

# Bases de données

Le langage SQL et Oracle

[jd@ecomms.fr](mailto:jd@ecomms.fr)

Version du 8/10/07

# Gestion des dates

- Le type DATE intègre date et heure
  - Doit être formaté pour tout voir
  - Ex : `SELECT TO_CHAR(datejour,'DD/MM/YYYY')` as datejour FROM...
  - Heure système : `CURRENT_DATE`
- Pour les intervalles, on préférera un `TIMESTAMP`

# Génération automatisée de clés

- On peut utiliser la notion de séquence :

```
create sequence matable_seq  
start with 1  
increment by 1  
nomaxvalue;
```

- On utilise ensuite la séquence lors de l'insertion :

```
insert into matable values(matable_seq.nextval, 'blabli');
```

# Les jointures

- Permettent d'effectuer des requêtes entre plusieurs tables
- Élément donc indispensable...
- ...mais aussi source de ralentissements et divers dysfonctionnements

# Jointures “à la main”

- Ces jointures étaient les seules disponibles à l'époque des premiers SGBD
- Sont encore très utilisées
- Principe : on explicite dans chaque requête les liaisons entre clés primaires et clés secondaires

# Exemple de jointure manuelle

Utilisation d'alias

```
SELECT * FROM client c, facture f
WHERE c.nom='bob'
AND f.refcli=c.refcli
```

La jointure est explicitée

# Problèmes des jointures “manuelles”

- Incohérence : WHERE est fait pour restreindre, pas pour élargir !
- Syntaxe parfois lourde lorsqu'on enchaîne X jointures
  - Ex. des tables de nomenclature
- Certains cas sont difficiles à traiter
  - Ex : afficher tous les clients y compris ceux sans facture

# Les jointures normalisées

- Nouveauté de SQL2 (1992)
- Utilisation du mot-clé JOIN pour expliciter les jointures utilisées
  - Plus cohérent que dans WHERE
  - Plus efficaces (optimisations du SGBD)
  - Permet le traitement de cas particuliers

# Jointure interne

- C'est l'équivalent de la jointure manuelle
- Mais l'utilisation de JOIN facilite les optimisations et libère le WHERE
- Ex :  

```
SELECT * FROM client c  
INNER JOIN facture f  
ON f.refcli=c.refcli
```

# Jointure naturelle

- Syntaxe ultra simplifiée
- “Devine” la jointure si les clés ont les mêmes noms dans les deux tables
- Ex :  
`SELECT * FROM client c  
NATURAL JOIN facture f`

# Jointure naturelle explicitée

- Idem solution précédente
  - Mais on précise le nom du champ représentant la jointure avec *USING*
- Ex :

```
SELECT * FROM client c
NATURAL JOIN facture f
USING refcli
```

# Jointures externes

- Les jointures externes permettent de prendre en compte des données en dehors des jointures 'normales'
- Par ex : inclure les clients n'ayant pas de facture

# Right, Left, Full

- Doit on inclure tous les éléments de la table :
  - à gauche du mot clé JOIN

```
SELECT * FROM client c  
LEFT OUTER JOIN facture f  
ON f.refcli=c.refcli
```

On incluera les  
clients sans facture

# Right, Left, Full

- Doit on inclure tous les éléments de la table :
  - à droite du mot clé *JOIN* (factures sans client)
  - des deux côtés (*FULL OUTER JOIN*)

```
SELECT * FROM client c  
FULL OUTER JOIN facture f  
ON f.refcli=c.refcli
```

On inclut tous les clients et toutes les factures

# Sous requêtes

- Il est possible d'avoir des “poupées russes” de requêtes en SQL
- Utiliser le résultat d'une requête en tant que condition d'une autre requête
- Peut provoquer des problèmes de performances
- Parfois remplaçable par des jointures

# Sous requête renvoyant un seul résultat

- On peut l'intégrer dans le SELECT ou le WHERE (ou le HAVING)

- Ex :

```
SELECT * FROM xx
```

```
WHERE tarif >
```

```
(SELECT sum(*) FROM ...)
```

# Existence d'une sous requête

- On peut utiliser les mots clés :
  - EXISTS pour vérifier qu'une sous requête renvoie quelque chose
  - UNIQUE pour vérifier qu'un seul résultat est renvoyé
- Ex : **SELECT ...**  
**WHERE EXISTS(SELECT ...)**

# Sous requête renvoyant une liste de résultats

- On utilise les clauses IN (ou NOT IN), ANY, ou ALL
- Ex : **SELECT \* FROM xx**  
**WHERE nom IN**  
**(SELECT nom FROM ...)**
- Ex : **SELECT \* FROM xx**  
**GROUP BY ....**  
**HAVING sum(xx) > ANY**  
**(SELECT sum(xx) FROM ... GROUP BY...)**

# Groupements

- Ex : on veut faire la somme des clients par leur statut
- On utilise pour cela la clause GROUP BY :

```
SELECT sum(*) FROM client  
GROUP BY statut
```

# Conditions sur les données regroupées

- Ex : on veut les sommes des clients par statut, mais uniquement s'il y en a plus de 10 par statut
- On ne peut utiliser le WHERE, car WHERE regarde les données, pas les résultats du GROUP BY
- On utilise HAVING

# Exemple d'utilisation de HAVING

- `SELECT SUM(*) FROM client  
GROUP BY statut  
HAVING SUM(*) > 10`

# Vues sous Oracle

- Une vue est la possibilité de nommer une requête revenant fréquemment

- Exemple :

```
CREATE VIEW mavue  
AS SELECT .....
```

- Utilisation :

```
SELECT * FROM mavue;
```

# Vues paramétrées

- Les paramètres passés à la vue sont concaténés au SELECT d'origine
- Ex : **SELECT \* FROM mavue WHERE num=3**
- va rajouter num=3 aux paramètres du SELECT d'origine

# Triggers

- Oracle gère la notion de triggers :
  - Automates lancés suite à des actions particulières
  - Ex : décrémenter un compteur après un DELETE sur une table
  - Ex : mettre à jour un champ s'il est mis à jour dans une autre table

# Ex de trigger

```
CREATE TRIGGER declencheur_insertion
```

```
AFTER INSERT ON tbl_1
```

Condition

```
FOR EACH ROW
```

```
WHEN (NEW.col_a <= 10)
```

```
BEGIN
```

```
    INSERT INTO tbl_2 VALUES (:NEW.col_b, :NEW.col_a);
```

```
END;
```

Action a accomplir

# Transaction

- Une transaction est une suite d'instructions SQL (INSERT, UPDATE, DELETE...)
- Il est possible de revenir en arrière, c'est à dire d'annuler toutes les modifications au sein d'une transaction

# Validation d'une transaction

- Une transaction est validée avec l'instruction COMMIT
- Toutes les modifications sont validées et rendues disponibles pour les autres utilisateurs
- La transaction est fermée. Toutes les instructions ultérieures feront partie d'une autre transaction

# Annulation d'une transaction

- On annule une transaction avec l'instruction `ROLLBACK`
- On revient à l'état du début de transaction
- Toutes les modifications sont annulées
- On peut démarrer une nouvelle transaction

# Sectionnement d'une transaction

- Il est possible de découper une transaction en plusieurs étapes

UPDATE...

SAVEPOINT SI

INSERT...

ROLLBACK TO SI

L'INSERT est annulé, mais l'UPDATE reste d'actualité

# Transactions et verrous

- Les transactions impliquent la mise en place de verrous :
  - L'utilisateur U1 modifie une donnée
  - L'utilisateur U2 ne peut modifier cette donnée tant que U1 n'a pas validé

# Verrous explicites

- En plus des verrous posés par INSERT, UPDATE, DELETE, il est possible de définir “à la main” certains verrous :
- Instruction **SELECT FOR UPDATE** permet de lire une donnée et de garantir que personne n’ira la modifier avant son propre update
- *lock table XX in YY mode* verrouille manuellement une table

# Modes de verrous

- lock table XX in exclusive mode
  - Les autres transactions ne peuvent que lire la table
- lock table XX in exclusive mode nowait
  - Les transactions interdites ne sont pas bloquées
- lock table XX in share mode
  - Ne verrouille pas, mais interdit aux autres de verrouiller