Bases de données documentaires et distribuées, http://b3d.bdpedia.fr

Cassandra: étude de cas

Le modèle de données Cassandra

Cassandra organise les données en **tables** dénormalisées

Principales différences avec le modèle relationnel

- Pas de clé étrangère (et pas de jointure)
- Imbrication (*nesting*) de valeurs structurées
- Pas d'indépendance logique/physique : la clé détermine le placement
- Sauf exception, les recherches se font sur la clé

Principe de conception

Pas de jointure, donc pas de symétrie d'accès : on doit organiser les données en fonction des accès applicatifs.

Imbrication de structures

C'est le principe de dénormalisation : on regroupe les données le plus possible dans des lignes pour éviter les jointures.

Une valeur d'attribut peut correspondre :

- à un ensemble de valeurs (non ordonnées) : **SET**
- à une liste de valeurs : **LIST**
- à un dictionnaire : **MAP**
- à un nuplet : **TUPLE**
- à une instance d'un type utilisateur

Les ensembles doivent rester de taille raisonnable

Principes de conception

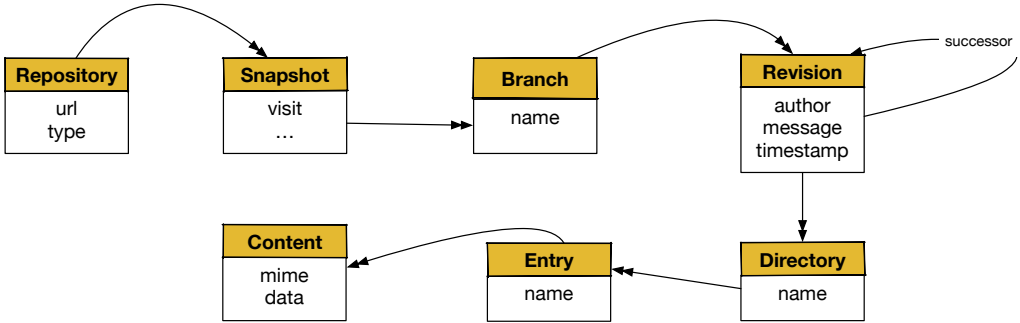
On adopte un des nombreux schémas possibles en fonction des besoins.

- On conçoit les **chemins d'accès**, ou séquence des requêtes émises par une application
- On organise les tables pour que chaque requête s'exécute **localement** et **séquentiellement**
- On peut créer ponctuellement des index ou des vues matérialisées
- Au pire (mais inévitable ?) on multiplie les organisations physiques et donc la redondance

Pour quelle type d'application ?

Acceptable pour une approche WORM (*Write Once, Read many*)

Notre modèle



Pour chaque table on a un point d'accès (pe `'repository'`) et l'identification relative à ce point d'accès (pe `'snapshot'`). On nommera la table `snapshot_by_repository`

Quelques exemples : cherchons les visites

Une première approche purement relationnelle ne fonctionne pas.

Schéma

```
create table Repository
   (Repository_id uuid,
    url varchar,
    description text,
    primary key (Repository_id))

create table Visit
   (visit_id uuid,
    Repository_id
    visit_ref int,
    date date,
     primary key (visit_id) )
```

Requêtes

Il faut deux requêtes (pas de jointure)

```
select Repository_id in oid
from Repository
where url ='github.org/cassandra'

select * from Visit
where Repository_id = oid
```

Et Cassandra refuse les deux !

Organisons les clés et chemins d'accès

Les critères de recherche doivent impliquer la clé

```
create table Repository
   (url varchar,
    description text,
    primary key (url)
    )

create table Visit_by_Repository
   (url_Repository varchar,
    visit_ref int,
    date date,
    primary key (url_Repository,
                 visit_ref)
    )
```

```
select * from Repository
where  url ='github.org/cassandra'

select * from Visit_by_Repository
where  url ='github.org/cassandra'

select * from Visit_by_Repository
where  url ='github.org/cassandra'
and visit_ref < 3
```

Imbriquons pour avoir moins de tables

Un type `Visit`

```
create type Visit (
    number int,
    date_visit date)
```

Imbriqué dans `Repository`

```
create table Repository
   (url varchar,
    description text,
    visites list<Visit>,
    primary key (url)
    )
```

Suppose un nombre limité de visites.

Une requête localisée et unique pour trouver toutes les visites d'un dépôt.

```
select * from Repository
where  url ='github.org/cassandra'
```

Les dépôts visités au moins 10 fois

```
select * from Repository
where  url ='github.org/cassandra'
and count(visit.number) >= 10
```

Tentons une modélisation complète : Snapshot

```
create type Repository (
    description int,
    other_infos text)

create table Snapshot_by_Repo
   (url varchar,
    snapshot_date,
    snapshot_id uuid,
    Repository Repository,
    branches map<string, uuid>,
    primary key (url,
                 snapshot_date)
    )
```

Avec peu de branches par snapshot

Tous les snapshots d'une Repository.

```
select * from Snapshot_by_Repo
where  url ='github.org/cassandra'
```

Snapshots après une date et contenant une branche `master`

```
select * from Snapshot_by_Repo
where  url ='github.org/cassandra'
and snapshot_date > '01-MARCH-2024'
and branches contains key 'master'
```

On obtient les id des snapshots et des branches

Connaissant une branche : les Revisions

On poursuit la même démarche. Cette fois on a affaire à un graphe. Pas facile à modéliser efficacement.

```
create table Revision_by_branch
   (branch_id uuid,
    revision_id uuid,
    parents list<uuid>,
    author varchar,
    message varchar,
    tstamp date,
    primary key (branch_id,
     revision_id)
    )
```

Toutes les révisions d'une branche par Stefano.

```
select * from Revision_by_branch
where  branch_id ='bxyz'
and author = 'Stefano'
```

Les révisions dont un parent est la révision 'abcd'

```
select * from Revision_by_branch
where   branch_id ='bxyz'
and parents contains 'abcd'
```

On obtient les id des révisions d'une branche.

Et si je veux naviguer dans le graphe ?

Les parents et ascendants de la révision 'lklklk'. On obtient d'abord la liste des parents avec la requête

```
select parents from Revision_by_branch
where   branch_id ='bxyz'
and revision_id = 'lklklk'
```

Puis il faut faire une requête pour **chaque** parent

```
select parents from Revision_by_branch
where   branch_id ='bxyz'
and revision_id = 'id_du_parent'
```

On a peut-être intérêt à charger une bonne fois le graphe dans l'application.

En résumé (provisoire)

En synthèse :

- La clé primaire détermine le stockage *et* les requêtes possibles
  $\Rightarrow$ **clé de partitionnement** = point d'accès, **clé de regroupement** = identifiant relatif au point d'accès
- Il faut parfois "retourner" une clé primaire si on souhaite un accès symétrique à une table existante (cf. exemple `révision/Directory`)
  $\Rightarrow$ automatisable avec une vue matérialisée (forte redondance)
- L'imbrication d'ensembles ou listes peut limiter le nombre d'étapes
  $\Rightarrow$ leur taille doit être restreinte, cf. exemple des branches