NOSQL – NORMALISATION / DÉNORMALISATION

NOSQL – NORMALISATION / DÉNORMALISATION

- la modélisation traditionnelle d'une base de données relationnelle a un objectif de normalisation qui vise à éviter à la fois toute redondance et toute perte d'information
- la redondance est évitée en découpant les données et en les stockant indépendamment les unes des autres dans des tables séparées
- la perte d'information est évitée en utilisant un système de référencement basé sur les clés primaires et clés étrangères
- les données sont contraintes par un schéma qui impose des règles sur le contenu de la base

NOSQL – NORMALISATION / DÉNORMALISATION

- il n'y a aucune hiérarchie dans la représentation des entités
- prenons des Films, des Réalisateurs, des Acteurs et des Pays auxquels appartiennent réalisateurs et acteurs
- une entité comme Pays, qui peut être considérée comme secondaire, a droit à sa table dédiée, tout comme l'entité Film qui peut être considérée comme essentielle
- on ne tient pas compte en relationnel de l'importance respective des entités représentées
- la distribution des données dans plusieurs tables est compensée par la capacité de SQL à effectuer des jointures qui exploitent le plus souvent le système de référencement (clé primaire, clé étrangère) pour associer des lignes stockées séparément

NOSQL – NORMALISATION / DÉNORMALISATION

```
create table Artiste (idArtiste integer not null,
                      nom varchar (30) not null,
                      prenom varchar (30) not null,
                      anneeNaiss integer,
                      primary key (idArtiste),
                      unique (nom, prenom);
```

```
create table Realisateur (idReal integer not null,
                          nomReal varchar (30) not null,
                          prenomReal varchar (30) not null,
                          anneeNaissReal integer,
                          primary key (idReal),
                          unique (nomReal, prenomReal);
```

NOSQL – NORMALISATION / DÉNORMALISATION

```
create table Film (idFilm integer not null,
        titre varchar (50) not null,
        annee integer not null,
        idReal integer not null,
        genre varchar (20) not null,
        resume varchar (255),
        codePays varchar (4),
        primary key (idFilm),
        foreign key (idReal) references Realisateur);
```

```
create table Role (idFilm integer not null,
        idActeur integer not null,
        nomRole varchar (30),
        primary key (idActeur,idFilm),
        foreign key (idFilm) references Film,
        foreign key (idActeur) references Artiste);
```

NOSQL – DOCUMENTS ET COLLECTIONS

- en conception relationnelle, on ne trouve que des valeurs dites atomiques, non décomposables
- par exemple, il ne peut y avoir qu'un seul genre pour un film
- si ce n'est pas le cas, il faut (processus de normalisation) créer une table des genres et la lier à la table des films
- cette nécessité de distribuer les données dans plusieurs tables est une lourdeur souvent reprochée à la modélisation relationnelle
- avec un document structuré, il est très facile de représenter les genres comme un tableau de valeurs, ce qui casse la première règle de normalisation

NOSQL – DOCUMENTS ET COLLECTIONS


```
{
  "titre": "Pulp fiction",
  "annee": "1994",
  "genre": ["Action", "Policier", "Comédie"]
  "pays": "USA"
}
```

NOSQL – DOCUMENTS ET COLLECTIONS

Il est également facile de représenter une table par une collection de documents structurés. Voici la table des artistes en notation JSON :

artiste: {"id": 11, "nom": "Travolta", "prenom": "John"},

artiste: {"id": 27, "nom": "Willis", "prenom": "Bruce"},

artiste: {"id": 37, "nom": "Tarantino", "prenom": "Quentin"},

artiste: {"id": 167, "nom": "De Niro", "prenom": "Robert"},

artiste: {"id": 168, "nom": "Grier", "prenom": "Pam"}

NOSQL – DOCUMENTS ET COLLECTIONS

- on peut donc encoder une base relationnelle sous la forme de documents structurés
- chaque document peut être plus complexe structurellement qu'une ligne dans une table relationnelle
- une telle représentation, pour des données régulières, n'est pas du tout efficace à cause de la redondance de l'auto-description : à chaque fois on répète le nom des clés alors qu'on pourrait les factoriser sous forme de schéma relationnel et les représenter indépendamment, ce que fait un système relationnel classique

NOSQL – IMBRICATION DES STRUCTURES

Dans une modélisation relationnelle, nous avons séparé les films et les artistes dans 2 tables distinctes, et nous avons lié chaque film à son réalisateur par une clé étrangère. Grâce à l'imbrication des structures, il est possible avec un document structuré de représenter l'information de la manière suivante :


```
"titre": "Pulp fiction",
"annee": "1994",
"genre": "Action",
"pays": "USA",
"realisateur": {
    "nomReal": "Tarantino",
    "prenomReal": "Quentin",
    "anneeNaissReal": "1963"
}
```


On a imbriqué un objet dans un autre, ce qui ouvre la voie à la **représentation d'une entité par un unique document complet**

NOSQL – IMBRICATION DES STRUCTURES

- nous n'avons plus besoin du système de référencement par clés primaires / clés étrangères, remplacé par l'imbrication qui associe physiquement les entités films et artistes
- prenons l'exemple du film "Pulp Fiction" et son metteur en scène et ses acteurs
- en relationnel, pour reconstituer l'ensemble du film "Pulp Fiction", il suffit de suivre les références entre clés primaires et clés étrangères
- c'est ce qui permet de voir que Tarantino (clé = 37) est réalisateur de Pulp Fiction (clé étrangère idRéal dans la table Film, avec la valeur 37) et joue également un rôle (clé étrangère idArtiste dans la table Rôle)
- tout peut être représenté par un unique document structuré, en tirant parti de l'imbrication d'objets dans des tableaux

NOSQL – IMBRICATION DES STRUCTURES

Nous obtenons une unité d'information autonome représentant l'ensemble des informations relatives à un film.

```
{
  "titre": "Pulp fiction",
  "annee": "1994",
  "genre": "Action",
  "pays": "USA",
  "realisateur": {
      "nomReal": "Tarantino",
      "prénomReal": "Quentin",
      "anneeNaissReal": "1963"
   }
```

imbrication " réalisateur "

NOSQL – IMBRICATION DES STRUCTURES

```
"acteurs": {
  {"prénom": "John",
     "nom": "Travolta",
     "anneeNaiss": "1954",
     "role": "Vincent Vega"},
  {" prénom ": "Bruce",
     " nom ": "Willis",
     " anneeNaiss ": "1955",
     "role": "Butch Coolidge" },
  {"prenom": "Quentin",
     "nom": "Tarantino",
     "anneeNaiss ": "1963",
     "role": "Jimmy Dimmick"}
  }
}
```

imbrication " acteurs "

NOSQL – IMBRICATION DES STRUCTURES : AVANTAGES

- plus besoin de jointures : il est inutile de faire des jointures pour reconstituer l'information puisqu'elle n'est plus dispersée dans plusieurs tables
- plus besoin de transactions : une seule écriture du document suffit pour créer toutes les données du film Pulp fiction alors qu'en relationnel, il faudrait écrire 1 fois dans la table Film, 3 fois dans la table Artiste, 3 fois dans la table Rôle
- une seule lecture suffit pour récupérer l'ensemble des informations
- on s'adapte à la distribution : si les documents sont autonomes, il est très facile des les déplacer pour les répartir au mieux dans un système distribué. L'absence de lien avec d'autres documents donne la possibilité d'organiser librement la collection

NOSQL – IMBRICATION DES STRUCTURES : INCONVÉNIENTS

- hiérarchisation des accès : la représentation des films et des artistes n'est pas symétrique. Les films apparaissent près de la racine des documents, les artistes sont enfouis dans les profondeurs. L'accès aux films est donc privilégié : on ne peut pas accéder aux artistes sans passer par les films … ce qui peut ou non convenir à l'application
- perte d'autonomie des entités : il n'est plus possible de représenter les informations sur un réalisateur si on ne connaît pas au moins un de ses films. Inversement, en supprimant un film (par exemple Pulp Fiction), on risque de supprimer définitivement les données sur un artiste qui n'aurait tourné que dans ce film là
- redondance : la même information est représentée plusieurs fois. Quentin Tarantino sera représenté autant de fois qu'il a tourné de films (ou fait l'acteur dans plusieurs films)

NOSQL – IMBRICATION DES STRUCTURES : INCONVÉNIENTS

- on privilégie, en modélisant les données comme des documents, une certaine perspective de la base de données (ici, les films), ce qui n'est pas le cas en relationnel où toutes les informations sont au même niveau
- avec la représentation ci-dessus, comment connaître par exemple tous les films tournés par Tarantino ? → il n'y a pas vraiment d'autre solution que de lire tous les documents
- ce sont des inconvénients majeurs qui risquent à terme de rendre la base de données inexploitable
- il faut bien les prendre en compte avant de se lancer dans l'aventure du NoSQL

NOSQL – IMBRICATION DES STRUCTURES : SCHÉMA ?

- les systèmes NoSQL ne proposent pas forcément de schéma, ou en tout cas rien d'équivalent aux schémas relationnels
- il existe un gain apparent : on peut tout de suite, sans modélisation, commencer à insérer des documents
- mais rapidement la structure de ces documents change, et on ne sait plus trop ce que l'on a mis dans la base qui peut rapidement devenir une poubelle de données
- si on veut éviter cela, c'est au niveau de l'application effectuant des insertions qu'il faut effectuer la vérification des contraintes qu'un système relationnel peut nativement prendre en charge