

# Cours de bases de données

## Stockage de données

P. Rigaux

CNAM Paris

April 27, 2011

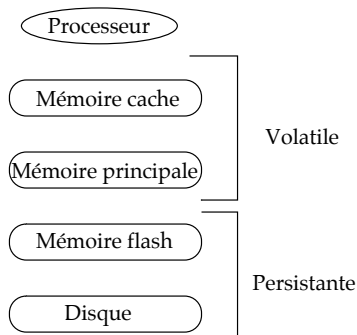
# Techniques de stockage

Contenu de ce cours :

- ➊ **Stockage de données.** Supports, fonctionnement d'un disque, technologie RAID
- ➋ **Organisation des fichiers.** Champs, enregistrements, blocs, techniques d'accès.
- ➌ **Un exemple concret : Oracle.**

# Les mémoires d'un ordinateur

Les mémoires dans un ordinateur forment une hiérarchie :



Plus une mémoire est rapide, moins elle est volumineuse.

# Quelques ordres de grandeur

Mémoire	Taille (en Mo)	Temps d'accès (secondes)
cache	Quelques Mégas	$\approx 10^{-8}$ (10 nanosec.)
principale	$O(10^{12})$ (Gigas)	$\approx 10^{-8} - 10^{-7}$ (10-100 nanosec.)
Mémoire "flash"	$O(10^{12})$ (Gigas)	$\approx 10^{-4}$ (0,1 millisc.)
disque magnétique	$O(10^{15})$ (Téras)	$\approx 10^{-2}$ (10 millisc.)

Un accès disque est (environ) un million de fois plus coûteux qu'un accès en mémoire principale (effet de la **latence**) !

C'est vrai pour un accès *aléatoire*, mais très souvent les accès se font *séquentiellement* car les données sont bien rangées.

Dans ce dernier cas, le **débit maximal** reste limité (env. 100 MO/s).

# Importance pour les bases de données

La problématique des performances, en quelques phrases:

- un SGBD doit ranger sur disque les données ;
  - ① parce qu'elle sont trop volumineuses (de moins en moins vrai) ;
  - ② parce qu'elles sont **persistantes** (et doivent survivre à un arrêt du système).
- le SGBD doit **toujours** amener les données en mémoire pour les traiter ;
- si possible, les données utiles devraient résider le plus possible en mémoire.

La capacité à gérer efficacement les transferts disque-mémoire est un facteur important de performance d'une application bases de données.

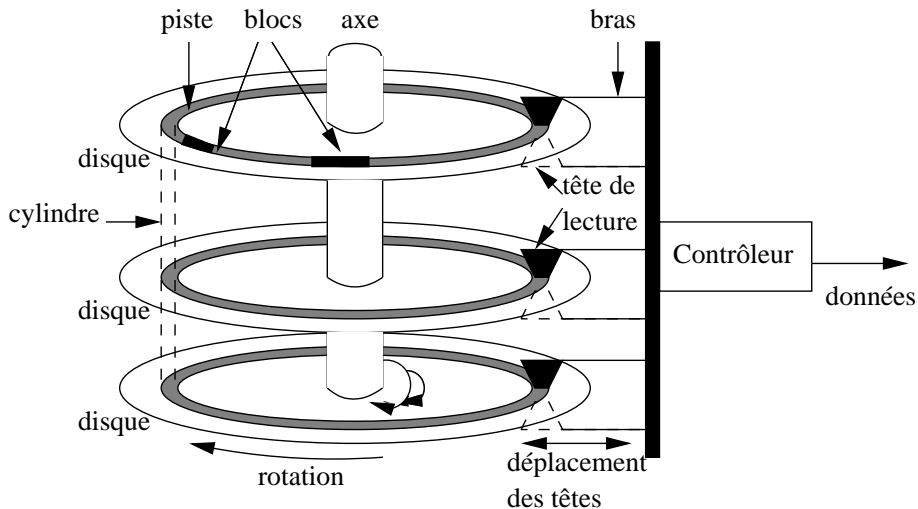
# Organisation d'un disque

**Disque** : surface magnétique, stockant des 0 ou des 1, divisé en secteurs.

**Dispositif** : les surfaces sont entraînées dans un mouvement de rotation ; les têtes de lecture se déplacent dans un plan fixe.

- le **bloc** est un ensemble de secteurs, sa taille est en général un multiple de 512 ;
- la **piste** est l'ensemble des blocs d'une surface lus au cours d'une rotation ;
- le **cylindre** est un ensemble de pistes situées sous les têtes de lecture.

# Structure d'un disque



# Disque = mémoire à accès direct

Adresse = numéro du disque ; de la piste où se trouve le bloc ; du numéro du bloc sur la piste.

- ➊ **délai de positionnement** pour placer la tête sur la bonne piste ;
- ➋ **délai de latence** pour attendre que le bloc passe sous la tête de lecture ;
- ➌ **temps de transfert** pour attendre que le (ou les) bloc(s) soient lus et transférés.

**Important** : on lit toujours au moins un bloc, même si on ne veut qu'un octet ; c'est **l'unité d'entrée/sortie** !

**Essentiel: le principe de localité** : si deux données sont « proches » du point de vue applicatif, alors elles doivent être proches sur le disque (idéalement, dans le même bloc).



# Exemple de spécification d'un disque

Caractéristique	Performance
Taux de transfert	110 Mo par seconde
Cache	32 Mo
Nbre de disques	5 (10 têtes)
Nombre total secteurs (512K)	$2 \times 10^9$
Nombre de cylindres	50 000
Vitesse de rotation	10 000 rpm (rot. par minute)
Délai de latence	En moyenne 3 ms
Temps de positionnement moyen	5.2 ms
Déplacement de piste à piste	0.6 ms

## Quelques calculs...

À partir des spécifications, on calcule :

- 1 nombre de secteurs par face ;
- 2 nombre (moyen) de secteurs par piste ;
- 3 nombre (moyen) d'octets par piste et par cylindre ;
- 4 temps de rotation, et donc délai de latence ;
- 5 délai de transfert pour 1 ou  $n$  blocs.

On en déduit les temps *minimal*, *maximal* et *moyen* de lecture.

Le temps d'accès peut varier en fonction de l'organisation des blocs, de l'ordre d'accès aux blocs et de la conservation de certains blocs en mémoire.

# Le principe de localité et ses conséquences

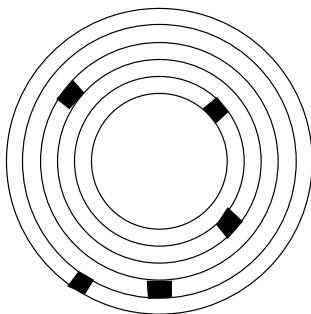
**Principe de localité** : l'ensemble des données utilisées par une application pendant une période donnée forme souvent un groupe bien identifié.

- ❶ **Localité spatiale** : si une donnée  $d$  est utilisée, les données proches de  $d$  ont de fortes chances de l'être également ;
- ❷ **Localité temporelle** : quand une application accède à une donnée  $d$ , il y a de fortes chances qu'elle y accède à nouveau peu de temps après.
- ❸ **Localité de référence** : si une donnée  $d_1$  référence une donnée  $d_2$ , l'accès à  $d_1$  entraîne souvent l'accès à  $d_2$ .

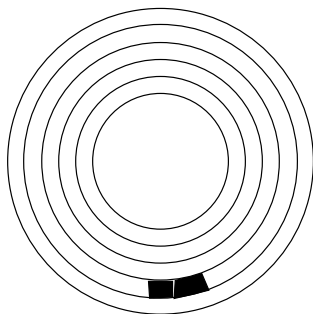
Les systèmes exploitent ce principe en déplaçant dans la hiérarchie des mémoires des groupes de données proches de la donnée utilisée à un instant  $t \Rightarrow$  le pari est que l'application accèdera à d'autres données de ce groupe.

## Localité spatiale : placement des blocs

Les données qui doivent être lues *ensemble* doivent être *proches* sur le disque.



(a)



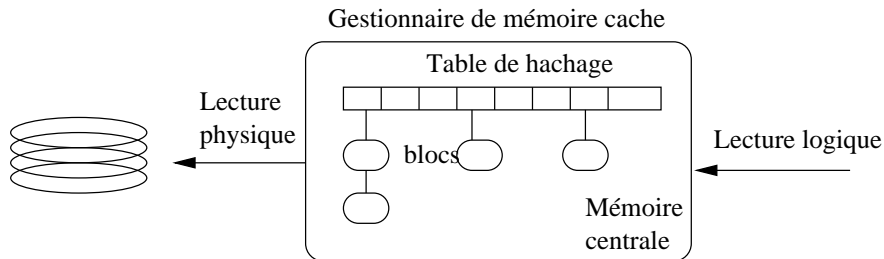
(b)

Proximité sur un disque : (1) dans le même bloc, (2) dans deux blocs consécutifs, (3) sur la même piste, (4) sur le même cylindre, (5) fonction de l'éloignement des pistes.

Très important en cas de lecture à l'avance (appelée *read ahead* ou *prefetching*).

## Localité spatiale et temporelle : mémoire cache

Le SGBD garde en mémoire les blocs après utilisation, afin d'exploiter à la fois la localité spatiale (les autres données du bloc) et la localité temporelle.



Un des paramétrages des SGBD consiste à leur attribuer une partie de la mémoire centrale qui sert – en grande partie – de *cache*.

## Lectures logiques, lectures physiques : le *Hit ratio*

Le paramètre qui mesure l'efficacité d'une mémoire cache est le *hit ratio*, défini comme suit :

$$\textit{hit ratio} = \frac{\textit{nb de lectures logiques} - \textit{nb lectures physiques}}{\textit{nb de lectures logiques}}$$

Si toutes les lectures logiques (demande de bloc) aboutissent à une lecture physique (accès au disque), le *hit ratio* est 0.

S'il n'y a aucune lecture physique, le *hit ratio* est de 1.

Le HR est le premier indicateur à surveiller: il devrait être d'au moins 0.8, voire 0.9.

# Un peu de détente : exercices!

On considère un fichier de 1 Go et un buffer de 100 Mo.

- quel est le *hit ratio* en supposant que la probabilité de lire les blocs est uniforme ?
- même question, en supposant que 80 % des lectures concernent 200 Mo, les 20 % restant étant répartis uniformément sur 800 Mo ?
- avec l'hypothèse précédente, jusqu'à quelle taille de buffer peut-on espérer une amélioration significative du *hit ratio* ?

# Technologie RAID

Stockage sur disque : point *sensible* des SGBD, pour les performances *et* pour la sécurité. On estime :

- risque de panne pour un disque pendant les prochains 10 ans = 1 ;
- risque de panne pour *deux* disques pendant les prochains 5 ans = 1 ;
- risque de panne pour *cent* disques pendant le prochain mois = 1 !

La technologie RAID vise principalement à limiter le risque dû à une défaillance.



# Les niveaux RAID

Il existe 7 niveaux, numérotés de 0 à 6.

- ❶ niveau 0 : rien !
- ❷ niveau 1 : duplication – brutale – des données ;
- ❸ niveau 4 : reprise sur panne basée sur la parité ;
- ❹ niveau 5 : répartition de l'information de parité ;
- ❺ niveau 6 : prise en compte de défaillances simultanées.

# RAID 1

Principe trivial : on a deux disques, sur lesquels on écrit et lit en parallèle

- deux fois plus coûteux ;
- pas d'amélioration notable des performances.

## RAID 4

Hypothèses : on dispose de  $n$  disques, tous de même structure. On introduit un *disque de contrôle* contenant *la parité*. Exemple :

D1: 11110000

D2: 10101010

D3: 00110011

Chaque bit du disque de contrôle donne la parité, pour le même bit, des autres disques :

DC: 01101001

⇒ permet la reprise sur panne en cas de défaillance d'un seul disque.

## Performances du RAID 4

- **Lectures** : elles s'effectuent de manière standard sur les disques de données ;
- **Répartition** : le RAID 4 distribue les blocs sur les  $n$  disques, ce qui permet d'effectuer des lectures en parallèle.

**Écritures** il faut tenir compte des versions *avant* et *après* mise à jour d'un octet. Exemple :

avant : 11110000

après : 10011000

Octet de mise à jour : 01101000.

On doit inverser les bits 2, 3, 5 du disque de parité.

## RAID 5 et RAID 6

Problème 1 du RAID 4 :  $n$  fois plus d'écritures sur le disque de contrôle.

Solution RAID 5 : les blocs de parité sont distribués sur les  $n + 1$  disque.

Exemple :

D1: 11110000

D2: 10101010

D3: 00110011

DC: 01101001

Problème 2 : et si deux disques tombent en panne en même temps ? RAID 6 : un codage plus sophistiqué permet de récupérer deux défaillances simultanées.

# Fichiers

Une base de données = un ou plusieurs **fichiers**.

Un fichier = un ou plusieurs **blocs**.

Le SGBD choisit **l'organisation des fichiers** :

- 1 l'espace est-il bien utilisé ?
- 2 est-il facile et efficace de faire une **recherche** ?
- 3 est-il facile et efficace de faire une **mise à jour** ?
- 4 les données sont-elles correctement représentées, et en sécurité ?

La plupart des SGBD prennent en charge la gestion des fichiers et de leur contenu.

# Enregistrements

Un enregistrement = une suite de *champs* stockant les valeurs des attributs.

Type	Taille en octets
INTEGER	4
FLOAT	4
DOUBLE PRECISION	8
DECIMAL ( $M, D$ )	$M, (D+2 \text{ si } M < D)$
CHAR( $M$ )	$M$
VARCHAR( $M$ )	$L+1 \text{ avec } L \leq M$

## Tailles variables et valeurs NULL

Si tous les champs sont de taille fixe et ont une valeur : pas de problème :

En pratique : certains champs ont une taille variable ou sont à NULL.

- pour les champs de taille variable : on précède la valeur par la taille exacte ;
- pour les valeurs NULL : on peut indiquer une taille 0 (Oracle) ; on peut créer un « masque » de bits.

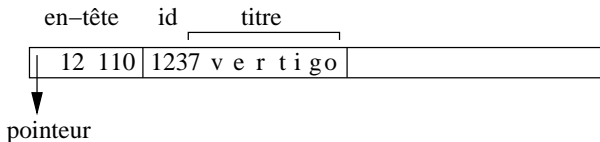
La norme n'impose pas de choix, mais dans tous les cas il faut gérer une *information complémentaire* sur les enregistrements.



## En-tête d'enregistrement

Les informations complémentaires sont stockées dans l'en-tête d'un enregistrement. Exemple :

- table *Film* (id INT, titre VARCHAR(50), année INT)
- Enregistrement (123, 'Vertigo', NULL)



Le pointeur donne par exemple l'adresse du *schéma* de l'enregistrement (de la table).

# Blocs et enregistrements

- on essaie d'éviter qu'un enregistrement chevauche deux blocs ;
- on veut envisager le cas où la taille d'un enregistrement varie ;
- **on affecte une adresse à un enregistrement** pour pouvoir y accéder en une seule lecture ;
- on détermine une méthode pour gérer un **déplacement**.

Les deux derniers problèmes sont particulièrement cruciaux pour *l'indexation des enregistrements*.

## Enregistrements de taille fixe

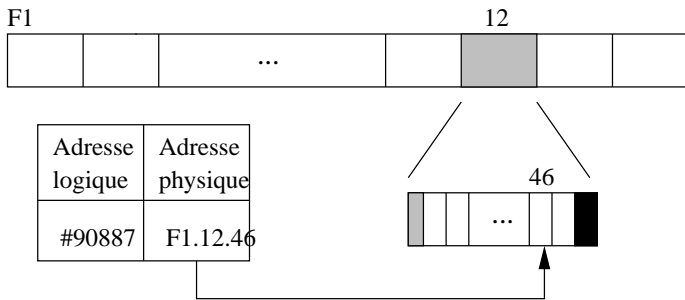
- Pour une taille de bloc  $B$  et d'enregistrement  $E$ , on a  $\lfloor B/R \rfloor$  enregistrements par bloc ;
- Exemple :  $B = 4096$ ,  $E = 84$ ,  $\lfloor \frac{4096-100}{84} \rfloor = 47$  enregistrements par bloc.
- Le 563 est dans le bloc  $\lfloor 563/47 \rfloor + 1 = 12$ ,
- Le bloc 12 contient les enregistrements  $11 \times 47 + 1 = 517$  à  $12 \times 47 = 564$  ;
- le 563 est donc l'avant-dernier du bloc.

NB : on peut aussi le référencer par le fichier + le bloc + le numéro interne.  
Soit F1.12.46.

# Enregistrements de taille variable

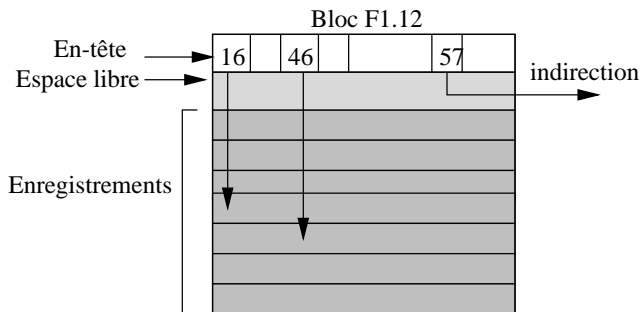
- un enregistrement peut changer de taille, avec réorganisation interne du bloc ;
- un enregistrement peut être déplacé.

Première solution indirection :



## Solution intermédiaire

- on a un *adressage physique* pour le bloc ;
- au sein du bloc on a une *indirection* pour adresser les enregistrements.



Solution adoptée par Oracle.

# Réorganisation du stockage

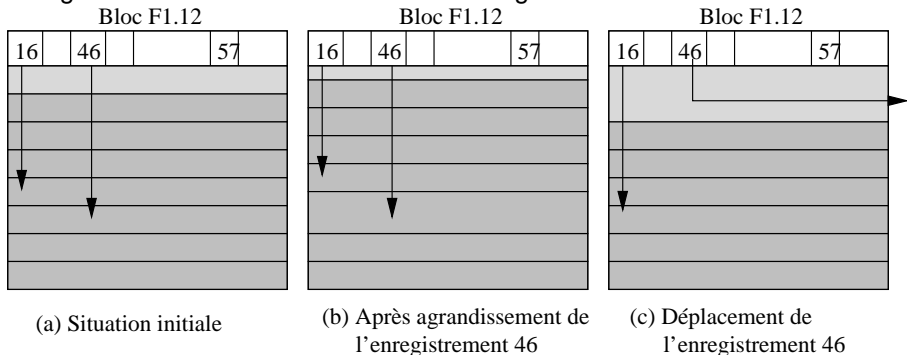
L'adressage indirect donne une certaine souplesse :

- ❶ si un enregistrement s'agrandit, mais qu'il reste de la place dans le bloc :
  - ❶ On lit le bloc en mémoire (coût : 1 lecture)
  - ❷ On modifie l'organisation interne : l'enregistrement occupe une partie de la place libre ; les autres sont éventuellement déplacés *localement* ; la table d'adressage est modifiée (coût : 0!)
  - ❸ on écrit le bloc.
- ❷ sinon on déplace l'enregistrement *et on crée un chaînage dans l'en-tête du bloc*.

La création de chaînage pénalise les performances  $\Rightarrow$  si possible laisser de l'espace libre dans un bloc.

# Exemple d'évolution avec chaînage

On agrandit deux fois successivement l'enregistrement F1.12.46.



## Recherche dans un fichier

En l'absence d'index approprié, le seul moyen de rechercher un enregistrement est de parcourir **séquentiellement** le fichier.

La performance du parcours est conditionnée par :

- la bonne utilisation de l'espace (idéalement tous les blocs sont pleins) ;
- le stockage le plus contigu possible (même piste, même cylindre, etc).

On peut faire beaucoup mieux si le fichier est trié sur la clé de recherche (recherche par dichotomie).



# Recherche par dichotomie

Algorithme (pour un fichier avec  $n$  blocs) :

- 1 On lit le bloc situé au milieu du fichier ;
- 2 Si on trouve l'enregistrement : terminé ;
- 3 Sinon on sait qu'il est soit à droite (si la clé cherchée est supérieure à celles du bloc) ou à gauche : on recommence en (1).

Coût (au pire) : nombre de fois où on peut diviser  $n$  en 2.

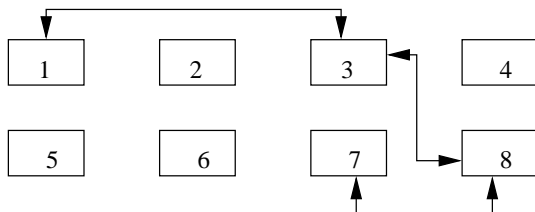
Soit le plus petit  $m$  tel que  $2^m \geq n$ , donc  $m = \lceil \log_2 n \rceil$ .

# Mise à jour d'un fichier

Pour les UPDATE et DELETE : on se ramène à une recherche.





Pour les INSERT, problème : on ne peut pas se permettre de parcourir le fichier à chaque fois !

Première solution : liste doublement chaînée.



## Mise à jour (2)

Seconde solution : garder une table des pages libres.

libre ?	espace	adresse	
O	123	1	
N		2	
		...	
			
O	1089	7	
			

Avantage : on peut savoir facilement où trouver l'espace nécessaire.

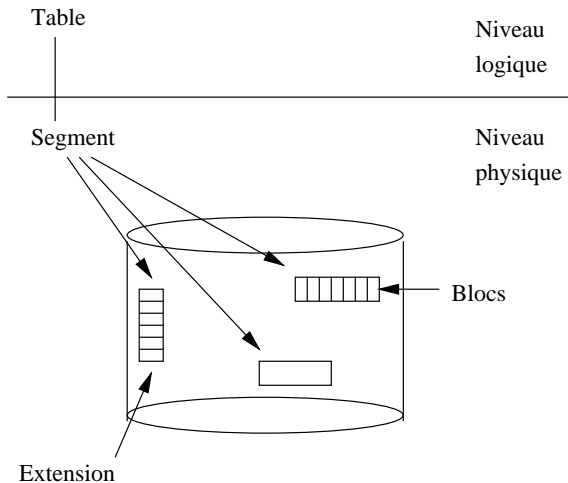
# Oracle

## Principales structures physiques dans ORACLE :

- ❶ **bloc**: unité physique d'E/S.  
La taille d'un bloc ORACLE est un multiple de la taille des blocs du système sous-jacent.
- ❷ **Extension**: ensemble de blocs contigus contenant un même type d'information.
- ❸ **Segment**: ensemble d'extensions stockant un objet logique (une table, un index ...).

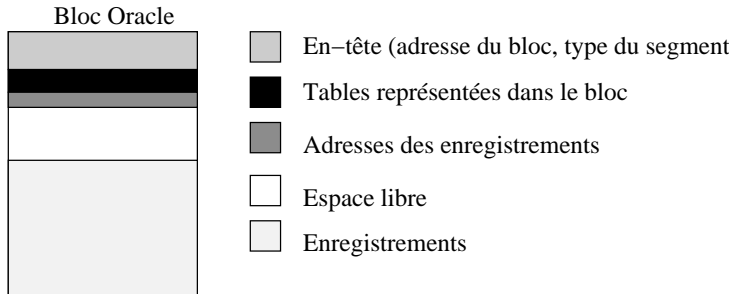
Le paramétrage du stockage des données (tables et index) est spécifié dans un *tablespace*

# Tables, segments, extensions et blocs



# Les blocs Oracle

La structure d'un bloc repose sur un adressage physique/logique, chaque enregistrement ayant une adresse interne.



Un chaînage est créé quand il faut déplacer un enregistrement.

# Gestion de l'espace

- PCTFREE donne l'espace libre à préserver au moment de la création d'une table ou d'un index.
- PCTUSED indique à quel moment le bloc est disponible pour des insertions.

Oracle maintient un répertoire (?) des blocs disponibles pour insertions.

Exemple :

- 1 PCTFREE = 30% et PCTUSED=70%
- 2 PCTFREE = 10% et PCTUSED=80%

Le second choix est plus efficace, mais plus risqué et plus coûteux.

# Stockage et adressage des enregistrements

En règle générale un enregistrement est stocké dans un seul bloc.

L'adresse physique d'un enregistrement est le *ROWID*:

- 1 Le numéro de la page dans le **fichier**.
- 2 Le numéro du n-uplet dans la page.
- 3 Le numéro du fichier.

Exemple : 00000DD5.000.001 est l'adresse du premier n-uplet du bloc DD5 dans le premier fichier.



# Extensions et segments

L'**extension** est une suite de blocs contigus. Le **segment** est un ensemble d'extensions contenant un objet logique.

Il existe quatre types de segments :

- 1 Le segment de données.
- 2 Le segment d'index.
- 3 Le *rollback segment* utilisé pour les transactions.
- 4 Le segment temporaire (utilisé pour les tris).

Moins il y a d'extensions dans un segment, plus il est efficace.

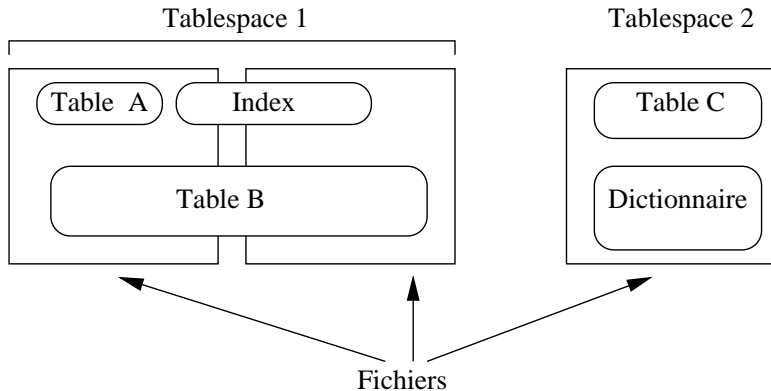
# Les *tablespaces*

Une base est divisée par l'administrateur en *tablespace*. Chaque *tablespace* consiste en un (au moins) ou plusieurs fichiers.

La notion de *tablespace* permet :

- ❶ De contrôler l'emplacement physique des données. (par ex. : le dictionnaire sur un disque, les données utilisateur sur un autre) ;
- ❷ de régler l'allocation de l'espace (extensions) ;
- ❸ de faciliter la gestion (sauvegarde, protection, etc).

## Exemple de *tablespaces*



## Création de *tablespaces*

```
CREATE TABLESPACE TB1
  DATAFILE 'fichierTB1.dat' SIZE 50M
  DEFAULT STORAGE (
    INITIAL 100K, NEXT 40K, MAXEXTENTS 20,
    PCTINCREASE 20);
```

```
CREATE TABLESPACE TB2
  DATAFILE 'fichierTB2.dat' SIZE 2M
  AUTOEXTEND ON NEXT 5M MAXSIZE 500M
  DEFAULT STORAGE (INITIAL 128K
    NEXT 128K MAXEXTENTS UNLIMITED);
```

# Maintenance d'un *tablespace*

Quelques actions disponibles sur un *tablespace* :

- 1 On peut mettre un *tablespace* hors-service.  
`ALTER TABLESPACE TB1 OFFLINE;`
- 2 On peut mettre un *tablespace* en lecture seule.  
`ALTER TABLESPACE TB1 READ ONLY;`
- 3 On peut ajouter un nouveau fichier.  
`ALTER TABLESPACE ADD DATAFILE  
'fichierTB1-2.dat' SIZE 300 M;`

## Affectation de tables à un *tablespace*

On peut placer une table dans un *tablespace*. Elle prend alors les paramètres de stockage de ce dernier.

On peut aussi remplacer certaines valeurs.

```
CREATE TABLE Film (...)  
    PCTFREE 10  
    PCTUSED 40  
    TABLESPACE TB1  
    STORAGE ( INITIAL 50K  
              NEXT 50K  
              MAXEXTENTS 10  
              PCTINCREASE 25 );
```