CENTRE NATIONAL D'ÉTUDES SPATIALES

REFERENTIEL NORMATIF du CNES

RNC

Référence: **RNC-CNES-Q-HB-80-508**
Version 5
02 Juin 2008

MANUEL

ASSURANCE PRODUIT

REGLES POUR L'UTILISATION DU LANGAGE SQL

ACCORD du Bureau de Normalisation	**BN n° 34 du 25/06/07 – BN n°44 du 08/09/08**
APPROBATION Président du CDN Alain CUQUEL	

PAGE D'ANALYSE DOCUMENTAIRE

TITRE : **REGLES POUR L'UTILISATION DU LANGAGE SQL**
MOTS CLES : SQL, Bases de Données, Relationnel, Algèbre de CODD,Accès ensembliste, Langage, Requête, Interrogation, ANSI, ISO, Règles, Transaction.
NORME EQUIVALENTE : Néant
OBSERVATIONS : Néant
RESUME : Ce document complète un ensemble de règles fondamentales applicables pour toute utilisation du langage SQL. SQL peut être utilisé seul, ou intégré dans un environnement procédural. Il s'appuie sur la norme SQL 2 alias SQL 92. Les nouveautés introduites par SQL 3 alias SQL 1999 y sont également présentées.
SITUATION DU DOCUMENT : Ce document fait partie de la collection des Manuels approuvés du Référentiel Normatif du CNES (RNC). Il est affilié au document « RNC-ECSS-Q-ST-80 Software Product Assurance ».
NOMBRE DE PAGES : 96 — **LANGUE** : Française
Progiciels utilisés / version : Word 2002
SERVICE GESTIONNAIRE : Inspection Générale Direction de la Fonction Qualité (IGQ)
AUTEUR(S) : **DATE : 02/06/2008** **Repris par J-C. DAMERY**

PAGES DES MODIFICATIONS

VERSION	DATE	PAGES MODIFIEES	OBSERVATIONS
PR.1	20/08/93	Toutes pages créées	Version initiale CR2A complétée et validée par le Responsable.
PR.2	10/09/93	Toutes pages modifiées	Numérotation automatique des règles et ajout de l'attribut d'applicabilité.
PR.3	07/12/93	Toutes pages remplacées	Prise en compte des remarques émises en interne CNES (CT/AQ/QL et CT/TI)
1.0	21/04/94	i.1, i.2, 1	Approbation du Comité Technique Référentiel et du Comité de Validation
2	01/03/00	Toutes	Nouvelle codification des documents
3	20/01/05	Toutes	Suite à la FEB 13/2004 acceptée au BN n°7 du 04/10/04. Renommage des règles. Introduction de règles relatives à SQL3. Description et justification des règles. Mise en forme et répartition des règles. Avec le support de CS SI.
4	24/01/2007	Toutes	Mise en forme, avec le support de T. Leydier (Virtualité Réelle). Modification du titre. Cf. FEB 48/2006 acceptée au BN n° 22 du 06/03/06. Document accepté au BN n° 34 du 25/06/07 pour introduction dans le RNC.
5	02/06/2008	Toutes	Changement de nomenclature suite à la phase de benchmarking ECSS (ancienne référence RNC-CNES-Q-80-508).

TABLE DES MATIERES

1. INTRODUCTION

Le document « Règles pour l'utilisation du Langage SQL » est rattaché à la norme RNC-ECSS-Q-ST-80 « Software Product Assurance ». Il décrit les règles applicables pour un logiciel utilisant le langage SQL.

2. OBJET

Ce document décrit les règles pour l'utilisation du langage SQL.
Elle ne liste que les règles indispensables à appliquer, pour contribuer à obtenir un bon niveau de qualité sur les logiciels développés selon ce langage.
Ce document n'est pas un manuel de référence du langage SQL. Il nécessite, pour être utilisé, la connaissance et la maîtrise du langage. Le présent document s'appuie sur la norme ISO/IEC SQL2 ([DA1]).
Les nouveautés introduites par la norme SQL3 ([DA2]) sont également présentées et clairement identifiées.
Le présent document a été rédigé indépendamment de toute implémentation particulière de SQL: il ne s'appuie que sur la norme SQL2 ([DA1]) et SQL3 ([DA2]). Toutefois, les exemples illustrant les règles - bien que le plus souvent génériques - s'appuient sur une syntaxe ORACLE.
Notons également que parmi les règles SQL3, certaines notions qui sont abordées dans ce doument ne sont pas forcément implémentées par tous les SGBD du marché.

3. DOMAINE D'APPLICATION

Ce document est applicable au développement et à la maintenance des systèmes informatiques sol pour leurs parties réalisées en SQL.
Toutes les règles présentes dans ce document peuvent être sélectionnées. Elles ne sont pas contradictoires entre elles, mais peuvent en revanche contrarier certaines pratiques de codage visant à obtenir une performance maximale, ou un temps de codage minimum.
Les responsables d'un projet peuvent ajouter des règles à ce document à la condition qu'elles ne remettent pas en cause les règles existantes. Ils peuvent également restreindre le nombre de règles applicables (i.e. impératives) par une directive d'application du présent document, ou définir des cas de dérogation.
Certaines règles nécessitent une adaptation spécifique au Projet. Elles sont citées ci-après: [1]

CO.Présentation.Requête
CO.Présentation.CaractèresInterdits
CO.Schéma.Profils

[1] Attention: cette liste peut ne pas être exhaustive. Un passage en revue des règles est nécessaire pour déterminer celles qui peuvent nécessiter une adaptation de la part du Projet.

4. DOCUMENTS

4.1. DOCUMENTS DE REFERENCE

DR	Identification	Titre
(DR1)	RNC-ECSS-Q-ST-80	Software Product Assurance
(DR2)		SQL Language Reference Manual - ORACLE7 Server - déc 92
(DR3)		Administrator's Guide - ORACLE7 Server - déc 92
(DR4)		Programmer's guide to Oracle Precompilers - Version 1.5 - déc 92
(DR5)	Z 67-133=ISO/CEI 9126. 10/1992	AFNOR Traitement de l'information -Evaluation des produits logiciels
(DR6)	ISO/CEI 10026	Technologie de l'information - Interconnection de systèmes ouverts - Traitement transactionnel réparti. Partie N° 1 : Modèle OSI-TP

4.2. DOCUMENTS APPLICABLES

DA	Identification	Titre
(DA1)	NF ISO/CEI 9075	Technologie de l'information – Langage de base de données SQL

4.3. AUTRES DOCUMENTS

DA	Titre
(DR7)	*SQL Performance Tuning* (Peter Gulutzan, Trudy Pelze – Addison Wesley)
(DR8)	http://www.ncb.ernet.in/education/modules/dbms/SQL99
(DR9)	http://www.dba-oracle.com/art_oracle_obj.htm
(DR10)	http://sqlpro.developpez.com/SQL_AZ_991.html

5. TERMINOLOGIE

5.1. GLOSSAIRE

Terme	Définition
Accélérateur	Objet permettant d'augmenter les performances d'accès aux données.
Binary Large Object (blob)	Contenu d'un champ d'une table qui correspond à une image binaire stockée "en l'état" (chaîne de bits dont la structure interne est volontairement ignorée du dictionnaire.
Clé primaire	Attribut (colonne) utilisé pour identifier une et une seule occurrence d'une table.
Clé étrangère	Attribut (colonne) identique à une clé primaire d'une autre table.
Cluster	Accélérateur jouant sur l'organisation des tables: on rassemble dans une même page disque des tables ou éléments de table accédés simultanément par les transactions que l'on désire accélérer.
Colonne	Attribut d'une table.
Colonne qualifiée	Colonne dont l'appartenance à une table est spécifiée (client.nom).
Commande	Désignation large: Ordre SQL cohérent adressé au SGBD.
Commandes de contrôle de session	Ensemble de commandes SQL permettant de définir le profil d'un utilisateur au cours d'une session SQL.
Commandes de contrôle de transaction	Ensemble de commandes permettant d'implémenter la notion de transaction en conformité avec la norme OSI TP (CF [DR6])
Contrainte	Restriction de l'ensemble des valeurs possibles d'une colonne.
Critère	Condition sur une colonne.
Expression	Une expression est une combinaison de variables, de constantes et de fonctions au moyen d'opérateurs. On distingue globalement trois types d'expression SQL correspondant chacune à un type de données: - Les expressions arithmétiques, les expressions chaînes de caractères, et les expressions de dates.
Fonction de groupe	Fonction s'appliquant à plusieurs occurrences.
Index	Accélérateur SQL.

Terme	Définition
Interpréteur	Analyseur des commandes SQL, parfois nommé 'parseur'.
Jointure	Mécanisme relationnel permettant d'accéder en une seule requête à des colonnes appartenant à plusieurs tables.
Langage de définition des données	Ensemble de commandes permettant de créer tous les objets de la base (tables, vues, index, clusters, ...).
Langage de manipulation des données	Ensemble de commandes permettant l'utilisation de la base, pour un accès en lecture ou en création/modification/destruction de données.
Locator	Pointeur sur une donnée de type Character Large Object Binary Large Object Binary.
NULL	La valeur NULL signifie 'indéterminé' ou 'sans objet': elle s'oppose aux attributs spécifiés 'NOT NULL', où une valeur est toujours exigée en INSERT ou UPDATE.
Occurrence	Ensemble de valeurs identifiant une ligne d'une table.
Optimiseur	Analyseur effectuant le choix du chemin d'accès le moins coûteux pour l'exécution d'une requête.
PRO*Langage	Interprétation de l'anglais 'Embedded SQL', qui désigne le sous-ensemble de SQL permettant une utilisation intégrée au sein d'un langage algorithmique de "troisième" génération (par exemple: C, Ada, Pascal, Fortran, PL1 etc..)
Requête	Commande SQL de sélection, commençant par SELECT.
Script SQL	Procédure ou ensemble d'ordres SQL, visualisable et éditable, constituant une unité de traitement.
Sous-interrogation	Interrogation secondaire dont le résultat permet de réaliser une interrogation principale..
Table	Objet de la base de données contenant les données.
Trigger	Mécanisme de déclenchement automatique d'une procédure stockée lors de l'arrivée d'un événement sur une colonne ouune table (création, suppression, modification)
Type Abstrait	Type dont la structure est entièrement défini par la développeur et permettant de typer des variables PSM ou des colonnes d'une table.
Vue	Table virtuelle, nommée, dérivée d'une ou plusieurs tables, qui n'est pas rémanente mais dont la définition- exprimée par la clause SELECT- est rémanente.

5.2. ABREVIATIONS

Abréviation	Définition
BD	Base de Données.
BLOB	Binary Large Object.
CCT	Commandes de contrôle de transaction.
CLI	Call-Level Interface
LDD	Langage de définition des données.
LMD	Langage de manipulation (modification + interrogation) des données.
LOB	Large OBject
L3G	Langage de troisième génération (C, Fortran ..).
SGBD	Système de gestion de base de données.
PL/SQL	Nom donné par Oracle au PSM (cf. ci-dessous)
PSM	Langage procédural de haut niveau dont les programmes sont directement stockés dans la base et permettant l'interaction avec la base de données relationnelle (Persistent Stored Module).
SQL	Langage d'interaction avec une base de données relationnelle (Structured Query language).
SQLCA	Structure de communication SQL.
UDT	User Data Type

6. ADEQUATION EVENTUELLE AUX NORMES ET STANDARDS

Les règles présentées dans ce document sont conformes au langage SQL défini dans le document DA1.

7. REGLES

Organisation des règles

L'organisation des règles n'est pas conforme à l'organisation standard utilisée dans les autres langages.
SQL est un langage permettant d'interagir avec une base de données relationnelle.
Le champ d'action de SQL peut être divisé en 2 espaces fonctionnels distincts :

- L'espace des commandes d'administration et de gestion de tous les objets de la BD.
- L'espace des commandes d'utilisation simple de la BD.

Du point de vue **"valeur d'usage"**, seul le deuxième espace est intéressant. Le premier n'existe que pour permettre le fonctionnement sûr et répétitif des fonctions d'utilisation de la base de données par ses exploitants (utilisateurs en interactif et chaînes batch).
Pour cette raison, l'espace d'administration et de gestion est confié à une et une seule personne morale: l'administrateur. En fonction de contraintes diverses échappant à la compétence de ce document, la personne morale Administrateur peut correspondre à une ou plusieurs personnes physiques, mais jamais aucune.
Les utilisateurs sont, eux, en nombre variable. Ils disposent de droits plus ou moins étendus sur les tables applicatives, mais ne disposent d'aucun droit sur les tables du dictionnaire de données, hormis leur consultation, qui elle même ne doit pas être encouragée (une règle va dans ce sens).
Le présent document divise donc les règles en 2 principales rubriques :

- Règles portant sur les commandes SQL d'administration de la Base de Données.
- Règles portant sur les commandes SQL d'utilisation de la Base de Données.

En préalable à ces deux rubriques, le lecteur trouvera un ensemble limité de règles de bon sens, communes aux deux espaces d'utilisation de SQL: les règles générales de mise en oeuvre de SQL.
Ce document présentant certaines règles relatives à la norme SQL3 (clairement identifiées par « ***Règle spécifique SQL 3 / 1999 »***), les projets mettant en œuvre un SGBD n'intégrant pas ces fonctionnalités se limiteront aux règles SQL2.
Notons enfin que les règles peuvent s'appliquer dès les phases de conception préliminaire, détaillée ou lors du codage. La table récapitulative en fin de document ordonne ces règles par phase.

7.1. REGLES GENERALES RELATIVES A LA MISE EN OEUVRE DE SQL

7.1.1. L'accès à SQL

CP.SQL.Accès	L'accès direct à SQL par les utilisateurs finaux de la base doit être évité.

Description

Pour effectuer son travail, l'utilisateur de l'application ne doit jamais être en contact direct avec SQL. Il doit toujours avoir à sa disposition une interface fonctionnelle (commandes ou environnement) masquant et encapsulant les commandes SQL qui permettent d'extraire de la base (ou placer dans la base) les informations qu'il manipule. Cette interface est un filtre permettant de n'utiliser de SQL que ce qui est nécessaire, et un optimiseur offrant la garantie que SQL est utilisé de manière convenable.

Justification

SQL est un langage riche, complexe et très puissant, et tend à devenir de plus en plus complet. L'utilisation directe de SQL ne peut être sûre et efficace que si l'utilisateur maîtrise parfaitement les règles du langage et les principes de fonctionnement du moteur SQL.

Exemple

```
Des écrans SQL*FORMS constituent un environnement masquant l'existence de SQL aux
utilisateurs.
```

CD.SQL.Requête	Les requêtes SQL sont spécifiées par les concepteurs qui en ont le besoin. Elles sont écrites par un spécialiste de SQL. Elles sont optimisées par un spécialiste du SGBD.

Description

Le concepteur de l'application spécifie ce qui doit être fait par SQL. Le spécialiste SQL a la responsabilité de définir l'ensemble des requêtes SQL pour tous les concepteurs. Le spécialiste du SGBD se prononce sur l'adéquation du SQL obtenu par rapport au fonctionnement spécifique du moteur du SGBD.

On ne vise ici que les requêtes, transactions, procédures etc. qui sont critiques pour le système (fréquentes, complexes, ou pénalisantes sur le plan des accès).

Justification

Les requêtes sont plus performantes, et leur style est homogène. La structure du code est améliorée car le spécialiste peut centraliser l'ensemble des besoins en requêtes, et regrouper des requêtes similaires.

Le fonctionnement en mode Client-Serveur est amélioré, car le spécialiste SQL peut déterminer avec complétude ce qui est du domaine du client et ce qui est du domaine du serveur. La performance de traitement est garantie, car le spécialiste du SGBD, disposant de scripts clairs, bien écrits, et déjà optimisés sur le plan du fonctionnement de SQL "pur", est capable de valider ces scripts dans de bonnes conditions.

7.1.2. Soumission de requêtes SQL

CD.SQL.Soumission	Eviter la soumission d'ordres LDD en mode conversationnel. Toujours passer par des fichiers sources.

Description

En mode conversationnel, l'ordre SQL envoyé au noyau est saisi dynamiquement à l'écran; si l'ordre comporte la moindre erreur de syntaxe, il faut le ressaisir entièrement, ou utiliser les "facilités" de mémorisation/correction de l'environnement SQL (type SQL*Plus d'ORACLE).
Passer par la soumission de fichiers ASCII contenant les ordres LDD (voir glossaire).

Justification

L'émission de requêtes en mode conversationnel est une opération lourde et dangereuse. Certaines requêtes occupent plusieurs lignes et sont complexes. La moindre erreur rend nécessaire une nouvelle saisie ou implique une maîtrise parfaite des mécanismes de correction et ressoumission intégrés dans l'environnement.
La saisie de l'ensemble des ordres SQL sous un éditeur permet de disposer d'une base rémanente permettant :

- la validation à priori du script,
- la ressoumission totale ou partielle en cas d'erreur ou de rejet,
- la garantie d'une trace pour investigation ou compte-rendu ultérieurs.

CD.SQL.Script	Rendre le déroulement des scripts visibles et interruptibles.

Description

Le déroulement d'un script d'ordres SQL doit être synchronisé avec des envois de messages vers l'utilisateur ou l'administrateur, pour lui permettre de suivre le déroulement de son traitement.
Il doit également être interruptible, si le déroulement du traitement n'est pas conforme, ou si on ne désire pas le continuer.
Cette règle vaut en particulier pour les procédures d'administration du dictionnaire. Elle ne s'applique pas aux traitements batch ou automatiques.

Justification

L'exécution de commandes SQL "en aveugle" peut être dangereuse. Une séquentialisation par étapes, avec reprise possible à la fin de chaque étape, permet de maîtriser l'évolution du traitement.

Exemple

Cas d'une utilisation sous SQL*Plus d'ORACLE.

```
… ordres SQL*Plus …

REMARK  Execution de l'etape de creation initiale
START ini_crea.sql
PROMPT 'Etape de creation initiale effectuee: consulter C.R'
PAUSE presser RETURN pour continuer

REMARK  Execution de l'étape de mise à jour du schema
START maj.sql
PROMPT 'Etape de mise à jour effectuee: consulter C.R'
PAUSE presser RETURN pour continuer


… suite ordres SQL*Plus …
```

7.1.3. Style de Présentation des scripts et des requêtes SQL

CO.Présentation.Requête	Le style de présentation d'une requête SQL complète respecte les principes suivants : ? Il fait ressortir les sous-requêtes composant la requête, ? Il met en évidence les connecteurs, ? Il isole les mots clés du langage, ? La requête est indentée sans excès. ? Il privilégie la compréhension.

Description

Un style classique de présentation d'une commande SQL respecte les conventions suivantes :

? Les mots clés sont en majuscules.
? Les variables, noms de tables, de colonnes, propriétaires, alias et synonymes sont en minuscules.
? Les constructions ou mots clés structurants FROM, WHERE, AS, HAVING, ORDER BY, GROUP BY, CONNECT BY, START WITH, FOR UPDATE OF,....sont en début de ligne, sauf lorsque la requête tient en une ligne.
? Les mots clés INTERSECT, MINUS et UNION sont seuls sur une ligne, sauf lorsque la requête tient en une ligne.
? Les fonctions sont- soit toutes en minuscules -soit toutes en majuscules.
? Les opérateurs sont en majuscules.
? La requête est indentée. Il est utile d'indenter une requête en respectant les règles suivantes :

Si la requête ne tient pas sur une ligne les lignes suivantes seront décalées vers la droite d'une indentation.

1. Les mots clé devront être placés en début de ligne tout en respectant l'indentation.
2. Si la partie de requête qui suit un mot clé ne tient pas sur une ligne, les lignes suivantes de l'énumération seront décalées, par rapport au mot clé, vers la droite d'une indentation.
3. La valeur de l'indentation est au minimum de 2 caractères et doit être la même dans toute l'application. 3 caractères constituent un idéal.

Justification

La structuration d'une requête facilite la relecture et la maintenance. Elle permet d'avoir une présentation identique quel que soit le programmeur.

Exemple

```
SELECT nom, adresse, ville
    FROM client
    WHERE pays = 'FRANCE'
    ORDER BY code_postal

SELECT no_client, COUNT('X')
    FROM commande
    GROUP BY no_client
    HAVING COUNT('X') =
                      (SELECT MAX(COUNT('X'))
                      FROM commande
                      GROUP BY no_commande)

SELECT no_client, nom
    FROM client
    WHERE  no_client IN
          ( SELECT no_client
                FROM commande
                WHERE etat = 'EN COURS')

UPDATE commande
    SET etat = 'ARCHIVE'
    WHERE date_commande < '01-JAN-93'

INSERT INTO article (no_article, description, prix)
    VALUES
          (123,'crayon couleur vert clair',5.20)

DECLARE curseur_commande
    CURSOR FOR
          SELECT no_client, date_commande
          FROM commande
          WHERE etat = 'EN COURS'
```

CO.Présentation.Ligne	Aucune ligne ne dépasse 79 caractères.

Description

Garder la longueur de chaque ligne inférieure à 80 caractères.

Justification

Certains éditeurs sont limités à 80 caractères.

CO.Présentation.CaractèresInterdits	Les lettres O, o et 0 sont à éviter partout. La lettre I est à éviter en fin de symbole. D'autres limitations peuvent dépendre du Projet ...

Description

La règle se passe de commentaires.

CO.Présentation.Editeur	L'utilisation d'un éditeur syntaxique est conseillée.

Description

Le style et la structure générale d'un SCRIPT SQL doit suivre un standard projet; une grande partie du travail de mise en conformité peut être automatisé par l'utilisation d'un éditeur syntaxique.
Un tel éditeur est réactif au niveau du mot ou de la ligne: il permet une mise en forme automatique au niveau du respect de la forme majuscule/minuscule, de l'indentation, de l'isolation des patterns des sous-requêtes.
Le paramétrage d'un éditeur tel que EMACS(bien connu du monde UNIX) peut être envisagé. A défaut, une mise en forme à postériori (i.e: après saisie) est envisageable: celle-ci peut être faite au moyen de tout utilitaire de traitement de fichier ASCII tel que `sed, awk, grep` etc...

Justification

L'utilisation d'un éditeur syntaxique décharge le concepteur du souci fastidieux de formattage, tout autant qu'elle permet la production d'un code homogène par l'ensemble des concepteurs.

CO.Présentation.Commentaire	Les requêtes sont commentées, sans excès.

Description

Des commentaires sont introduits dans la requête, à l'extrémité des lignes, sous forme /* xxxxxxxxxxxxxxx */
Ils ne sont introduits que lorsque la compréhension de la requête l'exige.
Un commentaire par requête est souhaitable. Certaines requêtes plus complexes peuvent en requérir plusieurs.

Justification

Un commentaire précis et concis fait perdre quelques secondes au concepteur, mais peut faire gagner plusieurs heures au mainteneur.

Des commentaires triviaux alourdissent inutilement la requête et dévaluent les commentaires importants.

Exemple

```
SELECT /* Recherche des commandes incluant article
no 115 */
     c.no_commande, c.no_client, c.date_commande,
c.etat
    FROM commande c, ligne_commande l
    WHERE l.no_article = 115
          AND c.no_commande=l.no_commande
```

est plus valorisant que:

```
SELECT  c.no_commande, c.no_client, c.date_commande,
c.etat
    FROM commande c, ligne_commande l
    WHERE l.no_article = 115 /* l= alias de
no_article */
          AND c.no_commande /* c=alias de
ligne_commande */
               = l.no_commande
```

7.1.4. Conformité aux standards et normes.

CD.Programmation.Norme	Toujours utiliser un SGBD dans sa configuration la plus proche de la norme SQL2 voire SQL3.

Description

Toujours utiliser un SGBD dans sa configuration la plus proche de la norme SQL2 ou SQL3 suivant les capacités du SGBD utilisé à intégrer cette norme.

Justification

La conformité à la norme SQL2 et SQL3 améliore la portabilité.

Exemple

L' utilisation de l'option MODE=ANSI13 de ORACLE rend conforme à la norme la syntaxe des clauses "WHENEVER"

CO.Programmation.Forme	L'utilisation des nouvelles formes syntaxiques de la norme SQL3 doit être favorisée.

Description

Règle spécifique SQL 3 / 1999

Une nouvelle syntaxe est disponible pour faciliter l'écriture de jointures et améliorer leur lisibilité :

- CROSS JOIN
- NATURAL JOIN
- la clause USING
- la clause ON
- les clauses LEFT OUTER JOIN, RIGHT OUTER JOIN et FULLL OUTER JOIN

De nouveaux prédicats sont également disponibles :

- IS DISTINCT FROM
- IS TRUE
- IS SIMILAR TO

Ces nouvelles formes d'écritures doivent être utilisées car elles permettent de simplifier l'écriture des requêtes et les rendent plus lisibles.

Justification

La simplification et l'ammélioration de la lisibilité facilite la maintenance.

Exemple

```
La jointure croisée ou produit cartésien :

SELECT nom,
   prenom,
   service
FROM employes,
   services ;
```

```
Avec la syntaxe normalisée, cette requête s'écrit :
SELECT nom,
   prenom,
   service
FROM employes
   CROSS JOIN services ;
```

```
L'opérateur NATURAL JOIN permet d'éviter de préciser les
colonnes concernées par la jointure :

SELECT d.departement,
   v.ville
FROM departements d,
   villes v
WHERE d.localisation = v.localisation
AND      d.pays = v.pays ;
```

```
Avec la syntaxe normalisée, cette requête s'écrit :

SELECT departement,
   ville
FROM departements
   NATURAL JOIN villes ;
```

```
La clause USING permet de restreindre les colonnes
concernées, lorsque plusieurs colonnes servent à définir
la jointure naturelle :

SELECT d.departement,
   v.ville
FROM departements d,
   villes v
WHERE d.localisation = v.localisation ;
```

```
Avec la syntaxe normalisée, cette requête s'écrit :

SELECT departement,
   ville
FROM departements
   NATURAL JOIN villes USING localisation ;
```

CO.Programmation.HorsNorme	Identifier et baliser les commandes d'extension de la norme.

Description

Les commandes ou paramètres n'appartenant pas à la norme sont balisées dans le texte.Un balisage automatique par éditeur syntaxique peut être envisagé.

Justification

En cas de portage ou de réutilisation: on peut tout de suite identifier les requêtes non standards, donc probablement non portables et justifiant une intervention.

L'usage de balises commentaires permet une automatisation de la recherche par un outil de traitement. Le balisage peut être plus ou moins précis et ne porter que sur la commande.

Exemple

Cas de l'utilisation d'une extension ORACLE V7: l'option ON DELETE CASCADE de la clause CONSTRAINT est non SQL2, la clause STORAGE et les identifieurs des contraintes non plus.

```
CREATE TABLE facture /*Attention: certains paramètres sont non SQL2 */
(     no_facture INTEGER NOT NULL
      quantité INTEGER NOT NULL
      montant INTEGER NOT NULL
      no_client INTEGER NOT NULL
            CONSTRAINT pk_nofacture       /*#SQL2*/   PRIMARY KEY(no_facture)
            CONSTRAINT fk_noclient        FOREIGN KEY(no_client)
                                          REFERENCES client(no_client)
                                          ON DELETE CASCADE /*#SQL2*/
            CONSTRAINT ck_23 CHECK (quantité > 0 AND montant > 0)
      STORAGE     /*#SQL2*/   (INITIAL 100K     NEXT 50K
                  MINEXTENTS 1      MAXEXTENTS 50     PCTINCREASE 5)    )
```

7.2. ADMINISTRATION DES DONNEES

7.2.1. La définition du schéma

GENERALITES

CP.Schéma.Définition	Toutes les commandes de définition du schéma doivent être groupées.

Description

Toutes les commandes de définition du schéma (autorisations, structures des tables, vues, index, ...) sont groupées dans un ou plusieurs composants logiciels qui constituent la référence pour la structure de la base.

Justification

Il est nécessaire de disposer de composants logiciels centralisés et cohérents pour installer ou ré-installer la base.

Ces ensembles de commandes de définition sont gérés en configuration avec les applicatifs. C'est à dire qu'une version des applicatifs est gérée conjointement avec une version des commandes de définition du schéma.

CD.Schéma.Table	La définition d'une table comporte une commande CREATE TABLE suivie d'une commande ALTER TABLE.

Description

La procédure de création d'une table comporte 2 étapes:

1. une étape de création pure par CREATE TABLE, qui se limite à créer l'objet physique (la table, ses colonnes, attributs de stockage et d'allocations de ressources.
2. une étape de définition logique de l'objet (clés, valeurs par défaut, contraintes...) par ALTER TABLE.

Ceci vaut également pour une vue.

Sur cette base, il est possible, au choix :

d'exécuter tous les ordres CREATE, puis tous les ordres ALTER.
d'exécuter, table après table, l'ordre CREATE suivi de l'ordre ALTER.

Justification

Mélanger dans une même procédure les ordres de création physique et les ordres de définition logique rend l'ensemble de la procédure inutilisable une fois la table (ou vue) créée.

Les ordres ALTER TABLE ainsi définis sont réappliquables tels quels ou après modification. Le processus de création des objets gagne en clarté et rigueur. Il est davantage contrôlable.

Exemple

```
CREATE TABLE facture
(     no_facture INTEGER
      quantité INTEGER
      montant INTEGER
      no_client INTEGER
      STORAGE /*#SQL2*/       (INITIAL 100K     NEXT 50K
                               MINEXTENTS 1      MAXEXTENTS 50
      PCTINCREASE 5)
      TABLESPACE  tb_human_resource
      CLUSTER cl_facturation (no_client)
      )

ALTER TABLE facture
(     MODIFY no_facture CONSTRAINT NOT NULL
      MODIFY quantité DEFAULT 1 CONSTRAINT NOT NULL
      MODIFY montant CONSTRAINT NOT NULL
      MODIFY no_client CONSTRAINT NOT NULL
            ADD CONSTRAINT pk_nofacture   /*#SQL2*/   PRIMARY
KEY(no_facture)
            ADD CONSTRAINT fk_noclient          FOREIGN
KEY(no_client)
                                          REFERENCES
client(no_client)
                                          ON DELETE CASCADE
/*#SQL2*/
            ADD CONSTRAINT ck_23 CHECK (quantité > 0 AND montant
> 0)        )
```

CD.Nommage.Objets	Les noms des objets créés dans la base ne doivent pas être des noms réservés.

Description

Les noms des objets de la base (tables, colonnes, vues, ...) ne doivent pas être identiques à des noms utilisés par le SGBD pour sa gestion interne.

Justification

Des ambiguïtés de nommage entre les objets applicatifs et les objets gérés par le système du SGBD peuvent entraîner de graves dysfonctionnements, tant pour le développement que durant des phases d'exploitation.

Exemple

CREATE TABLE utilisateur ----------

Dans l'exemple, on a préféré utiliser le nom 'UTILISATEUR' plutôt que 'USER', qui est souvent utilisé par les systèmes des SGBD.

CD.Nommage.Unicité	Deux objets différents ne doivent pas avoir le même nom, ni des orthographes très voisines.

Description

Chacun des objets de la base (tables, vues, colonnes, contraintes, ...) doit être nommé de façon distincte et non ambigu.

Justification

Des nommages non ambigus facilitent à la fois les phases de développement, de documentation, d'exploitation et de maintenance.

L'ensemble des objets est référencé dans le dictionnaire des données du SGBD. Des nommages non ambigus améliorent donc la lisibilité de ce dictionnaire.

CO.Nommage.Abréviation	Les abréviations utilisées pour le nommage ou l'aliasing des objets de la base doivent être normalisées.

Description

Les noms des objets de la base (tables, vues, colonnes, contraintes, ...) peuvent comporter des abréviations. Si c'est le cas, ces abréviations doivent être normalisées. Tous les noms d'objets obéissent alors à la norme.

Justification

Une nomenclature claire et normalisée des abréviations facilite le développement et la maintenance. Plus particulièrement, le dictionnaire de données est plus facilement exploitable et les applicatifs de type infocentre sont plus efficaces.

Exemple

? Les tables commencent par *t*, vues par *v*, Index par *i*, tablespaces par *c*, clusters par *cu*, exception par x, ...
? Les contraintes suivent un standard strict d'identification à définir par le projet,
? Toutes les colonnes définissant des dates commencent par *'dt_'*,
? Toutes les colonnes définissant des commentaires commencent par *'cmt_'*,
? Toutes les colonnes définissant des indicateurs (Oui, Non) commencent par *'boo_'*,
? ...

DEFINITION DES TABLES

CD.Table.Taille	Les tables doivent contenir un nombre limité de colonnes.

Description

Le nombre de colonnes par table, ainsi que la dimension des colonnes, doivent être maintenus minimaux. Des tables très plates (nombreux attributs) sont plus pénalisantes que des tables très profondes (nombreuses occurences).

Justification

Selon les SGBD et les outils utilisés, les performances sont influencées par le nombre de colonnes des tables : plus le nombre de colonnes augmente, plus les performances se dégradent.
Un nombre élevé d'attributs d'une table cache souvent un défaut d'analyse conceptuelle et un niveau de normalisation insuffisant..

Exemple

```
Une table ne devrait pas excéder une dizaine de colonnes. Lorsque une de ces colonnes
est de type RAW ou BLOB, la table doit si possible être dédiée au stockage de cette
information (c'est toujours possible à partir d'un modèle conceptuel "travaillé").
```

DEFINITION DES VUES

CD.Vue.MiseEnOeuvre	On doit utiliser des vues lorsque les restrictions d'accès aux données se font sur des critères dynamiques.

Description

On doit utiliser des vues lorsque les restrictions d'accès s'établissent à partir de domaines de valeurs pour des colonnes ou des occurrences de tables.

Justification

Les privilèges créés au niveau des définitions de données sont uniquement statiques, interdisant des types d'accès pour des tables ou des colonnes de tables. Les vues permettent d'établir des types d'accès à partir de constructions dynamiques d'informations.

Exemple

```
CREATE VIEW client_francais
          AS SELECT no_client, nom
                FROM client
                WHERE pays = 'FRANCE'

GRANT UPDATE ON client_francais TO andre
```

Dans l'exemple, l'utilisateur andre a des accès sélectifs sur les occurrences de la table client.

CD.Vue.Masquage	On doit utiliser les vues lorsqu'on veut masquer la structure réelle des données.

Description

Lorsqu'on veut masquer la structure réelle des données, on doit utiliser des vues qui, à partir de ces structures réelles, construisent des définitions "logiques" des données plus proches de l'utilisateur. L'utilisateur exploitera ces vues logiques en ignorant les structures réelles qui les sous-tendent.

Justification

Lorsque l'utilisateur a accès à des outils qui lui rendent les structures de données visibles (infocentre, ...), les vues permettent de lui masquer la complexité des données (jointures, ...) et les éléments de structure auxquelles il n'a pas accès (colonnes de tables).

Exemple

```
CREATE VIEW client_vendeur
          AS SELECT client.nom, client.ville,
vendeur.nom
                FROM client, vendeur
                WHERE client.id_vendeur =
vendeur.id_vendeur
```

```
Dans l'exemple, la vue 'client_vendeur' réalise en fait une collection d'informations
à partir des deux tables 'client' et 'vendeur'.
```

CO.Vue.Contrôle	Pour restreindre la modification des données à travers les vues, la clause WITH CHECK OPTION doit être utilisée de façon systématique.

Description

La clause WITH CHECK OPTION, qui est un élément de la définition de la vue, empêche qu'une modification réalisée à travers la vue (UPDATE, INSERT), soit contradictoire avec sa définition.

Justification

La mise en place de la clause permet de compléter le filtre d'accès aux données réalisé par la création des vues.

Exemple

```
CREATE VIEW client_francais
          AS SELECT no_client, nom
                FROM client
                WHERE pays = 'FRANCE'
                WITH CHECK OPTION

GRANT UPDATE ON client_francais TO andre
```

Dans l'exemple, andre ne peut pas insérer dans la vue des occurrences dont le pays serait différente de FRANCE, de même qu'il lui est impossible de modifier la valeur de pays à une autre valeur que FRANCE.

CD.Vue.Modification	Une vue permettant des modifications de données doit avoir une définition simple.

Description

Si une vue est utilisée pour mettre à jour les données de la base, sa définition ne doit pas être complexe. Les jointures de plus de 2 tables sont déconseillées, de même que les structures calculées, les opérateurs ensemblistes, les structures de groupe, l'opérateur DISTINCT, et les expressions en général.

Justification

La mise à jour des tables à travers les vues ne doit pas entraîner des ambiguïtés dues à des définitions complexes.

Exemple

```
CREATE  VIEW  client_français
    AS
          SELECT no_client, nom, adresse,
code_postal, ville, pays
          FROM client
          WHERE pays= 'FRANCE'
          WITH CHECK OPTION
```

Dans l'exemple, la vue a une définition simple et pourra être utilisée pour modifier les données

CO.Vue.Accès	Les accès à travers les vues doivent être soumis à optimisation.

Description

Lorsque des accès aux données sont réalisés au travers d'une vue, il est important d'analyser le comportement de l'optimiseur des requêtes du SGBD. On devra étudier l'optimisation des chemins d'accès utilisés par le SGBD au moment de l'exploitation de la vue.

Justification

Le chemin d'accès utilisé par le SGBD pour atteindre les informations de la vue dépend de la structure de cette vue. Selon les cas, on pourra donc être amené à simplifier la définition de la vue ou à créer des accélérateurs d'accès aux données.

Exemple

Syntaxe spécifique ORACLE)

Exemple 1.

```
CREATE  VIEW  client_français
          AS SELECT nom
          FROM client
          WHERE pays= 'FRANCE'

CREATE INDEX I1_client
          ON client(pays)
```

Dans l'exemple 1, l'index 'I1_client' permet d'optimiser les accès réalisés à partir de la vue 'client_français'.

Exemple 2.

```
CREATE VIEW  v_commandes_toulousains
      AS
            SELECT commande.no_commande,commande.date_commande, client.no_client
            FROM commande, client
            WHERE
                  (client.ville = 'TOULOUSE') AND( commande.no_client =
client.no_client)

avec commandes et clients appartenant à un cluster sur la colonne no_client.
```

Dans l'exemple 2, l'appartenance des 2 tables constituant la vue à un cluster accélère cette vue.

CO.Vue.Définition	La définition d'une vue doit être explicite: l'accès universel * est fortement déconseillé pour définir une vue.

Description

Selon les SGBD, différentes syntaxes sont possibles pour la commande CREATE VIEW. Certaines syntaxes imposent le nommage explicite des colonnes qui composent la vue (la définition de la vue est alors statique), d'autres tolèrent le caractère générique '*' (la définition de la vue est alors dynamique). La syntaxe détermine donc le degré de portabilité de la vue vis-à-vis des changements de structure des tables sous-jacentes.

Justification

Si la définition de la vue nomme explicitement les colonnes qui la composent, alors les ajouts ou suppressions de colonnes dans les tables composant la vue n'auront pas de répercussion sur la structure de la vue.

Si la définition de la vue intègre le caractère générique '*', les suppressions et ajouts de colonnes ont des répercussions immédiates sur la structure de la vue. Celle-ci est impactée: il faut la créer à nouveau pour qu'elle conserve sa structure antérieure.

Exemple

```
CREATE  VIEW  client_français
          AS SELECT *
          FROM client
          WHERE pays= 'FRANCE'
```

Dans l'exemple, la vue devra être créée à nouveau si la structure de la table 'client' est modifiée.

CD.Vue.Composition	Les vues composées à partir de vues sont interdites.

Description

La plupart des SGBD autorisent une composition récurrente des vues. Cette possibilité ne doit pas être exploitée car elle complexifie le logiciel, le rendent peu performant et freine considérablement son évolutivité.

CO.Vue.Forçage	Le forçage de la définition d'une vue est interdit.

Description

Certains SGBD (dont ORACLE et INGRES) permettent de forcer la création de la vue dans le DD, même si la requête qui la définit n'est pas valide à l'instant de la création de la vue (les tables n'existent pas encore, où ne sont pas conformes à la définition de la vue).

Justification

Il ne doit jamais être nécessaire de se placer en situation telle qu'une vue puisse être définie sans que les conditions permettant de l'exploiter soient réunies.

Exemple

sous ORACLE V7, une syntaxe du type:

```
CREATE  VIEW  client_français
          AS SELECT *
          FROM client
          WHERE pays= 'FRANCE'
FORCE                , avec table client
inexistante
```

devra être interdit, et le mot clé FORCE impitoyablement éliminé.

DEFINITION DES ACCELERATEURS D'ACCES : INDEX ET CLUSTERS

CD.Accélérateur.Définition	Un accélérateur d'accès doit être implémenté si un accès sélectif et rapide aux données est exigé.

Description

Lorsqu'un accès sélectif sur des colonnes d'une table doit être rapide, un accélérateur d'accès doit être implémenté sur ces colonnes.

Justification

Si aucun accélérateur n'est défini sur les colonnes accédées, la lecture des informations s'effectue par un parcours séquentiel de la table. Dès que les tables atteignent un volume important, la durée de cette lecture séquentielle devient prohibitive. Au contraire, dès qu'un accélérateur d'accès est créé sur les colonnes accédées, les requêtes interrogeant ces colonnes utilisent le classement réalisé par l'accélérateur. La table n'est plus alors lue séquentiellement du début jusqu'à la fin, mais est accédée via l'accélérateur (les algorithmes d'implantation de ces accélérateurs dépendent du SGBD : B-TREE, H-CODE, ...). Les temps d'accès aux informations sont diminués d'autant.

Exemple (syntaxe spécifique ORACLE)

```
CREATE INDEX i1_ligne_commande
          ON ligne_commande(no_article)

SELECT no_commande, quantite
    FROM ligne_commande
          WHERE no_article = 9001
```

```
Dans l'exemple, l'index 'i1_ligne_commande' permet d'accélérer la requête de lecture
des quantités commandées pour l'article '9001'.
```

CD.Accélérateur.Complexité	Le nombre et la complexité des accélérateurs d'accès doivent être aussi faibles que possible.

Description

On peut créer un nombre important d'accélérateurs d'accès par table, avec un nombre également important de colonnes impliquées. Il est pourtant très important de réduire le nombre et la complexité de ces accélérateurs, de manière à ce qu'ils "collent" parfaitement aux accès réalisés en utilisation opérationnelle de la base. Un accélérateur d'accès ne doit jamais être superflu.

Justification

Coût de gestion : Lorsqu'un accélérateur d'accès est créé sur des colonnes d'une table, la mise à jour de ces colonnes déclenche la modification de tous les accélérateurs où elles interviennent. Les transactions de gestion intégrant de telles modifications sont donc ralenties d'autant.

Coût de stockage : Les accélérateurs d'accès peuvent se traduire au niveau physique, par des données spécifiques qui sont effectivement stockées dans la base. Par suite, toute création d'accélérateur d'accès se traduit par un coût de stockage supplémentaire.

Exemple (Syntaxe spécifique ORACLE)

```
CREATE INDEX  i1_article
          ON article(reference)
```

Dans l'exemple, l'accélérateur n'est justifié que si des requêtes devant être rapides, portent sur la référence des articles.

CD. Accélérateur.Discrimination	Les accélérateurs d'accès doivent être discriminants.

Description

Les colonnes objets de l'accélérateur doivent avoir de nombreuses valeurs distinctes d'une occurrence à l'autre.

Justification

Lorsqu'on crée un accélérateur, les données se trouvent triées et organisées par rapport aux colonnes qui constituent l'accélérateur. Si ces colonnes prennent souvent les mêmes valeurs (colonnes peu discriminantes), les répartitions des données sont moins fines et les algorithmes d'accès deviennent moins efficaces.

Exemple

```
CREATE INDEX i1_commande
          ON commande(no_client)
```

Dans l'exemple, l'index sera d'autant plus efficace que les clients distincts objets des commandes seront nombreux. Dans l'hypothèse où l'intégralité des commandes serait partagée entre très peu de clients distincts, l'accélérateur deviendrait très peu efficace.

CD.Accélérateur.Application	Les accélérateurs d'accès doivent concerner des tables importantes.

Description

Les accélérateurs d'accès doivent être utilisés sur des tables dépassant une certaine taille (en nombre d'occurrences). Cette taille "plancher" est variable d'un SGBD à l'autre, et dépend aussi du spectre de dispersion des données. En dessous de cette taille, il n'est pas nécessaire (il est même nuisible) de créer un accélérateur. Des essais opérés sur une maquette de la base permettront de préciser, pour le SGBD et la typologie des données du Projet, la valeur approximative de ce seuil.

Justification

En dessous d'une certaine taille de table, l'accès par les accélérateurs est moins rapide qu'un accès séquentiel. L'emploi d'accélérateurs devient donc superflu. Pour choisir définitivement l'adoption ou non d'accélérateurs sur ces petites tables, il faudra considérer les fonctionnements spécifiques des optimiseurs de requêtes des SGBD.

Exemple (Syntaxe spécifique ORACLE)

```
CREATE INDEX iu_article
          ON article(no_article)
```

Dans l'exemple, l'accélérateur est inutile si, par exemple, la table des articles ne contient que 50 articles et si elle est accédée de manière unitaire (on considère ici que l'index n'est pas utilisé pour assurer l'unicité (UNIQUE INDEX)).
Par contre, avec ORACLE , si la table article est accédée en jointure avec une autre table, le fait qu'elle n'ait pas d'index implique qu'elle sera lue après les autres tables indexées, même si celles-ci sont très importantes. La décision de créer l'index ou non doit donc intégrer ces deux réflexions.

CO.Accélérateur.Optimisation	Les requêtes associées à des accélérateurs, doivent être testées vis-à-vis de l'optimiseur du SGBD sélectionné.

Description

Les requêtes réalisant les accès sur des colonnes objets d'accélérateurs, n'utilisent pas toujours les chemins d'accès souhaités. La syntaxe de ces requêtes ainsi que les opérateurs utilisés doivent être validés, certaines caractéristiques pouvant en fait désactiver l'accélérateur.

Justification

Certaines syntaxes ou opérateurs peuvent, selon les SGBD, désactiver l'utilisation des accélérateurs. Les performances des applicatifs sont alors pénalisées d'autant.

Exemple (Syntaxe spécifique ORACLE)

```
CREATE INDEX i1_article
          ON article(reference)

1.  SELECT prix
               FROM article
               WHERE reference = 'REF_001'

2.  SELECT reference,prix
               FROM article
               WHERE substr(reference,1,3) = 'REF'
```

Dans l'exemple, l'accélérateur sera utilisé pour la requête 1. La requête 2, de par le fonctionnement de l'optimiseur d'ORACLE, n'utilisera pas l'accélérateur car la requête réalise une opération sur la colonne 'reference'.

CO. Accélérateur.Création	Dans les procédures d'initialisation de données, les accélérateurs qui créent des fichiers de pointeurs doivent être créés après le chargement des données.

Description

Dans beaucoup de procédures d'initialisation de données, les opérations de création des tables, d'insertion des occurrences et de création des accélérateurs d'accès sont intégrées dans la même procédure. Dans ce cas, la création des accélérateurs d'accès, lorsqu'ils créent des fichiers de pointeurs, doit être postérieure à l'insertion des occurrences.

Justification

Il est beaucoup plus performant d'insérer "en masse" les occurrences dans des tables dépourvues de fichiers de pointeurs. Ces fichiers seront créés a posteriori, sur les données existantes. Cette règle n'est pas valable pour les accélérateurs de type cluster, qui sont définis au moment de la création des tables.

Exemple (Syntaxe spécifique ORACLE)

```
CREATE TABLE article (--------)

INSERT INTO article (-----)
          VALUES (-------------)

CREATE INDEX iu_article
          ON article(no_article)
```

```
Dans l'exemple, les données sont créées dans la table ARTICLE avant la création de
l'accélérateur d'accès.
```

CD. Accélérateur.Rémanence	Les accélérateurs d'accès sont rémanents: ils ne sont pas créés au démarrage de l'application.

Description

La définition des accélérateurs d'accès est permanente, et stockée dans le dictionnaire. Il ne doit jamais exister de fichier "prologue" ou "épilogue" créant et détruisant de tels objets. Lorsqu'on ne désire pas utiliser certains de ces objets (comme un index par exemple), il est possible de désactiver leur usage en passant par un des cas de non utilisation par l'optimiseur.

Justification

Les accélérateurs font partie des objets de la base. Leur création/suppression exige des privilèges. Dans une approche rigoureuse de la notion de base de données, seules les données doivent pouvoir être créées ou supprimées par une application.

CD. Accélérateur.Définition2	La définition des accélérateurs s'appuie sur les conditions pratiques d'utilisation de la base.

Description

Opérer un bilan des accès, effectif et prévision de croissance pour chaque table ou vue.
Identifier sa criticité sur le plan des contraintes de performance.
En déduire les accélérateurs nécessaires et suffisants.
La liste des index est déterminée en fonction des requêtes qui s'exercent sur les objets de la base.
Généralement, on choisit de créer ou pas des index selon les usages suivants :

- on crée un index sur chaque clé primaire,
- on crée un index sur chaque clé étrangère pour les tables de gros volumes (> 20 KO),
- on évite d'indexer des colonnes prenant peu de valeurs différentes,
- on limite au minimum les index posés sur des tables souvent accédées en mise à jour,
- les clusters ne sont utilisés que si les tables correspondantes sont exploitées de façon régulière.

Justification

Un accélérateur n'a de sens qu'associé à un type d'accès donné. Il ne faut pas définir d'accélérateurs sans avoir au préalable étudié sur chaque table ou vue son profil d'accès, ses estimations de volume et de croissance, etc ...

DEFINITION DES LOBS

CP.LOB.Définition	Pour un LOB interne, les caractéristiques de stockage doivent être précisées lors de sa définition.

Description

Règle spécifique SQL 3 / 1999

Les LOBs sont des objets de grande taille divisés en deux catégories :

- les LOBs internes stockés à l'intérieur de la base de données
- les LOBs externes stockés comme des fichiers par le système d'exploitation et dont la base connaît le nom et le chemin d'accès.

Justification

Pour un LOB interne, ses caractéristiques de stockage seront définies même si celles-ci ne sont pas obligatoires pour des raisons de contrôle des performances et de l'espace disque.

Exemple

```
Utilisation d'un LOB dans une table :

CREATE TYPE ty_image AS OBJECT
    (image BLOB (100K) ;
```

```
CREATE TABLE personne (
    nom VARCHAR2(50),
    prenom VARCHAR2 (50),
    photo ty_image) ;
```

CO.LOB.Locator	L'utilisation des « locator » doit être limité à la manipulation de données de grande taille (comme les LOB).

Description

Règle spécifique SQL 3 / 1999

Un «locator » est un pointeur permettant d'accéder à des éléments de type variable, paramètre, opération externe. L'intérêt d'utiliser un pointeur doit être réservé à la manipulation de données de grande taille.

Justification

L'utilisation de «locator » sur des éléments de taille moyenne ou réduite ne présente pas d'intérêt, l'organisation de la base de donnée devant résoudre les problèmes qui pourrait justifier leur utilisation dans ce cas.

Dans le cas de la manipulation de données de grande taille, l'intérêt du locator prend toute sa signification, dans la mesure où il retarde le chargement de la donnée et le tranfic réseau qui en découle au moment où l'information doit être réellement utilisée.

Notons enfin que certains SGBD (comme Oracle) implémente de façon transparente les LOBS sous forme de locator.

Exemple

```
Create table test
(id number,
large_lob clob default empty_clob());
Le type de données CLOB dans cette exemple est initialisé en utilisant
la procedure empty_clob() , afin d'initialiser le locator.
Dans cet exemple Oracle gére le type CLOB grace à un locator. Une fois
la table crée, il est possible de lire et écrire comme dans les
exemples ci dessous :
- Ecriture du Clob:
DECLARE
v1clob clob;
BEGIN
SELECT  large_lob into v1clob FROM test
WHERE  id =1
FOR UPDATE;
dbms_lob.writeappend(v1clob,4000,'1234567890....');
```

```
END;
/

- Lecture du Clob

declare
v1clob clob;
history_len binary_integer := 4000;
offset_var integer:=1;
buffer_stream varchar2(4000);
begin

select large_lob into v1clob from test
where id=1;

dbms_lob.read(v1clob,history_len,offset_var,buffer_stream);
dbms_output.put_line(buffer_stream);
end;
/
```

DEFINITION DES CONTRAINTES SUR LES TABLES

CD.Table.CléPrimaire	Il doit exister au moins une contrainte d'unicité par table.

Description

Pour une table donnée, il doit exister au moins une contrainte d'unicité portant sur une ou plusieurs de ses colonnes.

Justification

L'existence des contraintes d'unicité assure qu'indépendamment des applicatifs, ces contraintes sont toujours vérifiées par le noyau au moment des validations des modifications (COMMIT).

L'existence d'au moins un sous-caractéristique d'unicité par table prémunit contre tout risque de doublon total (toutes les colonnes d'une occurrence égalent toutes les colonnes d'une autre occurrence dans la même table) durant les phases d'exploitation.

Exemple

```
CREATE TABLE compteur_facture(
          cle_compteur_facture char(1),
          prochain_numero_facture number(6),
          CONSTRAINT unq_compteur_facture
                UNIQUE (cle_compteur_facture))

INSERT INTO compteur_facture(cle_compteur_facture,
prochain_numero_facture)
          VALUES ('A',1)
```

Une table mono-occurrence de description de paramètres généraux, ne nécessite pas fonctionnellement une contrainte d'unicité. Néanmoins, la création d'une telle

contrainte prémunira cette table contre une importation intempestive de données par exemple, et donc contre la création "sauvage" de doublon.

CO.Table.Contrainte	Toutes les contraintes relatives aux données doivent être stockées au niveau du noyau.

Description

La variété et la complexité des contraintes qui peuvent être définies au niveau du noyau dépendent du SGBD. Il est néanmoins important de s'assurer que les principales règles d'intégrité des données y sont définies (domaines de valeurs, contraintes d'unicité, contraintes d'intégrité référentielles, ...)

Justification

Le stockage des contraintes au niveau du noyau permet de s'assurer que les données resteront cohérentes indépendamment des applicatifs.

Les contraintes définies au niveau du noyau sont centralisées à l'intérieur du dictionnaire de données, et non réparties dans les applicatifs. Par suite, il est plus facile de documenter ces contraintes et de les faire évoluer.

Dans les contextes Client/ Serveur, le serveur joue pleinement son rôle de garant de la sécurité, d'intégrité et de cohérence des données alors que le client se spécialise dans les tâches de gestion de l'ergonomie et de mise en forme des informations pour des sorties sur écran ou sur imprimante.

Exemple (Syntaxe spécifique ORACLE)

```
CREATE TABLE facture
    (no_facture INTEGER PRIMARY KEY,
     quantite INTEGER NOT NULL,
     montant INTEGER NOT NULL,
     no_client INTEGER REFERENCES client
(no_client),
     CHECK (quantite > 0
     AND montant >0)
```

CO.Table.Contrôle	La vérification des contraintes doit être anticipée par rapport au COMMIT lorsque des contraintes opérationnelles l'exigent.

Description

La vérification des contraintes, lorsqu'elles sont définies au niveau du noyau, s'effectue au moment du COMMIT. Dans certains cas, cette vérification est trop tardive (cas d'un opérateur devant être prévenu avant la fin de transaction). Il faut alors déclencher la vérification des contraintes avant le COMMIT de fin de transaction. La contrainte doit donc être dupliquée ou déplacée sur le code client.

Justification

Une vérification trop tardive des contraintes d'unicité peut avoir des effets très négatifs sur le fonctionnement d'un applicatif. C'est notamment le cas pour les transactions interactives comportant plusieurs écrans.

7.2.2. La modification du schéma

GENERALITES

CP.Schéma.Modification	Les commandes de modification du schéma ne sont pas autorisées à l'intérieur des applicatifs, pour les objets permanents.

Description

Les commandes de modification du schéma (ALTER TABLE, ALTER VIEW, ...) ne sont pas autorisées à l'intérieur des applicatifs, lorsque les informations concernées sont des informations permanentes.
Au cas où des exigences fonctionnelles nécessitent ce type de commandes, elles doivent être rassemblées dans des procédures d'administration spécifiques.

Justification

Sécurité des données : Durant une opération de type ALTER, la base de données se trouve dans un état instable qui échappe aux mécanismes traditionnels de gestion des transactions (SAVEPOINT, puis ROLLBACK ou COMMIT).
Confidentialité des données : Une opération de type ALTER nécessite des privilèges d'un niveau élevé, car afférant à la définition des données. Ces privilèges devront néanmoins être concédés à l'utilisateur au moins durant l'exécution des commandes concernées.
Intégrité des données : Une commande de type ALTER, exécutée durant une transaction, réalise une validation (COMMIT) implicite des modifications de la base déjà effectuées depuis le début de transaction. A moins de prévoir un point de reprise particulier, une telle commande perturbe donc l'ensemble d'une transaction.

Exemple (Syntaxe spécifique ORACLE)

```
ALTER TABLE client
          ADD (telephone char(10))
```

7.2.3. La sécurité des données

GESTION DES TRANSACTIONS

CO.Transaction.Finalisation	Une transaction doit être achevée par COMMIT ou ROLLBACK

Description

Toute transaction doit s'achever soit par une validation dans la base des modifications effectuées (COMMIT), soit par leur annulation (ROLLBACK).

Justification

Les données modifiées par une transaction (INSERT, UPDATE, DELETE) ne sont validées dans la base qu'au moment du COMMIT.
Les données modifiées durant la transaction ne sont rendues disponibles aux autres transactions que par COMMIT ou ROLLBACK.

Exemple

```
SAVEPOINT X
--------------- (modifications)
COMMIT
```

```
Dans l'exemple, les modifications sont validées dans la base et rendues disponibles
pour les autres processus utilisateurs, au moment du COMMIT.
```

CO.Transaction.Décomposition	Une transaction modifiant des volumes de données dépassant les capacités de journalisation de la base de données, doit comporter des COMMIT intermédiaires

Description

Une transaction peut entraîner des modifications de données (INSERT, UPDATE, DELETE) dont les volumes saturent les capacités de journalisation de la base de données. Dans œ cas, et si les journaux eux-mêmes ne peuvent être ajustés aux traitements, la transaction doit impérativement comporter des COMMIT intermédiaires pour libérer ces journaux.
Ce problème ne se rencontre que dans le cas de traitements transactionnels lourds (de type financier ou centrale de services).

Justification

Les COMMIT intermédiaires sont nécessaires pour un déroulement sûr et performant de la transaction.

Exemple

```
SAVEPOINT X              ---> début de transaction
--------------- (modifications)
COMMIT                   ---> validation intermédiaire  (libération des journaux)
-------------- (modifications)
COMMIT                   ---> validation finale dans la base de données
```

```
Dans l'exemple, le COMMIT intermédiaire valide dans la base le premier lot de
modifications et libère les journaux des modifications.
```

CO.Transaction.Cohérence	Une transaction comportant plusieurs COMMIT doit implémenter les points de reprise correspondants.

Description

Toute transaction qui réalise des validations de données dans la base, alors même que la transaction n'est pas finie, doit comporter les procédures de reprise nécessaires aux fins anormales de traitement.

Justification

Des données modifiées sont validées dans la base avant la fin de la transaction, lors des COMMIT intermédiaires. En cas de fin de transaction anormale, le retour au dernier état cohérent (ROLLBACK) ne se fait donc pas sur le SAVEPOINT du début de transaction, mais sur ces COMMIT intermédiaires. La cohérence d'ensemble de la base n'étant alors plus assurée, il est nécessaire de prévoir des procédures de reprise de la transaction, qui intègrent les informations déjà validées.

Exemple

```
SAVEPOINT X                 ---> début de transaction
--------------- (modifications)
COMMIT                      ---> validation intermédiaire
-------------- (modifications)
**** Fin anormale ****
```

Dans l'exemple, le ROLLBACK ne se réalise pas par rapport au SAVEPOINT X, mais par rapport à l'état des données après le COMMIT intermédiaire.

CD.Transaction.Homogénéité	Une transaction ne doit pas combiner des commandes de modification de données (LMD) et des commandes de définition de données (LDD).

Description

Une transaction ne doit pas comporter conjointement des commandes de modification de données (INSERT, UPDATE, DELETE) et des commandes de définition de données (CREATE TABLE, ...)

Justification

Les commandes de définition de données (CREATE TABLE, ALTER TABLE, CREATE VIEW, ...) génèrent un COMMIT implicite.

Par suite, si des données ont été modifiées dans la transaction avant ces commandes de définition, les modifications sont validées dans la base. Il est alors impossible de revenir à l'état des données stable de début de transaction en cas de fin anormale.

Exemple (Syntaxe spécifique ORACLE)

```
SAVEPOINT X
-------------- (modifications)
---------------- (commandes de définition )
ROLLBACK TO X   (-------- Retour au contexte impossible)
```

Dans l'exemple, le retour au contexte de début de transaction, imposé par la commande de ROLLBACK, est impossible.

CD.Transaction.Trigger	A l'intérieur d'un trigger, le `COMMIT` ou `ROLLBACK` est interdit.

Description

Règle spécifique SQL 3 / 1999

Bien que l'appel à `COMMIT` ou `ROLLBACK` soit autorisé lors de l'analyse du code, une erreur est générée lors de l'exécution de ces commandes.

Justification

Un trigger fait partie intégrante d'une transaction, si la transaction effectue un rollback, le trigger subit ce rollback, si la transaction effectue un commit, le changement lié au trigger est aussi commité, le trigger est un atome d'une transaction et c'est pourquoi il est interdit de faire un commit ou un rollback dans un trigger.

Exemple

Une erreur ORA-04092 est levée sous Oracle lors de l'exécution d'un commit ou un rollback dans un trigger.

```
SQL> create or replace trigger Shipment_Ins_Before
before
 insert on Shipment for each row
 declare
 begin
 :new.Manual_Check := 'Y';
 commit;
 end;
 /
Trigger created.
SQL> insert into Shipment  (Shipment_ID)  values
('ABCD1000');
insert into Shipment
```

```
ERROR at line 1:
ORA-04092: cannot COMMIT in a trigger
ORA-06512: at line 2
ORA-04088: error during execution of trigger
'FRAYED_WIRES.SHIPMENT_INS_BEFORE'
```

Une erreur ORA-04092 est également levée sous Oracle si un trigger appelle une procédure stockée, une fonction ou un sous-programme dans lequel se trouve un commit ou un rollback.

GESTION DES ACCES CONCURRENTS (VERROUS)

CO.Transaction.Verrou	Toute modification de données doit être précédée par un verrouillage de ces données en écriture.

Description

Toute modification de données (UPDATE, DELETE) doit préalablement verrouiller ces données en écriture vis-à-vis des autres processus utilisateurs de ces données, même si l'analyse des accès ne fait pas apparaître de conditions de conflit prévisibles.

Justification

Les modifications de données existantes ne doivent pas se faire de manière simultanée par des processus différents. Le verrouillage des données assure la sérialisation des modifications entre les process.

Selon les outils de développements et les SGBD, les verrouillages peuvent être réalisés implicitement par les commandes de mise à jour (UPDATE, DELETE). Le déverrouillage s'effectue par la validation des modifications dans la base (COMMIT) ou l'annulation des modifications (ROLLBACK).

Exemple

```
UPDATE TABLE x SET --------- WHERE --------(------ Verrouillage implicite
                                              des occurrences modifiées )

COMMIT           (------ Libération des occurrences modifiées)
```

CO.Transaction.Exclusion	Toute lecture exclusive des données doit verrouiller ces données durant la lecture.

Description

Toute lecture des données qui exige que ces données ne soient pas modifiées durant la lecture, doit préalablement verrouiller ces données, même si l'analyse des accès ne fait pas apparaître de conditions de conflit prévisibles.

Justification

Une donnée consultée à un instant T peut être modifiée à l'instant T+1 par un autre processus utilisateur. Pour les cas où une telle modification n'est pas tolérable, la donnée doit être verrouillée durant le temps de la lecture.

Exemple

Cas de la sélection/ mise à jour d'un compteur devant être unique :

```
UPDATE table_compteur SET compteur = compteur +1 /* verrouillage implicite du
compteur*/
SELECT compteur+1 FROM table_compteur            /* lecture du compteur*/
COMMIT                                           /* libération du compteur
modifié*/
```

ou

```
SELECT compteur+1 FROM table_compteur     /*verrouillage explicite du compteur
*/
            FOR UPDATE OF compteur NOWAIT

UPDATE table_compteur SET compteur = compteur+1 /*mise à jour du compteur*/

COMMIT      /*libération du compteur modifié*/
```

CO.Transaction.Verrou2	Toute modification explicite des mécanismes implicites de verrouillage du SGBD doit être justifiée. En cas de doute: un verrouillage excessif est préférable à un verrouillage insuffisant.

Description

Les SGBD actuels possèdent une gestion fine et optimisée des verrouillages de données, basée sur une analyse statistique des principaux cas d'accès aux données. On ne modifiera pas ces mécanismes sans avoir au préalable mis en évidence leur inadaptation au besoin.
Il est nécessaire de connaître parfaitement les mécanismes implicites (par défaut) du SGBD. Dans le cas où ces mécanismes sont jugés trop -ou pas assez- contraignants, il existe dans SQL deux ordres permettant de les modifier: LOCK TABLE ou SELECT ..... FOR UPDATE.
Certains mécanismes peu connus (ex: SET TRANSACTION READ ONLY) permettent d'obtenir un bon niveau de cohérence, sans pour autant poser de verrous pénalisants.

Justification

Trop de verrouillage immobilise des ressources: les autres processus utilisateurs sont potentiellement pénalisés par la quantité de ressources indisponibles et la durée de cette indisponibilité. Pas assez de verrouillage conduit à des données qui ne sont plus intègres. Ce cas de figure se révèle dans presque tous les cas beaucoup plus critique qu'un excès de verrouillage.

Exemple

```
LOCK TABLE mpm_sql.article IN
                  ROW SHARE MODE  /*Forçage d'un verrou  au niveau occurrence*/
                  NOWAIT


      ..............

UPDATE  mpm_sql.article
            SET prix=prix*1.05
            WHERE  num_article = 45
```

Dans l'exemple, le ROW SHARE MODE ne constitue pas une erreur, mais pourrait être avantageusement remplacé (du point de vue compréhension) par un SELECT ..... FOR UPDATE, qui opère le même niveau de verrouillage.

CO.Transaction.Verrou3	Tout verrouillage de données doit être aussi sélectif et bref que possible.

Description

Tout verrouillage de données doit être libéré aussitôt que possible pour ne pas bloquer les autres utilisateurs. Il ne doit porter que sur les données réellement utilisées.

Justification

Plus longtemps le verrouillage immobilise les ressources, plus les autres processus utilisateurs sont potentiellement pénalisés par la durée d'immobilisation de ces ressources.

Exemple

```
SELECT ------- (lectures de la base ne nécessitant pas de blocage)
   FROM -------

--------- (traitements locaux)

INSERT INTO ---------- (mises à jour de la base)
   UPDATE ----------
   DELETE FROM ------
   COMMIT
```

Dans l'exemple, les mises à jour de la base sont groupées après les traitements locaux afin d'immobiliser les ressources modifiées le moins longtemps possible.

CO.Transaction.Verrou4	Toute demande explicite d'allocation de verrou doit être interruptible.

Description

Toute demande explicite d'allocation de verrou (LOCK, FOR UPDATE OF, ...) doit être interruptible dans le temps (option NOWAIT ou WAIT assorti d'une possibilité d'interruption).

Justification

Un processus demandant explicitement un verrou peut ne pas l'obtenir pendant une durée indéterminée (par exemple si l'utilisateur déjà allocataire s'est absenté). Si le processus se met en attente et qu'il n'est pas interruptible, le blocage devient définitif et peut exiger un arrêt système.

Exemple

```
SELECT compteur +1
          FROM table_compteur
          FOR UPDATE OF compteur NOWAIT
```

(Dans cet exemple, le processus demandeur du verrou reçoit immédiatement un code erreur si l'occurrence concernée est déjà bloquée par un autre process.)

CD.Transaction.MutEx	Les transactions doivent être organisées de manière à éviter les étreintes fatales.

Description

Les transactions immobilisant des données diverses de manière consécutive doivent être organisées de manière à ne pas se bloquer mutuellement. Elles doivent demander les ressources identiques selon une même chronologie. Cet ordre à respecter dans les modifications des tables peut être réalisé, par exemple, en utilisant les tables par ordre alphabétique de nom.

Justification

Deux processus qui se bloquent mutuellement dans une étreinte fatale ne peuvent être dénoués par les SGBD que par des ROLLBACK arbitraires. Il faut donc éviter ce type de dénouement en organisant selon une même chronologie d'accès, les transactions utilisatrices de données communes, et en gérant les verrous explicitement (demander tous les verrous en début de transaction, et attendre leur attribution.

Exemple : A éviter : peut provoquer un DEADLOCK :

```
Process 1                                   Process 2

UPDATE table_compteur                       UPDATE  client
            SET compteur = compteur+1                   SET nom = 'paul'
                                                        WHERE no_client =
                                                              412001

UPDATE client                               UPDATE table_compteur
            SET no_client=100000                      SET compteur = compteur+1
            WHERE no_client=412001
```

Dans l'exemple, le processus 1 obtient son verrou sur l'occurrence de table_compteur, puis demande un verrou sur le client 412001. Dans le même temps, le process 2 obtient son verrou sur le client 412001 et demande un verrou sur l'occurrence de table_compteur. Les deux processus se bloquent mutuellement.
Les tables critiques monooccurences, ou à faible nombre d'occurences, peuvent être verrouillées en EXCLUSIVE MODE pendant le temps nécessaire à la transaction :

```
Chacun des 2 process pourra avoir la forme :

LOCK TABLE table_compteur IN EXCLUSIVE MODE NOWAIT
      En cas d' echec, on reviendra plus tard .....
      En cas de succès, on continue.....
UPDATE table_compteur  SET compteur = compteur + 1

UPDATE client ..etc

COMMIT
```

CD.Transaction.Critique	Une maquette doit implémenter les mécanismes de verrouillage les plus critiques.

Description

Lorsque certaines transactions s'avèrent (à l'avance) critiques sur le plan des conflits d'accès, et présente des risques de mauvaise performance liée aux poses de verrous, une maquette doit pouvoir implémenter les principaux mécanismes incriminés, dans des conditions de charge représentatives.

Justification

Les problèmes liés aux conflits d'accès aux données sont trop souvent sous-évalués, et repoussés en phase de codage et de test. Ce défaut de prise en compte peut entraîner une 'ré-ingénierie' du schéma de la base non prévue, dommageable pour le Planning.

Exemple

```
Sans objet.
```

7.2.3.1. L'accès a la base de donnees et La confidentialité des données

L'ACCES A LA BASE DE DONNEES

CD.Schéma.Profils	En préalable au codage et à la génération de la base, le Projet doit avoir établi des conventions de noms des utilisateurs sur lesquels s'établissent les différents droits de gestion et d'accès à la base.

Description

En préalable au codage et à la génération de la base de données, le Projet doit avoir établi des conventions sur les noms des utilisateurs en fonction de leur profil, sur lesquels s'établissent les différents droits de gestion et d'accès à la base.

Justification

Une convention sur les noms en fonction des droits donnera à l'administrateur une meilleure vision globale sur les utilisateurs et leur degrés de liberté sur la base.

Exemple

```
Le nom du propriétaire du Schéma peut être le nom du Projet, ou un nom significatif
illustrant la mission du système. Le nom de l'administrateur, non trivial (éviter
ADMIN!!!) et distinct du nom physique de l'administrateur du moment, doit refléter sa
mission et ses prérogatives.
Des appelations humoristiques sont admises.
```

CD.Schéma.Propriétaire	Le nom du propriétaire du schéma ne doit jamais servir pour les accès utilisateurs: tous les droits d'accès doivent être explicitement accordés.

Description

Le schéma, les tables, ... appartiennent à un propriétaire déclaré qui correspond à une personne virtuelle et pas physique. Aucune utilisation applicative de la base ne doit se faire sous le compte de cet utilisateur-propriétaire. Si possible, les possibilités de connection de ce dernier doivent être nulles, ou très réduites.
Les commandes GRANT et REVOKE doivent être utilisées afin d'attribuer à chaque utilisateur physique les droits qui lui correspondent, et seulement eux.

Justification

Chaque utilisateur intervenant dans la base doit être identifiable et identifié. Les permissions doivent pouvoir être accordées conformément à chaque profil (ou rôle) utilisateur.
Le découplage entre le schéma de la base et les données qu'elle contient est garanti.

Exemple

```
Sans objet.
```

CD.Schéma.Concepteur	Les noms des concepteurs ne doivent pas être utilisés dans le schéma de la base.

Description

Les concepteurs créent les tables, les vues, les différents objets du schéma de la base. Mais le nom du ou des propriétaire(s) de ces objets ne saurait être le nom de ces concepteurs, ni le nom de l'utilisateur informatique sous lequel s'est opéré le développement.

Justification

Le nom des propriétaires et administrateurs.

OCTROI DE PRIVILEGES

CD.Privilège.Utilisateur	Les privilèges attribués aux utilisateurs doivent être réduits au minimum nécessaire et suffisant. On définira si possible des classes de privilèges.

Description

Les privilèges attribués aux utilisateurs de la base de données doivent être nécessaires et suffisants pour leur utilisation des données et du SGBD en général.
Plutôt que d'attribuer des privilèges au cas par cas, il est préférable de définir des classes d'appartenance des utilisateurs dotées de privilèges. Ces classes sont souvent appellés *'rôles'*.

Justification

Les contrôles de sécurité assurés par le SGBD doivent, pour être efficaces, s'appuyer sur des définitions de privilèges "au plus juste" pour chacun des utilisateurs, ou des classes d'utilisateurs.

Exemple

```
GRANT SELECT, INSERT,UPDATE
          ON client
```

```
          TO rl_agent_ventes      /* role agent des
ventes*/
GRANT rl_agent_ventes TO andre
```

Dans l'exemple, l'utilisateur *andré* a les droits de lecture, d'insertion et de mise à jour sur la table *client*. Ces droits lui sont attribués via un rôle. Il ne peut par exemple pas supprimer d'occurrence, ni modifier la structure de la table.

CD. Privilège.Octroi	Les programmes doivent éviter les commandes réclamant des privilèges importants.

Description

Tout programme applicatif doit éviter des commandes qui réclament des privilèges importants pour leur exécution.

Justification

L'existence de telles commandes (CREATE TABLE, CREATE VIEW, ...) exige d'accorder des privilèges importants pour des utilisateurs de bas niveau. Une telle dissémination des privilèges pénalise le niveau de sécurité globale de la base.

Exemple

```
CREATE TABLE table_temporaire
INSERT INTO  table_temporaire ------------
-------------------  (suite du traitement)
DROP TABLE  table_temporaire
```

Dans l'exemple, la création et la suppression de la table temporaire exigent que les utilisateurs du programme aient le privilège adéquat. On préférera donc, aux considérations de temps de réponse près, la commande DELETE FROM table_temporaire.

CD. Privilège.Restriction	On choisira de restreindre les accès par les privilèges plutôt que par les vues.

Description

Pour certains scénarios de restrictions d'accès (accès sélectif à certaines colonne d'une table, ...), on a le choix entre la réalisation d'une vue ou la création de privilèges. Il faut préférer les privilèges.

Justification

La vue se constitue dynamiquement au moment de son utilisation. Cette constitution est plus lente que la vérification de l'accès de l'utilisateur aux données concernées.

Exemple

```
GRANT SELECT, UPDATE
          ON client(no_client, pays)
          TO andre
```

Dans l'exemple, l'utilisateur a seulement la possibilité de manipuler les colonnes colonne1 et colonne2 de la table. L'autre solution était "CREATE VIEW client1 AS SELECT no_client,pays FROM client", puis "GRANT SELECT, UPDATE ON client1 TO andre".

REVOCATION DES PRIVILEGES

CD.Privilège.Révoquation	On ne doit révoquer de privilège que si cette révocation est compatible avec l'utilisation opérationnelle des données.

Description

Toute révocation d'accès ne doit être réalisée qu'après vérification qu'elle est compatible avec l'ensemble des traitements opérant sur la base de données.

Justification

En cas de révocation d'un privilège pourtant nécessaire à l'utilisation opérationnelle des données, les traitements concernés deviennent infructueux.

Exemple

```
REVOKE UPDATE
          ON client
          FROM andre
```

Dans l'exemple, si un traitement de mise à jour de la table client est lancé par l'utilisateur andre, ce traitement va être interrompu et tomber en erreur.

7.2.4. La connexion à la base

CO.Sécurité.MotDePasse	Aucun mot de passe ne doit figurer en clair dans un programme.

Description

Aucun mot de passe utilisateur ne doit figurer en clair dans un fichier programme accessible en lecture, et qui résiderait sur une machine d'exploitation de manière permanente.

Justification

La présence en clair d'un mot de passe dans des fichiers, accessibles par les systèmes d'exploitation, est contraire aux principes de confidentialité des données.

Exemple

```
CONNECT  andre/dupont
-------------- (traitement)
```

Dans l'exemple, le mot de passe de connexion est inclus dans le traitement et est donc accessible à une lecture externe.

CO. Sécurité.MotDePasse2	La saisie interactive d'un mot de passe ne doit pas laisser ce mot de passe visible à l'écran.

Description

La saisie du mot de passe d'un utilisateur ne doit pas laisser apparaitre ce mot de passe en clair sur l'écran.

Justification

L'affichage en clair d'un mot de passe utilisateur sur un écran est contraire aux principes de confidentialité des données.

Exemple

```
Entrez votre nom/mot_de_passe : andre/dupont
```

Dans cet exemple qui simule une saisie simultanée de nom et de mot de passe, le mot de passe est probablement visible à l'écran, ce qui est proscrit.

7.2.5. Le dictionnaire de données

CD.Dictionnaire.Modification	Le dictionnaire de données ne doit pas être modifié par les applicatifs.

Description

Les objets du dictionnaire de données (tables, vues, contraintes, ...) sont modifiables au moyen des requêtes SQL. Les applicatifs ne doivent pas utiliser ce moyen pour modifier des valeurs du dictionnaire des données, ou pire en modifier la structure.

Justification

La modification du dictionnaire des données demande des privilèges d'accès peu compatibles avec les privilèges habituels de l'utilisateur courant.
Des modifications directes dans des tables du dictionnaire risquent d'entraîner des incohérences d'ensemble dans le comportement de la base.

Exemple (Syntaxe spécifique ORACLE)

```
INSERT INTO user$(name,------)
          VALUES ('VINCENT', -------)
```

```
Dans l'exemple, une table du dictionnaire de données ORACLE est mise à jour
directement pour insérer un utilisateur. La véritable commande de création
d'utilisateur est en fait :
```

```
 'GRANT ---- TO vincent ------).
```

CO.Dictionnaire.Consultation	La consultation explicite des objets du dictionnaire de données doit être limitée.

Description

Les objets du dictionnaire de données (tables, colonnes, vues, ...) doivent être utilisés le moins possible par les applicatifs.

Justification

Les structures et le nommage des objets du dictionnaire de données sont susceptibles d'évolutions avec les versions successives des SGBD. Par suite, l'utilisation explicite de ces objets limite la portabilité des applicatifs.

La consultation explicite d'objets du dictionnaire de données nécessite dans la plupart des SGBD des privilèges particuliers. Par conséquent, l'intégration de telles consultations dans les applicatifs contraint souvent à accorder aux utilisateurs des privilèges peu compatibles avec leur profil.

Exemple (Syntaxe spécifique ORACLE)

```
SELECT  *
          FROM user$
          WHERE name = 'VINCENT'
```

```
Dans l'exemple, une table du dictionnaire de données ORACLE est utilisée pour vérifier
qu'un utilisateur est bien référencé dans la base. Ceci est néfaste.
Si une interrogation de ce type est nécessaire (ce qui doit d'abord être établi..),
l'application doit définir une vue applicative qui sera accédée par les utilisateurs.
Cette vue peut s'appuyer directement sur la table du DD, ou être mise à jour à
intervalles réguliers à partir du DD.
```

7.3. REQUETES D'INTERROGATION DES DONNEES

7.3.1. Généralités

CO.Requête.Colonne	Les colonnes sélectionnées doivent être désignées nominativement chaque fois que cela est possible.

Description

Dans une requête, il est possible de sélectionner toutes les colonnes d'une table en utilisant le caractère générique "*" (par exemple dans une requête "SELECT * FROM commande"). Cette possibilité est à proscrire. Il est impératif de désigner toutes les colonnes nominativement.

Justification

En spécifiant toutes les colonnes sélectionnées d'une requête d'interrogation, l'interpréteur des commandes SQL n'a pas à rechercher la liste des colonnes dans le dictionnaire des données. Un gain de temps est observé.

En maintenance, l'ajout de colonnes à une table en oubliant de corriger la requête de sélection évite l'erreur de mettre en correspondance un nombre différent de colonnes sélectionnées et de variables réceptrices.

La compatibilité ascendante est améliorée dans le cas de modifications ultérieures du schéma des tables de la BD.

Au contraire, le caractère "*" est autorisé dans les fonctions de groupe (par exemple dans une requête "SELECT COUNT(*) ...").

CD.Requête.Portée	Les requêtes d'interrogation doivent porter sur un nombre limité de tables et de colonnes.

Description

Le nombre de colonnes sélectionnées et de tables sélectionnées doivent être aussi faibles que possible. Ces nombres dépendent du SGBD utilisé, du système hôte, ...

Justification

Plus la requête est complexe, plus l'interpréteur des commandes SQL aura de possibilités pour trouver la façon optimale d'accéder aux données. Il vaut mieux décomposer le traitement et exécuter plusieurs traitements plus simples.

D'autre part, le nombre d'enregistrements susceptibles d'être ramenés par une requête effectuant la jointure entre plusieurs tables est le produit du nombre d'enregistrements de ces tables. La complexité d'une requête peut se doubler d'un volume important de données sélectionnées.

7.3.2. Usage des expressions

CD.Requête.Expression	L'usage des expressions à l'intérieur d'ordres SQL doit être réduit au minimum.

Description

SQL offre de nombreuses possibilités de fonctions et d'opérateurs de manipulation et conversion de variables de tout type. Il ne faut pas céder à la tentation d'opérer des calculs ou des traitements applicatifs dans le cadre de commandes SQL. Seuls doivent être admises les expressions indispensables pour la présentation ou l'acquisition de données. Les autres traitements doivent être opérés hors SQL.

Justification

SQL n'est pas un moyen de calcul. Il permet l'accès aux données et leur stockage. L'aspect faussement universel de SQL ne doit pas masquer qu'il ne peut s'agir d'un langage performant lorsque des traitements arithmétiques sont visés.

7.3.3. Les fonctions intégrées

CO.Requête.Distinction	Sur des volumes importants de données, ne pas rechercher les valeurs distinctes d'une colonne ou d'un groupe de colonnes en utilisant l'opérateur DISTINCT.

Description

Selon le volume des occurrences, sur lesquelles on veut rechercher les valeurs distinctes d'une colonne ou d'un groupe de colonnes, le moyen à utiliser pour effectuer la requête est différent :

- Si le volume sélectionné est faible (selon le système de gestion de base de données, les performances de la machine, les paramètres système, les tailles de la base de données et des fichiers pour les données temporaires), on peut utiliser l'opérateur DISTINCT.
- Si le volume sélectionné est très important (> 1000), il vaut mieux rechercher les occurrences triées avec une requête SQL et comparer chaque occurrence à l'occurrence précédente sans utiliser le langage SQL.

Justification

La recherche des valeurs distinctes d'une colonne ou d'un groupe de colonnes utilise des zones de données temporaires pour consigner les occurrences déjà traitées et comparer avec chaque nouvelle occurrence sélectionnée.

7.3.4. Les critères de recherche

CO.Requête.ExprCplx	Les critères de recherches complexes et comportant des opérateurs logiques ou arithmétiques doivent comporter des parenthèses.

Description

Les critères de recherches complexes qui comportent des opérateurs logiques (OR, AND NOT.....) ou arithmétiques doivent utiliser des parenthèses.

Justification

La lisibilité est améliorée et la maintenance facilitée.

L'ordre d'évaluation des critères peut varier d'une implémentation à l'autre. Le parenthèsage oblige le programmeur à penser à l'ordre d'évaluation des critères. La lecture du code est simplifiée.

CO.Requête.Critère	Les critères de recherche doivent être ordonnés.

Description

Les critères de recherche doivent être ordonnés en respectant les règles suivantes :

- grouper les critères de recherche table par table en commençant par les tables sur lesquelles les critères sont les plus sélectifs,
- dans chaque critère de recherche, mettre dans la partie gauche les éléments à rechercher et dans la partie droite les constantes ou les éléments des tables déjà recherchés.

Justification

La lecture de la requête informe sur la procédure de recherche des données. La correction d'une requête et sa maintenance sont simplifiées.

Si l'optimiseur des requêtes SQL est influencé par l'ordre des critères de recherche, les critères les plus sélectifs étant mis en premier sont exécutés en premier. Les volumes temporaires sont les volumes minimums pour parvenir au résultat.

Exemple (syntaxe spécifique ORACLE)

Rechercher les commandes en cours des clients de la ville de Toulouse :

```
SELECT commande.no_commande, commande.date_commande,
client.no_client
    FROM commande, client
    WHERE client.ville = 'TOULOUSE'
    AND commande.no_client = client.no_client
    AND commande.etat = 'EN_COURS'
```

CO.Requête.Négation	Eviter l'utilisation des opérateurs NOT IN, NOT EXISTS et NOT LIKE, sauf dans les cas où l'on désire faire ressortir la logique de la négation.

Description

Eviter autant que faire se peut les formes négatives NOT IN, NOT EXISTS et NOT LIKE. Leur préférer les formes affirmatives associées ou les opérateurs ensemblistes.
Toutefois, cette règle ne s'applique pas lorsque des objectifs de compréhension (maintenance) imposent au contraire de maintenir l'expression négative.

Justification

Soit un ensemble fini d'éléments { 'A', 'B', 'C', 'D', 'E' }. Dire que la colonne <colonne> ne prend pas ses valeurs dans le sous-ensemble {'A', 'B' } équivaut à dire que la colonne <colonne> prend ses valeurs dans le sous-ensemble complémentaire {'C', 'D', 'E' }.
Ainsi, à chaque fois que l'ensemble des éléments comparables est un ensemble fini, il ne faut pas écrire "WHERE colonne NOT IN ('A', 'B')" mais plutôt "WHERE colonne IN ('C','D', 'E')"
L'opérateur unaire NOT EXISTS teste qu'il n'existe pas d'éléments correspondants à un ensemble de critères. Il est souvent plus performant de rechercher l'ensemble complet des éléments et de soustraire les éléments vérifiant l'ensemble de critères en utilisant l'opérateur ensembliste EXCEPT (norme SQL2), ou MINUS (syntaxe ORACLE).
Tester qu'une colonne ne correspond pas à un modèle 'ABC%' en utilisant l'opérateur NOT LIKE n'est pas performant, sauf si l'ensemble de données à comparer est très petit. Là aussi, il peut être plus performant de rechercher l'ensemble complet des éléments et de soustraire les éléments correspondants au modèle 'ABC%' en utilisant l'opérateur ensembliste EXCEPT (MINUS pour ORACLE).

Exemple (syntaxe spécifique ORACLE)

```
Rechercher les commandes qui ne sont pas archivées :

SELECT no_commande, date_commande
    FROM commande
    WHERE etat IN ('EN COURS','PREVISION')

 Rechercher les articles qui n'existent dans aucune
                    commande :

SELECT no_article
    FROM article
MINUS
SELECT no_article
    FROM ligne_commande

   Rechercher les clients qui ne sont pas dans le
                 département 31 :

SELECT nom, ville
          FROM client
          WHERE pays = 'FRANCE'
MINUS
    SELECT nom, ville
```

```
FROM client
WHERE pays = 'FRANCE'
      AND code_postal LIKE '31%'
```

CO.Requête.OpEnsembliste	Préférer l'utilisation de IN à l'utilisation de OR.

Description

Si on désire rédiger un critère du type "rechercher les données de la table <table> pour lesquelles les valeurs de la colonne <colonne> sont soit 'A', soit 'B', soit 'C', il vaut mieux écrire : "WHERE colonne IN ('A', 'B','C')" plutôt que : "WHERE colonne = 'A' OR colonne = 'B' OR colonne = 'C'"

Justification

L'opérateur logique OR lie deux sous-ensembles de l'ensemble résultat, ces deux sous-ensembles étant d'abord évalués séparément.
L'opérateur IN précède une liste d'éléments à laquelle chaque valeur de la première partie du critère est évaluée.

Exemple

```
Rechercher les clients français et italiens :

SELECT nom, ville, pays
    FROM client
    WHERE pays IN ('FRANCE', 'ITALIE')

est plus performant que :

SELECT nom, ville, pays
    FROM client
    WHERE pays = 'FRANCE'
    OR pays = 'ITALIE'
```

CO.Requête.Pattern	N'utiliser LIKE que si un des caractères joker % ou _ apparaît dans la comparaison.

Description

L'opérateur LIKE peut toujours être remplacé par une expression plus simple lorsque aucun de ces caractères n'est spécifié.

Justification

L'interêt de LIKE réside dans la capacité de "pattern matching" du moteur de la base. Cette capacité repose essentiellement sur la notion de "distance" lexicale (avec LIKE) ou phonétique (avec SOUNDEX). Cette distance ne peut être exprimée que par les caractères % ou _.

CO.Requête.Pattern2	Utiliser si possible avec l'opérateur LIKE une chaîne dont les premiers caractères sont spécifiés.

Description

L'opérateur LIKE est plus performant, en particulier lorsque la colonne comparée est munie d'un accélérateur, si la chaîne recherchée est du type 'ABC%' plutôt que du type '%ABC'.

Justification

L'opérateur LIKE permet de rechercher des chaînes de caractères comparables à un modèle incluant des caractères de remplacement de chaîne de longueur quelconque "%" ou de longueur fixe "_".

Ainsi, le modèle 'ABC%' correspond aux variables de type chaîne commençant par les caractères 'ABC' : par exemple 'ABC', 'ABCD', 'ABCde'.

Le modèle '%ABC' correspond aux variables de type chaîne se terminant par les caractères 'ABC' : par exemple 'ABC', 'DABC', 'deABC'

Certains SGBD disposent d'accélérateurs qui sont utilisés pour comparer les premiers caractères de la colonne aux premiers caractères constants du modèle 'ABC%'. La recherche des éléments est plus rapide.

Exemple

```
Rechercher les clients du département 31 :

SELECT nom, ville
    FROM client
    WHERE pays = 'FRANCE'
    AND code_postal LIKE '31%'
```

CO.Requête.Pattern3	Toujours préférer l'emploi de '_' à celui de '%', dans les cas où les 2 conduisent au même résultat..

Description

Lorsque cela revient au même, on préfèrera toujours le caractère joker '_' (exactement un) au caractére '%' (un ou plus ou rien), même si on doit remplacer un '%' par plusieurs '_'.

Justification

'_' étant plus sélectif, le filtre de sélection de l'optimiseur peut être plus performant.

CO.Requête.Comparaison	Toujours comparer dans un critère de recherche des éléments de types identiques.

Description

Ne pas utiliser les conversions implicites obtenues si on met en correspondance dans un critère de recherche des éléments de types différents : par exemple une partie numérique et une partie chaîne de caractères.

Justification

La mise en correspondance de deux parties de types différents dans un critère de recherche est une erreur qui, selon le système de gestion de base de données, peut ne pas entrainer d'erreur d'exécution, mais peut avoir pour résultat des temps d'exécution très long.

Exemple

```
Par exemple, le critère "WHERE colonne1 = colonne2" avec colonne1 de type chaîne de
caractères et colonne2 de type numérique met en correspondance deux colonnes de types
différents.
Pour éviter une erreur d'exécution, certains interpréteurs de commandes SQL (ORACLE
par exemple) traduisent ce critère par le critère suivant "WHERE TO_NUMBER (colonne1)
= colonne2"
Si les colonnes "colonne1" ou "colonne2" supportent un accélérateur, l'accélérateur
est désactivé et l'évaluation du critère se fait par lecture de tous les
enregistrements de la table. Si les volumes extraits sont importants, les temps
d'exécution sont pénalisés.
```

CO.Requête.FctColonne	Reporter les fonctions de colonnes dans une même partie du critère de recherche.

Description

Si un critère met en correspondance des fonctions de colonnes, reporter si possible les fonctions dans une même partie du critère.

Justification

Le critère "WHERE colonne1 * 3 = colonne2 + 4" entraine pour chaque valeur de colonne1 et colonne2 l'évaluation des fonctions sur les colonnes "colonne1" et "colonne2" :

- f1(colonne) = colonne *3
- f2(colonne) = colonne + 4

Si les colonnes "colonne1" et "colonne2" supportent des accélérateurs (index ou cluster), l'accélérateur est désactivé et l'évaluation du critère se fait par lecture de tous les enregistrements de la table. Si les volumes extraits sont importants, les temps d'exécution sont très pénalisés.

Ainsi, selon les accélérateurs présents et la recherche que l'on veut privilégier, on peut choisir de remplacer par les critères :

- "WHERE colonne1 = (colonne2 + 4) / 3"
- "WHERE colonne2 = (colonne1 * 3) - 4

Exemple

Rechercher les articles pour lesquels la réduction de niveau "A" est inférieure aux 2/3 de la réduction de niveau "B" :

```
SELECT no_article, reference
    FROM article
    WHERE reduction_a < (reduction_b * 2) / 3
```

7.3.5. Les jointures

CO.Requête.Jointure	Les colonnes des tables jointes doivent être qualifiées.

Description

Dans les requêtes avec jointures il faut qualifier le nom des colonnes par le nom ou un alias de la table auxquelles elles appartiennent.

Justification

- La qualification permet d'améliorer les performances en diminuant les accès au descripteur de tables.
- La lisibilité du code est améliorée.
- Le risque d'erreur sur l'appartenance d'une colonne à l'une ou l'autre des tables est supprimé si ce nom est utilisé dans deux tables différentes.

Exemple *(syntaxe spécifique ORACLE)*

Rechercher les commandes en cours des clients de la ville de Toulouse :

```
SELECT commande.no_commande, commande.date_commande,
client.no_client
    FROM commande, client
    WHERE client.ville = 'TOULOUSE'
    AND commande.no_client = client.no_client
    AND commande.etat = 'EN_COURS'
```

7.3.6. Les sous-interrogations

CO.Requête.Existence	Privilégier l'utilisation de l'opérateur EXISTS.

Description

Faire précéder une sous-interrogation de l'opérateur EXISTS permet de tester l'existence de données vérifiant un ensemble de critères.

Justification

Lorsqu'une requête d'interrogation recherche l'existence de données vérifiant certains critères, il n'est pas utile de balayer la table entière.

Exemple

Par exemple, pour rechercher si l'article de référence interne 115 apparait dans au moins une commande, on peut exécuter :

```
SELECT 'OK'
    FROM ligne_commande
    WHERE no_article = 115
```

Mais il est plus rapide d'exécuter :

```
SELECT 'OK'
    FROM table_vide
    WHERE EXISTS
          (SELECT no_commande
                FROM ligne_commande
                WHERE no_article = 115)
```

CO.Requête.Interrogation	Une requête SQL pouvant être indifféremment écrite sous forme de jointure ou sous forme de sous-interrogation doit être écrite sous forme de jointure.

Description

Parfois, une requête SQL peut être écrite de différentes façons. Certaines requêtes d'interrogation écrites en utilisant des sous-interrogations peuvent aussi être écrites avec des jointures. Si tel est le cas, préférer l'écriture utilisant des jointures.

Justification

Une requête SQL incluant une sous-interrogation peut (suivant l'optimiseur des requêtes SQL) entrainer, pour chaque valeur de la partie principale de la requête, l'évaluation de la sous-partie. Les performances ne sont pas optimales. Certains optimiseurs (ORACLE à partir de la version 6) traduisent ces requêtes avec sous-interrogations en requêtes avec jointures. Ecrire directement la requête avec la jointure permet dans ces cas d'économiser le temps de traduction de la requête.

Exemple (syntaxe spécifique ORACLE)

Rechercher les commandes contenant l'article de numéro interne 115 :

```
SELECT no_commande, no_client, date_commande, etat
    FROM commande
    WHERE no_commande IN
          (SELECT no_commande
                FROM ligne_commande
                WHERE no_article = 115)
```

est identique (quant au résultat), mais peut être moins performant que la requête :

```
SELECT c.no_commande, c.no_client, c.date_commande,
c.etat
    FROM commande c, ligne_commande l
    WHERE l.no_article = 115
          AND c.no_commande = l.no_commande
```

CO.Requête.Résultat	Les opérateurs "=" et "IN" doivent être utilisés en cohérence avec le nombre d'occurences en retour de la sous- interrogation de la base.

Description

Une sous-interrogation peut avoir pour résultat aucune, une ou plusieurs occurrences.
Il est important de bien choisir, parmi les 2 opérateurs "=" et "IN" pouvant précéder la sous-interrogation, celui-qui est adapté :

? L'opérateur "=" dans "WHERE colonne = (SELECT ...)" signifie que la sous-interrogation peut avoir pour résultat aucune ou une seule occurrence.

? L'opérateur "IN" dans "WHERE colonne IN (SELECT ...)" signifie que la sous-interrogation peut avoir pour résultat aucune, une seule, ou plusieurs occurrences.

Justification

Les opérateurs "=" et "IN" ont la même valeur syntaxique dans la grammaire SQL lorsqu'ils précèdent une sous-interrogation .

Les tests effectués avant livraison d'un logiciel se font généralement sur des jeux de tests qui peuvent ne pas correspondre à des données réelles, en particulier les cardinalités entre les éléments peuvent ne pas être respectées.

Les performances étant meilleures en utilisant l'opérateur "=", utiliser cet opérateur chaque fois que cela est possible, mais bien vérifier les cardinalités afin de ne pas déclencher d'erreur lorsque les données seront des données réelles.

Exemple

```
Rechercher les clients ayant des commandes en cours :
```

```
SELECT no_client, nom, adresse, ville
    FROM client
    WHERE no_client IN
          (SELECT no_client
                FROM commande
                WHERE etat = 'EN COURS')
```

```
Un client peut avoir aucune, une, ou plusieurs commandes en cours; il faut donc
utiliser l'opérateur "IN".
```

7.3.6.1. Les interrogations de groupes

CO.Requête.FctGroupe	Simplifier le plus possible les requêtes sur lesquelles on veut implémenter des fonctions de groupe.

Description

Minimiser le nombre de tables et de colonnes dans les requêtes SQL contenant des fonctions de groupe.

Justification

Si la requête SQL incluant une fonction de groupe opère sur des tables en jointure, il faudra ajouter les clés de jointure aux colonnes de l'opérateur GROUP BY.

Les colonnes pour l'opérateur GROUP BY sont extraites et triées dans des zones de données temporaires. Minimiser le nombre de colonnes permet de minimiser le volume des données temporaires et de rendre le traitement plus performant.

CO.Requête.Critère2	Privilégier le positionnement des critères dans la clause WHERE plutôt que dans la clause HAVING.

Description

La clause HAVING permet d'évaluer des critères sur le résultat de la fonction de groupe. Si le critère peut être positionné dans la clause WHERE, il est plus performant de le placer à cet endroit.

Justification

Un critère positionné dans la clause WHERE diminue d'autant la population sur laquelle le classement par groupes est effectué. Les performances sont améliorées.

Seuls les critères sur les résultats de la fonction de groupe doivent être inscrits dans la clause HAVING.

Exemple

Rechercher les clients ayant plus de deux commandes en cours :

```
SELECT no_client
    FROM commande
    WHERE etat = 'EN COURS'
    GROUP BY no_client
    HAVING count('OK') > 2
```

CO.Requête.Occurrence	Eviter d'utiliser des requêtes "SELECT COUNT(*)....." si elles ne sont pas indispensables.

Description

Il est de mauvais style de rechercher préalablement le nombre d'enregistrements correspondant à la requête en utilisant une requête "SELECT COUNT(*) ...".Pour tester si une requête SQL ramène au moins un enregistrement ou pour rechercher le nombre exact d'enregistrements d'une requête, il faut utiliser les statuts de retour d'exécution de ces requêtes (en ouverture ou en lecture de curseur).

Justification

L'exécution des deux requêtes "SELECT COUNT(*) ..." puis "SELECT ..." exécute deux fois la même évaluation d'enregistrements dans deux curseurs différents.

L'économie de la requête "SELECT COUNT(*) ..." supprime l'interprétation de la requête, la recherche des données correspondantes, le comptage de ces données.

7.3.6.2. Les opérateurs ensemblistes

CO.Requête.Parenthésage	Si plus d'un opérateur ensembliste est utilisé dans une requête,les différentes parties doivent être placées entre parenthèses.

Description

Si plusieurs parties d'un ordre SQL sont liées par des opérateurs ensemblistes (UNION, INTERSECT), il faut placer ces parties entre parenthèses, même si l'ordre d'exécution résultant est l'ordre d'exécution par défaut.

Justification

La lisibilité est améliorée et la maintenance facilitée.
L'ordre d'évaluation des parties peut varier d'une implémentation à l'autre. Le parenthèsage oblige le programmeur à penser à l'ordre d'évaluation des parties. La lecture du code est simplifiée.

Exemple

Rechercher les clients français ou anglais habitant Londres :

```
(SELECT nom
    FROM client
    WHERE pays = 'FRANCE')
UNION
((SELECT nom
    FROM client
    WHERE pays = 'ANGLETERRE')
INTERSECT
(SELECT nom
    FROM client
    WHERE ville = 'LONDRES'))
```

7.3.7. Les tris

CO.Requête.Trie	Forcer l'ordre des lignes retournées si et seulement si cet ordre est important pour l'application.

Description

Forcer par une clause "ORDER BY" l'ordre des lignes retournées par une requête si et seulement si cet ordre est important pour l'application.

Justification

Cet ordre peut changer d'une exécution d'une requête à une autre si une clause "ORDER BY" n'est pas utilisée et évite des disfonctionnements en cas de portage. Mais attention, le tri des occurences peut être coûteux en temps.

Exemple

Rechercher les articles du catalogue de prix supérieur à 300 F et les trier par prix décroissants :

```
SELECT no_article, reference, prix
    FROM article
    WHERE prix > 300
    ORDER BY prix DESC
```

CO.Requête.Position	Ne pas désigner les colonnes par leur position dans une clause ORDER BY.

Description

L'utilisation d'un entier pour désigner la position d'une colonne définie dans une requête d'interrogation ne sera autorisée que dans le cas où il n'est pas possible de donner le nom d'une colonne. Ce cas se présente lors de l'utilisation d'opérateurs ensemblistes (UNION, INTERSECT), eux-mêmes peu conseillés.

Justification

La possibilité d'une erreur dans le rang de la colonne est supprimée.
La maintenance et la lisibilité de la requête sont améliorées.

Exemple

Rechercher les articles coutant plus de 300 F et les trier par prix décroissant :

```
SELECT no_article, reference, prix
    FROM article
    WHERE prix > 300
    ORDER BY prix DESC
```

est plus facile à lire que :

```
SELECT no_article, reference, prix
    FROM article
    WHERE prix > 300
    ORDER BY 3 DESC   /* mauvais */
```

CO.Requête.Trie2	Ne pas ordonner les occurences sur de trop nombreuses colonnes.

Description

La limite du nombre de colonnes critères de tri peut être fixée pour un applicatif donné. Elle dépend du SGBD utilisé. Un maximum de 3 parait raisonnable.

Justification

Les valeurs des colonnes critères de tri sont copiées dans des zones de données temporaires. Plus leur volume est important, plus le tri sera long.

Les performances du tri sont aussi fonction de la taille des colonnes critères de tri. Il faut donc éviter de prendre comme critère de tri une colonne de grande taille.

Si les colonnes critères de tri sont trop nombreuses, rechercher les colonnes discriminantes et celles qui le sont moins (les colonnes discriminantes sont les colonnes qui peuvent prendre beaucoup de valeurs différentes).

Exemple

Pour trier les clients de la table "client" par leur localisation, on peut exécuter la requête :

```
SELECT nom, adresse, code_postal, ville, pays
    FROM client
    ORDER BY pays, ville
```

Si la presque totalité des clients est en France, la valeur "pays" n'est pas discriminante, on peut exécuter la requête, dont le résultat sera différent :

```
SELECT nom, adresse, code_postal, ville, pays
    FROM client
    ORDER BY ville
```

7.4. MODIFICATION DES DONNEES

7.4.1. L'insertion des données

CO.Modification.MultiInsertion	Pour des insertions multiples utiliser des tableaux de variables à la place d'une itération.

Description

Il est préférable d'utiliser un tableau de variables pour réaliser une insertion multiple, plutôt que de réaliser une boucle sur des insertions occurrence par occurrence.
Les tableaux de variables utilisés dans une même requête doivent avoir le même nombre d'éléments.

Justification

Les performances sont améliorées car il n'y a qu'un seul appel au SGBD.

Exemple

```
Insérer trois nouveaux articles :

int var_article [3]   = {124 , 125 , 126 };

char var_reference [3] [30] = { "stylo encre bois" ,
"stylo à bille rouge" , "gomme" };

float var_prix [3] = { 61.5, 23 , 0.25 };

INSERT INTO article
   (no_article, reference, prix)
   VALUES ( :var_article, :var_reference, :var_prix )
```

CO.Modification.Insertion	Spécifier toutes les colonnes de la table réceptrice dans le cas d'une insertion par 'VALUES' de valeurs.

Description

Dans tous les ordres d'insertion par 'VALUES', faire suivre les mots "INSERT INTO <table>" de la liste des colonnes recevant des valeurs.
"INSERT INTO <table> (colonne1, colonne2, ..., colonneN)"
Toutes les colonnes doivent apparaître, même celles qui sont optionnelles, mais qui reçoivent à ce moment une valeur.

Justification

Pour être certain d'attribuer la bonne valeur au bon champ, il est préférable de spécifier, dans l'ordre,toutes les colonnes recevant des valeurs, dans une disposition permettant un rapprochement facile avec les valeurs apportées par 'VALUES' ou par un 'SELECT' de sous-interrogation.

Spécifier toutes les colonnes permet une relecture plus aisée, en particulier si le nombre des tables est important et leur structure complexe.
En maintenance, l'ajout de colonnes à une table en oubliant de corriger la requête d'insertion évite l'erreur de mettre en correspondance un nombre différent de colonnes réceptrices et émettrices.

Exemple

Insérer un article dont le prix n'est pas connu :

```
INSERT INTO article
    (no_article, reference)
    VALUES
          (127 , 'taille crayon')
```

CO.Modification.Copie	Pour copier des données d'une table dans une autre, utiliser l'ordre "INSERT ...SELECT".

Description

Pour copier un ensemble de données d'une table dans une autre, utiliser l'ordre "INSERT ... SELECT", plutôt que d'exécuter une requête de sélection "SELECT" et d'insérer les éléments extraits enregistrement par enregistrement "INSERT ... VALUES".

Justification

En n'exécutant qu'une seule requête, les accès à la base de données sont moins nombreux donc plus performants.
Si les volumes copiés sont très importants, il peut être plus performant de fractionner la copie d'occurences en itérant sur des sélections dans des tableaux de variables et des insertions.

Exemple

Archiver les commandes :

```
INSERT INTO sav_commande
    (no_commande, no_client, date_commande, etat)
    SELECT no_commande, no_client, date_commande,
etat
          FROM commande
          WHERE etat = 'ARCHIVE'
```

CO.Modification.Ordre	Il est inutile d'ordonner les données avant de les insérer dans une table.

Description

Insérer les données dans une table en respectant un ordre des données n'apporte aucune amélioration lors de l'exécution d'une requête d'interrogation. Il est donc inutile de trier les données dans ce seul but.

Justification

Le tri des données est une action effectuée à l'exécution des requêtes d'interrogation de données de la base de données. Si aucun tri n'est demandé sur la table, l'ordre d'affichage dépend du fonctionnement de l'optimiseur et des positions relatives des données dans le fichier base de données.

Très exceptionnellement, pour des tables absolument statiques, certains SGBD disposent de fonctionnalités rendant intéressant le fait d'insérer les données dans l'ordre : par exemple, "création d'un index sans tri préalable".

7.4.1.1. La suppression des données

CO.Modification.Suppression	Toujours spécifier 'DELETE FROM'

Description

La norme SQL2 utilise la syntaxe "DELETE FROM .....". Certains SGBD (ORACLE par exemple) définissent comme optionnel le mot clé "FROM". Dans ce cas, il faut respecter la norme SQL2.

Justification

Respecter la norme SQL2 pour éviter d'avoir des problèmes de portage.

Exemple

```
DELETE FROM commande
    WHERE etat = 'ARCHIVE'
```

7.5. MODES DE PROGRAMMATION

7.5.1. La définition de types utilisateur ou UDT

En complément à la règle de nommage *CO.Nommage.Abréviation*, les types utilisateurs devront faire l'objet d'une règle de nommage propre pour :

- le typage fort,
- les types hérités,
- les types abstraits,
- les LOBs.

CO.Type.Booleen	Lors de l'utilisation d'un booléen, le test explicite à `True` ou `False` doit être écrit.

Description

Règle spécifique SQL 3 / 1999

Lors de l'écriture d'une condition dans une structure de contrôle, le test sur une variable booléenne doit être écrit de façon explicite en testant la valeur à True ou False

Justification

L'écriture de façon explicite d'une condition favorise la lisibilité du code et permet de distinguer plus facilement les différentes conditions introduites dans le test.

Exemple

```
Exemple d'utilisation d'un booléen pour  écrire des
traces :

debug BOOLEAN  := FALSE ;

IF debug = TRUE THEN
    dbms_output. colname_v||' c1 '||col_1_v||' c2
'||col_250_v) ;
    dbms_output.new_line;
END IF ;
```

CP.Type.Définition	L'utilisation du typage fort avec le mot clé DISTINCT doit être favorisée.

Description

Règle spécifique SQL 3 / 1999

Un type DISTINCT est un type construit depuis un type ordinaire de SQL. Il ne peut être combiné dans des opérations de calcul ou de comparaisons qu'avec des types de même nature.

Justification

Le type DISTINCT permet de créer de nouveaux types de données instanciables et donc utilisables dans la définition d'une colonne de tables. Désormais, toute comparaison et opération entre types distincts différents est impossible.

Exemple

```
Exemple de distinction des types température et voltage
:
CREATE DISTINCT TYPE TEMPERATURE AS FLOAT;
CREATE DISTINCT TYPE VOLTAGE AS FLOAT ;
```

CP.Type.Factorisation	L'utilisation du typage fort permet de factoriser la définition des types des colonnes dans un type abstrait pour la gestion de la table et dans le code de création de la base.

Description

Règle spécifique SQL 3 / 1999

Une description des types des colonnes est factorisée dans un module inclut par les codes de création de la base et de gestion des données de la base ce qui évite toute duplication du type des colonnes.

Justification

Une modification du type d'une colonne consiste simplement à modifier le type fort et garantit ainsi la cohérence entre la table de la base de données et le code de gestion des données.

Exemple

```
Sans objet
```

CP.Type.Abstrait	La définition d'un type abstrait doit se faire via la notation 'AS OBJECT' permettant de conserver la visibilité sur ces objets dans le langage hôte.

Description

Règle spécifique SQL 3 / 1999

L'utilisation des types abstraits permet de construire des objets sous forme d'une structure de données et de routines qui lui sont applicables. Dans le cas où les instances de ces types abstraits doivent pouvoir transiter entre le langage hôte et le SQL, on utilisera la notation « AS OBJECT ».

Justification

Cette utilisation doit être faite en particulier pour mapper avec les types du langage Java par exemple via JDBC.

Exemple

```
Create type T_INDIVIDU as object ( Nom Varchar2(40),
                        Prenom Varchar2(30),
                        Adresse Varchar2(250)) ;
```

CO.Type.Constructeur	Pour chaque type abstrait, un constructeur et un destructeur doivent être définis explicitement.

Description

Règle spécifique SQL 3 / 1999

Le constructeur a pour but d'initialiser les champs du type abstrait soit à des valeurs par défaut qui permet de créer une instance vide qu'il faudra assigner, soit aux valeurs passées en paramètre.
Le desctructeur permet de nettoyer le type abstrait.

Justification

Si le constructeur n'assigne pas par défaut les champs d'une instance de type abstrait, ceux-ci ne sont pas consistents et risquent d'être ainsi stockés dans la base de données.

Exemple

```
Exemple de constructeur :

CONSTRUCTOR FUNCTION ADT_PERSONNE_CONSTRUCTEUR
   RETURNS ADT_PERSONNE

BEGIN
   DECLARE p ADT_PERSONNE
   SET p.PRS_NOM = NULL ;
   SET p.PRS_PRENOM = NULL ;
   SET p.PRS_DATE_NAISSANCE = NULL ;
   RETURN p ;
END

Exemple de desctructeur :

DESTRUCTOR FUNCTION ADT_PERSONNE_DESTRUCTEUR
   RETURNS ADT_PERSONNE

BEGIN
   DESTROY p ;
   RETURNS p ;
END
```

CP.Type.Imbrication	La définition de types abstraits à partir d'autres types abstraits doit être maîtrisée.

Description

Règle spécifique SQL 3 / 1999

La définition d'un type abstrait à partir d'autres types abstrait (notion de référence) doit être limitée à un seul niveau d'imbrication.

Justification

Bien que mécanisme puissant et séduisant d'un point de vue conceptuel, les types abstraits imbriqués sont difficiles à gérer et sont source de problèmes difficiles à déceler (segmentation fault)

Exemple

Sans objet

CP.Type Modélisation	Les types abstraits doivent être modélisés au niveau de la conception.

Description

L'identification des types abstrait doit apparaître au niveau de la conception globale de l'application et pas seulement dans la description de la BD. En effet, ces types seront la plupart du temps utilisé comme des vecteurs d'échange entre la partie applicative développée dans un langage hôte et la base de donnée pour leur archivage dans des tables correspondant au type abstrait ou dans des colonnes de ce type abstrait.

Justification

Cette identification permet de tracer tout le cheminement d'une donnée du plus haut niveau de la conception jusqu'au niveau traitant du stockage de la donnée (3ème tier d'une conception 3 tier).

Exemple

```
Sans objet
```

7.5.2. La programmation sans curseur

CO.Curseur.SélectionMultiple	Pour des sélections multiples utiliser des tableaux de variables plutôt que d'utiliser un curseur.

Description

Il est préférable d'utiliser un tableau de variables plutôt que d'utiliser un curseur et faire un "fetch" pour chaque ligne, lorsque le nombre maximum de ligne retournée est connu.

Justification

Les performances sont améliorées car il n'y a qu'un seul appel au SGBD.

Exemple (langage C)

```
Une table client contient des références de clients associés à des villes. Dans chaque
ville, le nombre maximum de clients est 50.
Une aide permet d'afficher tous les clients d'une ville.
La lecture de cette table est plus rapide est plus simple sous la forme suivante.
```

```
char nom_client [ 50 ] [ 30 ] ;

EXEC SQL
    SELECT nom
          INTO :nom_client
          FROM client
          WHERE ville = 'TOULOUSE';
```

7.5.3. La programmation par curseurs

OUVERTURE DE CURSEUR

CO.Curseur.Utilisation	Les curseurs fréquemment utilisés ne doivent pas être fermés.

Description

Il ne faut pas fermer les curseurs fréquemment utilisés sauf si leur nombre déclenche un problème de saturation mémoire.

Justification

L'ouverture et la fermeture d'un curseur consomment du temps machine préjudiciable aux performances.

FERMETURE DE CURSEUR

CO.Curseur.Fermeture	Les curseurs doivent être fermés dès qu'ils ne sont plus utiles.

Description

Il faut fermer un curseur lorsqu'il n'est plus utilisé.

Justification

Un curseur ouvert occupe de la place en mémoire. Un grand nombre de curseurs ouverts sature la mémoire et peut entraîner, selon les SGBD des permutations préjudiciables aux performances.

7.5.3.1. La programmation avec SQL dynamique

CO.SQL.Dynamique	Utiliser des ordres SQL dynamiques à chaque fois que la requête à exécuter n'est connue qu'au moment de son exécution.

Description

La programmation avec SQL dynamique est obligatoire dès que la requête SQL n'est pas définissable exactement.

Justification

Pour exécuter une requête SQL fabriquée en fonction de certains choix utilisateur (par exemple, l'utilisateur saisit les critères de recherche qu'il veut appliquer), seule la programmation avec SQL dynamique permet l'exécution.

En conception modulaire, il peut être intéressant de regrouper les opérations d'accès à la base de données dans des modules séparés utilisant des ordres SQL dynamiques.

CO. SQL.Constante	Utiliser des ordres SQL dynamiques paramétrés pour l'inclusion de constantes SQL .

Description

Lorsqu'une même requête SQL est exécutée plusieurs fois pour des données différentes, définir une requête avec paramètres et l'exécuter pour chaque donnée.

Justification

La requête n'est analysée qu'une fois. A partir de la deuxième exécution, les temps de réponse sont plus rapides.

Exemple (langage C)

```
Définir une requête supprimant les clients d'une ville donnée, et l'appliquer aux
villes Paris et Lyon :Copier dans la chaîne :chaine_requete la requête de suppression
des clients d'une ville
```

```
EXEC SQL PREPARE requete_suppr
    FROM :chaine_requete ;

Copier dans le descripteur :liste_var la ville de
Paris

EXEC SQL EXECUTE requete_suppr
    USING :liste_var ;
```

Recommencer en copiant dans le descripteur :liste_var la ville de Lyon

7.5.4. La programmation avec trigger

En complément à la règle de nommage *CO.Nommage.Abréviation*, les triggers devront faire l'objet d'une règle de nommage propre.

CP.Trigger. Modélisation	L'utilisation des triggers sur une table doit être justifiée et doit apparaître dans la description du schéma BD.
	Leur nommage doit inclure les informations sur le type de mise à jour, le séquencement et le type de Trigger.

Description

Règle spécifique SQL 3 / 1999

Un Trigger est un traitement se déclenchant automatiquement lors de la mise à jour d'une table. Il permet de poser du code exécutable avant ou après les événements d'insertion, de suppression ou de modification avec dans ce dernier cas un filtre d'exécution sur les colonnes modifiées.

Les triggers ne doivent être utilisés que lorsque l'on désire qu'une action spécifique s'exécute automatiquement lorsqu'un évenement spécifique ce produit.

Justification

Le déclenchement automatique des triggers rend leur utilisation non visible au sein du code SQL de gestion des données de la base. Il est nécessaire d'être vigilant sur la définition des triggeurs de manière à ne pas engendrer d'appel en cascade (qui pourrient notamment produire des appels en cycle infini !).

Exemple

```
Exemple de règle de nommage :
      Tg_<Type_Mise_A_JOUR>_<Séquencement>_<Type>_<LibelleLibre>
      Avec :
            Type_Mise_A_JOUR = [I,U,D]
                  pour insertion, mise à jour ou suppression.
            Séquencement = [A,B]
                  pour avant ou après la mise à jour de la table.
            Type = [R,S]
                  pour un déclenchement sur chaque ligne mise à
jour ou une seule fois pour la mise à jour de la table (par
ordre).
```

```
Exemple de trigger de mise à jour de remise sur les prix :
CREATE OR REPLACE TRIGGER Tg_d_a_r_equip_superv
  AFTER DELETE
  ON PA_EQUIP_SUPERV
  FOR EACH ROW
  DECLARE
  BEGIN
      DELETE FROM PA_EQ_FICTYP WHERE peq_nom = :old.peq_nom;
  END;
  /
```

CD.Trigger.Affectation	A l'intérieur d'un trigger, la modification de tables ou colonnes sur lesquelles des triggeurs sont associés, est interdite.

Description

Règle spécifique SQL 3 / 1999

On evite dans la mesure du possible le déclenchement de triggers en cascade : trigger déclenchant eux-mêmes d'autres triggers.

Justification

Un ordre SQL à l'intérieur d'un triggeur de la base de donnée peut à son tour déclencher une succession de triggers qu'il sera difficile de suivre lors des tests de mise au point et qui peuvent ammener à créer des cycles entre eux ou de la récursivité produisant des boucles infinies.

Exemple

```
Non applicable.
```

7.5.5. La programmation par procédures stockées

En complément à la règle de nommage *CO.Nommage.Abréviation*, les routines devront faire l'objet d'une règle de nommage propre.

CO.PSM.CodeRetour	Dans la mesure du possible, une routine doit retourner un code retour permettant de connaître l'état d'exécution de la routine.

Description

Règle spécifique SQL 3 / 1999

La clause RETURNS permet de définir le type retourné. Pour retourner une valeur, il suffit d'utiliser le mot clef RETURN.

L'ajout de la spécification RETURN NULL ON NULL INPUT indique qu'en cas de présence d'un paramètre NULL, la routine renvoie immédiatement le marqueur NULL.

Justification

Le test des paramètres en entrée est important dans le cas de routines écrites dans un langage hôte car ces langages ne permettent pas facilement de gérer des marqueurs NULL dans les variables.

Toute routine devra au moins retourner une information sur le déroulement de celle-ci. : correct ou en erreur.

Exemple

```
CREATE PROCEDURE credite (SQLCODE client_id, montant
DECIMAL (16,2))
   RETURN NULL ON NULL INPUT

   INSERT INTO cpt_bancaire (CLI_ID, CPT_CREDIT,
DATE_TRANSACTION)
   VALUES (client_id, montant, current_date) ;

   RETURNS OK ;
```

CO.PSM.Récursivité	Le mécanisme de récursivité ne doit être utilisé que pour le parcours de structure BD arborescente en utilisant les mécanismes introduits par SQL 3.

Description

Règle spécifique SQL 3 / 1999

La notion d'arbre de donnée peut être facilement exploitée en utilisant les mécanismes de récursivité introduits par SQL 3. Elle offre une solution séduisante et permet d'éviter des jointures par l'écriture d'une seule requête.

Ce mécanisme ne doit pas être réalisé par l'écriture de procédures PSM récursives (qui s'appellent elles-mêmes) mais par la mise en œuvre de mots clés nouvellement introduits par SQL 3.

Justification

L'implémentation de ce mécanisme est spécifique à chaque SGBD et peut se présenter sous les formes suivantes :

- WITH RECURSIVE en Sybase,
- CONNECT BY en Oracle …

pouvant comporter une clause d'arrêt lorsqu'un nombre d'item est trouvé (clause CYCLE).

Exemple

```
Déterminer s'il est possible de se rendre de NICE à
BORDEAUX :

WITH RECURSIVE trajet (depart, arrivee)
AS (SELECT VLS_DEPART, VLS_ARRIVEE
    FROM T_VOLS_VLS
    UNION ALL
    SELECT debut.depart, suite.VLS_ARRIVEE
    FROM trajet debut
          INNER JOIN T_VOLS_VLS suite
          ON debut.arrive = suite.VLS_DEPART)
SELECT *
FROM trajet
WHERE depart = 'NICE'
    AND arrivee = 'BORDEAUX'
```

CO.PSM.Présentation	Une routine doit être pourvue d'un en-tête de description. Son code doit être indenté et commenté.

Description

Règle spécifique SQL 3 / 1999

Cette règle rejoint les règles générales de tout langage de haut niveau pour lequel une présentation doit être défini au niveau de la description de chaque routine, mais également sur la présentation du code (identation, aération, parenthésage des expressions complexes etc…) et des commentaires.

Justification

La présentation homogénéise les structures de codes, favorise la lecture et donc la compréhension des traitements et de ce fait facilite la maintenance.

CO.PSM.Boucle	L'utilisation de EXIT et RETURN est interdit à l'intérieur d'une boucle sauf pour les LOOP

Description

Règle spécifique SQL 3 / 1999

SQL3 est doté d'une syntaxe riche en termes de structures conditionnelles : IF, FOR, LOOP, REPEAT, DO WHILE, DO UNTIL, WHILE, LEAVE. Il est donc important de choisir la structure conditionnelle qui convient afin d'éviter de sortir d'une structure via EXIT ou RETURN.

Justification

L'utilisation de EXIT ou RETURN destructure la terminaison d'une boucle ayant pour effet de nuire à la lisibilité et au debugage de l'opération.

Exemple

```
Cependant dans cet exemple, la boucle est terminée lorsqu'on ne trouve plus de donnees
pour satisfaire le SELECT du curseur c_line_item. L'emploi de %NOTFOUND ou %FOUND peut
causer des boucles infinies si l'on ne verifie pas que ces attributs sont  evaluées  à
NULL dans un test logique EXIT-WHEN
```

```
open c_line_item;
loop
fetch c_line_item
into li_info;
EXIT WHEN (c_line_item%NOTFOUND) or
(c_line_item%NOTFOUND is NULL);
End loop;
```

CO.PSM.Prédicat	Lors du parcours d'une collection dans une boucle FOR, utiliser les prédicats .FIRST et .LAST à la place du premier et dernier élément de la collection.

Description

Règle spécifique SQL 3 / 1999

L'utilisation de ce présicats permet de s'affranchir de la valeur réelle du premier et dernier élément d'une collection, permettant de réaliser ainsi des algorithmes « génériques ».

Justification

Favorise les évolutions et la réutilisation du code source

Exemple

```
FOR i IN emp_tab.FIRST .. emp_tab.LAST LOOP
...
END LOOP;
```

7.5.6. La structuration des programmes L3G

CD.Modélisation.Interfaçage	Les accès à la base doivent être déportés dans des modules spécialisés.

Description

Les requêtes devront être "déportées" dans des modules qui contiendront toutes les fonctions d'accès à la base. Les fonctions d'accès à la base devront avoir un nom qui rappelle que ce sont des fonctions d'accès à la base.

Justification

Seuls les modules d'accès à la base seront à modifier si le projet devait utiliser un autre SGBD que celle qui a été utilisée lors de la conception du produit.

Lors de la réalisation, les développeurs de la partie application n'ont pas à connaître les problèmes de mise en oeuvre de SQL.

La pré-compilation n'est à appliquer que sur les modules d'accès à la base. La maintenance est facilitée

Exemple (langage C)

Soit un applicatif utilisant les modules suivants :

```
module_calcul.c
module_affichage.c
module_reception.c
module_émission.c
module_sql.pcc
```

Seul le dernier module contiendra les fonctions d'accès au SGBD :

```
    int sql_rechercher_nom_client ( ref_client ,
nom_client )

    int sql_creer_client ( ref_client , nom_client ,
adresse )
```

Le seul module à être précompilé sera le module module_sql.pcc pour donner le module

```
module_sql.c
```

CO.Modélisation.Objet	Les stratégies de modélisation relationnel classique et relationnel objet ne doivent pas être mélangées au sein d'une même conception.

Description

Règle spécifique SQL 3 / 1999

Dès le début de la phase de conception d'un modèle de base de données, le type de modélisation doit être choisi entre le nouveau modèle objet relationnel introduit par la norme SQL 3 ou le modèle relationnel classique.

Justification

Le choix sur l'un ou l'autre des types de modélisation assure l'homogénéité de la gestion de la base de donnée.

Le modèle relationnel classique ne supporte pas les objets complexes : les LOBs ne sont pas structurés ce qui impose une lecture séquentielle de l'objet ; il sépare complètement les données des traitements ce qui rend impossible l'encapsulation des données. Il ne connaît pas la notion de pointeurs visibles.

En modélisation objet, l'identité objet permet de référencer de manière unique n'importe quel objet d'une base. L'encapsulation ainsi introduite par la modélisation objet permet de présenter une structure de donnée stable à l'utilisateur, même si la structure de la base change complètement. Les données, au lieu d'être directement accessibles par l'utilisateur, sont cachées et des méthodes permettant la lecture et modification sont mises en place.

La notion de référence (REF) permet de faire directement référence à un autre objet pour typer une colonne de table sans passer par une jointure alors que dans une modélisation relationnelle classique, cette notion sera modélisée sous forme d'une jointure liant une clé externe à sa cible de clé primaire.

CD.Modélisation.Homogénéité	On ne doit pas mélanger l'utilisation de procédures stockées et de requêtes écrites en langage hôte.

Description

Règle spécifique SQL 3 / 1999

L'accès à la base de donnée peut être réalisé sour forme de requêtes SQL introduites dans des procédures du langage hôte (comme du Pro*C) ou être totalement décrites dans des procédures stockées (PSM) dont seul le nom de la procédure apparaît au niveau du langage.

Ces deux types d'accès à la BD ne doivent pas cohabiter, et l'on doit choisir l'une ou l'autre des représentations pour toute l'application. Lorsqu'un SGBD implémentant les procédures stockées est utilisé, on favorise l'utilisation de ce mécanisme.

Justification

Le choix entre l'une des solution permet d'avoir une représentation homogène de la conception de l'application ainsi que de la BD et facilite la maintenance dans la mesure où ce sont les mêmes mécanisme qui sont utilisés.

CO.Modélisation.Paquetage	Les procédures stockées doivent être regroupées au sein de paquetages traitant de thèmes donnés et jouant le rôle d'interface avec le langage hôte (Java ou C++).

Description

Règle spécifique SQL 3 / 1999

SQL 3 devient un véritable langage procédural avec l'introduction des notions de blocs, de structures conditionnelles et de gestion des exceptions.

Les procédures stockées peuvent être regroupées au sein de paquetage permettant de gérer de façon centralisée et cohérentes l'ensemble des procédures relatives à une même thématique ou traitement.

Les paquetages doivent être de faible couplage et de forte cohésion. Ils seront aussi indépendant que possible : les cycles d'utilisation entre paquetages sont à proscrire.

Ce type de regroupement permet d'implémenter les modules qui seront l'interface directe avec les classes objets du langage hôte.

Justification

Le regroupement de procédures stockées dans des paquetages facilite le maintien en cohérence des évolutions et leur maintenance. D'autre part, il permet d'introduire la notion de modularité entre paquetages et apporte une rigueur et une structure dans la démarche de codage qui se rapproche des démarches de conception classique de logiciel.

L'optimisation des performances est également améliorée dans la mesure où ces paquetages sont chargés dans une zone mémoire dédiée (shared pool sous Oracle) et analysés une fois pour toute lors du «parsing » de la requête. Ces paquetages peuvent être «épinglés » en mémoire afin d'y rester et ne pas être ré-analysés.

CO.Modélisation.Seuils	Des seuils de métriques doivent être déterminés en phase de codage concernant : le taux de commentaire, le nombre d'imbrication, le nombre de lignes, …

Description

Règle spécifique SQL 3 / 1999

Cette règle vient renforcer la règle *CO.Routine.Présentation* relatives aux règles générales applicables à tout langage de haut niveau. La maîtrise de la complexité d'un traitement peut être rélisée par le respect de certains seuils.

Justification

Ces métriques permettent de mesurer la complexité d'un code et d'identifier les traitements critiques afin de mettre en oeuvre des dispositions spécifiques (découpage de code, augmentation des cas de tests …)

7.6. GESTION DES ERREURS

CO.Erreur.Gestion	Le traitement des erreurs ne doit pas pouvoir boucler.

Description

Les traitements d'erreurs activés par une commande de branchement "WHENEVER ... GOTO" doivent commencer par invalider le branchement en cas d'erreur par la commande "WHENEVER ... CONTINUE" avant l'utilisation d'une commande SQL.

Justification

Si une commande SQL est utilisée dans le traitement de l'erreur et qu'elle provoque une erreur le traitement de l'erreur sera réactivé ce qui provoquera une nouvelle erreur et ainsi de suite.

Exemple *(syntaxe SQL2 ou ORACLE avec l'option MODE=ANSI13)*

```
..........
EXEC SQL WHENEVER SQLERROR GOTO : sqlerreur ;
..........

sqlerreur :
    EXEC SQL WHENEVER SQLERROR CONTINUE ;
    EXEC SQL ROLLBACK WORK RELEASE ;

..........
```

CO.Erreur.Détection	La détection des erreurs doit utiliser WHENEVER.

Description

Les traitements d'erreurs doivent être activés par une commande WHENEVER plutôt que par un test sur des données spécifiques d'une implémentation de SQL. .

Justification

La commande WHENEVER est conforme à la norme SQL2 alors qu'un test sur une donnée est spécifique d'une implémentation et non portable.

Exemple

Utiliser

```
EXEC SQL WHENEVER NOT FOUND GOTO  :traitement_erreur  ;
```

Plutôt que de tester la structure d'erreur (syntaxe spécifique ORACLE Pro*c)

```
if ( sqlca.sqlcode == 1403 )
    traiter_erreur ( ) ;
```

CO.Erreur.Exception	Dans les procédures PSM, le traitement des erreurs doit être réalisé par le mécanisme des exceptions.

Description

Règle spécifique SQL 3 / 1999

Le PSM introduit la notion d'exception pouvant être levée ou trappées. La mise en œuvre du traitement d'exception doit être chaque fois que possible centralisé en fin de procédure et suivre des règles similaires à celles des langages hôtes tels que C++ et Java.

Justification

La lisibilité et la maintenabilité du code sont facilitées par une présentation similaire du code PSM.

Exemple

```
Exemple de présentation d'une routine :

FUNCTION totalvente (annee IN INTEGER) RETURN NUMBER
IS
    xreturn_nothing EXCEPTION ;
    xreturn_the_value EXCEPTION ;
    retval NUMBER ;

BEGIN
    retval  := calc_totals (annee) ;

    IF retval = 0 THEN
          RAISE xreturn_nothing ;
    ELSE
          RAISE xreturn_the_value ;
    END IF ;

EXCEPTION
    WHEN xreturn_the_value THEN RETURN retval ;
    WHEN xreturn_nothing THEN return 0 ; ;
END ;
```

CO.Erreur.Nommage	La gestion des exceptions doit donner lieu à une règle de nommage et à des règles générales sur la gestion des exceptions.

Description

Règle spécifique SQL 3 / 1999

SQL3 introduit désormais la notion d'exception semblable aux langages objets. Le traitement des cas dégradés doit être réalisé par ce biais, pour lequel quelques règles doivent êtres définis par chaque projet.

Justification

La notion d'exception permet d'élaguer le code de traitements plus lourds et explicites tels que le test des compte rendu et le déroutement en traitement d'erreur. Pour ce faire, une règle sur le nommage des exception doit être identifiée afin de les discerner clairement de tout autre élément de codage (on pourra convenir par exemple que tout nom d'exception débute par « x ») ainsi qu'une règle générale comme le regroupement des traitement d'exception en fin d'opération plutôt que dans chaque bloc.

8. AUTRES ASPECTS SPECIFIQUES AU LANGAGE

8.1. BASE DE REFERENCE POUR LE DOCUMENT

Les exemples donnés dans ce document utilisent les tables principales suivantes :

`Liste des clients et caractéristiques associées :`

Table	Colonne	Type	Longueur	Optionnalité
CLIENT	*NO_CLIENT*	*entier*	*6*	*obligatoire*
	NOM	*caractère variable*	*30*	*obligatoire*
	ADRESSE	*caractère variable*	*80*	*obligatoire*
	CODE_POSTAL	*caractère fixe*	*5*	*obligatoire*
	VILLE	*caractère variable*	*20*	*obligatoire*
	PAYS	*caractère variable*	*20*	
	ID_VENDEUR	*caractère fixe*	*10*	

`Liste des articles :`

Table	Colonne	Type	Longueur	Optionnalité
ARTICLE	*NO_ARTICLE*	*entier*	*6*	*obligatoire*
	REFERENCE	*caractère fixe*	*30*	*obligatoire*
	PRIX	*décimal*	*6,2*	
	REDUCTION_A	*décimal*	*4,2*	
	REDUCTION_B	*décimal*	*4,2*	
	REDUCTION_C	*décimal*	*4,2*	

`Liste des commandes :`

Table	Colonne	Type	Longueur	Optionnalité
COMMANDE	*NO_COMMANDE*	*entier*	*6*	*obligatoire*
	NO_CLIENT	*entier*	*6*	*obligatoire*
	DATE_COMMANDE	*date*		
	ETAT	*caractère variable*	*10*	*obligatoire Valeurs possibles: ('EN COURS', 'ARCHIVE', 'PREVISION')*

Liste des lignes de commandes :

Table	Colonne	Type	Longueur	Optionnalité
LIGNE_COMMANDE	*NO_COMMANDE*	*entier*	*6*	*obligatoire*
	NO_ARTICLE	*entier*	*6*	*obligatoire*
	QUANTITE	*entier*	*6*	
	REDUCTION	*décimal*	*4,2*	

Liste des vendeurs :

Table	Colonne	Type	Longueur	Optionnalité
VENDEUR	*ID_VENDEUR*	*caractère fixe*	*10*	*obligatoire*
	NOM	*caractère variable*	*30*	*obligatoire*

Certains exemples utilisent d'autres tables :
La table SAV_COMMANDE est une table d'archive des commandes. Sa structure est identique à celle de la table COMMANDE.
La table TABLE_VIDE est une table à une colonne et un enregistrement. Elle peut être utilisée pour accéder à des constantes de la base de données.
La table TABLE_COMPTEUR contient une colonne COMPTEUR de type numérique.

9. SYNTHESE

9.1. TABLE RECAPITULATIVE DES REGLES

Les règles sont récapitulées ici, classées par ordre alphabétique.

Id. Règle	Intitulé	Page
CD. Accélérateur.Définition2	La définition des accélérateurs s'appuie sur les conditions pratiques d'utilisation de la base.	35
CD. Accélérateur.Discrimination	Les accélérateurs d'accès doivent être discriminants.	32
CD. Accélérateur.Rémanence	Les accélérateurs d'accès sont rémanents: ils ne sont pas créés au démarrage de l'application.	34
CD. Privilège.Octroi	Les programmes doivent éviter les commandes réclamant des privilèges importants.	51
CD. Privilège.Restriction	On choisira de restreindre les accès par les privilèges plutôt que par les vues.	51
CD.Accélérateur.Application	Les accélérateurs d'accès doivent concerner des tables importantes.	32
CD.Accélérateur.Complexité	Le nombre et la complexité des accélérateurs d'accès doivent être aussi faibles que possible.	31
CD.Accélérateur.Définition	Un accélérateur d'accès doit être implémenté si un accès sélectif et rapide aux données est exigé.	31
CD.Dictionnaire.Modification	Le dictionnaire de données ne doit pas être modifié par les applicatifs.	53
CD.Modélisation.Homogénéité	On ne doit pas mélanger l'utilisation de procédures stockées et de requêtes écrites en langage hôte.	84
CD.Modélisation.Interfaçage	Les accès à la base doivent être déportés dans des modules spécialisés.	83
CD.Nommage.Objets	Les noms des objets créés dans la base ne doivent pas être des noms réservés.	23
CD.Nommage.Unicité	Deux objets différents ne doivent pas avoir le même nom, ni des orthographes très voisines.	24
CD.Privilège.Révoquation	On ne doit révoquer de privilège que si cette révocation est compatible avec l'utilisation opérationnelle des données.	52
CD.Privilège.Utilisateur	Les privilèges attribués aux utilisateurs doivent être réduits au minimum nécessaire et suffisant. On définira si possible des classes de privilèges.	50
CD.Programmation.Norme	Toujours utiliser un SGBD dans sa configuration la plus proche de la norme SQL2 voire SQL3.	18
CD.Requête.Expression	L'usage des expressions à l'intérieur d'ordres SQL doit être réduit au minimum.	56
CD.Requête.Portée	Les requêtes d'interrogation doivent porter sur un nombre limité de tables et de colonnes.	55
CD.Schéma.Concepteur	Les noms des concepteurs ne doivent pas être utilisés dans le schéma de la base.	50
CD.Schéma.Profils	En préalable au codage et à la génération de la base, le Projet doit avoir établi des conventions de noms des utilisateurs sur lesquels s'établissent les différents droits de gestion et d'accès à la base.	49
CD.Schéma.Propriétaire	Le nom du propriétaire du schéma ne doit jamais servir pour les accès utilisateurs: tous les droits d'accès doivent être explicitement accordés.	49
CD.Schéma.Table	La définition d'une table comporte une commande CREATE TABLE suivie d'une commande ALTER TABLE.	22
CD.SQL.Requête	Les requêtes SQL sont spécifiées par les concepteurs qui en ont le besoin. Elles sont écrites par un spécialiste de SQL. Elles sont optimisées par un spécialiste du SGBD.	13
CD.SQL.Script	Rendre le déroulement des scripts visibles et interruptibles.	14

Id. Règle	Intitulé	Page
CD.SQL.Soumission	Eviter la soumission d'ordres LDD en mode conversationnel. Toujours passer par des fichiers sources.	14
CD.Table.ClePrimaire	Il doit exister au moins une contrainte d'unicité par table.	37
CD.Table.Taille	Les tables doivent contenir un nombre limité de colonnes.	25
CD.Transaction.Critique	Une maquette doit implémenter les mécanismes de verrouillage les plus critiques.	48
CD.Transaction.Homogénéité	Une transaction ne doit pas combiner des commandes de modification de données (LMD) et des commandes de définition de données (LDD).	42
CD.Transaction.MutEx	Les transactions doivent être organisées de manière à éviter les étreintes fatales.	48
CD.Transaction.Trigger	A l'intérieur d'un trigger, le COMMIT ou ROLLBACK est interdit.	43
CD.Trigger.Affectation	A l'intérieur d'un trigger, la modification de tables ou colonnes sur lesquelles des triggeurs sont associés, est interdite.	79
CD.Vue.Composition	Les vues composées à partir de vues sont interdites.	30
CD.Vue.Masquage	On doit utiliser les vues lorsqu'on veut masquer la structure réelle des données.	26
CD.Vue.MiseEnOeuvre	On doit utiliser des vues lorsque les restrictions d'accès aux données se font sur des critères dynamiques.	25
CD.Vue.Modification	Une vue permettant des modifications de données doit avoir une définition simple.	27
CO. Accélérateur.Création	Dans les procédures d'initialisation de données, les accélérateurs qui créent des fichiers de pointeurs doivent être créés après le chargement des données.	34
CO. Sécurité.MotDePasse2	La saisie interactive d'un mot de passe ne doit pas laisser ce mot de passe visible à l'écran.	53
CO. SQL.Constante	Utiliser des ordres SQL dynamiques paramétrés pour l'inclusion de constantes SQL .	78
CO.Accélérateur.Optimisation	Les requêtes associées à des accélérateurs, doivent être testées vis -à-vis de l'optimiseur du SGBD sélectionné.	33
CO.Curseur.Fermeture	Les curseurs doivent être fermés dès qu'ils ne sont plus utiles.	77
CO.Curseur.SélectionMultiple	Pour des sélections multiples utiliser des tableaux de variables plutôt que d'utiliser un curseur.	76
CO.Curseur.Utilisation	Les curseurs fréquemment utilisés ne doivent pas être fermés.	77
CO.Dictionnaire.Consultation	La consultation explicite des objets du dictionnaire de données doit être limitée.	54
CO.Erreur.Détection	La détection des erreurs doit utiliser WHENEVER.	86
CO.Erreur.Exception	Dans les procédures PSM, le traitement des erreurs doit être réalisé par le mécanisme des exceptions.	87
CO.Erreur.Gestion	Le traitement des erreurs ne doit pas pouvoir boucler.	86
CO.Erreur.Nommage	La gestion des exceptions doit donner lieu à une règle de nommage et à des règles générales sur la gestion des exceptions.	88
CO.LOB.Locator	L'utilisation des « locator » doit être limité à la manipulation de données de grande taille (comme les LOB).	36
CO.Modélisation.Objet	Les stratégies de modélisation relationnel classique et relationnel objet ne doivent pas être mélangées au sein d'une même conception.	84
CO.Modélisation.Paquetage	Les procédures stockées doivent être regroupées au sein de paquetages traitant de thèmes donnés et jouant le rôle d'interface avec le langage hôte (Java ou C++).	85

Id. Règle	Intitulé	Page
CO.Modélisation.Seuils	Des seuils de métriques doivent être déterminés en phase de codage concernant : le taux de commentaire, le nombre d'imbrication, le nombre de lignes, …	85
CO.Modification.Copie	Pour copier des données d'une table dans une autre, utiliser l'ordre "INSERT ...SELECT".	70
CO.Modification.Insertion	Spécifier toutes les colonnes de la table réceptrice dans le cas d'une insertion par 'VALUES' de valeurs.	69
CO.Modification.MultiInsertion	Pour des insertions multiples utiliser des tableaux de variables à la place d'une itération.	69
CO.Modification.Ordre	Il est inutile d'ordonner les données avant de les insérer dans une table.	71
CO.Modification.Suppression	Toujours spécifier 'DELETE FROM'	71
CO.Nommage.Abréviation	Les abréviations utilisées pour le nommage ou l'aliasing des objets de la base doivent être normalisées.	24
CO.Présentation.CaractèresInterdits	Les lettres O, o et 0 sont à éviter partout. La lettre I est à éviter en fin de symbole. D'autres limitations peuvent dépendre du Projet ...	16
CO.Présentation.Commentaire	Les requêtes sont commentées, sans excès.	17
CO.Présentation.Editeur	L'utilisation d'un éditeur syntaxique est conseillée.	17
CO.Présentation.Ligne	Aucune ligne ne dépasse 79 caractères.	16
CO.Présentation.Requête	Le style de présentation d'une requête SQL complète respecte les principes suivants : Il fait ressortir les sous-requêtes composant la requête, Il met en évidence les connecteurs, Il isole les mots clés du langage, La requête est indentée sans excès. Il privilégie la compréhension.	15
CO.Programmation.Forme	L'utilisation des nouvelles formes syntaxiques de la norme SQL3 doit être favorisée.	19
CO.Programmation.HorsNorme	Identifier et baliser les commandes d'extension de la norme.	20
CO.PSM.Boucle	L'utilisation de EXIT et RETURN est interdit à l'intérieur d'une boucle sauf pour les LOOP	82
CO.PSM.CodeRetour	Dans la mesure du possible, une routine doit retourner un code retour permettant de connaître l'état d'exécution de la routine.	80
CO.PSM.Prédicat	Lors du parcours d'une collection dans une boucle FOR, utiliser les prédicats .FIRST et .LAST à la place du premier et dernier élément de la collection.	82
CO.PSM.Présentation	Une routine doit être pourvue d'un en-tête de description. Son code doit être indenté et commenté.	81
CO.PSM.Récursivité	Le mécanisme de récursivité ne doit être utilisé que pour le parcours de structure BD arborescente en utilisant les mécanismes introduits par SQL 3.	80
CO.Requête.Colonne	Les colonnes sélectionnées doivent être désignées nominativement chaque fois que cela est possible.	55
CO.Requête.Comparaison	Toujours comparer dans un critère de recherche des éléments de types identiques.	61
CO.Requête.Critère	Les critères de recherche doivent être ordonnés.	57
CO.Requête.Critère2	Privilégier le positionnement des critères dans la clause WHERE plutôt que dans la clause HAVING.	65

Id. Règle	Intitulé	Page
CO.Transaction.Verrou2	Toute modification explicite des mécanismes implicites de verrouillage du SGBD doit être justifiée. En cas de doute: un verrouillage excessif est préférable à un verrouillage insuffisant.	45
CO.Transaction.Verrou3	Tout verrouillage de données doit être aussi sélectif et bref que possible.	47
CO.Transaction.Verrou4	Toute demande explicite d'allocation de verrou doit être interruptible.	47
CO.Type.Booleen	Lors de l'utilisation d'un booléen, le test explicite à True ou False doit être écrit.	72
CO.Type.Constructeur	Pour chaque type abstrait, un constructeur et un destructeur doivent être définis explicitement.	74
CO.Vue.Accès	Les accès à travers les vues doivent être soumis à optimisation.	28
CO.Vue.Contrôle	Pour restreindre la modification des données à travers les vues, la clause WITH CHECK OPTION doit être utilisée de façon systématique.	26
CO.Vue.Définition	La définition d'une vue doit être explicite: l'accès universel * est fortement déconseillé pour définir une vue.	29
CO.Vue.Forçage	Le forçage de la définition d'une vue est interdit.	30
CP.LOB.Définition	Pour un LOB interne, les caractéristiques de stockage doivent être précisées lors de sa définition.	35
CP.Schéma.Définition	Toutes les commandes de définition du schéma doivent être groupées.	22
CP.Schéma.Modification	Les commandes de modification du schéma ne sont pas autorisées à l'intérieur des applicatifs, pour les objets permanents.	40
CP.SQL.Accès	L'accès direct à SQL par les utilisateurs finaux de la base doit être évité.	13
CP.Trigger. Modélisation	L'utilisation des triggers sur une table doit être justifiée et doit apparaître dans la description du schéma BD. Leur nommage doit inclure les informations sur le type de mise à jour, le séquencement et le type de Trigger.	78
CP.Type Modélisation	Les types abstraits doivent être modélisés au niveau de la conception.	76
CP.Type.Abstrait	La définition d'un type abstrait doit se faire via la notation 'AS OBJECT' permettant de conserver la visibilité sur ces objets dans le langage hôte.	74
CP.Type.Définition	L'utilisation du typage fort avec le mot clé DISTINCT doit être favorisée.	73
CP.Type.Factorisation	L'utilisation du typage fort permet de factoriser la définition des types des colonnes dans un type abstrait pour la gestion de la table et dans le code de création de la base.	73
CP.Type.Imbrication	La définition de types abstraits à partir d'autres types abstraits doit être maîtrisée.	75

REFERENTIEL NORMATIF REALISE PAR :

Centre National d'Etudes Spatiales

Inspection Générale Direction de la Fonction Qualité

18 Avenue Edouard Belin

31401 TOULOUSE CEDEX 9

Tél. :05 61 27 31 31 - Fax : 05 61 28 28 49