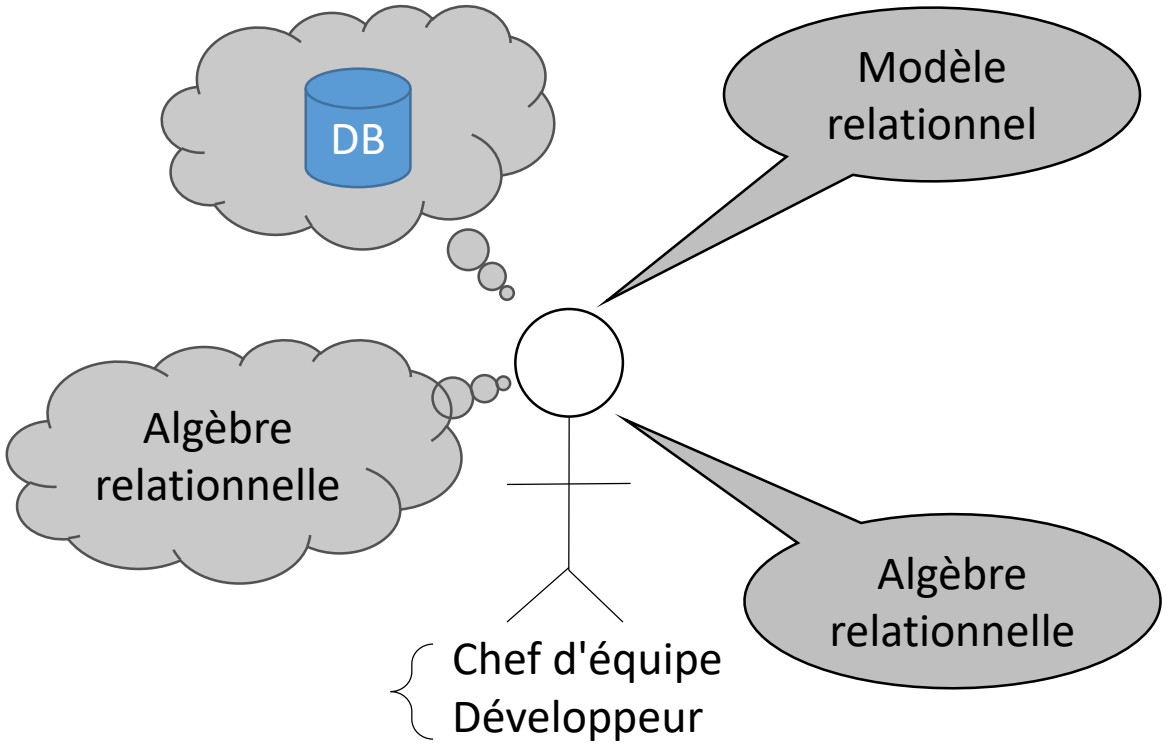
Des variantes du joint externe ne complètent que les tuples de la relation de gauche ($\overset{\circ}{\bowtie}_L$) ou de celle droite ($\overset{\circ}{\bowtie}_R$). Le θ-joint peut aussi être étendu de façon similaire.

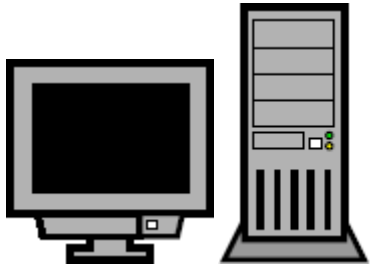
Le langage SQL : Pourquoi?



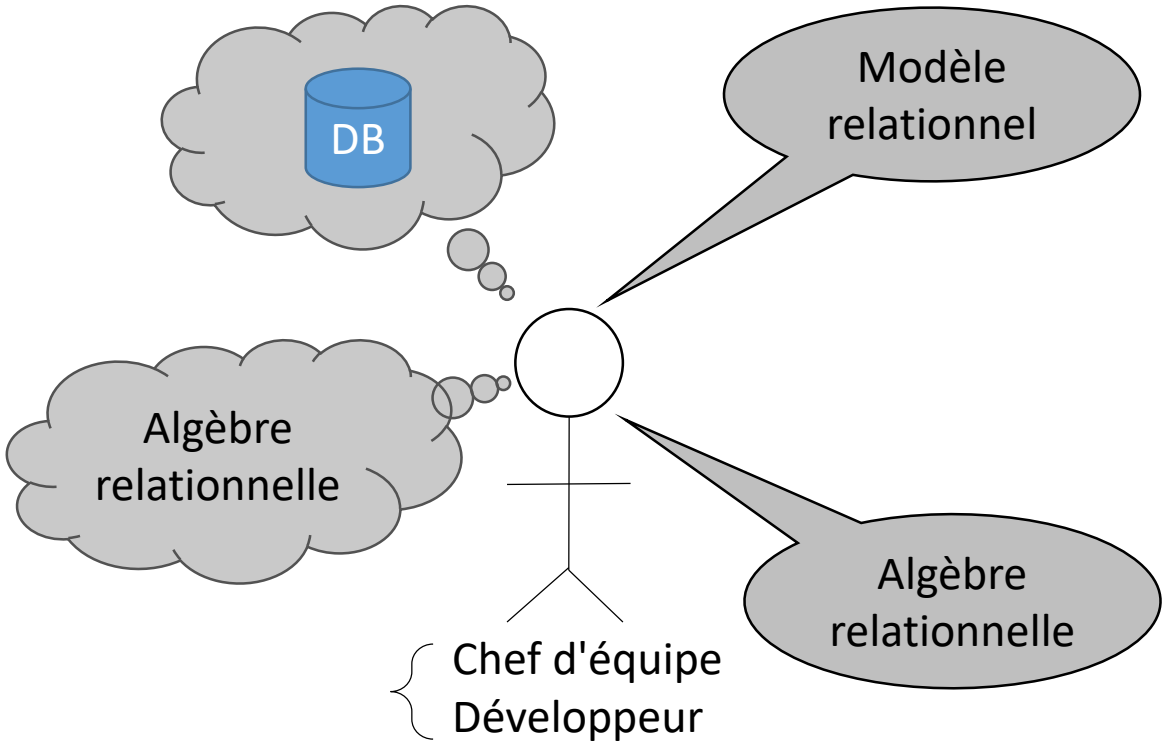


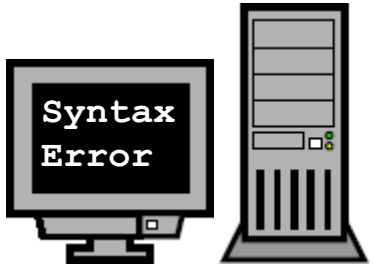
Le langage SQL : Pourquoi?



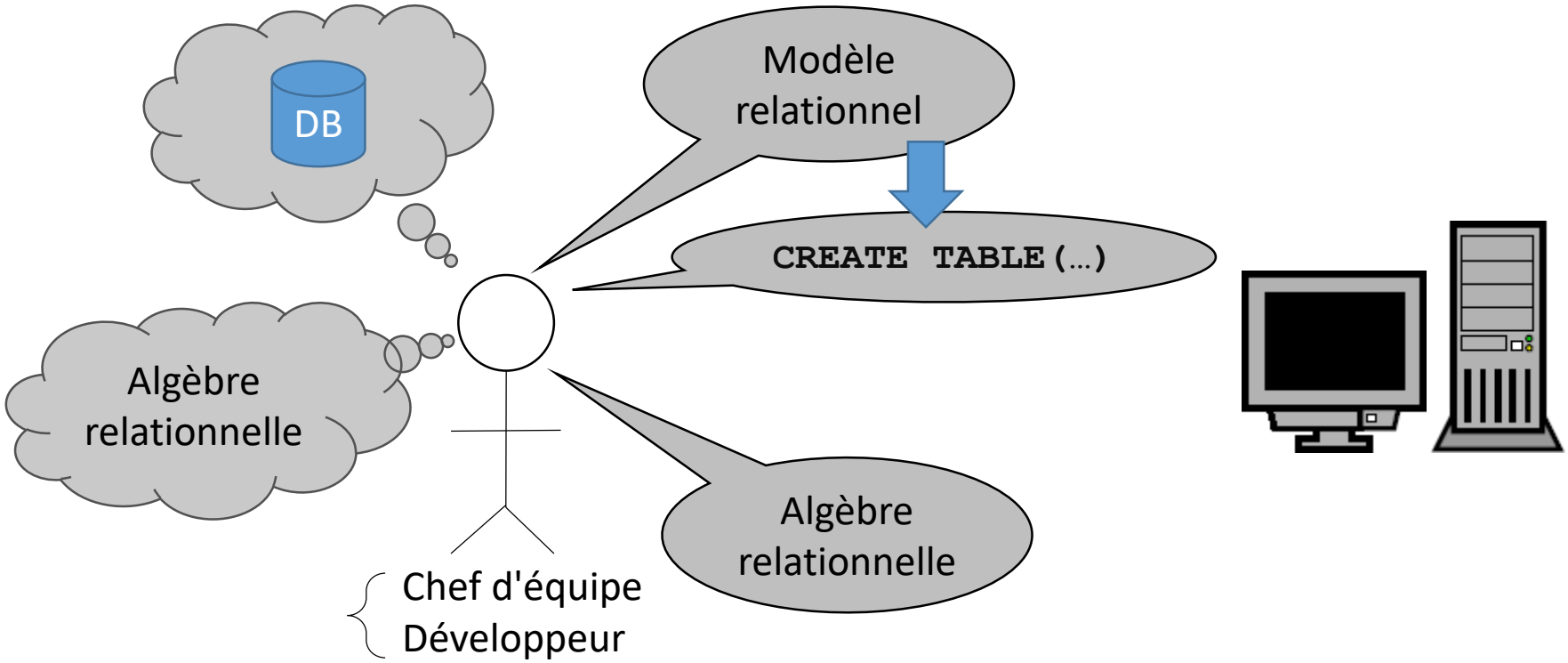


Le langage SQL : Pourquoi?

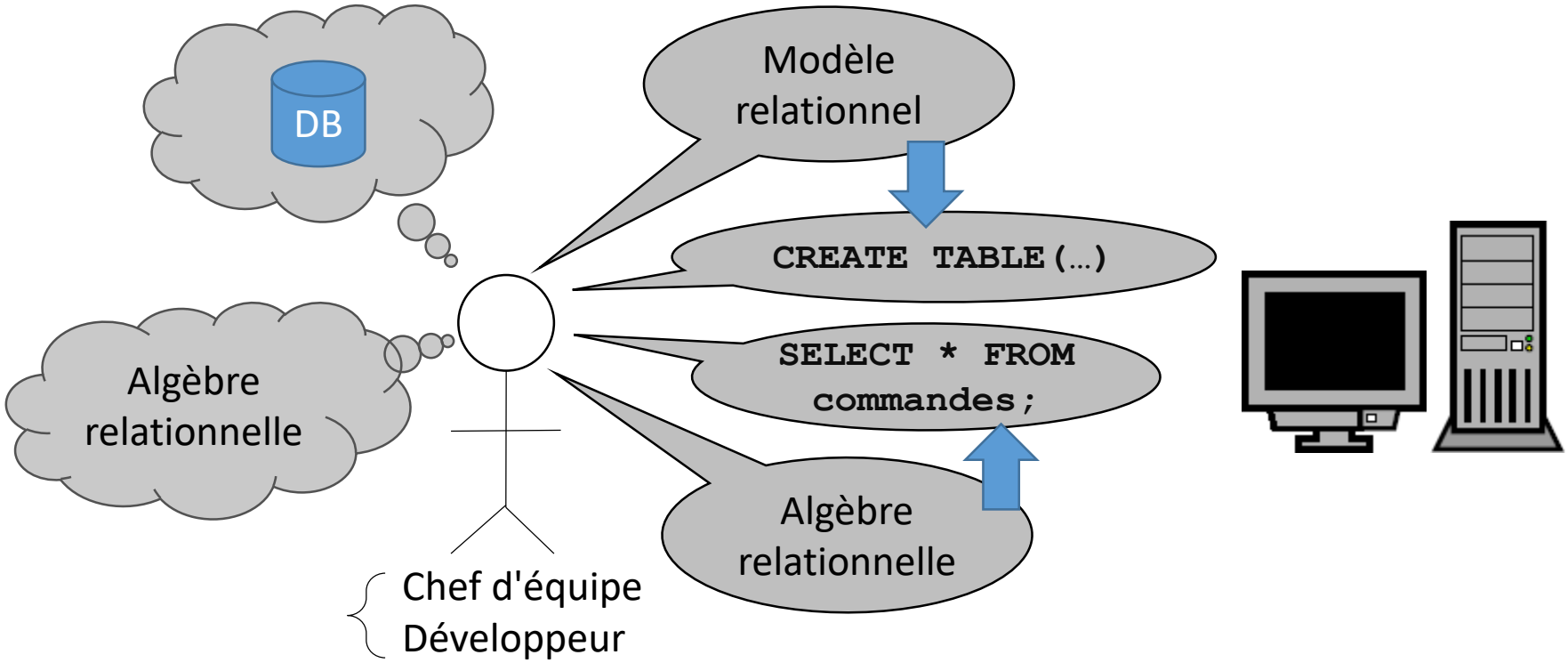




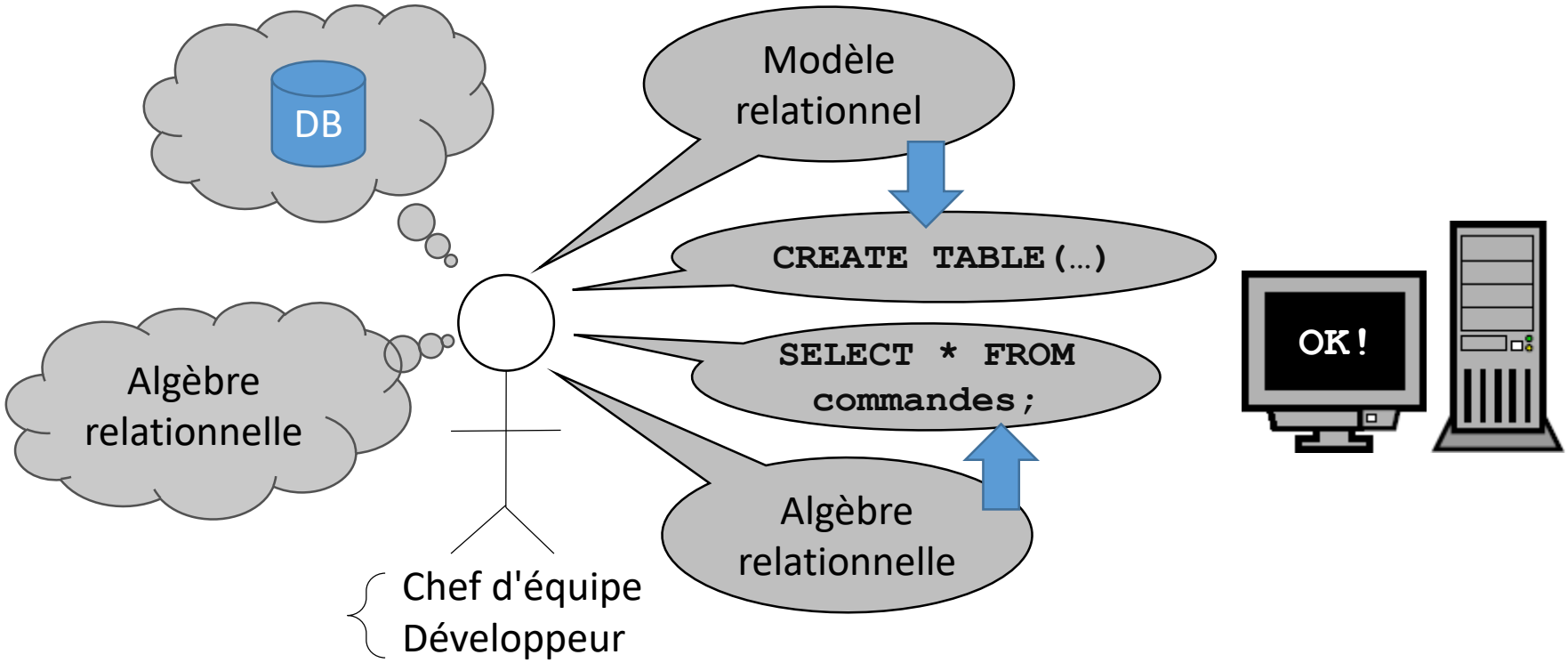
Le langage SQL : Pourquoi?



Le langage SQL : Pourquoi?



Le langage SQL : Pourquoi?



Le langage concret des bases de données relationnelles : SQL

- Langage de haut niveau proche de l'algèbre relationnelle.
- Permet d'exprimer les requêtes sans (trop) se soucier de leur implémentation : le système de gestion de bases de données se charge de l'optimisation et de l'exécution des requêtes.
- Permet des traitements élaborés des données qui ne doivent plus être manipulées au niveau de l'application.

L'origine de SQL remonte aux premiers systèmes de bases de données relationnelles (1970-1980). Le langage est standardisé par des comités de l'ISO et de l'ANSI :

- SQL1, 1986
- SQL2, 1992 (SQL-92)
- SQL3, 1999 (SQL-99), ...

Les versions implémentées diffèrent légèrement suivant les systèmes de gestion de bases de données et sont plus ou moins complètes.

SQL (Structured Query Language)

Langage de base de données très complet :

- *Langage interactif de manipulation de données* (*DML*) :
  - Langage d'interrogation (requêtes),
  - Modification du contenu de la base de données : insertion, suppression, modification de tuples.
- *Langage de définition de données* (*DDL*) :
  - *Schémas* : définition, modification de schémas, suppression de relations.
  - Définition de *vues*.
  - Gestion des *autorisations*.
  - Définition de contraintes d'*intégrité*.
- SQL peut être utilisé seul, ou à partir d'un langage de programmation (*"embedded" SQL*).
- SQL permet de gérer le *contrôle de transactions* (accès simultanés à la base de données).

Les requêtes en SQL

Structure de base des requêtes :

select $A_1, \ldots, A_n$
from $r_1, \ldots, r_m$
where P

En algèbre relationnelle : $\pi_{A_1,\ldots,A_n}(\sigma_P(r_1 \times \cdots \times r_m))$

NB : **select** correspond à une *projection* qui peut être une projection généralisée.

Exemple :

FOURNISSEURS (NOM_F, ADRESSE_F, COMB, PRIX)

Qui fournit du *charbon* ?

select *NOM_F*
from *fournisseurs*
where *COMB = 'charbon'*

La structure des conditions (where)

- les conditions sont composées de conditions plus simples par les opérateurs de conjonction **and** ($\wedge$), disjonction **or** ($\vee$) et négation **not** ($\neg$).
- Les condition simples s'expriment par des relations ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$) entre expressions arithmétiques ($+$, $-$, $*$, $/$) construites à partir de constantes et attributs.

Exemple :
CLIENTS (NOM, ADRESSE, SOLDE_DU)
COMMANDES (NO, NOM, COMB, QUANT)

Quelle est l'adresse des clients qui ont commandé du *mazout* ?

select *ADRESSE*
from *clients, commandes*
where *COMB = 'mazout'*
and *clients.NOM = commandes.NOM*

Variantes de la requête de base

- **select** $A_1, \ldots, A_n$
  from $r_1, \ldots, r_m$
  revient à supposer $P = \mathit{true}$ (donc pas de sélection)
  En algèbre relationnelle : $\pi_{A_1,\ldots,A_n}(r_1 \times \cdots \times r_m)$

- **select** $*$
  from $r_1, \ldots, r_m$
  where P
  revient à projeter sur tous les attributs de toutes les relations concernées
  En algèbre relationnelle : $\sigma_P(r_1 \times \cdots \times r_m)$

Comment interpréter les requêtes SQL ?

L'équivalence avec l'algèbre relationnelle donne une interprétation des requêtes, mais une vue opérationnelle est aussi utile.

Le résultat d'une requête **select** ... **from** ... **where** peut se déterminer comme suit :

1. On examine toutes les combinaisons de tuples de relations $r_1, \ldots, r_m$ (cela revient à considérer $r_1, \ldots, r_m$ comme des variables qui parcourent les tuples de leurs relations respectives).
2. Pour chaque combinaison de tuples, on détermine si la clause **where** est satisfaite.
3. Lorsque la clause **where** est satisfaite, on produit un tuple comportant les attributs spécifiés dans la clause **select**.

Opérations booléennes en SQL

union **intersect** **except** (différence)

Les relations auxquelles on applique ces opérations doivent être de même schéma.

(**select** $A_1, \ldots, A_n$
from $r_1, \ldots, r_m$
where P)
union
(**select** $A_1, \ldots, A_n$
from $r'_1, \ldots, r'_{m'}$
where P')

Exemple :

EMPLOYEE (EMP_ID, FNAME, LNAME, BDATE, ADDRESS, SALARY, DEPT_NO)

DEPARTMENT (DNO, DNAME, MGR_ID)

PROJECT (PNUM, PNAME, DNO)

WORKS_ON (EMP_ID, PNO, HOURS)

Donner les numéros et les noms de tous les projets dans lesquels l'employé *Smith* est impliqué, soit parce qu'il est le manager du département qui contrôle ce projet, soit parce qu'il travaille sur ce projet.

```
( select PNUM, PNAME from PROJECT, DEPARTMENT, EMPLOYEE
  where PROJECT.DNO = DEPARTMENT.DNO
        and MGR_ID = EMP_ID
        and LNAME = 'Smith'  )
 union
( select PNUM, PNAME
  from PROJECT, WORKS_ON, EMPLOYEE
  where PNUM = PNO and WORKS_ON.EMP_ID = EMPLOYEE.EMP_ID
        and LNAME = 'Smith'  )
```

Copies multiples en SQL

Par défaut, SQL n'élimine pas les copies multiples de tuples lors du calcul d'un **select**. . .**from**. . .**where**, mais le fait lors du calcul de l'opérateur **union**.

Ces choix s'expliquent par des considérations d'implémentation. On peut forcer un autre comportement en utilisant les mots clés **all** et **distinct**.

select distinct $A_1, \ldots, A_n$
from $r_1, \ldots, r_m$
where P

select . . .
union all
select . . .

Utiliser plusieurs fois la même relation en SQL

Il est parfois nécessaire de comparer les tuples d'une relation entre eux.

- En algèbre relationnelle cela peut se faire en prenant un produit de la relation avec elle même.
- En SQL, on mentionne deux fois la relation dans la clause **from**, en attribuant un identificateur à chaque occurrence de la relation.
- On parle parfois de "variable tuple", faisant référence au fait que chaque occurrence de la relation correspond à un parcours des tuples de celle-ci.

Exemple :

CLIENTS (NOM, ADRESSE, SOLDE_DU)

Quels sont le nom et l'adresse de tous les clients dont le solde dû est inférieur à celui de Dupont, A. ?

select *C1.NOM*, *C1.ADRESSE*
from *CLIENTS C1*, *CLIENTS C2*
where *C1.SOLDE_DU* $<$ *C2.SOLDE_DU*
and *C2.NOM* = *'Dupont, A.'*

Requêtes imbriquées en SQL

La condition **where** de l'instruction **select** peut être une condition relative à un ensemble qui est lui-même le résultat d'un **select**.

Exemple :

FOURNISSEURS (NOM_F, ADRESSE_F, COMB, PRIX)

COMMANDES (NO, NOM, COMB, QUANT)

Quels sont les fournisseurs qui fournissent au moins un combustible commandé par 'Dupont, A.' ?

select *NOM_F*
from *FOURNISSEURS*
where *COMB* **in**
(**select** *COMB*
from *COMMANDES*
where *NOM = 'Dupont, A.'*)

$$\pi_{NOM\_F}\Big(\pi_{COMB}(\sigma_{NOM='Dupont,\ A.'}(commandes)) \bowtie fournisseurs\Big)$$

Exemple :

EMPLOYEE (EMP_ID, FNAME, LNAME, BDATE, ADDRESS, SALARY, DEPT_NO)

DEPARTMENT (DNO, DNAME, MGR_ID)

PROJECT (PNUM, PNAME, DNO)

WORKS_ON (EMP_ID, PNO, HOURS)

Donner les numéros et les noms de tous les projets dans lesquels l'employé *Smith* est impliqué, soit parce qu'il travaille sur ce projet, soit parce qu'il est le manager du département qui contrôle ce projet.

```
select distinct PNUM, PNAME
from PROJECT
where PNUM in
      ( select PNUM from PROJECT, DEPARTMENT, EMPLOYEE
        where PROJECT.DNO = DEPARTMENT.DNO
              and MGR_ID = EMP_ID and LNAME = 'Smith'  )
or     PNUM in
      ( select PNUM
        from PROJECT, WORKS_ON, EMPLOYEE
        where PNUM = PNO and WORKS_ON.EMP_ID = EMPLOYEE.EMP_ID
              and LNAME = 'Smith'  )
```

Condition sur le résultat d'une requête imbriquée : Ensemble vide ou non

exists **not exists**

Exemple :

CLIENTS (NOM, ADRESSE, SOLDE_DU)
COMMANDES (NO, NOM, COMB, QUANT)

Quel est le nom des clients qui ont émis au moins une commande de bois ?

select *NOM*
from *CLIENTS*
where exists
(**select** * **from** *COMMANDES*
where *CLIENTS.NOM = COMMANDES.NOM* **and** *COMB = 'bois'*)

Quel est le nom des clients qui n'ont rien commandé ?

select *NOM* **from** *CLIENTS*
where not exists (**select** *
from *COMMANDES*
where *CLIENTS.NOM = COMMANDES.NOM*)

Condition : Test d'absence de valeur

Il s'agit de détecter si la valeur d'un attribut est absente, c'est-à-dire s'il n'y a pas de tuple contenant une valeur pour cet attribut, ou s'il a une *valeur nulle (null value)*.

Exemple :

EMPLOYEE (EMP_ID, FNAME, LNAME, BDATE, ADDRESS, SALARY, DEPT_NO)
SUPERVISION (EMP_ID, BOSS_ID)

Donner le nom et le prénom des employés qui n'ont pas de patron ?

```
select FNAME, LNAME
from EMPLOYEE
where not exists
      ( select *
        from SUPERVISION
        where EMPLOYEE.EMP_ID = SUPERVISION.EMP_ID )
```

Implémenter la division avec not exists

- Imaginons la requête suivante en algèbre relationnelle

$$\pi_{BOSS\_ID}(supervision) \div \pi_{EMP\_ID}(employee)$$

- On peut comprendre cette requête comme « Donnez les identifiants des patrons qui ont, au moins une fois, supervisé chaque employé »

- Ou, en utilisant la double négation « Donnez les identifiants des patrons pour lesquels il n'existe pas d'employés qui n'ont pas été supervisés par eux ».

- Cela devient, en SQL:

```
SELECT BOSS_ID
FROM supervision s1
WHERE NOT EXISTS
        (SELECT EMP_ID
        FROM employee e
        WHERE NOT EXISTS (SELECT EMP_ID
                                FROM supervision s2
                                WHERE e.EMP_ID = s2.EMP_ID
                                AND s1.BOSS_ID = s2.BOSS_ID))
```

Et si le schéma avait été :

EMPLOYEE (EMP_ID, FNAME, LNAME, BDATE,
ADDRESS, SALARY, DEPT_NO, BOSS_ID)

Donner le nom et le prénom des employés qui n'ont pas de chef ?

select *FNAME, LNAME*
from *EMPLOYEE*
where *BOSS_ID* **is null**

Valeurs nulles en SQL

En SQL lorsque l'on teste un attribut dont la valeur est nulle (**null**), le résultat n'est ni `vrai`, ni `faux`, mais `inconnu`.

Exemple :

EMPLOYEE (EMP_ID, FNAME, LNAME, BDATE, ADDRESS, SALARY, DEPT_NO, BOSS_ID)

select *FNAME, LNAME*
from *EMPLOYEE*
where $SALARY \leq 50.000$ **or** $SALARY > 50.000$

Ne sélectionne pas les tuples pour lesquels la valeur de *SALARY* est **null**.

Joints explicites en SQL

En SQL il est possible de calculer directement les différentes forme de joint. Un joint peut former une requête ou être utilisé dans une clause **from**.

- Produit cartésien : **cross join**.
- joint naturel : **natural join**.
- θ-joint : r **join** s **on** ...
- Joint externe : on ajoute les mots clés **full outer**, **left outer** ou **right outer**.

Fonctions d'agrégation en SQL

avg **count** **sum** **min** **max** **stddev** **variance**

Exemples :
CLIENTS (NOM, ADRESSE, SOLDE_DU)

FOURNISSEURS (NOM_F, ADRESSE_F, COMB, PRIX)

- Quelle est la moyenne des soldes dûs par les clients ?

  select avg(*SOLDE_DU*) *AV_SOLDE*
  from *CLIENTS*

- Combien y a-t-il de fournisseurs ?

  select count(**distinct** *NOM_F*) *#FOURN*
  from *FOURNISSEURS*

- Combien y a-t-il de fournisseurs de mazout ?

  select count(*NOM_F*) *#FOURN_MAZOUT*
  from *FOURNISSEURS*
  where *COMB = 'mazout'*

Opérateurs de Groupement en SQL

Partitionnement des tuples d'une relation en groupes sur lesquels on peut appliquer séparément des fonctions d'agrégation.

- **group by** $A_1, \ldots, A_k$
  partitionne la relation en groupes de telle manière que 2 tuples sont dans le même groupe s'ils ont des valeurs identiques pour les attributs $A_1, \ldots, A_k$

  Exemple :

  FOURNISSEURS (NOM_F, ADRESSE_F, COMB, PRIX)

  Table des combustibles avec leur prix moyen ?

  select *COMB*, **avg**(*PRIX*) *PM*
  from *FOURNISSEURS*
  group by *COMB*

Comparaison de chaînes de caractères en SQL

Quelques caractères spéciaux :
% dénote un nombre quelconque de caractères quelconques,
_ dénote un caractère quelconque.

Comparaison : **like** **not like**.

Exemple :

FOURNISSEURS (NOM_F, ADRESSE_F, COMB, PRIX)

COMMANDES (NO, NOM, COMB, QUANT)

Quel est le nom des fournisseurs dont l'adresse contient 'Laville' comme sous-chaîne ?

select *NOM_F*
from *FOURNISSEURS*
where *ADRESSE_F* **like** *'%Laville%'*

Exemple :

EMPLOYEE (EMP_ID, FNAME, LNAME, BDATE, ADDRESS, SALARY, DEPT_NO)

Donner le nom et le prénom des employés qui sont nés dans les années 60 (sachant qu'une date de naissance est une chaîne de caractères du type `aaaammjj`).

select *FNAME, LNAME*
from *EMPLOYEE*
where *BDATE* **like** '196_____'

Opérateur de Tri en SQL

Exemple :

EMPLOYEE (EMP_ID, FNAME, LNAME, BDATE, ADDRESS, SALARY, DEPT_NO)
DEPARTMENT (DNO, DNAME, MGR_ID)
PROJECT (PNUM, PNAME, DNO)
WORKS_ON (EMP_ID, PNO, HOURS)

Donner la liste des employés et des projets sur lesquels ils travaillent, classée selon le département auxquels les employés sont attachés (ordre alphabétique sur le nom de département) et pour chaque département, classée par nom de famille et prénom (ordre lexicographique).

select *DNAME, LNAME, FNAME, PNAME*
from *DEPARTMENT, EMPLOYEE, WORKS_ON, PROJECT*
where *DEPT_NO = DEPARTMENT.DNO*
and *WORKS_ON.EMP_ID = EMPLOYEE.EMP_ID* **and** *PNUM = PNO*
order by *DNAME, LNAME, FNAME*

Options : **asc** (ordre croissant) et **desc** (ordre décroissant)

Requêtes SQL

SELECT $< [DISTINCT]^{?} * | (Attr)^{+} >$
FROM $< (table | (table\ [\text{NATURAL}|\text{LEFT}|\text{RIGHT}|\text{OUTER}]^{?}\ \text{JOIN}\ table))^{+} >$
(WHERE $< (condition)^{+} >)^{?}$
(GROUP BY $< (Attr)^{+} >$
(HAVING $< (condition\ sur\ le\ groupe)^{+} >)^{?})^{?}$
(ORDER BY $< (Attr\ [\text{ASC}|\text{DESC}]^{?})^{+} >)^{?}$
(LIMIT $< nombre\ de\ tuples >)^{?}$
;

1. Inclusion des tables (FROM)
2. Sélection (WHERE; optionnel)
3. Groupement (GROUP BY; optionnel)
4. Sélection sur les groupes (HAVING; optionnel; seulement après un GROUP BY)
5. Tri (ORDER BY; optionnel; peut entrer en conflit avec GROUP BY)
6. Limitation du nombre de tuples (LIMIT; optionnel)
7. Projection (SELECT)

Modification du contenu de la base de données en SQL

Insertion :

insert into r
values $(v_1, \ldots, v_k)$

où r : nom d'une relation
$v_1, \ldots, v_k$: valeurs pour les attributs de r

Exemple : COMMANDES (NO, NOM, COMB, QUANT)

insert into *COMMANDES*
values (*7, 'Lebon,R.', 'bois', 8*)

Si l'on autorise les *valeurs nulles* : l'insertion peut ne fournir de valeurs que pour certains attributs (précisés).

Exemple : *FOURNISSEURS (NOM_F, ADRESSE_F, COMB, PRIX)*

insert into *FOURNISSEURS (NOM_F, ADRESSE_F)*
values (*'New_Brule', 'Rue au Bois 1 - Laville'*)

On peut insérer plus d'un tuple à la fois.

Exemple :

FOURNISSEURS (NOM_F, ADRESSE_F, COMB, PRIX)

Ajouter à la relation *tout_brule_vend* les combustibles vendus par 'Tout-Brule' avec leur prix.

insert into *TOUT_BRULE_VEND*
    select *COMB*, *PRIX*
    from *FOURNISSEURS*
    where *NOM_F = 'Tout-Brule'*

Ajout de tuples via un fichier

```
LOAD DATA
    [LOW_PRIORITY | CONCURRENT] [LOCAL]
    INFILE 'file_name'
    [REPLACE | IGNORE]
    INTO TABLE tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [CHARACTER SET charset_name]
    [{FIELDS | COLUMNS}
        [TERMINATED BY 'string']
        [[OPTIONALLY] ENCLOSED BY 'char']
        [ESCAPED BY 'char']
    ]
    [LINES
        [STARTING BY 'string']
        [TERMINATED BY 'string']
    ]
    [IGNORE number {LINES | ROWS}]
    [(col_name_or_user_var
        [, col_name_or_user_var] ...)]
    [SET col_name={expr | DEFAULT},
        [, col_name={expr | DEFAULT}] ...]
```

Suppression :

delete from r
where ψ

Modification :

update r
set $A_1 = \mathcal{E}_1, \ldots, A_k = \mathcal{E}_k$
where ψ

Chaque tuple μ de r qui satisfait ψ est modifié : $\mu[A_i] := \mathcal{E}_i$ pour $i = 1, \ldots, k$.

Exemple :

FOURNISSEURS (NOM_F, ADRESSE_F, COMB, PRIX)

Changements de prix :

update *FOURNISSEURS*
set *PRIX*=4300
where *NOM_F = 'Tout-Brule'*
and *COMB = 'bois'*

update *FOURNISSEURS*
set *PRIX = .9 * PRIX*
where *NOM_F = 'Tout-Brule'*