Bases Relationnelles vs Bases NoSQL

Exercices

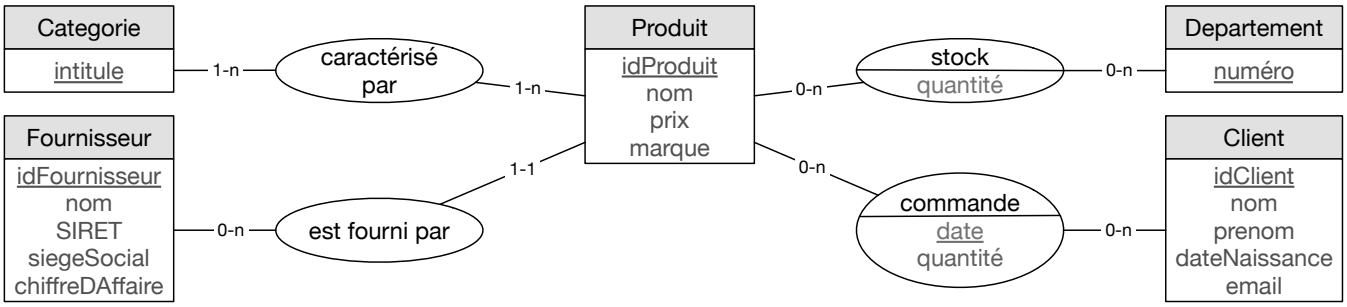
ESILV
nicolas.travers (at) devinci.fr

Chapitre 1

Schéma relationnel vs NoSQL

Ce chapitre est construit de manière à faire comprendre les problématiques liées au passage d'une base de données relationnelle vers une base NoSQL. Etant donné un contexte fortement distribué, nous serons confrontés aux problèmes de jointures traditionnelles du relationnel.

Dans ce chapitre, nous utiliserons une base gérant une vente de produits à des clients sur internet. Le schéma Entité/Association est le suivant :



Par ailleurs, nous avons quelques statistiques associées qui nous servirons de base pour les calculs :

— 10^7 clients, 10^4 produits, 10^9 produits commandés, 100 départements ;

— Un client fait en moyenne 33 commandes contenant chacune 3,3 produits différents ;

— 1 à 5 catégories par produit (en moyenne 2) ;

— 50 produits de marque "Apple", répartis sur tous les départements.

1.1 Relationnel vers Documents JSon

JSon (JavaScript Object Notation) est un modèle de représentation de données semi-structurées très utilisé sur le Web. Il est notamment utilisé dans les systèmes NoSQL orientés documents.

Dans cette section, nous allons étudier la transformation d'un schéma relationnel vers des documents JSon. Nous prendrons pour exemple la table `Produit`.

1.1.1 Donner pour l'entité `Produit` la correspondance directe en JSon sous forme d'exemples de documents (nous n'avons pas de schéma JSon) ;

1.1.2 Le prix doit être détaillé avec une devise et un taux de TVA ;

1.1.3 Les catégories sont indépendantes les unes des autres. Fusionner les deux entités dans le document JSon ;

1.1.4 Le fournisseur d'un produit est fréquemment interrogé avec le produit.

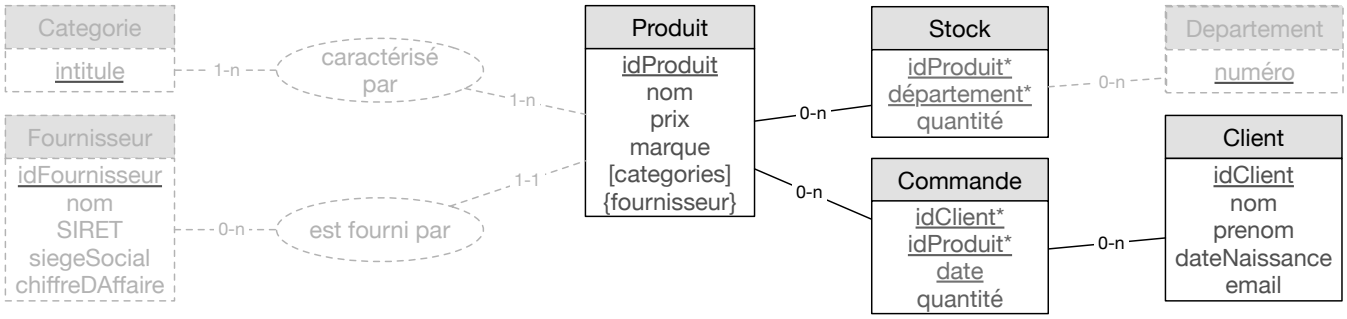
Correction :

```
{
  "_id" : 1,
  "nom" : "iPhone 6",
  "prix" : {"tarif" : 699.99, "devise" : "EUR", "TVA": 20.0},
  "categories" : ["Téléphone", "Smartphone", "Appareil Photo", "Caméra"],
  "fournisseur" : {
    "Nom" : "FNAC",
    "SIRET" : 123456789,
    "siegeSocial" : "La Défense",
    "chiffreDAffaire" : 100000000
  },
  "marque" : "Apple"
}
```

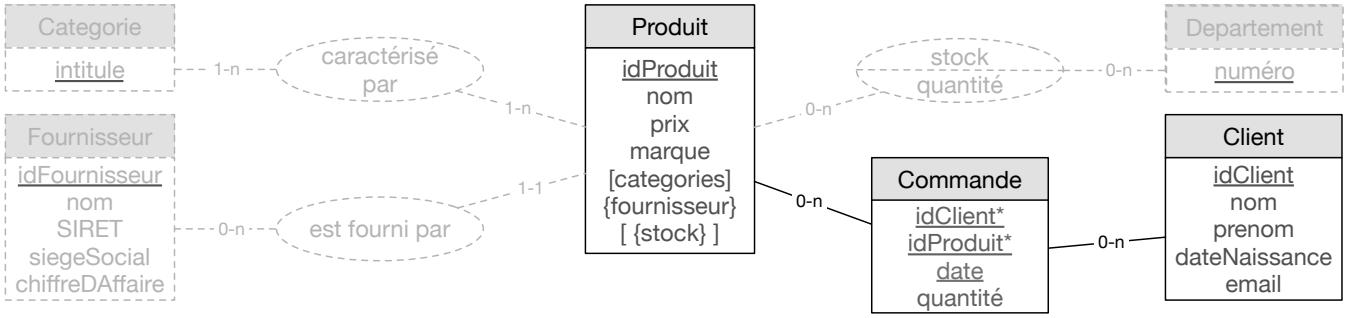
1.2 Dénormalisation

Nous souhaitons éviter de faire de trop nombreuses communications réseaux pour des requêtes soumises à notre base de données. Pour ce faire, proposez un nouveau schéma Entité/Association en appliquant les fusions nécessaires en fonction des informations fournies ci-dessous. Vous y associerez un document JSon exemple.

1.2.1 Au vu des informations données dans la section 1.1, donner le nouveau schéma Entité/Association. Nous noterons en plus que les numéros de département sont indépendant les uns des autres ;
Correction :

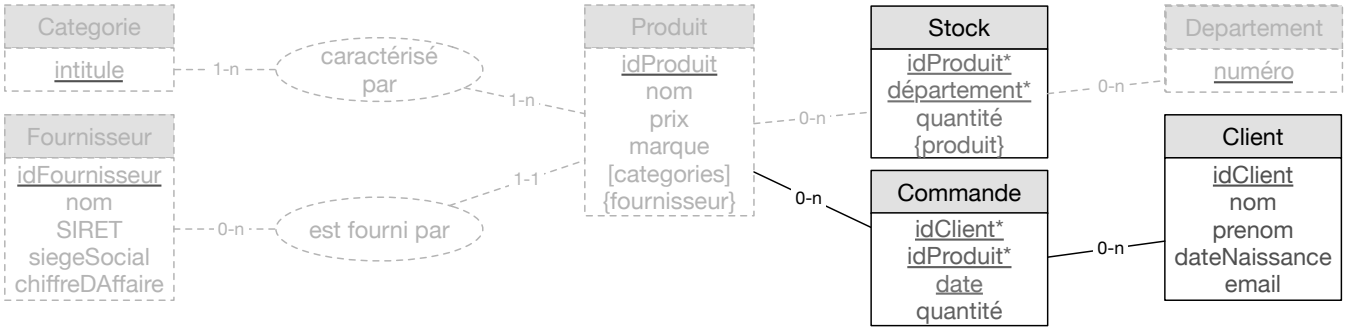


1.2.2 Les produits sont fréquemment interrogés pour y consulter le stock ;
Correction :



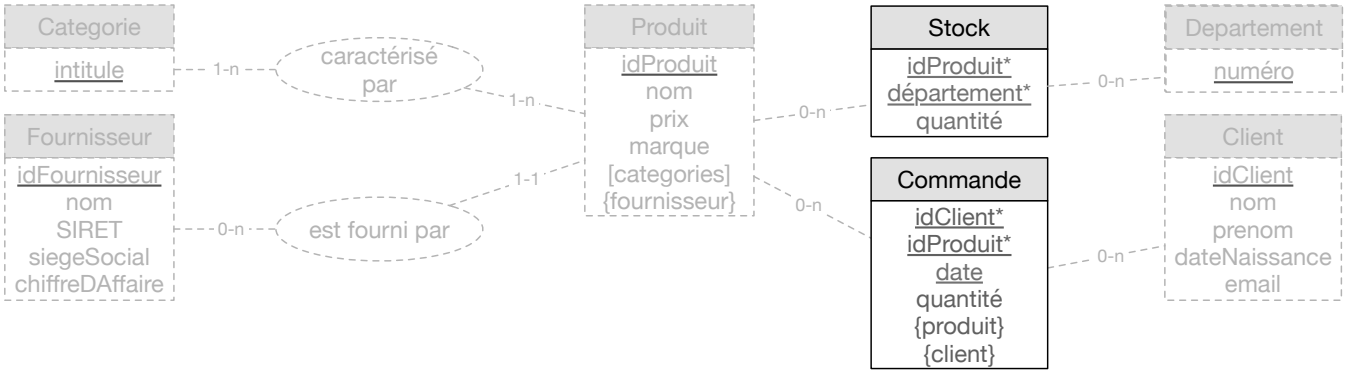
```
{
  "id_produit" : 1, "nom" : "iPhone", "prix" : 699.99, "marque" : "Apple",
  "stock" : [
    {"departement" : 92, "quantite" : 10},
    {"departement" : 75, "quantite" : 1000},
    {"departement" : 12, "quantite" : 5}
  ]
}
```

1.2.3 Le stock d'un département est fréquemment interrogé, associé aux produits ;
Correction :




```
{
  "id_produit" : 1, "departement" : 92, "quantite" : 10,
  "produit" : {"nom" : "iPhone", "prix" : 699.99, "marque" : "Apple"}
}
```

1.2.4 Les commandes sont fréquemment interrogées pour y retrouver le produit et le client ;
Correction :




```
{
  "id_client" : 125,  "date" : "2017-05-04", "quantite" : 1,
  "produit":{"id_produit":1, "nom":"iPhone", "prix":699.99, "marque":"Apple"},
  "client" : {"nom":"Dupont", "prenom":"Toto"}
}
```

1.2.5 Quels sont les problèmes liés à cette dénormalisation ?
Correction :

— Le principal problème est lié à la mise à jour des données. En effet, à partir du moment où une donnée est répliquée plusieurs fois dans différents documents, il est nécessaire de les mettre à jour pour garder

la cohérence des données (catégories d'un produit identique, changement de marque...). De fait, plus l'information est dupliquée, plus les mises à jour vont couter cher.
A contrario, si les données ne sont pas mises à jour, il n'y aura pas de cohérences des données. Le système d'information doit tenir compte des versions des valeurs présentes dans la base de données.

— La disponibilité du système est dépendant des mises à jour. En effet, si l'on considère qu'une mise à jour doit être synchronisée pour garantir la cohérence globale, le nombre de mise à jour risque de réduire considérablement la disponibilité du système. Un mode asynchrone diminue la cohérence mais permet de garantir cette disponibilité (Théorème de CAP).

1.3 Jointures

Les systèmes NoSQL ne favorisent pas les jointures du fait de nombreuses communications réseaux générées pour regrouper les données corrélées. Nous allons comparer différentes solutions pour en estimer le coût.

Pour cela nous utiliserons les 4 requêtes de jointure suivantes :

R1 Stock (liste des noms et prix de produits, ainsi que leur quantité) du département n°92 ;

R2 Distribution des produits (nom et quantité) de marque "Apple" dans les départements ;

R3 Nom du produit le plus commandé par le client n°125 ;

R4 Les 100 noms de produits les plus commandés (somme des quantités).

Le coût de chaque requête prendra en compte le nombre de messages échangés sur le réseau. Il se décompose en :

— Messages "**Aller**", interrogeant des données spécifiques ou l'ensemble des serveurs. Cette partie peut être considérée également comme le "*Map*"

— Messages "**Retour**" donnant le nombre de documents retournés

— "**Shuffle**" pour le regroupement des données en préparation d'un "*Reduce*". Les données sont redistribuées sur le réseau selon une clé de regroupement.

— Messages "**Reduce**" correspond au nombre de messages retournés par l'opération *Reduce*. Il est souvent égal au nombre de clé de regroupement.

s et **retour**s.

Les collections sont réparties sur 1000 serveurs. Les choix de répartitions dans les serveurs seront précisés pour chaque requête.

1.3.1 Jointure applicative

Calculer le nombre de messages échangés sur le réseau avec l'algorithme suivant :

1) On interroge la première collection (`Aller` : nb de serveurs interrogés, `Retour` : nb de documents qui répondent)

2) On interroge la seconde collection avec le résultat de la première (`Aller` : nb de documents retournés précédemment, `Retour` : nb de documents qui répondent)

Si une agrégation est nécessaire, il sera effectué par un regroupement par clé (i.e. Map/Reduce). Le nombre de messages sera réparti en "regroupement" et retour de groupement (après agrégation).

Les choix de stockage pour chaque requête est le suivant :

R1 La collection "stock" est placée sur le réseau grâce au département (hachage sur la clé "département"), et les "produits" selon leur identifiant de produit ;
Correction :

— **Aller : 1**. Le hachage permet d'interroger un seul serveur (département = 92)

— **Retour : 10 000** produits avec leur quantité

— **Aller : 10 000** requêtes (chaque produit)

— **Retour : 10 000** nom et prix

— **Total : 30 001** échanges sur le réseau

R2 Stockage identique au précédent ;
Correction :

— **Aller : 1000**. Interrogation de tous les serveurs pour récupérer les produits "Apple" (aucune connaissance sur leur emplacement - si index sur la marque, alors il y aura 50 requêtes)

— **Retour : 50**. Récupération des 50 produits "Apple".

— **Aller : 5 000**. Pour chaque produit (50), les 100 départements sont interrogés : 50×100

— **Total : 6 050**

R3 Les commandes sont placées grâce à l'identifiant du client (hachage). Le comptage groupé par `idproduit` sera effectué de manière distribuée (i.e. MapReduce), il y a 20 idproduits différents.
Correction :

— **Aller : 1**. On retrouve les commandes du client sur le serveur correspondant ($33 \times 3,33 = 100$ produits commandés).

— **Shuffle : 100**. 100 produits commandés envoyés pour regroupement (à 20 serveurs différents)

— **Reduce : 20**. Chaque groupe (serveur) retourne la valeur agrégée

— **Aller + Retour : 1 + 1**. On interroge le produit correspondant (présent sur un seul serveur).

— **Total : 123**.

Si les produits commandés n'étaient pas répartis en fonction du client mais du produit, cela couterait beaucoup plus cher ! (min 10^4)

R4.1 Même stockage, on groupe par `idclient`.
Correction :

— **Aller : 1000**. Tous les serveurs sont interrogés

— **Shuffle :** 10^9 produits commandés (envoi de id-produit+quantité).
Il est possible d'optimiser cette étape en effectuant avant des regroupements locaux par `idproduit` sur chaque serveur. Maximum 10^4 idproduit par serveur, donc $10^4 \times 1000 = 10^7$ messages.

— **Reduce :** 10^4 produits retournés, chacun fait la somme des quantités. On prend les 100 premiers au niveau de l'application.

— **Aller + Retour :** $100 + 100$ Noms des 100 produits les plus commandés

— **Total :** $10^9 + 1,1 \times 10^4 + 200$**.**

R4.2 Stockage par `idproduit`, le groupement se fera alors localement (pas de messages pour regroupement).
Correction :

— **Aller : 1000**. Tous les serveurs sont interrogés

— **Shuffle : 0**. Tous les groupements sont réalisés localement (même clé de hachage).

— **Reduce : 10 000** produits retournés

— **Aller + Retour : 100 + 100**. Noms des 100 produits les plus commandés

— **Total : 11 200.**

1.3.2 Jointure par Map/Reduce

Une autre possibilité est de réunir les deux types de documents dans une seule et même collection. Les deux "collections" co-existent donc et peuvent répondre aux requêtes.

Pour effectuer la jointure, il faut dans le `MAP` émettre les documents avec la clé de jointure et les données demandées. Dans le `REDUCE`, les données sont jointes pour produire le résultat demandé. Pour améliorer le coût de la jointure, les données sont regroupées grâce à la clé de la jointure.

A chaque requête, tous les serveurs sont interrogés, le reduce est effectué localement grâce à la clé de regroupement sur la jointure. Le résultat est envoyé (s'il existe).

Calculer pour chaque requête le coût en nombre de messages :

R1 Produit et Stock, regroupés sur le numéro de département
Correction :

— **Aller/Map : 1000**. Tous les serveurs reçoivent le MAP

— **Shuffle : 0**. Données regroupées localement, sur prédicat de jointure.

— **Retour : 10 000**. Chaque produit du département 92 répond à la requête.

— **Total : 11 000**. Gain de 63% (par rapport à 1.3.1.R1)

Cette solution n'est toutefois pas recommandée car seuls 100 serveurs sont utilisés pour la répartition des données. Avec une distribution inégale, la charge de traitement local pourrait être trop lourde.

R2 Produit et Stock, regroupés sur le numéro de département
Correction :

— **Aller/Map : 1000**. Tous les serveurs reçoivent le MAP

— **Shuffle : 0**. Données regroupées localement, sur prédicat de jointure.

— **Reduce : 5 000**. Les 50 produits "Apple" répondent pour chacun des 100 départements.

— **Total : 6 000**. Gain de 0,8% (par rapport à 1.3.1.R2)

Idem, le regroupement sur le département est une mauvaise solution. On gagne le coût de regroupement des produits Apple par département.

R3 Produit et Commande, regroupés sur l'identifiant de produit
Correction :

— **Aller/Map : 1000**. Tous les serveurs reçoivent le MAP

— **Shuffle : 0**. Données regroupées localement, s'il y a au moins un produit pour le client 125, la somme des quantités est calculée.

— **Reduce : 20**. Les 20 produits différents commandés (correspondant à 20 serveurs) sont retournés pour calculer du max.

— **Total : 1 020**. Perte de 87% (par rapport à 1.3.1.R3). Ceci est du au fait que l'on interroge tous les serveurs, tandis qu'on pourrait regrouper les données par client. Toutefois, cette solution est 10 fois meilleure que sans regroupement de données.

La répartition par "idproduit" est un bon choix car dépasse largement le nombre de serveurs. Toutefois, attention à la répartition des commandes associées qui peuvent créer un déséquilibre.

R4 Produit et Commande, regroupés sur l'identifiant de produit
Correction :

— **Aller/Map : 1000**. Tous les serveurs reçoivent le MAP

— **Shuffle : 0**. Données regroupées localement, calcul de la somme des quantités par produit. Jointure pour obtenir le nom du produit associé.

— **Reduce : 10 000**. Tous les produits répondent. Les 100 plus grands seront calculés au niveau de l'application.

— **Total : 11 000**. Gain de 0,1% (par rapport à 1.3.1.R4.2).

1.3.3 Dénormalisation

En utilisant des collections contenant des documents normalisés (ceux obtenus dans la section 1.2), nous pouvons éviter d'avoir à effectuer des jointures.

Pour chaque requête, calculer le nombre de messages générés sur chacune des collections de documents générées, avec les répartitions suivantes :

R1 `Produit-Stock` : regroupement sur "idproduit"
Correction :

— **Aller : 1000**. On interroge les 1000 serveurs

— **Retour : 10 000**. Chacun des 10^4 produits retourne le stock du département correspondant.

— **Total : 11 000**. Idem que 1.3.2.R1

R1 `Stock-Produit` : regroupement sur "département"
Correction :

— **Aller : 1**. On interroge le serveur contenant "92".

— **Retour : 10 000**. Chacun des 10^4 produits stockés retourne le document intégralement.

— **Total : 10 001**. Gain de 9% par rapport à 1.3.2.R1

Toutefois, la clé de regroupement n'est pas recommandée car il n'y a que 100 départements. Donc, uniquement 100 serveurs pourront être utilisés maximum (avec une distribution inégale).

R2 `Produit-Stock` : regroupement sur la "marque"
Correction :

— **Aller : 1**. On interroge le serveur contenant "Apple".

— **Retour :** 5000. Chacun des 50 produits "Apple" produit un stock par département : 50×100.

— **Total : 5 001**. Gain de 16% par rapport à Idem que 1.3.2.R2

Etant donné qu'il y a plus de marques que de départements, la répartition des documents sera meilleure que la soluion précédente (mais limitée au nombre de marques différentes).

R2 `Stock-Produit` : regroupement sur la "marque"
Correction :

— **Aller : 1**. On interroge le serveur contenant "Apple".
— **Retour :** 5000. Chacun des 5 000 produits stockés sont retournés.
— **Total : 5 001**. Gain de 16% par rapport à Idem que 1.3.2.R2 Cette solution est préférable à la précédente car elle passe mieux à l'échelle en nombre de produits et sur plus de possibilités de requêtes d'agrégation.

R3 `Produit-Commande` : regroupement sur "id"

Correction :

— **Aller :** 10^4. Tous les produits sont interrogés. Seuls ceux contenant "idclient = 125" sont retournés. A chaque fois, la somme des quantités est effectuée sur le produit.
— **Retour : 20**. Les 20 produits différents commandés par le client sont retournés.
— **Total : 10 020**. Perte de 70% !!! On ne peut plus répartir les données par client (on retrouve le pire des cas sans répartition)

R3 `Commande-Produit` : regroupement sur "idclient". Le comptage groupé par `idproduit` sera effectuée de manière distribuée (i.e. MapReduce).

Correction :

— **Aller : 1**. On interroge le serveur contenant le client 125.
— **Shuffle : 100**. 100 produits commandés envoyés pour regroupement (20 serveurs différents)
— **Reduce : 20**. Les 20 produits différents commandés par le client sont retournés.
— **Total : 121**. Gain de 1,2% Gain de 16% par rapport à Idem que 1.3.1.R3 (il n'y a plus les informations du produit le plus commandé à aller chercher)

R4 `Produit-Commande` : regroupement sur "id"

Correction :

— **Aller : 1000**. Tous les serveurs sont interrogés. La somme des quantités est calculée directement sur le document.
— **Retour : 10 000** . Chaque produit répond pour ne garder que les 10 premiers (fait sur un serveur ou localement).
— **Total : 11 000** . Gain identique à Gain 1.3.2.R4. Les regroupements locaux ont déjà été optimisés grâce à la dénormalisation. On y gagne l'accès aux noms des produits concernés.

R4 `Commande-Produit` : regroupement sur "idproduit".

Correction :

— **Aller : 1000**. On interroge tous les serveurs
— **Shuffle : 0**. Les sommes de quantités par produit sont calculées localement.
— **Reduce : 10 000**. Chaque produit répond à la requête.
— **Total : 11 000**. Idem. La dénormalisation fait gagner le temps de recherche du nom. Cette solution est préférable car elle passe plus facilement à l'échelle que l'autre dénormalisation.

1.3.4 Conclusions

Créer un tableau récapitulant pour chaque type de jointure les coûts en nombre de messages échangés. Que peut-on en conclure ?

Correction :

Le tableau ci-dessous résume pour chaque requête le coût correspondant pour avoir une vision plus globale du coût des jointures. La clé de regroupement est précisée dans chaque case.

Jointure	**Applicative**		**Map/Reduce**		**Dénormalisation**	
R1	departement	30 001	département	11 000	P-S/idproduit	11 000
					S-P/departement	**10 001**
R2	departement	6 050	département	6 000	**P-S/marque**	**5 001**
					S-P/marque	**5 001**
R3	idproduit	123	idproduit	1 020	P-C/id	10 020
					C-P/idclient	**121**
R4	idclient	10^9			**P-C/id**	**11 000**
	idproduit	11 200	**idproduit**	**11 000**	**C-P/idproduit**	**11 000**

Avec ce tableau, nous pouvons conclure les points suivants :

1 - La dénormalisation reste la solution la plus efficace

2 - La dimension la plus grande est généralement préférable pour l'imbrication

3 - Le choix du regroupement des documents est primordial pour l'optimisation soit de l'interrogation directe, soit du *shuffle*.