SQL

Université de Nice - Sophia Antipolis
Richard Grin
Version 2.15 L3 – 14/10/07

IMPORTANT

- L'environnement Oracle doit être installé avant le début du TP 6
- Notice d'installation à la fin du TP 6
- Nom de login Oracle :
  - info ou iup
  - 7 premières lettres du nom
  - initiale du prénom
- Exemple : infogrinr ou iupgrinr
- Mot de passe Oracle : le nom de login Oracle (pas le mot de passe Unix)

Introduction

Présentation de SQL

- SQL = Structured Query Language
  = langage d'interrogation structuré
- Langage de gestion de bases de données relationnelles pour
  - interroger
  - mettre à jour (LMD ; Langage de Manipulation des Données)
  - définir les données (LDD ; Langage de Définition des Données gérées)
  - contrôler l'accès aux données (LCD ; Langage de Contrôle de l'accès aux Données)

Norme SQL92 (ou SQL2)

- Norme adoptée en 1992 par l 'ISO (*International Organisation for Standardization*)
- Presque complètement implémentée par les principaux SGBD : Oracle, DB2, Informix, MySQL, PostsgreSQL, Access, SQL Server,…
- La norme SQL99 (ou SQL3) est déjà là depuis longtemps mais elle est loin d'être implémentée par tous ces SGBD

Oracle

- SGBDR (Relationnel) qui utilise le langage SQL
- Langage procédural PL/SQL (dit L4G) propriétaire
- Nombreux programmes utilitaires
  - SQL*PLUS : SQL interactif, avec quelques ajouts
  - SQL*FORMS : saisir/voir des données avec des formulaires
  - SQL*REPORTWRITER : rapports imprimés
  - WebDB pour l'interface avec le Web
  - ...

Connexion/Déconnexion

- Connexion par sqlplus :
  sqlplus [*nom*[/*mot de passe*]]
- Base de travail « **INFO** » pour vous
- Déconnexion :
  exit
  (ou EXIT)

Identificateurs

- Identificateurs pour les objets manipulés
  - 30 caractères au plus
  - lettres, chiffres, _, $ ou # (commence par une lettre)
  - pas un mot clef
- Quelques mots clefs : ASSERT, ASSIGN, AUDIT, COMMENT, DATE, DECIMAL, DEFINITION, FILE, FORMAT, INDEX, LIST, MODE, OPTION, PARTITION, PRIVILEGES, PUBLIC, SELECT, SESSION, SET,TABLE.

Tables

- Relations stockées sous forme de tables composées de lignes et de colonnes
- SQL92 : le nom d'une table est précédé du nom d'un schéma (pour réunir tous les objets liés à un même thème)
- Sous Oracle, le schéma est remplacé par le nom de l'utilisateur qui a créé la table : BERNARD.DEPT
- Par défaut, le schéma est le nom de l'utilisateur connecté

Exemple de table ; DEPT

Dept	NomD	Lieu
10	Finances	Paris
20	Recherche	Grenoble
30	Ventes	Lyon

Exemple de table ; EMP

Matr	**Nom**	**Sal**	**Com**	**Sup**	**Dept**
1200	Dupond	2500	300	2200	10
2200	Durand	3000	500		10
1780	Boisier	2500		2200	20

Colonne

- Toutes les données d'une colonne sont d'un même type
- Identificateur unique pour les colonnes d'une table, mais 2 colonnes dans 2 tables différentes peuvent avoir le même nom
- Le nom complet d'une colonne comprend le nom de la table à laquelle elle appartient (obligatoire en cas d'ambiguïté) : DEPT.dept, BERNARD.DEPT.dept

Types de données

Types de données SQL2

- Types numériques
- Types pour les chaînes de caractères
- Types temporels (dates, heures, minutes,…)
- SQL2 n'a pas de type pour les données très volumineuses telles que les images et les sons
- SQL2 ne permet pas à l'utilisateur de créer ses propres types

Types numériques (1)

- Nombres entiers :
  - **`SMALLINT`** sur 2 octets
  - **`INTEGER`** sur 4 octets
- À virgule flottante :
  - **`REAL`**
  - **`DOUBLE PRECISION`** (ou **`FLOAT`**)
- Constantes : 253.8, -10, 1.3E-5

Types numériques (2)

- Nombres décimaux à nombre fixe de décimales :
  - **`DECIMAL(nbChiffres, nbDécimales)`**
  - **`NUMERIC(nbChiffres, nbDécimales)`**

Précision imposée pour les calculs

- NUMERIC(**8**, 2) ou DECIMAL(**8**, 2) : **6** chiffres avant la virgule et 2 après
- Constantes : 253.8, -10

Types chaînes de caractères

- **`CHAR`**(*longueur*)
  chaînes de caractères avec un nombre fixe de caractères
- **`VARCHAR`**(*longueurMaximum*)
  chaînes de caractères avec un nombre variable de caractères (mais un nombre maximum de caractères)
- CHAR(5) : chaîne de 5 caractères
- VARCHAR(20) : chaîne de 20 caractères au plus
- Constante : **`'Comptabilité'`**, **`'Aujourdh''ui'`**

' doublé

Types temporels

- **`DATE`** pour les dates
- **`TIME`** pour les heures, minutes et secondes
- **`TIMESTAMP`** pour un moment précis : date et heures, minutes et secondes, avec une précision jusqu'à la microseconde (un millionième de seconde)

Type booléen

- **`BIT`** permet d'enregistrer un bit
- Exemples :
  BIT(1)
  BIT(4)
- Pas supporté par Oracle

Types numériques d'Oracle

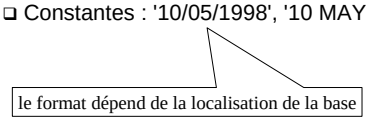
- Oracle accepte les types numériques SQL2 mais il les traduit dans ses propres types
- **`NUMBER`**
  nombre à virgule flottante avec jusqu'à 38 chiffres significatifs
- **`NUMBER`**(*nbChiffres [, nb-décimales]*)
  nombre décimal d'au plus *nbChiffres* chiffres dont *nb-décimales* après la virgule

Types chaînes de caractères d'Oracle

- CHAR, comme la norme SQL2
- VARCHAR est accepté mais Oracle conseille d'utiliser VARCHAR2 qui a les mêmes propriétés

Types temporels d'Oracle

- Le type DATE remplace les types DATE et TIME de SQL2
- DATE correspond à une date avec une précision jusqu'à la seconde
- Constantes : '10/05/1998', '10 MAY 1998'



Interrogations simples

Select simple

SELECT expression1, expression2, …
FROM table
WHERE prédicat

```
select nomE, poste from emp;
select * from dept;
select nomE, sal + comm from emp;
select matr, nomE, sal * 1.15 from emp
  where sal + comm >= 12500;
```

Expressions

- Ces expressions se trouvent à la suite du **`select`** ou du **`where`** ; elles peuvent comporter des
  - noms de colonnes
  - constantes
  - opérateurs arithmétiques
  - concaténations de chaînes de caractères (||)
  - calculs sur les dates (+ et -)
  - fonctions

Valeur NULL

- Valeur attribuée aux attributs qui n'ont pas reçu de valeur
- Les expressions qui contiennent la valeur NULL ont la valeur NULL :
  `comm + sal` est NULL si **`comm`** est NULL
- Sauf la fonction **`COALESCE`**

COALESCE

- COALESCE(*expr1*, *expr2*,…)
  retourne la valeur de la 1ère expression non NULL parmi les expressions *expr1*, *expr2*,…
- **`COALESCE(comm, 0)`**
- **`COALESCE(comm, salaire, 0)`**
- **`COALESCE(poste, 'simple soldat')`**
- Supporté par Oracle 10g mais pas par Oracle 9i

NVL

- NVL peut être utilisé à la place de coalesce dans Oracle 9i
- NVL(expression, valeur)
  renvoie la valeur de l'expression si elle n'est pas NULL, et valeur sinon
- Exemple :
  NVL(comm, 0)

NULLIF

- Permet de récupérer des données d'une BD qui n'utilisait pas la valeur NULL
- Exemple :

```
select nomE, NULLIF(comm, -1) from emp
```


Signification de NULL

- NULL signifie qu'une donnée est manquante et qu'on ne connaît donc pas sa valeur
- Cependant, dans la pratique il peut signifier (entre autres) :
  - pas applicable (seuls les commerciaux touchent une commission) ; à éviter, mais possible pour simplifier le schéma de la base
  - valeur 0 : on peut considérer que la valeur NULL correspond à la valeur 0 ; **à éviter !**

Logique à 3 valeurs

- La valeur NULL implique une logique à 3 valeurs :
  - vrai
  - faux
  - on ne sait pas si c'est vrai ou faux
- Par exemple, la condition « salaire > 1000 » pour un employé peut être
  - vrai, si le salaire est 1200
  - faux, si le salaire est 700
  - on ne sait pas, si le salaire est NULL

Implication de cette logique

- Soit une liste L qui contient NULL : (1, 8, NULL, 78, 7)
- Est-ce que 1 appartient à L ?
- Évidemment oui
- Est-ce que 10 appartient à L ?
- On ne sait pas !
- Est-ce que 10 n'appartient pas à L ?
- On ne sait pas !

Pseudo-table DUAL

- Particularité d'Oracle
- Contient une seule ligne et une seule colonne
- Ne peut être utilisée qu'avec une requête select
- Pour afficher une expression dont la valeur ne dépend d'aucune table en particulier
- **`select user from dual;`**
- **`select sysdate from dual;`**

Création d'une table (LDD)

Création d'une table

CREATE TABLE *table* (
colonne1 type1,
colonne2 type2,
. . .
. . .)

- Exemple :

```
create table article (
  ref char(5) not null,
  nom varchar(20),
  prix numeric(9,2),
  dateMAJ date);
```

Option **`not null`** si la colonne doit obligatoirement être renseignée

Valeur par défaut

- On peut donner une valeur par défaut pour une colonne :

```
create table dept (
  numDept integer not null,
  nomDept varchar(20),
  ville varchar(30) default 'Nice');
```

- On peut aussi donner une fonction comme valeur par défaut ; par exemple, **`default sysdate`**

DESCRIBE

- Cette commande SQL*PLUS d'Oracle (pas SQL) affiche une description des colonnes d'une table :

```
SQL > describe article;
Name                Null ?     Type
REF                 Not null   CHAR(5)
NOM                            VARCHAR(20)
PRIX                           NUMBER(9,2)
DATEMAJ                        DATE
```

Même si le type donné à la création est DECIMAL

Contraintes d'intégrité

Définition

- Une contrainte d'intégrité est une contrainte que doivent vérifier les données d'une table
- Une commande est annulée par le SGBD si son exécution viole une des contraintes

Types de contraintes d'intégrité

- PRIMARY KEY : clé primaire
- FOREIGN KEY … REFERENCES : clé étrangère
- UNIQUE : 2 lignes ne peuvent avoir la même valeur pour les colonnes spécifiées
- CHECK : contrainte de domaine, ou autre ; porte sur une seule ligne

Types de contraintes d'intégrité

- 2 types de contraintes :
  - contrainte de colonne (concerne une seule colonne)
  - contrainte de table

Définition des contraintes

- Les contraintes sont définies dans les commandes CREATE (ou ALTER) TABLE
  - à l'intérieur des définitions de colonnes pour les contraintes de colonne
  - au même niveau que les définitions de colonnes pour les contraintes de table
- CONSTRAINT *nomContrainte définitionContrainte*

Clé primaire

- Si la clé primaire n'est formée que d'une seule colonne, le plus simple est d'ajouter une contrainte de colonne :

```
create table emp (
  matr integer constraint pkemp primary key,
  . . .
```

- Sinon, il faut ajouter une contrainte de table :

```
create table participation (
  matr integer,
  codeP integer,
  . . .,
  constraint pkpar primary key(matr, codeP))
```


Une erreur à ne pas faire

- Si une table a une clé primaire formée de 2 colonnes, il ne faut pas déclarer 2 contraintes de colonne
- Il faut déclarer une seule contrainte de table portant sur les 2 colonnes :
  `constraint pkpar primary key(matr, codeP)`

Contrainte sur les clés primaires

- Aucune des colonnes de la clé primaire ne peut avoir la valeur **`null`**

Contrainte UNIQUE

- 2 lignes de la table ne pourront avoir la même valeur (sauf NULL)
- Correspond à un identificateur (clé candidate si minimal), si on ajoute une contrainte NOT NULL

Clé étrangère

- Si une seule colonne forme la clé étrangère, le plus simple est d'utiliser une contrainte de colonne :

```
create table emp (
  . . .,
  dept integer
    constraint r_dept references dept(dept))
```

Optionnel si colonne référencée est clé primaire

Clé étrangère

- Peut être une contrainte de table :
  FOREIGN KEY (*colonne1*, *colonne2*,…)
  REFERENCES *table-ref* [(*col1*, *col2*,…)]
- Exemple :

```
create table emp (
  . . .,
  dept integer,
  constraint r_dept
    foreign key(dept) references dept(dept))
```

Il faut ajouter foreign key

Clés étrangères

- Les colonnes de l'autre table référencées (*col1*, *col2*,…) doivent avoir la contrainte PRIMARY KEY ou UNIQUE

```
constraint r_dept references dept(dept)
```

dept doit être clé primaire, ou unique

Option ON DELETE CASCADE (sans)

```
create table emp (
   . . .
  dept integer
    constraint r_dept references dept)
```

- On ne peut supprimer un département s'il est référencé par une ligne de la table **emp**

Option ON DELETE CASCADE (avec)

```
create table emp (
   . . .
  dept number(2)
    constraint r_dept references dept
      on delete cascade)
```

- La suppression d'un département entraîne automatiquement la suppression de toutes les lignes de la table **emp** qui référencent ce département

Autres options pour les clés étrangères

- on delete set null
- 4 autres options de SQL2 pas implémentées par Oracle :
- on delete set default
- on update cascade
- on update set null
- on update set default

Exemples divers de contraintes

```
create table emp (

  matr integer constraint pkemp primary key,

  nomE varchar(30) constraint nom_unique unique
       constraint maj check (nomE = upper(nomE)),

  dept smallint constraint r_emp_dept references dept
       constraint ndept check (dept in (10, 20, 30, 40)));
```


Exemples divers de contraintes

```
create table participation (

  matr integer constraint r_part_emp references emp,

  codeP varchar(5) constraint r_part_projet
          references projet,
  …,

  constraint pkpart primary key(matr, codeP));
```

Modification des contraintes

```
ALTER TABLE emp
    DROP CONSTRAINT nom_unique
    ADD (CONSTRAINT sal_min
          check(sal + coalesce(comm, 0) > 5000))
    RENAME CONSTRAINT truc TO machin;
```

- On ne peut ajouter que des contraintes de table

Vérification des contraintes

- En fonctionnement normal les contraintes sont vérifiées à chaque requête SQL
- Cette vérification peut être gênante, en particulier lors de l'ajout de plusieurs lignes de données
- Exemple : si on a cette contrainte sur la colonne SUP de la table EMP :
  `constraint sup_ref_emp references EMP`
- La contrainte oblige à ajouter les supérieurs en premier

Contraintes « différables »

- Pour pallier ce problème, la vérification d'une contrainte peut être différée à la fin de la transaction
- Exemple :

  CONSTRAINT **`sup_ref_emp`** *references dept* DEFERRABLE

Contraintes « différables »

- La syntaxe :

  CONSTRAINT *nom-contrainte def-contrainte*
  [NOT] DEFERRABLE
  [INITIALLY {DEFERRED | IMMEDIATE}]
- La valeur par défaut est NOT DEFERRABLE

Différer une contrainte

- Indiquer qu'une contrainte est différable ne suffit pas pour la différer si elle n'a pas été déclarée « INITIALLY DEFERRED »
- Par défaut, une contrainte différable ne l'est que si on la diffère par la commande

  SET CONSTRAINT nom-contrainte DEFERRED;
- Elle ne sera différée que pour la durée d'une transaction

Invalider des contraintes avec Oracle

- Oracle permet aussi d'invalider des contraintes
- Utile pour, par exemple, améliorer les performances lors de l'ajout d'une grande quantité de données dans la base :

  ALTER TABLE table
  {DISABLE | ENABLE} constraint *nom-contrainte*

Dictionnaire des données

Définition

- Tables qui stockent les descriptions des objets de la base
- Tenues à jour automatiquement par le SGBD
- Peuvent être consultées au moyen du langage SQL

2 types de vues

- Vues nommées USER_* décrivent les objets du schéma qui appartiennent à l'utilisateur ; par exemple, USER_TABLES
- Vues nommées ALL_* décrivent les objets du schéma qui appartiennent à l'utilisateur, ou sur lesquels l'utilisateur a reçu des droits ; ALL_TABLES

Tables ou vues du dictionnaire

- DICTIONARY (DICT) : vues du dictionnaire
- USER_TABLES (TAB) : tables et vues crées par l'utilisateur
- USER_CATALOG (CAT) : tables et vues sur lesquelles l'utilisateur a des droits, sauf celles du dictionnaire des données
- USER_TAB_COLUMNS (COLS) : colonnes des tables ou vues créées par l'utilisateur
- USER_INDEXES (IND) : index créés par l'utilisateur ou indexant des tables de l'utilisateur
- ...

Exemple d'utilisation

- Recherche d'informations sur les contraintes
- On commence par chercher les noms des vues :

```
select table_name from dict
where table_name like '%CONSTRAINTS%';
```

- On fait afficher les noms des colonnes :

```
describe USER_CONSTRAINTS
```

- On cherche les informations que l'on veut :

```
select constraint_name, constraint_type,
table_name
from user_constraints
```


Langage de manipulation des données (LMD)

Commandes de manipulation des données

- **INSERT** pour ajouter des lignes
- **UPDATE** pour modifier des lignes
- **DELETE** pour supprimer des lignes

Insertion

```
INSERT INTO table [(colonne1, colonne2,…)]
  VALUES (valeur1, valeur2,…)
```

ou

```
INSERT INTO table [(colonne1, colonne2,…)]
  select …
```

- La liste des colonnes est optionnelle ; par défaut, toutes les colonnes sont dans l'ordre donné lors de la création de la table
- Si la commande comporte une liste, les colonnes qui ne sont pas dans la liste auront la valeur NULL

Insertion

- Dans les programmes, il faut toujours donner la liste des colonnes dont on donne les valeurs pour faciliter la maintenance de l'application
- En interactif, on peut s'en passer

Exemples

```
insert into dept
  values (10, 'Finance', 'Paris');
insert into dept (lieu, nomD, dept)
  values ('Grenoble', 'Recherche', 20);
insert into emp (matr, nomE, dept, sal)
  select matr + 100, nomE, 60, sal * 0.15
    from emp
    where dept = 10;
```


Modification

```
UPDATE table
  SET colonne1 = expr1, colonne2 = expr2, …
  [ WHERE prédicat ]
```

Toutes les lignes par défaut

ou

```
UPDATE table [ synonyme ]
  SET (colonne1, colonne2, …) = (select …)
  [ WHERE prédicat ]
```

Ne doit renvoyer qu'une seule ligne

Exemples

```
update emp
  set dept = 10
  where nomE = 'Martin';
update emp
  set sal = sal * 1.1
  where poste = 'Commercial';
```

Exemples (2)

```
update emp
  set sal = (select avg(sal) * 1.1
                from emp
                where poste = 'Secrétaire')
  where nomE = 'Clément';
update emp E
  set (sal, comm) =
    (select avg(sal), avg(comm) from emp
       where dept = E.dept)
  where poste = 'Secrétaire';
```

Synchronisation

Suppressions

```
DELETE FROM table
  [ WHERE prédicat ]
```

- Attention ! s'il n'y a pas de clause **WHERE**, toutes les lignes sont supprimées
- Exemple :

```
delete from emp
where dept = 10;
```


Transactions

Une requête forme un tout indivisible

- Si une erreur survient pendant l'exécution d'une requête SQL, toutes les modifications déjà effectuées par la requête sont annulées automatiquement
- On peut généraliser ce comportement à un ensemble de requêtes SQL en utilisant les transactions

Transaction

- Ensemble de modifications de la base qui forment un tout indivisible :

  à la fin de la transaction, toutes les modifications effectuées pendant la transaction sont sauvegardées ou annulées

Début d'une transaction

- Dans la norme SQL2 toute modification appartient à une transaction
- Une transaction démarre au début de la session
- Une transaction démarre automatiquement après la fin d'une transaction (pas de commande pour démarrer une transaction)
- La structure des transactions est « plate » : les transactions ne peuvent se chevaucher

Terminer une transaction

- Pour terminer une transaction on peut
  - valider la transaction (COMMIT) : toutes les modifications deviennent effectives
  - annuler la transaction (ROLLBACK) : toutes les modifications sont annulées
- Les ordres DDL (create table par exemple) provoquent un COMMIT automatique

Propriétés des transactions - ACID

- **A**tomicité : un tout indivisible
- **C**ohérence : une transaction doit laisser la base dans un état cohérent ; elle ne doit pas mettre les données dans un état « anormal »
- **I**solation : une transaction est isolée des autres transactions (dans une certaine mesure…)
- **D**urabilité : le SGBD doit garantir que les modifications d'une transaction validée seront conservées, même en cas de panne

Propriétés des transactions - ACID

- **AID** est du ressort du système transactionnel du SGBD
- **C** est du ressort de l'utilisateur mais il est aidé
  - par **I,** car il n'a pas à considérer les interactions avec les autres transactions
  - par la vérification automatique des contraintes d'intégrité par le SGBD
- **I** est assuré par le système de contrôle de la concurrence du SGBD et **AD** sont supportés par les procédures de reprise après panne du SGBD

Exemple d'annulation d'une transaction

```
insert into dept
    values (10, 'Finance', 'Paris');
delete from emp;
rollback;
```


Transaction « non terminée »

- La dernière transaction est automatiquement validée à la sortie de SQL*PLUS si elle ne se termine pas par un ROLLBACK
- Attention ! ce comportement dépend du logiciel utilisé ; avec d'autres logiciels une transaction non validée explicitement est annulée
- ⇒ Toujours terminer explicitement une transaction

Isolation des transactions

- En fonctionnement standard les modifications effectuées par une transaction T ne sont connues par les autres transactions qu'après validation de T
- En fait, il existe plusieurs niveaux d'isolation (voir cours sur la concurrence)

Transactions longues

- Exemple : organisation par une agence de voyage d'un voyage Nice – Wuhan (Chine)
- Nécessite la réservation de plusieurs billets d'avion : Nice – Paris ; Paris – Beijing ; Beijing – Wuhan
- On commence par réserver les 2 premiers mais si on ne peut trouver de Beijing – Wuhan, il faut tout annuler
- On met donc toutes ces réservations dans une transaction ; ça peut être long si l'agence discute avec le client pendant la transaction

Problèmes avec les transactions longues

- Manque de souplesse : si on ne trouve pas de voyage Beijing – Wuhan, on annule tout
- On aurait pu garder le Nice – Paris et essayer de passer par Shanghai pour aller à Wuhan, en annulant seulement le Paris – Beijing
- Autre problème : le contrôle de la concurrence effectue des blocages sur les tables et les lignes qui ne sont relâchés qu'à la fin de la transaction
- Un problème de communication peut provoquer l'annulation des premières réservations alors qu'on pourrait simplement réessayer le lendemain

Transactions emboîtées

- Extension de la notion de transaction « plate »
- Évite les annulations complètes de transactions
- Apporte plus de souplesse dans les transactions longues et multi-sites
- Permet de limiter la durée des blocages des ressources système

Définition des transactions emboîtées

- Une transaction globale (mère) peut contenir des sous-transactions filles qui, elles-mêmes, peuvent avoir des filles
- L'annulation d'une transaction n'annule pas nécessairement la transaction mère ; celle-ci peut
  - décider d'un traitement substitutif
  - reprendre la transaction annulée
  - s'annuler
  - ou même ignorer l'annulation (traitement pas indispensable)
- L'annulation d'une transaction provoque l'annulation automatique de toutes ses transactions filles

Points de reprise

- Sans passer au modèle des transactions emboîtées, on peut assouplir le modèle des transactions plates
- Désigner des points de reprise dans une transaction : **`savepoint nomPoint`**
- Possible d'annuler toutes les modifications effectuées depuis un point de reprise : **`rollback to nomPoint`**
- Évite d'annuler toute la transaction et permet d'essayer de pallier le problème

Exemple

```
insert into ….;
savepoint p1;
delete from …;
update …;
savepoint p2;
insert into …; -- Problème !
rollback to p2;
insert into …; -- on essaie autre chose
commit;
```

Interrogation de la base

Syntaxe générale

```
SELECT …
FROM …
WHERE …
GROUP BY …
HAVING …
ORDER BY …
```

- L'ordre des clauses est imposé
- SELECT et FROM sont obligatoires

Clause SELECT

```
select [distinct] expression1 [ [AS] nom1],
       expression2 [ [AS] nom2],
         . . .
```

- ***nom1*** :
  - en-tête de la colonne (entre guillemets si mot-clef ou si contient plusieurs mots)
  - alias pour désigner la colonne dans une autre partie du select

```
select *
```
Toutes les colonnes

Exemples

```
select distinct poste from emp;
select
  nomE,
  sal + coalesce(comm, 0) as "Salaire Total"
from emp;
```


select dans une expression

-
```
select nomE,
       sal / (select sum(sal) from emp) * 100
  from emp
```
- Le select de l'expression peut être synchronisé avec le select principal :

```
select nomE,
       (select count(*)
        from emp
        where sal > e1.sal) + 1 as rang
  from emp e1
```
synonyme de emp pour lever l'ambiguïté dans le select interne

select dans une expression

- On peut utiliser l'alias pour trier par ordre décroissant des salaires :

```
select nomE,
       (select count(*)
        from emp
        where sal > e1.sal) + 1 as rang
from emp e1
order by rang
```

Clause FROM

FROM ***table1*** [***synonyme1***], ***table2*** [***synonyme2***], …

- Produit cartésien des tables s'il y en a plusieurs
- Possible de se restreindre à un sous-ensemble du produit cartésien (voir jointure)


```
select B.dept, A.nomD
from dept A, dept B

      DEPT NOMD
---------- ------------
        30 VENTES
        20 VENTES
        10 VENTES
        30 RECHERCHE
        20 RECHERCHE
        10 RECHERCHE
        30 FINANCE
        20 FINANCE
        10 FINANCE
```

10 VENTES
20 RECHERCHE
30 FINANCE

Clause FROM

- Certains SGBDs (et la norme SQL-2) permettent l'utilisation d'un **SELECT** à la place du nom d'une table :

```
select nomE, sal,
       sal / total * 100 Pourcentage
  from emp,
       (select sum(sal) as total from emp);
```


Clause WHERE

- La clause WHERE comporte de nombreuses possibilités :
  - opérateurs de comparaison
  - opérateurs logiques
  - jointures
  - sous-interrogations

Opérateurs de comparaison

- **=**, **!=**, **<**, **>**, **<=**, **>=**, **BETWEEN**, **LIKE**, **NOT LIKE**, **IN**, **NOT IN**, **IS NULL**, **IS NOT NULL**
- **LIKE** permet d'utiliser des jokers :
  - **%** pour une chaîne de caractères de longueur quelconque
  - _ pour un seul caractère
- Attention,
  ~~expression = **NULL**~~ n'est jamais vrai,
  il faut utiliser expression **IS NULL**

Exemples

```
select * from emp where poste = 'Secrétaire';
select * from emp
  where sal between 10000 and 15000;
select * from emp where dept in (10, 30);
select * from emp where comm is not null;
select * from emp where nomE like '%A%';
```

Opérateurs logiques

- **AND**, **OR**, **NOT**
- Exemples :

```
select nomE from emp
where dept = 30
  and (sal > 10000 or comm is null);
select * from emp
where not (poste = 'Directeur'
           or poste = 'Secrétaire');
```

et si on met **and** à la place de **or** ?

Jointures

- Traduction de l'équi-jointure « emp J{dept} dept » :

```
select nomE, nomD
from emp, dept
where emp.dept = dept.dept
```

- Autre syntaxe :

```
select nomE, nomD
from emp JOIN dept
     ON emp.dept = dept.dept
```


Jointure de plus de 2 tables

- ```
  select nomE, nomP
  from emp, participation, projet
  where emp.matr = participation.matr
    and participation.codeP = projet.codeP
  ```
- Autre syntaxe :

```
select nome, nomp
from emp
  join participation
    on emp.matr = participation.matr
  join projet
    on participation.codep = projet.codep
```


Jointure naturelle

- La jointure s'effectue sur *toutes* les colonnes qui ont le même nom dans les 2 tables ; ces colonnes ne sont pas répétées dans la jointure
- Les colonnes qui participent à la jointure ne doivent être préfixées par un nom de table

Exemples de jointure naturelle

- ```
  select nomE, nomD, dept
  from emp NATURAL JOIN dept
  ```
- ```
  select nome, nomp
  from emp
       NATURAL JOIN participation
       NATURAL JOIN projet
  ```


Jointure d'une table avec elle-même

- Alias indispensable pour le nom de la table afin de lever l'ambiguïté sur les colonnes :

```
select emp.nomE "Employé",
       supe.nomE "Supérieur"
  from emp join emp supe
       on emp.sup = supe.matr
```
```

Jointures « non équi »

- Les jointures « non équi » peuvent être traduites comme les équi-jointures, en utilisant d'autres opérateurs de comparaison

```
select emp1.nomE, emp2.nomE
from emp emp1
     join emp emp2
       on emp1.sal < emp2.sal
```


Jointure externe

- Dans une jointure n'apparaissent que les lignes qui ont une ligne correspondante dans l'autre table
- Dans l'exemple suivant, un département qui n'a pas d'employé n'apparaîtra pas :

```
select nomE, nomD
  from emp join dept
       on emp.dept = dept.dept
```

- Si on veut qu'il apparaisse, on doit utiliser une jointure externe

Syntaxe SQL-2 de la jointure externe

-
```
select nomE, nomD
from emp RIGHT OUTER JOIN dept
         ON emp.dept = dept.dept
```
- RIGHT indique que l'on veut afficher toutes les lignes de la table de droite (dept)
- Ça revient à ajouter une « ligne fictive » dans l'autre table emp
- Cette ligne fictive aura toutes ses colonnes à **`null`**, sauf la colonne de jointure
- Il existe de même **`LEFT OUTER JOIN`** et **`FULL OUTER JOIN`**

Syntaxe Oracle de la jointure externe

- Syntaxe d'Oracle avant la version 9i
- On ajoute un (+) du côté où il « manque » une ligne
- Pour l'exemple précédent, il « manque des employés », donc on ajoute (+) du côté de la table **`emp`** :

```
select nomE, nomD
from emp, dept
where emp.dept (+) = dept.dept
```


Sous-interrogations

- Une clause WHERE peut comporter un ordre SELECT emboîté :

```
select nomE from emp
where poste = (select poste from emp
               where nomE = 'Martin');
```

- Cette sous-interrogation doit ramener une seule ligne et une seule colonne
- Remplace le select interne par NULL s'il ne renvoie aucune ligne (ou erreur, suivant les SGBD)

- Des variantes de sous-interrogations ramènent plusieurs colonnes ou plusieurs lignes

Sous-interrogation ramenant 1 ligne, 1 colonne

- WHERE *expression op* (SELECT …)
  où *op* est un des opérateurs de comparaison =, !=, <, >, <=, >=
- Exemple :

```
select nomE from emp
where sal >=
  (select sal from emp
   where nomE = 'Mercier')
```


Sous-interrogation ramenant plusieurs lignes

- **`WHERE` *`expression op`* `ANY (SELECT …)`**
  `WHERE` *`expression op`* `ALL (SELECT …)`
  `WHERE` *`expression`* `IN (SELECT …)`
  `WHERE` *`expression`* `NOT IN (SELECT …)`
  où *op* est un des opérateurs de comparaison =, **`!=`**, **`<`**, **`>`**, **`<=`**, **`>=`**
- **`ANY`** : vrai si la comparaison est vraie pour au moins une des valeurs ramenées par le **`SELECT`**
- **`ALL`** : vrai si la comparaison est vraie pour toutes les valeurs ramenées par le **`SELECT`**

- Remarque :
  - = ANY est équivalent à IN
  - != ALL est équivalent à NOT IN

Exemple

```
select nomE, sal from emp
where sal > all (select sal from emp
                 where dept = 30)
select nomE, sal from emp
where sal > all (select sal from emp
                 where dept = 38888)
```

Ce département n'existe pas !

Réflexion sur ALL

- Quand « x > all (x1, x2, …, xn) » est faux ?
- Quand ∃ un xi tel que xi ≥ x
- Si ∄ un xi tel que xi ≥ x, l'expression est vraie
- Donc si la liste (x1, x2, …, xn) est vide, l'expression est toujours vraie

Retour sur **`NULL`**

- La condition
  `expression not in (expr1, expr2, null)`
  n'est jamais vérifiée. Pourquoi ?
- Est-ce que la condition
  `expression in (expr1, expr2, null)`
  peut être vérifiée ?
- Rappel : la logique de SQL n'utilise pas seulement vrai et faux mais aussi « je ne sais pas », représenté par NULL
- Utile à savoir pour les sous interrogations qui renvoient NULL pour une des lignes

Sous-interrogations ; optimisation

- Soit un select qui comporte une sous-interrogation :

```
select nom from emp
where dept in
  (select dept from dept
   where lieu = 'NICE')
```

- Pour chaque employé, le select peut lancer la sous-interrogation pour savoir si l'employé est dans un département qui se trouve à Nice
- En fait, le moteur de recherche du SGBD va optimiser en lançant d'abord la sous-interrogation et en conservant en mémoire les départements de Nice

Sous-interrogations synchronisées

- Cette optimisation n'est pas possible quand la sous-interrogation utilise une des valeurs ramenées par l'interrogation principale
- On dit que la sous-interrogation est synchronisée avec l'interrogation principale
- Notation pointée utilisée pour se référer dans la sous-interrogation à une colonne de l'interrogation principale :

```
select nomE from emp E
where dept != (select dept from emp
               where matr = E.sup);
```

Considéré comme une constante par la sous-interrogation

Sous-interrogation ramenant 1 ligne de plusieurs colonnes

- **`WHERE (`*`expr1, expr2,…`*`) op (SELECT …)`**
  où *op* est = ou **`!=`** (mais pas **`<`**, **`>`**, **`<=`**, **`>=`**)

le select renvoie 1 seule ligne

- Exemple :

```
select nomE from emp
  where (poste, sal) =
    (select poste, sal from emp
     where nomE = 'Mercier');
```


Relation d'ordre

- Il n'y a pas de relation d'ordre totale « naturelle » sur les tuples
- 5 > 3, 12 > 7
- On ne peut comparer (5, 7) et (3, 12)

Sous-interrogation ramenant plusieurs lignes de plusieurs colonnes

- **`WHERE (`*`expr1, expr2,…`*`) op ANY (SELECT …)`**
  `WHERE (`*`expr1, expr2,…`*`) op ALL (SELECT …)`
  `WHERE (`*`expr1, expr2,…`*`) IN (SELECT …)`
  `WHERE (`*`expr1, expr2,…`*`) NOT IN (SELECT …)`
  où *op* est = ou **`!=`** (mais pas **`<`**, **`>`**, **`<=`**, **`>=`**)
- Exemple :

```
select nomE from emp
  where (poste, sal) in
    (select poste, sal from emp
     where dept = 10);
```


EXISTS

- La clause « EXISTS(select …) » est vraie si le select renvoie au moins une ligne
- La sous-interrogation est le plus souvent synchronisée :

```
select nomE from emp E
  where exists
    (select null from emp
     where sup = E.matr);
```

synchronisation

Division

R

A	B
x	1
y	2
z	1
x	3

S

C
1
3

R ÷B S

A
x

- **R ÷B S** = les A qui sont associés à tous les C
  = {a ∈ R[A] / ∀c ∈ S, (a, c) ∈ R}
  = {a ∈ R[A] / ∄ c ∈ S, (a, c) ∉ R}

Division avec NOT EXISTS

- R ÷B S = { a ∈ R[A] / ∄ c ∈ S, (a, c) ∉ R }
- Une traduction « mot à mot » en SQL donne :

```
select A from R R1
where not exists
  (select C from S
   where not exists
     (select A, B from R
      where A = R1.A and B = S.C)
```


Avec le dessin

R

A	B
x	1
y	2
z	1
x	3

S

C
1
3

R ÷B S

A
x

-
```
select A from R R1
  where not exists
    (select C from S
     where not exists
       (select A, B from R
        where A = R1.A and B = S.C)
```


Exemple

- Afficher la liste des numéros des départements qui ont tous les postes
- La solution est R ÷poste S où

  R = EMP [dept, poste] S = EMP [poste]
-
```
select distinct dept from emp E
 where not exists
    (select poste from emp E2
     where not exists
       (select dept, poste from emp
        where dept = E.dept
          and poste = E2.poste)
```


Avec le dessin

Dept | Poste

Poste

Dept

- R = EMP [dept, poste] S = EMP [poste]
-
```
select distinct dept from emp E
  where not exists
    (select poste from emp E2
     where not exists
       (select dept, poste from emp
        where dept = E.dept
          and poste = E2.poste)
```

Possible de simplifier

Simplification

- La traduction mot à mot peut souvent être simplifiée :

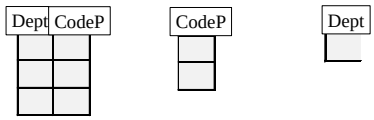
```
select dept from dept
 where not exists
    (select poste from emp E2
     where not exists
       (select dept, poste from emp
        where dept = dept.dept
          and poste = E2.poste)
```

Exemple avec jointure

- Quand la table dividende correspond à une jointure, on peut souvent simplifier le select le plus externe

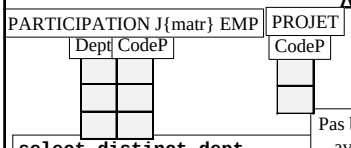
Exemple

- Donnez les numéros des départements qui ont participé à tous les projets



- R ÷ S où
  R = (PARTICIPATION J{matr} EMP) [dept, codeP]
  S = PROJET [codeP]

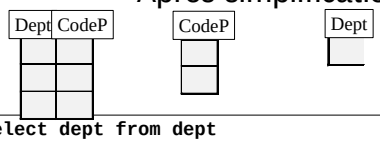
Avant simplification



Pas besoin de faire une jointure pour avoir un numéro de département

```
select distinct dept
from participation natural join emp E
  where not exists
    (select codep from projet
     where not exists
       (select dept, codep
        from participation natural join emp
        where dept = E.dept
          and codep = projet.codep))
```


Après simplification




```
select dept from dept
  where
    not exists
      (select codep from projet
       where not exists
       (select dept, codep
        from participation natural join emp
        where dept = dept.dept
          and codep = projet.codep))
```

Mais ici la jointure est indispensable

Division en comptant

- On peut obtenir le même résultat en comptant le nombre de valeurs distinctes (exemple à venir)

Fonctions de groupe

- Les fonctions de groupe peuvent apparaître dans une expression du select ou du having :

AVG	moyenne
SUM	somme
MIN	plus petite valeur
MAX	plus grande valeur
VARIANCE	variance
STDDEV	écart type
COUNT(*)	nombre de lignes
COUNT(col)	nombre de valeurs non NULL dans la colonne
COUNT(DISTINCT col)	nombre de valeurs distinctes

Exemples

```
select count(*) from emp;
select count(comm) from emp
  where dept = 10;
select sum(sal) from emp
  where dept = 10;
select max(sal) from emp
  where poste = 'INGENIEUR';

select nome, sal
from emp
where sal = max(sal);
```

Interdit, car `dept` et `max(sal)` ne sont pas au même niveau de regroupement

Niveaux de regroupement

- A un niveau de profondeur (relativement aux sous-interrogations) d'un SELECT, les fonctions de groupe et les colonnes doivent être toutes du même niveau de regroupement
- Par exemple, si on veut le nom et le salaire des employés qui gagnent le plus dans l'entreprise, la requête suivante provoquera une erreur :

```
select nome, sal from emp
  where sal = max(sal)
```

- Solution :

```
select nome, sal from emp
  where sal = (select max(sal) from emp)
```


Clause GROUP BY

- Permet de regrouper des lignes qui ont les mêmes valeurs pour des expressions :

```
GROUP BY expression1, expression2,…
```

- Il n'est affiché qu'une seule ligne par regroupement de lignes
- Exemple :

```
select dept, count(*)
from emp
group by dept;
```


Exemples

```
select dept, poste, count(*) from emp
  group by dept, poste;
select dept, count(comm) from emp
  group by dept;
```

```
select nome, dept, sal from emp
where (dept, sal) in
  (select dept, max(sal) from emp
   group by dept);
```

Schéma à retenir pour obtenir des optima sur des regroupements

Sans doute moins performant :

```
select nome, dept, sal from emp E
where sal = (select max(sal) from emp
             where dept = E.dept);
```


Exemples (2)

```
select nomD, avg(sal)
  from emp natural join dept
  group by nomD;
select dept, count(*) from emp
  where poste = 'SECRETAIRE'
  group by dept;
select count(*) "Nbre employés",
       count(comm) "Nbre commissions",
       count(comm) / count(*) "Ratio"
  from emp;
```


Restrictions pour les expressions

- Dans la liste des expressions du select ne peuvent figurer que des caractéristiques liées aux groupes :
  - des fonctions de groupes
  - des expressions figurant dans le GROUP BY
-

```
select dept, nomE, sal
from emp
group by dept;
```

Exemple

- ~~`select nomd, sum(sal)`~~
  `from emp natural join dept`
  ~~`group by dept.dept`~~
- Interdit ; il aurait fallu écrire :

```
select nomd, sum(sal)
from emp natural join dept
group by nomd
```


Clause HAVING

- Cette clause sert à sélectionner les groupes :
  HAVING *prédicat*
- Le prédicat ne peut porter que sur des caractéristiques de groupe

Exemples

```
select dept, count(*) from emp
  where poste = 'Secrétaire'
  group by dept
  having count(*) > 1;

select nomd "Département",
       count(*) "Nombre de secrétaires"
  from emp natural join dept
  where poste = 'Secrétaire'
  group by nomd
  having count(*) = (select max(count(*)) from emp
                     where poste = 'Secrétaire'
                     group by dept);
```


Exemples (2)

```
select dept, count(*) from emp
  group by dept
  having count(*) = max(count(*));
```

Interdit ! car pas le même niveau de regroupement que le reste du select

Exemples (2)

```
select dept, count(*) from emp
  group by dept
  having count(*) = (select max(count(*)) from emp
                     group by dept);
```


La division en comptant

- Lorsque la colonne « B » de la table dividende ne peut contenir que des valeurs de la colonne « C » de la table diviseur, on peut obtenir le quotient en comptant le nombre de valeurs distinctes :
- ```
  select dept
  from emp natural join participation
  group by dept
  having count(distinct codeP) =
    (select count(codeP)
     from projet)
  ```
  ```

Exemple

- 10 P1 P1
  10 P2 P2
  10 P1
  20 P1
- 10 est associé à 2 projets distincts
  20 est associé à 1 seul projet
  et il y a 2 projets
- ```
  select dept
  from emp natural join participation
  group by dept
  having count(distinct codeP) =
    (select count(codeP)
     from projet)
  ```


Contre-exemple

- Pas vrai si la table dividende peut contenir des valeurs qui ne sont pas dans la table diviseur
- Dans l'exemple suivant, le département 20 aurait été faussement sélectionné :
  10 P1 P1
  10 P2 P2
  20 P1
  20 P3

La division en comptant

- On pourrait tout de même s'en sortir par une requête un peu plus complexe :
- ```
  select dept
  from emp natural join participation
  where codeP in
    (select codeP from projet)
  group by dept
  having count(distinct codeP) =
    (select count(codeP)
     from projet)
  ```


Clause ORDER BY

- La clause ORDER BY précise l'ordre des lignes d'un SELECT :
  ORDER BY *expr1* [DESC], *expr2* [DESC], …
- On peut aussi donner le numéro de la colonne qui servira de clé de tri
- Exemples :
  ```
  select dept, nomD from dept order by nomD;
  select dept, nomD from dept order by 2;
  ```


Exemples

```
select nome, poste from emp
  order by dept, sal desc;

select dept, sum(sal) "Total salaires" from emp
  group by dept
  order by 2;

select dept, sum(sal) "Total salaires" from emp
  group by dept
  order by sum(sal);
```


Fonctions

- Fonctions arithmétiques :
  abs(n), mod(m, n), power(n, e), round(n, p), trunc(n, p), sign(n), sqrt(n), greatest(n1, n2,…), least(n1, n2,…)
- Conversions nombre - chaîne de caractères :
  to_char(n, format), number(chaîne)
- Fonctions date :
  round(date, précision), trunc(date, précision), sysdate
- Conversions date - chaîne de caractères :
  to_char(date, format), to_date(chaîne, format)

Fonctions (2)

- Fonctions sur les chaînes de caractères :
  length(chaîne), substr(chaîne, position, longueur),
  instr(chaîne, sous-chaîne, position, n),
  upper(chaîne), lower(chaîne),
  lpad(chaîne, long, car), rpad(chaîne, long, car),
  ltrim(chaîne, car), rtrim(chaîne, car),
  translate(chaîne, lesCar1, lesCar2),
  replace(chaîne, ancienne, nouvelle)

Exemple

```
select nomE, to_char(datemb, 'DD/MM/YYYY')
from emp
where round(sysdate – datemb) > 3;
```


« Fonction » de choix

- 2 variantes :
- CASE
  WHEN *condition1* THEN *resultat1*
  WHEN *condition2* THEN *resultat2*
  ELSE *resultat3*
  END
- CASE *expression*
  WHEN *valeur1* THEN *resultat1*
  WHEN *valeur2* THEN *resultat2*
  ELSE *resultat3*
  END

Exemples

```
SELECT nome, poste FROM emp
order by
  CASE poste
    WHEN 'Président' THEN 1
    WHEN 'Directeur' THEN 2
    ELSE 3
  END;
SELECT nome, poste FROM emp
order by
  CASE
    WHEN poste = 'Président' THEN 1
    WHEN poste = 'Directeur' THEN 2
    ELSE 3
  END;
```


Opérateurs ensemblistes

- On peut effectuer des opérations sur plusieurs select considérés comme des ensembles de lignes :
  select … UNION select …
  select … INTERSECT select …
  select … MINUS select …
- Ensembles au sens mathématique : si 2 lignes des 2 select d'une union ont les mêmes valeurs, l'union n'aura qu'une seule ligne avec ces valeurs

Exemple d'opérateur ensembliste

```
select dept from dept
minus
select dept from emp;
```

Exemple d'opérateur ensembliste

```
select nomE, 'salaire' TYPE, sal MONTANT
  from emp
union
select nomE, 'commission', comm
  from emp
  where comm is not null;

nomE         TYPE         MONTANT
Toto         salaire        1200
Bibi         salaire        5000
Toto         commission      240
```


UNION ALL

- Pour conserver les doublons avec UNION

Opérateurs ensembliste et order by

- Seul le dernier select peut recevoir une clause order by

Limiter le nombre de lignes

- Problème de portabilité si on veut ne faire afficher qu'une partie des lignes récupérées par un select

Exemple

- MySQL et Postgresql :

```
SELECT matr, nomE
FROM emp
LIMIT 10
```

- Oracle :

```
SELECT matr, nomE
FROM emp
WHERE ROWNUM <= 10
```

- SQL Server :

```
SELECT TOP 10 matr, nomE
FROM emp
```


Difficulté avec Oracle

- Si le select contient un order by : rownum numérote avant le tri, alors que top et limit numérotent après le tri
- Avec Oracle, il faut donc ruser : utiliser une sous-requête qui trie, alors que la requête principale limite le nombre de lignes :

```
select * from
  (select nomE, sal
   from emp
   order by sal)
where rownum <= 10
```

Langage de définition des données

Création de table par copie

- CREATE TABLE table (col1 type, …) AS select …
- **create table dept2**
  (cle integer, nom varchar(20))
  as select dept, nomd from dept;
- **create table dept10**
  as select * from emp
  where dept = 10;

Modifier la définition d'une table

- ALTER TABLE table
  ADD (col1 type1, col2 type2,…)
- ALTER TABLE table
  MODIFY (col1 type1, col2 type2,…)
- ALTER TABLE table
  DROP COLUMN colonne;

- On ne peut modifier une colonne que si la colonne ne contient que des valeurs null ou si la nouvelle définition est compatible avec les valeurs déjà entrées dans cette colonne

Exemples

- **alter table emp**
  add (situ_famille char(1),
  nbEnfants smallint);
- **alter table emp**
  modify (situ_famille char(2));

Supprimer une colonne

- Tous les SGBDs ne permettent pas de supprimer une colonne d'une table
- Si c'est impossible, on peut mettre toutes les valeurs de la colonne à null pour gagner de la place
- On peut aussi
  - utiliser create table as pour transférer les autres données dans une nouvelle table qui n'a pas cette colonne
  - supprimer la 1ère table
  - renommer la nouvelle table

Supprimer une colonne (Oracle 9i)

- ALTER TABLE emp DROP COLUMN comm n'est disponible sous Oracle que depuis la version 9i
- Il n'est pas possible de supprimer une colonne
  - référencée par une clé étrangère
  - sur laquelle un index a été construit
- Exemple :

```
alter table emp
drop column situ_famille;
```


Supprimer une table

- DROP TABLE table

Renommer une table

- On peut renommer une table :
  RENAME ancienNom TO nouveauNom
- Commande équivalente :
  ALTER TABLE ancienNom
  RENAME TO nouveauNom

Synonymes

- Si une table doit être utilisée par plusieurs utilisateurs, il peut être intéressant de lui donner un synonyme public :

```
CREATE PUBLIC SYNONYM employe
FOR toto.emp
```


Vues

Vues

- Une vue est une vision virtuelle partielle ou particulière des données d'une ou plusieurs tables
- La définition d'une vue est donnée par un select : les données de la vue sont celles retournées par le select
- Les utilisateurs peuvent consulter ou modifier la base à travers la vue comme si c'était une table réelle

Création et suppression d'une vue

- CREATE VIEW *vue* [(*col1*, *col2*,…)]
  AS select …
  [WITH CHECK OPTION]
- Le select peut contenir toutes les clauses d'un select sauf « order by »
- create view emp10 as
  select * from emp
  where dept = 10;
- DROP VIEW vue

Exemple de création de vues

```
create view
  deptStat (nom, inf, moy, max, total)
as
  select nomd,
         min(sal), avg(sal), max(sal), sum(sal)
  from emp natural join dept
  group by dept.nomd
```


Utilisation des vues dans un select

- Dans un select on peut utiliser une vue à la place d'une table
- **select * from emp10;**
- **select nom, total**
  from deptStat
  where total > 100000;

Suppression avec une vue

- On peut effectuer des delete à travers une vue, sous les conditions suivantes sur le select qui définit la vue :
  - une seule table
  - pas de group by
  - pas de distinct
  - pas de fonction de groupe

Modification avec une vue

- On peut effectuer des update à travers une vue, sous les conditions du delete, et en plus :
  - les colonnes modifiées sont des colonnes réelles de la table (pas des expressions)
- **update emp10**
  set sal = sal * 1.1;

Insertion avec une vue

- On peut effectuer des insert à travers une vue, sous les conditions du update, et en plus :
  - toute colonne « not null » de la table représentée par la vue est présente dans la vue
- **insert into emp10 (matr, nome, …)**
  values (1200, 'DUBOIS', …);

Option CHECK

- Si la vue a été créée avec "WITH CHECK OPTION", toute modification au travers de la vue ne peut donner des données qui ne seraient pas affichées par la vue
- Par exemple,

```
update emp10
set dept = 20;
```

sera interdit

Utilité des vues

- Les vues permettent de dissocier
  - la façon dont les utilisateurs voient les données
  - du découpage en tables
- On favorise ainsi l'indépendance entre les programmes et les données
- Si un programme utilise des vues, on peut, par exemple, remplacer une table par 2 tables sans modifier le programme de consultation des données ; il suffit de modifier la définition des vues

Utilité des vues (2)

- Peuvent simplifier la consultation de la base en enregistrant des select complexes
- Participent à la protection des données :
  - on peut donner accès à une vue, sans donner accès à la table sous-jacente
  - une vue peut ne donner accès qu'à certaines colonnes ou lignes d'une table
  - les modifications des données peuvent être restreintes avec la clause WITH CHECK OPTION

Index

Index

- Un index utilise des techniques informatiques pour rendre très rapides les accès aux valeurs d'une colonne
- `select * from emp where nomE = 'Dupond'` est très long si la table **emp** contient des millions de lignes
- Un index bien construit permet d'obtenir l'emplacement des informations sur Dupond en quelques accès disques (moins de 5, même s'il y a des millions de lignes dans la table EMP)

Création d'un index

- CREATE [UNIQUE] INDEX nomIndex ON table (col1, col2,…)
- `create index nomE on emp(nomE);`
- Le nom choisi doit être unique parmi tous les index de *toutes* les tables
- Oracle crée automatiquement un index sur les colonnes qui ont des contraintes Primary key et Unique

Utilisation d'un index

- Après sa création un index est géré automatiquement par le SGBD
- Il est transparent pour l'utilisateur : celui-ci interroge la base de la même façon que si l'index n'existait pas
- Le SGBD peut utiliser un index s'il pense que la requête sera accélérée
- Les index ralentissent les modifications des données

Suppression d'un index

- DROP INDEX nomIndex

Génération de clés

Utilité

- Les identifiants de lignes non significatifs sont préférables
- Le plus simple est d'avoir des clés qui sont des nombres entiers
- Le problème : générer des entiers sans que 2 lignes puissent avoir le même identifiant, même en situation de concurrence entre plusieurs transactions

Une mauvaise solution

- Prendre le plus grand nombre déjà utilisé dans la table comme identifiant et ajouter 1
- En pseudo code :

```
lock table;
val = select max(cle) from table;
insert into table
  values (val + 1,…);
commit;
```


Inconvénients de la solution 1

- Cette solution n'est pas performante :
  - nécessite un accès à la base
  - il faut trouver le plus grand
  - nécessite un blocage de la table pour éviter que 2 transactions n'obtiennent la même valeur
- Si on veut garder un historique, on peut se retrouver avec des lignes qui ont le même identificateur…
- … (si la ligne de plus grand identificateur est supprimée)

Solution 2

- Une table contient la prochaine clé à attribuer
- La valeur est incrémentée à chaque nouvelle clé

Variantes

- Variante 1 :
  Une seule table contient une seule valeur utilisée pour les identifiants de toutes les tables
- Variante 2 : Une table par clé
- Variante 3 :
  Une seule table qui contient une ligne par table qui a besoin d'une clé
  Quelle colonnes dans la table des clés ?

Pseudo-code de la variante 1

```
lock table_cle;
update table_cle
    set cle = cle + 1;
val = select cle from table_cle;
commit;

insert into table
    values (val,…);
commit;
```


Inconvénients de l'utilisation d'une table

- Cette solution n'est pas très performante :
  - nécessite un accès à la base (mais cette petite table est conservée en mémoire centrale dans le cache du SGBD)
  - il est nécessaire de bloquer l'accès à la valeur

Les séquences

- Les versions actuelles des SGBD offrent des solutions qui ne nécessitent pas d'accès aux données de la base
- Inconvénient : pas les mêmes solutions dans tous les SGBDs
- Les séquences sont disponibles avec Oracle, DB2 et PostgreSQL

Créer une séquence

- CREATE SEQUENCE *nom_séquence*
  [INCREMENT BY *entier1*]
  [START WITH *entier2*]
- **create sequence seqdept**
  increment by 10
  start with 10

Utilisation des séquences

- Deux pseudo-colonnes permettent d'utiliser les séquences :
  - CURRVAL retourne la valeur courante
  - NEXTVAL incrémente la séquence et retourne la nouvelle valeur
- **`insert into dept(dept, nomd)`**
  `values (seqdept.nextval, 'Finances')`

CURRVAL et NEXTVAL

- On ne peut utiliser CURRVAL qu'après avoir utilisé NEXTVAL au moins une fois dans la session de travail
- NEXTVAL modifie immédiatement la valeur future pour les autres transactions, même s'il est lancé dans une transaction non validée
- La valeur de CURRVAL ne dépend que des NEXTVAL lancés dans la même transaction

Modification des séquences

- ALTER SEQUENCE *nom_séquence*
  INCREMENT BY *entier1*
- **`alter sequence seqdept`**
  `increment by 5`
- On ne peut modifier la valeur de départ

Récupérer plusieurs identificateurs

- Mettre un incrément supérieur à 1 permet d'obtenir plusieurs identificateurs en un seul appel pour obtenir de meilleures performances
- Par exemple, si on veut des identificateurs pour des lignes de factures, on en a souvent besoin de plusieurs en même temps

Informations sur les séquences

- Afficher la valeur d'une séquence :
  `select seqdept.currval from dual`
- Tables du dictionnaire des données :
  `USER_SEQUENCES` et **`ALL_SEQUENCES`**

Autres solutions

- DB2 et SQL Server ont une clause « IDENTITY » pour dire qu'une colonne est un identifiant, avec une valeur qui est générée automatiquement par le SGBD
- La norme SQL 3 a normalisé cette possibilité
- Pas disponible sous Oracle
- MySQL permet de déclarer une colonne « AUTO_INCREMENT »

Comparaison séquences et autres solutions

- Les séquences sont souvent plus souples si on veut récupérer la valeur des identifiants pendant l'enregistrement des données
- Exemple : dans l'enregistrement d'une facture avec ses lignes de factures, on doit disposer de l'identifiant de la facture pour le mettre en clé étrangère dans les lignes de facture

Exemple

- facture(id, nom_client, date_facture,…)
- ligne_facture(id_facture, nb_ligne, quantite, id_article)

Procédures stockées

Procédures stockées

- Les SGBD modernes fonctionnent en client/serveur
- Chaque requête issue d'un client
  - transite sur le réseau
  - est « compilée » lorsqu'elle arrive au serveur : celui-ci recherche en particulier la meilleure façon de répondre à la requête
- La compilation peut être une étape complexe et longue à traiter et coûteuse en ressources
- Les procédures stockées sont des requêtes pré-compilées et stockées sur le serveur

3 étapes

1. Écriture du code de la procédure
2. Compilation et enregistrement sur le serveur ; on donne un nom à la procédure stockée
3. Exécution de la procédure en passant au serveur le nom de la procédure

Langages des procédures stockées

- Les procédures stockées peuvent inclure
  - des variables, des boucles et des tests
  - plusieurs requêtes SQL
- Le plus souvent, elles sont écrites dans un langage spécial
  - PL/SQL pour Oracle
  - PSM pour la norme SQL (PL/SQL lui ressemble mais n'est pas vraiment compatible)
  - Java pour les dernières versions d'Oracle

Avantages et inconvénients des procédures stockées

- Améliorent les performances et diminuent le trafic sur le réseau
- Encapsulent les processus métier (boîtes noires)
- Mais les langages d'écriture et les appels de ces procédures ne sont pas normalisés
- On perd donc de la portabilité si on utilise des procédures stockées

Exemple sous Oracle

```
create or replace procedure augmentation
  (unDept in integer, pourcentage in number,
   cout out number) is
begin
  select sum(sal) * pourcentage / 100
    into cout
    from emp
    where dept = unDept;
  update emp
    set sal = sal * (1 + pourcentage / 100)
    where dept = unDept;
end;
```


Utilisation d'une procédure stockée

- Depuis un langage de 3ème génération (C, Java,…) on les appelle presque comme des procédures ou des fonctions du langage
- Depuis SQL*PLUS le plus simple est de lancer **`execute`** <nom-procédure>

Paramètre « out » avec SQL*PLUS

- Pour récupérer un paramètre « out », appeler la procédure dans une autre procédure ou lancer l'exécution d'un bloc anonyme :

```
declare
  cout float;
begin
  augmentation(10, 5, cout);
  dbms_output.put_line('cout = ' || cout);
end;
```

lancer « **`set serveroutput on`** » pour permettre l'affichage sur l'écran

Supprimer une procédure

- DROP PROCEDURE nomProcédure

Compilation d'une procédure

- Une procédure stockée est compilée dès sa création
- Les erreurs éventuelles sont montrées par la commande
  SHOW ERRORS
- Si le schéma de la base ou la répartition des données a changé, il faut recompiler une procédure pour optimiser son exécution :
  ALTER PROCEDURE nomProcédure COMPILE

Informations sur les procédures

- Vues USER_PROCEDURES et USER_SOURCE du dictionnaire des données

Fonctions

- Comme les procédures stockées mais elles renvoient une valeur
- Elles peuvent être utilisées dans les requêtes SQL comme les fonctions prédéfinies

Exemple en PL/SQL

```
create or replace
  function euro_to_fr(somme in number)
  return number
is
  taux constant number := 6.55957;
begin
  return somme * taux;
end;
```


Exemple d'utilisation d'une fonction

```
select nome, sal, euro_to_fr(sal)
from emp
```


Triggers

Triggers (déclencheurs)

- Compilés et stockés sur le serveur
- Souvent écrits dans le même langage que les procédures stockées
- Leur exécution est déclenchée par des actions sur la base, par exemple une insertion dans une table
- Ils complètent les contraintes d'intégrité en permettant des contrôles et des traitements plus complexes
- Pas normalisés en SQL2 (normalisés en SQL3)

Exemple de trigger

```
CREATE OR REPLACE TRIGGER totalSalaire
  AFTER UPDATE OF salaire ON emp
  REFERENCING OLD AS ancien,
              NEW AS nouveau
  FOR EACH ROW
  update cumul
  set augmentation
      = augmentation + nouveau.salaire
        - ancien.salaire
   where matricule = ancien.matricule
```

Autre possibilité :
`for each statement`

Utilisation de :OLD et :NEW

```
CREATE OR REPLACE TRIGGER totalSalaire
  AFTER UPDATE OF salaire ON emp
  FOR EACH ROW
  update cumul
   set augmentation
      = augmentation + :NEW.salaire
        - :OLD.salaire
   where matricule = :OLD.matricule
```


Clause WHEN

```
create or replace trigger modifsalaire
before update of sal on emp
for each row
when (new.sal < old.sal)
begin
  raise_application_error(-20001,
    'Interdit de baisser le salaire ! ('
      || :old.nome || ')');
end;
```


- Afficher les erreurs de compilation :
  SHOW ERRORS
- Supprimer un trigger :
  DROP TRIGGER nomTrigger
- Tables du dictionnaire des données :
  USER_TRIGGERS, USER_TRIGGER_COLS

Ordres interdits

- Les ordres COMMIT et ROLLBACK sont interdits dans un trigger

Sécurité

Travail en sécurité

- Plusieurs utilisateurs peuvent travailler en toute sécurité sur la même base
- Les données peuvent être confidentielles ou partagées entre plusieurs utilisateurs

Contrôle de l'accès à la base

- Le contrôle de l'accès à la base est effectué en associant à chaque utilisateur
  - un nom de login
  - un mot de passe
- Chaque utilisateur a des privilèges d'accès à la base : droit ou non
  - de créer des tables ou des vues
  - de lire ou de modifier des tables ou des vues
  - ...

Rôles

- Pour faciliter la création de nouveaux utilisateurs avec certains droits, la plupart des SGBD permettent de créer des rôles
- Un rôle définit des droits et interdictions
- Quand on crée un nouvel utilisateur, on lui affecte un ou plusieurs rôles

Propriétaire des données

- Une table (et les données qu'elle contient) appartient à celui qui l'a créé
- Le propriétaire d'une table peut donner à d'autres le droit de travailler avec sa table
- Les vues permettent d'affiner les droits que l'on donne sur ses propres données : on peut donner des droits sur des vues et pas sur les tables sous-jacentes

Accorder des droits sur des objets

- GRANT privilège ON table/vue TO {utilisateur|rôle|PUBLIC} [WITH GRANT OPTION]

L'utilisateur qui reçoit le privilège pourra le donner à d'autres utilisateurs

- Des privilèges (il y en a d'autres) :
  - SELECT
  - INSERT
  - UPDATE [(col1, col2,…)]
  - DELETE
  - ALTER
  - ALL PRIVILEGES

Accorder des droits sur des actions

- GRANT privilège TO {utilisateur|rôle|PUBLIC} [WITH GRANT OPTION]
- Des privilèges (il y en a d'autres) :
  - CREATE INDEX
  - CREATE TABLE
  - ALL PRIVILEGES

Accorder des droits (exemples)

```
grant select on emp to clement;
grant select, update on emp
  to clement, chatel;
grant select on emp to public;
```

Changer son mot de passe :

```
grant connect to bibi
identified by motDePasse;
```


Reprendre les droits

- REVOKE privilège ON table/vue FROM utilisateur

Exemple de création de rôle

```
create role lmiage identified by
  lmiage;
grant create cluster,
  create procedure, create session,
  create sequence, create snapshot,
  create synonym, create table,
  create trigger,create view to lmiage;
```


Exemple de création d'utilisateur Oracle

```
CREATE USER toto
IDENTIFIED BY toto
DEFAULT TABLESPACE LINFO
TEMPORARY TABLESPACE USERTEMP
quota 5M on LINFO
profile default
/
GRANT LMIAGE TO toto
/
```


Injection de code SQL

- Tous les langages de programmation permettent de lancer des requêtes SQL pour interagir avec une base de données
- Il est alors possible de construire une requête SQL dont le texte contient une partie entrée par l'utilisateur (par concaténation de chaînes de caractères)
- Attention alors à l'injection de code SQL

Exemple d'injection de code SQL

- Un programme demande son nom et son mot de passe à un utilisateur, et les range dans 2 variables **nom** et **mdp**
- Il lance cette requête et accepte l'utilisateur si elle renvoie bien une ligne

```
"select * from utilisateur"
+ " where nom = '" + nom
+ "' and mdp = '" + mdp + "'"
```

- Quel est le problème ?

Le problème

- Un pirate sait qu'un des utilisateurs autorisés s'appelle Dupond
- Il saisit « **`Dupond' --`** » pour le nom et « **a** » pour le mot de passe
- La requête devient :

```
select * from utilisateur
where nom = 'Dupond' --' and mdp = 'a'
```

- Mais « -- » indique un commentaire avec le SGBD utilisé ; donc la requête exécutée sera :

```
select * from utilisateur
where nom = 'Dupond'
```


Les parades

- Toujours vérifier la saisie d'un utilisateur avant de s'en servir pour construire une requête SQL
- Pour l'exemple, il aurait suffit d'interdire le caractère « ' » ou de le doubler
- Les API fournies avec les langages pour accéder à une base de données offrent des facilités pour se protéger contre l'injection de code SQL
- Avec JDBC par exemple, il suffit d'utiliser des paramètres et **`PreparedStatement`** ; les caractères spéciaux comme « ' » sont alors traités comme des caractères ordinaires

Gestion des accès concurrents

- Un SGBD est prévu pour travailler avec de nombreux utilisateurs/transactions

Quelques questions

- A quel moment les modifications sont-elles vues par les autres transactions ?
- Que se passe-t-il lorsque plusieurs transactions veulent modifier les mêmes données ?
- Un ordre SQL (moyenne des salaires, par exemple) tient-il compte des modifications apportées par les autres transactions validées pendant son exécution ?

Problèmes liés aux accès concurrents

- Il peut se produire des pertes de données quand plusieurs processus veulent modifier les mêmes données en même temps
- Les principaux cas d'école sont :
  - mise à jour perdue
  - lecture inconsistante
  - lecture non reproductible
  - ligne fantôme

Mise à jour perdue

S
2500

Temps

Transaction T1	Transaction T2
s = Lire S	
	s = Lire S
s = s + 1000	
	s = s + 2000
Enregistrer s	
	Enregistrer s

Eviter les mises à jour perdues

S
3500

Temps

Transaction T1	Transaction T2
Bloquer S	
s = Lire S	
	s = Lire S ; attente...
s = s + 1000	
Enregistrer s	
Débloquer S	... s = Lire S
	s = s + 2000
	Enregistrer s

Problèmes liés aux blocages : interblocage

Temps

Transaction T1	Transaction T2
Bloquer A	
	Bloquer B
Bloquer B	
Attente...	
	Bloquer A
	Attente...

Lecture inconsistante

Temps

Transaction T1	Transaction T2
V = 100	
	v = Lire V (100)
ROLLBACK	
	Travaille avec v = 100

Ce cas n'arrive pas si les modifications ne sont visibles par les autres qu'après un COMMIT

Lecture non répétitive

Temps

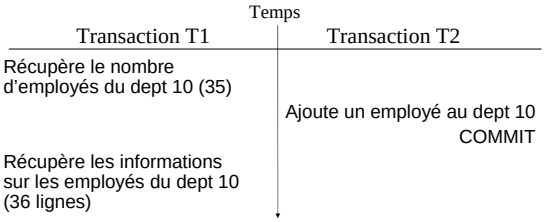
Transaction T1	Transaction T2
Lire V	
	V = V + 100
	COMMIT
Lire V	

Pour éviter cela, T1 devrait bloquer les données lues (V)

Lignes fantômes

- Ce problème survient quand une transaction n'a pas perçu la création d'une ligne par une autre transaction
- Par exemple, une transaction lit d'abord le nombre d'employés et lance ensuite la lecture d'informations sur chacun de ces employés
- Si une autre transaction a ajouté des employés, le nombre lu auparavant ne correspond plus à la réalité

Lignes fantômes



Pour éviter cela, T1 devrait bloquer la table des employés au début de la transaction (pas possible de bloquer des lignes qui n'existent pas !)

Transactions sérialisables

- On vient de voir que l'exécution de transactions entrelacées peut provoquer des pertes de données ou de cohérence
- Pour éviter ces problèmes, le SGBD doit s'arranger pour que l'exécution de plusieurs transactions entrelacées fournisse les mêmes résultats que l'exécution des mêmes transactions les unes à la suite des autres (dans un ordre quelconque)

Moyens de sérialiser

- Le moyen le plus courant s'appelle le verrouillage à 2 phases :
  - on doit bloquer un objet avant d'agir sur lui (lecture ou écriture)
  - on ne peut plus faire de blocage après avoir débloquer un objet
- On a donc 2 phases :
  1. acquisitions des verrous
  2. abandons des verrous (en pratique, souvent au COMMIT ou ROLLBACK)

Inter-blocage avec le verrouillage à 2 phases

- Les situations d'entrelacement des transactions qui auraient provoqué des problèmes vont se traduire alors par des attentes ou des inter-blocages (avec redémarrage de certaines transactions)

Estampillage

- L'estampillage est une autre stratégie que le verrouillage à 2 phases pour assurer la sérialisation des transactions
- L'ancienneté des transactions est repérée par une estampille donnée à la création de la transaction (un nombre incrémenté à chaque attribution)
- Chaque donnée accédée par une transaction reçoit l'estampille de cette transaction

Estampillage

- Les algorithmes qui utilisent l'estampillage assurent que l'exécution concurrente des transactions sera équivalente à l'exécution séquentielle de ces transactions dans l'ordre de leur estampille

Idée pour les algorithmes d'estampillage

- Une transaction T ne peut accéder à une donnée si cette donnée a déjà été accédée par une transaction plus jeune (donc d'estampille plus grande que celle de T)
- La transaction « trop vieille » T est annulée si elle s'est faite doubler par une transaction plus jeune
- Elle est relancée avec une estampille plus grande ; plus jeune, elle a plus de chance de pouvoir accéder à la donnée en passant « après »

Résultat de l'estampillage

- Cet algorithme est trop restrictif et ne va autoriser que peu d'exécutions parallèles des transactions
- La transaction redémarrée va peut-être entrer en conflit avec une autre transaction plus jeune
- D'autres algorithmes d'estampillage moins simplistes permettent d'améliorer le parallélisme, en particulier en distinguant les accès en lecture et les accès en écriture
- Malgré tout ce type de traitement est souvent trop restrictif et nuit à la concurrence

Autres traitements

- Pour améliorer les performances dans des situations de forte concurrence, les SGBD offrent la possibilité d'être plus permissif et de ne pas sérialiser les transactions

Niveaux d'isolation des transactions sous Oracle

- SET TRANSACTION ISOLATION LEVEL {SERIALIZABLE | READ COMMITED}
- SERIALIZABLE : les transactions s'exécutent totalement isolées des autres transactions et comme si elles s'exécutaient les unes après les autres

Niveaux d'isolation des transactions sous Oracle (2)

- READ COMMITED : les transactions ne voient les modifications des autres transactions qu'après les commit ;
  empêche les principaux problèmes de concurrence mais pas les lectures non répétitives ni les lignes fantômes

Autres niveaux d'isolation des transactions de SQL 2

- REPEATABLE READ : empêche les lectures non répétitives mais pas les lignes fantômes
- READ UNCOMMITED : les transactions voient les modifications avant même le commit

En résumé...

- Niveaux par isolation décroissante :
  - SERIALIZABLE (souhaitable, mais coûteux car provoque des blocages de tables)
  - REPEATABLE READ (sous Oracle, remplacé par les blocages explicites des lignes lues)
  - READ COMMITED (le plus souvent le niveau par défaut, comme avec Oracle)
  - READ UNCOMMITED (très rarement utilisé ; pas possible sous Oracle)

Au niveau d'un seul ordre SQL – Oracle

- Un instantané de la base est pris au début de chaque ordre SQL
- Cet instantané est utilisé pendant toute l'exécution de l'ordre,
  même si des données utilisées par l'ordre sont modifiées par une transaction validée avant la fin de l'exécution

Pratiquement...

- En fait, l'instantané n'est que virtuel
- Oracle s'arrange pour que l'ordre SQL voie les données comme elles étaient au début de l'exécution, en gardant plusieurs versions des données
- S'il s'aperçoit qu'une donnée a été modifiée depuis le début de l'ordre SQL, il utilise une ancienne version de la donnée

Traitement des accès concurrents par les SGBD

- Le but : favoriser au maximum les accès concurrents pour permettre l'exécution du plus grand nombre de transactions dans un temps donné (argument commercial important)
- Tous les SGBDs n'ont pas les mêmes solutions pour atteindre ce but

Durée des blocages

- Un blocage n'a lieu que le temps d'une transaction

Accès concurrents sous Oracle

- Oracle, par défaut, ne met aucun verrou pour effectuer une lecture
- Les lectures ne sont pas bloquées par les écritures, et vice-versa
- Pour cela, un historique des modifications est conservé (dans les segments de *rollback*)
- Une transaction est bloquée si elle veut modifier des données qui sont en cours de modification par une autre transaction
- La granularité minimum de blocage est la ligne

Autre façon de traiter les accès concurrents

- Beaucoup de SGBD bloquent les données lues
  - ce blocage n'interdit pas leur lecture par d'autres transactions mais interdit leur modification
- Une écriture bloque les données modifiées
  - elles ne peuvent être modifiées par d'autres transactions (comme avec Oracle)
  - mais elles ne peuvent pas non plus être lues par d'autres transactions

Blocages explicites et implicites

- Des blocages sont effectués implicitement par certaines commandes (en particulier UPDATE, INSERT et DELETE)
- Si le comportement par défaut du SGBD ne convient pas pour un traitement, on peut effectuer des blocages explicites des lignes ou des tables
- Les inter-blocages sont détectés par le SGBD qui annule un des ordres qui a provoqué l'inter-blocage

Granularité des blocages

- 2 types de blocages :
  - au niveau d'une table
  - au niveau d'une ligne
- Dans certains SGBDs le blocage d'une ligne implique le blocage de toute la page/bloc (plusieurs lignes) qui contient la ligne
- Certains SGBD transforment de trop nombreux blocages de lignes d'une table en un blocage de toute la table

Types de blocages

- Un blocage peut être
  - exclusif : seul celui qui a bloqué peut modifier ou bloquer les données bloquées
  - partagé : plusieurs transactions peuvent effectuer en même temps ce type de blocage

    C'est un blocage défensif qui marque les données pour interdire qu'une autre transaction ne les bloque dans un mode trop restrictif, ou ne les modifie

- Nous allons étudier quelques blocages dont disposent les utilisateurs d'Oracle
- Nous ne verrons pas tous les types de blocages fournis par Oracle

Blocages sur les tables

- LOCK TABLE table IN EXCLUSIVE MODE [NOWAIT]
  blocage complet de la table (à éviter si possible) ; on peut tout faire dans la table et les autres transactions ne peuvent que lire
- LOCK TABLE table IN SHARE MODE [NOWAIT]
  interdit aux autres transactions toute modification sur la table (sert pour faire des bilans) ;
  Autorise les autres transactions à bloquer elles aussi en mode SHARE

Blocages partagés sur les lignes

- SELECT colonnes FROM table
  WHERE condition
  FOR UPDATE OF colonnes
- Ce blocage permet de lire des valeurs, et en même temps de bloquer les lignes lues
- On peut ainsi travailler avec ces valeurs pour préparer leur modification
- Plusieurs transactions peuvent effectuer ce blocage sur des lignes différentes d'une même table

Blocages partagés sur les lignes

- SELECT FOR UPDATE n'interdit pas un blocage en mode partagé de la table mais les lignes bloquées ne pourront pas être modifiées ni bloquées par une autre transaction
- On peut modifier ensuite ces valeurs par UPDATE ou DELETE (après un éventuel déblocage d'un blocage partagé sur la table, effectué par une autre transaction)

Blocages implicites

- Les commandes INSERT, UPDATE et DELETE effectuent implicitement
  - un blocage exclusif des lignes modifiées
  - un blocage partagé sur la table ; ce blocage empêche un blocage de la table par une autre transaction, qui interdirait la modification des données

Lecture consistante pendant une transaction

- Si on veut travailler avec des valeurs inchangées pendant toute la durée de la transaction, on peut démarrer une transaction par :
  SET TRANSACTION READ ONLY
- On ne peut faire aucune modification pendant cette transaction ; les autres transactions peuvent faire ce qu'elles veulent mais leurs modifications ne seront pas visibles de la transaction qui a lancé cette commande
- Risque d'erreur si la transaction est trop longue et que les modifications des autres transactions sont trop nombreuses

Problèmes liés aux modifications concurrentes

- Situation typique : on souhaite
  1. lire une donnée D
  2. utiliser la valeur lue pour calculer une nouvelle valeur
  3. affecter à D la nouvelle valeur calculée
- Risque de modifications perdues si d'autres transactions ont modifié D entre l'étape 1 et l'étape 3

2 types de solutions

- Traitement pessimiste
- Traitement optimiste

Traitement pessimiste

- Pour éviter les problèmes liés aux accès concurrents, le comportement le plus simple est pessimiste :
  1. D est bloqué pour empêcher sa modification par d'autres transactions
  2. calcul de la nouvelle valeur
  3. affectation à D de la nouvelle valeur
  4. déblocage de D à la fin de la transaction

Inconvénients des blocages pessimistes

- Si l'utilisateur omet de valider rapidement la transaction, les blocages vont nuire aux autres transactions
- En règle générale, il faut essayer d'éviter le blocage pessimiste si un traitement interactif (donc long) doit être placé entre le début et la fin d'une transaction
- Des inter-blocages peuvent survenir avec d'autres transactions
- Coût d'un blocage pour les performances

Traitement optimiste

1. Lecture de D sans le bloquer, en espérant que d'autres transactions ne le modifieront pas
2. Calcul de la nouvelle valeur
3. Blocage court pendant lequel on vérifie si l'optimisme était justifié : D a t-il été modifié entre-temps par d'autres transactions ?
   si ça n'est pas le cas, on valide la transaction
4. sinon, on agit en conséquence ; par exemple en recommençant le traitement, ou en l'annulant

Vérification juste avant la validation (1)

- On peut comparer la valeur actuelle de D avec la valeur lue au début
- Mais si D est de grande taille, ça donne des mauvaises performances

Vérification juste avant la validation (2)

- On peut ajouter une information pour chaque ligne :
  - soit un « *timestamp* » qui indique le moment exact de la dernière modification de la ligne (peu fiable, surtout sur les bases distribuées)
  - soit un numéro de version de ligne qui est incrémenté à chaque modification de la ligne

Implémentation

- La comparaison peut se faire avec des « if » du langage hôte
- Souvent, on peut aussi utiliser ce type de update au moment où on effectue la modification (à la fin) :

```
n = update emp
      set salaire = nouveauSalaire
      where matr = 8000 and salaire = ancienSalaire
```

Si on récupère n = 0, l'optimisme n'était pas justifié

Avantages du traitement optimiste

- Plus de blocages longs
- Plus d'inter-blocages
- Moins coûteux en ressources
- Permet un granularité de « blocage » fine (au niveau d'un attribut et pas d'une ligne) en dehors du blocage final
- Dans certains cas, permet un traitement substitutif si l'optimisme n'était pas justifié

Inconvénients du blocage optimiste

- S'il y a beaucoup de « collisions » entre transactions, les annulations de traitements sont nombreux
- Ne convient pas si les annulations sont coûteuses (trop de modifications à annuler)
- Plus complexe à mettre en place que le blocage pessimiste

Traitement optimiste ou pessimiste ?

- Le choix dépend du contexte
- Pour les transactions qui comportent une intervention de l'utilisateur entre la lecture et la modification des données, il faut privilégier le traitement optimiste car le blocage pessimiste va bloquer des données pour un trop long moment
- S'il y a trop de risques de collisions entre transactions, on préférera souvent le blocage pessimiste pour éviter trop de calculs inutiles

Pratique des blocages pessimistes

- Avec Oracle un select simple ne bloque pas les données
- Si on veut effectuer des blocages pessimistes, on doit lire les données avec un « select … for update »
- On peut aussi plus rarement effectuer des blocages de table pour les tables dont on veut bloquer un grand nombre de lignes

Réponses à quelques questions

- A quel moment les modifications sont-elles vues par les autres transactions ?

  après la validation de la transaction (COMMIT)
- Que se passe-t-il lorsque plusieurs transactions veulent modifier les mêmes données ?

  blocages implicites du SGBD $\Rightarrow$ attente

Réponses à quelques questions

- Quand un ordre SQL tient compte de plusieurs lignes (moyenne des salaires, par exemple), cet ordre tient-il compte des modifications apportées par les autres transactions pendant l'exécution de l'ordre ?

  Ca dépend du SGBD. Avec Oracle, non