Université Paris Dauphine-PSL

Bases de données avancées

# Projet

*Auteurs :*
Marina Chau
Elise Chin
Mathilde Da Cruz

21 janvier 2022

# Contents

# Contexte

Un organisme de gestion de spectacles, de salles de concert et de vente de billets de spectacles gère une base de données dont le schéma relationnel est le suivant :

Spectacle(Spectacle_ID, Titre, DateDebut, Duree, Salle_ID, Artiste)
Concert(Concert_ID, Date, Heure, Spectacle_ID)
Salle(Salle_ID, Nom, Adresse, Capacite)
Billet(Billet_ID, Concert_ID, Num_Place, Categorie, Prix)
Vente(Vente_ID, Date_Vente, Billet_ID, MoyenPaiment)

Les attributs soulignés sont les attributs appartenant à la clé primaire. Ils sont de type entier.
L'attribut Spectacle_ID de la relation Concert est une clé étrangère qui fait référence à l'attribut Spectacle_ID de la relation Spectacle.
L'attribut Concert_ID de la relation Billet est une clé étrangère qui fait référence à l'attribut Concert_ID de la relation Concert.
L'attribut Billet_ID de la relation Vente est une clé étrangère qui fait référence à l'attribut Billet_ID de la relation Billet.
L'attribut Salle_ID de la relation Spectacle est une clé étrangère qui fait référence à l'attribut Spectacle_ID de la relation Spectacle.

Cette base de données permet de répondre, entre autres, aux requêtes suivantes :

1. Quels sont les dates du concert d'un artiste donné ?
2. Quels sont les noms des salles ayant la plus grande capacité ?
3. Quels sont les artistes n'ayant jamais réalisé de concert à une salle donnée ?
4. Quels sont les chanteurs ayant réalisé au moins un concert dans toutes les salles ?
5. Quels sont les chanteurs et les identificateurs des concerts pour lesquels il ne reste aucun billet invendu ?
6. Combien de billets d'une catégorie donnée ont été vendus par spectacles à une date donnée ?

# 1 SQL

***Question 1.*** *Créer un échantillon de données fictives d'au moins 10 nuplets pour chaque TABLE. (Vous pouvez choisir les types de données de tous les attributs sauf les IDs que sont déjà données dans l'énoncé.)*

Dans Spectacle, on suppose que la durée est un nombre de jours.

Dans Concert, nous avons renommé l'attribut "date" par Jour car c'etait plus simple pour notre interpréteur de ne pas avoir un attribut qui avait le même nom que le type.

```
CREATE TABLE Salle(
    Salle_ID INT PRIMARY KEY,
    Nom TEXT,
    Adresse TEXT,
    Capacite INT
);

CREATE TABLE Spectacle(
    Spectacle_ID INT PRIMARY KEY,
    Titre TEXT,
    DateDebut DATE,
    Duree INT,
    Salle_ID INT,
    Artiste TEXT,
    FOREIGN KEY(Salle_ID) REFERENCES Salle(Salle_ID)
);

CREATE TABLE Concert(
    Concert_ID INT PRIMARY KEY,
    Jour DATE,
    Heure TIME,
    Spectacle_ID INT,
    FOREIGN KEY(Spectacle_ID) REFERENCES Spectacle(Spectacle_ID)
);

CREATE TABLE Billet(
    Billet_ID INT PRIMARY KEY,
    Concert_ID INT,
    Num_Place INT,
    Categorie INT,
    Prix INT,
    FOREIGN KEY(Concert_ID) REFERENCES Concert(Concert_ID)
);

CREATE TABLE Vente(
    Vente_ID INT PRIMARY KEY,
    Date_Vente DATE,
    Billet_ID INT,
    MoyenPaiement TEXT,
    FOREIGN KEY(Billet_ID) REFERENCES Billet(Billet_ID)
);

INSERT INTO Salle VALUES (1, 'AccorHotalArena', 'Paris', 20000);
INSERT INTO Salle VALUES (2, 'ZenithAuvergne', 'Cournon', 10000);
INSERT INTO Salle VALUES (3, 'Cigale', 'Paris', 2000);
INSERT INTO Salle VALUES (4, 'Bataclan', 'Paris', 3000);
INSERT INTO Salle VALUES (5, 'Bikini', 'Toulouse', 1000);
INSERT INTO Salle VALUES (6, 'Astrada', 'Marsiac', 50000);
INSERT INTO Salle VALUES (7, 'LePlan', 'Ris', 2000);
INSERT INTO Salle VALUES (8, 'Ninkasi', 'Lyon', 500);
INSERT INTO Salle VALUES (9, 'Animatis', 'Issoire', 3);
INSERT INTO Salle VALUES (10, 'ZenithToulouse', 'Toulouse', 25000);

INSERT INTO Spectacle VALUES (1, 'Festival', '05-01-21', 2, 8, 'Bernard');
INSERT INTO Spectacle VALUES (2, 'Tournee', '02-20-22', 2, 3, 'Mireille');
INSERT INTO Spectacle VALUES (3, 'Concert', '01-17-22', 1, 2, 'Stephane');
INSERT INTO Spectacle VALUES (4, 'Tournee', '12-15-21', 3, 2, 'Jeanine');
```

```
INSERT INTO Spectacle VALUES (5, 'Tournee', '03-10-21', 2, 1, 'Jeanine');
INSERT INTO Spectacle VALUES (6, 'Festival', '09-10-22', 2, 10, 'Bruno');
INSERT INTO Spectacle VALUES (7, 'Concert', '02-06-22', 1, 5, 'Paulette');
INSERT INTO Spectacle VALUES (8, 'Anniv', '01-27-22', 1, 9, 'Yves');
INSERT INTO Spectacle VALUES (9, 'Retour', '01-05-20', 3, 6, 'Yves');
INSERT INTO Spectacle VALUES (10, 'Tournee', '03-25-21', 2, 3, 'Yves');

INSERT INTO Concert VALUES (1, '05-01-21', '18:00', 2);
INSERT INTO Concert VALUES (2, '05-02-21', '18:00', 2);
INSERT INTO Concert VALUES (3, '01-27-22', '20:00', 8);
INSERT INTO Concert VALUES (4, '12-15-21', '20:00', 4);
INSERT INTO Concert VALUES (5, '12-16-21', '20:00', 4);
INSERT INTO Concert VALUES (6, '12-17-21', '20:30', 4);
INSERT INTO Concert VALUES (7, '02-06-22', '19:00', 7);
INSERT INTO Concert VALUES (8, '01-05-21', '22:00', 9);
INSERT INTO Concert VALUES (9, '03-25-21', '21:00', 10);
INSERT INTO Concert VALUES (10, '03-10-21', '20:00', 5);

INSERT INTO Billet VALUES (1, 3, 1, 1, 30);
INSERT INTO Billet VALUES (2, 3, 2, 1, 30);
INSERT INTO Billet VALUES (3, 3, 3, 1, 30);
INSERT INTO Billet VALUES (4, 6, 1, 1, 50);
INSERT INTO Billet VALUES (5, 6, 2, 2, 35);
INSERT INTO Billet VALUES (6, 8, 1, 1, 70);
INSERT INTO Billet VALUES (7, 4, 1, 3, 15);
INSERT INTO Billet VALUES (8, 7, 1, 2, 25);
INSERT INTO Billet VALUES (9, 2, 1, 4, 5);
INSERT INTO Billet VALUES (10, 1, 1, 1, 100);

INSERT INTO Vente VALUES (1, '12-12-20', 1, 'Cash');
INSERT INTO Vente VALUES (2, '07-10-20', 2, 'Cash');
INSERT INTO Vente VALUES (3, '11-30-20', 3, 'CB');
INSERT INTO Vente VALUES (4, '11-30-20', 4, 'CB');
INSERT INTO Vente VALUES (5, '08-07-20', 5, 'Cheque');
INSERT INTO Vente VALUES (6, '06-17-20', 6, 'CB');
INSERT INTO Vente VALUES (7, '09-11-20', 7, 'Cash');
INSERT INTO Vente VALUES (8, '09-12-20', 8, 'CB');
INSERT INTO Vente VALUES (9, '12-25-20', 9, 'CB');
INSERT INTO Vente VALUES (10, '12-12-20', 10, 'CB')
```

***Question 2.*** *Répondre aux requêtes précédentes en SQL.*

1. Quelles sont les dates du concert de Mireille à la Cigale ?

```
SELECT Jour
    FROM Concert c, Spectacle s, Salle t
        WHERE c.Spectacle_ID = s.Spectacle_ID
            AND s.Artiste = 'Mireille'
            AND s.Salle_ID = t.Salle_ID
            AND t.Nom = 'Cigale';
```

   Résultat : 2021-05-01 2021-05-02

2. Quels sont les noms des salles ayant la plus grande capacité ?

```
Select Nom
    FROM Salle
    WHERE Capacite >= ALL(SELECT Capacite FROM Salle);
```

Résultat : Astrada

3. Quels sont les artistes n'ayant jamais réalisé de concert au Zénith d'Auvergne?

```
SELECT Artiste
    FROM Spectacle
    WHERE Artiste NOT IN (SELECT Artiste
                                FROM Spectacle sp, Salle sa
                                    WHERE sp.Salle_ID=sa.Salle_ID
                                        AND sa.Nom='ZenithAuvergne');
```

Résultat : Bernard Mireille Bruno Paulette Yves Yves Yves

4. Quels sont les chanteurs ayant réalisé au moins un concert dans toutes les salles ?

```
SELECT Artiste
    FROM Spectacle s
     WHERE NOT EXISTS
         (SELECT * FROM Salle sa WHERE NOT EXISTS
          (SELECT * FROM Spectacle sp
                    WHERE sp.Artiste = s.artiste
                        AND sa.Salle_ID = sp.Salle_ID
               )
           );
```

Aucun résultat (dans notre exemple)

5. Quels sont les chanteurs et les identificateurs des concerts pour lesquels il ne reste aucun billet invendu ?

```
SELECT Concert_ID, Jour
    FROM Concert co
        WHERE NOT EXISTS (SELECT * FROM Billet b
                            WHERE b.Concert_ID=co.concert_ID
                                AND NOT EXISTS (SELECT * FROM Vente v
                                                    WHERE b.Billet_ID=v.Billet_ID));
```

Resultat : Tous (car la table avait été créée ainsi)

6. Combien de billets de catégorie 1 ont été vendus par spectacles le '11-30-20' ?

```
SELECT s.Spectacle_ID, COUNT(Vente_ID)
    FROM Vente v, Billet b, Concert c, Spectacle s
        WHERE v.Date_Vente = '11-30-20'
            AND v.Billet_ID = b.Billet_ID
            AND b.Categorie = 1
            AND b.Concert_ID = c.Concert_ID
            AND c.Spectacle_ID = s.Spectacle_ID
                GROUP BY s.Spectacle_ID
```

Resultat :

| Spectacle_ID | Count(Vente_ID) |
|---|---|
| 4 | 1 |
| 8 | 1 |

# 2 Modèle clé-valeur - DynamoDB

## 2.1 Analyse du modèle

DynamoDB offre un service de base de données entièrement géré avec des systèmes intégrés de sécurité, de sauvegarde, de restauration et de mise en cache en mémoire pour les applications à l'échelle d'Internet. Cela signifie que l'on peut créer et gérer des applications dans le cloud à l'aide du logiciel sans avoir besoin de faire appel à un seul serveur. Tout cela est pris en charge par l'application. Il faut savoir que c'est une base de données multi-modèles : elle est orientée clefs/valeurs, mais aussi documents.

### 2.1.1 Architecture

DynamoDB divise les informations en tables (qui ne sont pas des tables relationnelles !), en articles et en attributs. Chaque table est composée d'articles, eux-mêmes contenant au moins un attribut. Les articles se différencient entre eux par les attributs qu'ils contiennent et les données que ces attributs stockent. Les bases de données Amazon DynamoDB ne sont donc pas régies par des schémas. Chaque article est identifié grâce à sa clé de partitionnement. Chaque article dispose aussi d'une clé de tri, qui permet de le classer avec les autres articles sur les partitions au sein des tables. Les requêtes sur les tables DynamoDB ne sont possibles que si l'on dispose d'une de ces deux clés. Toutes les données sont conservées sur des disques durs SSD et répliquées 3 fois. Comme DynamoDB est une base NoSQL, il est possible d'augmenter sa performance avec une mise à l'échelle horizontale ("horizontal scaling").

Afin de permettre cette mise à l'échelle horizontale, DynamoDB assigne des données sur des partitions différentes (gérées par des machines distinctes). Lorsqu'il y a ajout de données en masse, DynamoDB peut répartir les nouvelles données sur de nouvelles partitions en ajoutant de nouvelles machines au système.

DynamoDB utilise un cluster de machines, et chaque machine est responsable du stockage d'une partie des données dans ses disques locaux. Quand une nouvelle machine est ajoutée au réseau, elle se voit assignée un chiffre entier choisi aléatoirement. Il s'agit de son "token" (un identifiant ici). On note que DynamoDB utilise une topologie d'anneau pour la distribution des données sur différents serveurs. Il n'y a pas de maître dans le "ring" ni d'esclaves.

Dans cet exemple, on imagine que le cluster dynamoDB a 3 machines: A, B et C. On leur assigne respectivement les tokens suivants: 100, 2000, 10000 (voir figure 1).

*Comment est-ce que cela marche exactement ?*

Lorsque l'on insère une paire clé-valeur dans DynamoDB, la clé est d'abord hachée en un nombre entier, I. Ensuite, la paire est stockée sur la première machine que l'on rencontre lorsque l'on part de la position I sur le ring (dans le sens des aiguilles d'une montre).

### 2.1.2 Gestion de la réplication

Si la donnée est stockée sur une seule machine, on peut rencontrer d'importants problèmes de disponibilité et de fiabilité. En effet, si la machine est corrompue, ou si elle "crash", nous perdons les données. Afin de résoudre ce problème, DynamoDB a mis en place un système de réplication. Sur DynamoDB, les données sont répliquées N fois (N=3 est le nombre de réplication sur AWS). Chaque machine a une connaissance totale des tokens des autres machines disponibles sur le cluster DynamoDB.

Par exemple, chaque machine connaît le token de la machine A qui est 100.

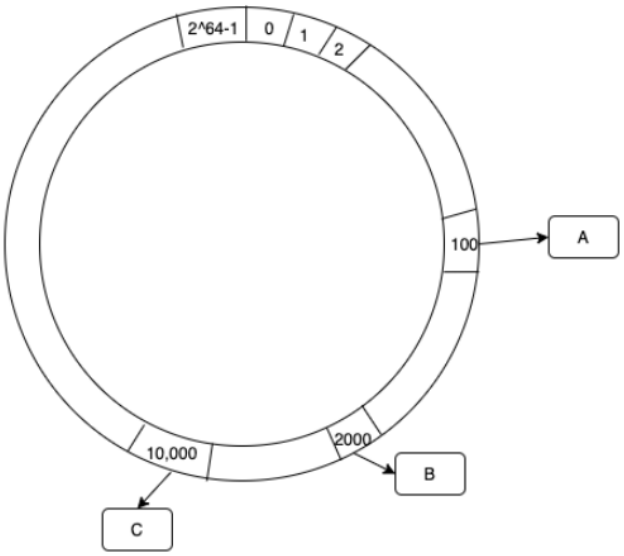
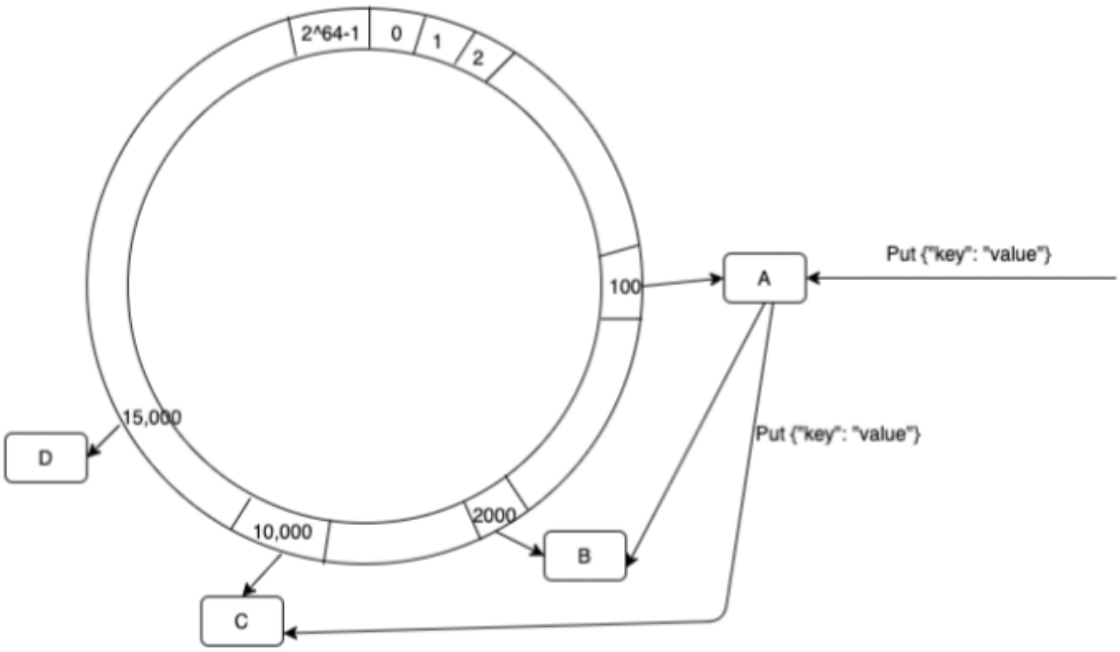



Figure 1

Lorsqu'une machine reçoit une requête pour stocker une nouvelle paire de clé-valeur, cette requête va être également passée à deux autres machines possédant un token supérieur à la machine initiale.




La machine A valide la réception de la nouvelle paire seulement après avoir reçu une réponse des machines B et C. Ce n'est qu'ensuite qu'il enverra une réponse au client.

Toutefois, on pourrait trouver des failles concernant ce système. En effet, si les 3 machines sont lentes, le temps d'écriture peut être très grand. Pour pallier ce problème, DynamoDB n'attend pas que les N machines lui répondent pour faire un retour au client : en effet, il n'y aura besoin que de W machines pour écrire la donnée dans le disque (W = 2 sur AWS). Dans notre cas, la machine A fait un retour au client dès lors que ces deux événements arrivent :

- La donnée est écrite sur le disque A.

- Une réponse reçue de la part de la machine B.
- Une réponse reçue de la part de la machine C.

### 2.1.3 Gestion des transactions et de reprise sur panne

***Gestion des transactions***

Les transactions offrent atomicité, cohérence, isolation et durabilité (propriétés ACID) dans DynamoDB. Les transactions Amazon DynamoDB permettent de regrouper plusieurs actions et de les soumettre comme une seule opération.

TransactWriteItems ou TransactGetItems tout ou rien. TransactWriteItems est une opération d'écriture synchrone et idempotente qui regroupe jusqu'à 25 actions d'écriture en une seule opération tout ou rien. Ces actions peuvent cibler jusqu'à 25 éléments distincts dans une ou plusieurs tables DynamoDB sur le même compte AWS et dans la même région. La taille d'agrégation des éléments dans la transaction ne peut pas dépasser 4 Mo. Les actions sont exécutées de manière atomique, de sorte qu'elles réussissent toutes ou aucune ne réussisse.

***Gestion de reprise sur panne***

L'infrastructure mondiale d'AWS s'articule autour de régions et de zones de disponibilité AWS. Les régions AWS fournissent plusieurs zones de disponibilité physiquement séparées et isolées, reliées par un réseau à latence faible, à débit élevé et à forte redondance. Avec les zones de disponibilité, l'utilisateur peut concevoir et exploiter des applications et des bases de données qui basculent automatiquement d'une zone de disponibilité à l'autre sans interruption. Les zones de disponibilité sont plus hautement disponibles, tolérantes aux pannes et évolutives que les infrastructures traditionnelles à un ou plusieurs centres de données.

Si l'utilisateur a besoin de répliquer ses données ou applications sur des distances géographiques plus importantes, il peut utiliser des régions locales AWS. Une région locale AWS est un centre de données unique conçu pour compléter une région AWS existante. Comme toutes les régions AWS, les régions locales AWS sont totalement isolées des autres régions AWS.

Outre l'infrastructure mondiale d'AWS, Amazon DynamoDB offre plusieurs fonctions qui contribuent à la prise en charge des besoins en matière de résilience et de sauvegarde de données.

*Sauvegarde et restauration à la demande*

DynamoDB fournit une capacité de sauvegarde à la demande. Il permet de créer des sauvegardes complètes des tables pour l'archivage et la conservation à long terme.

*Restauration à un instant dans le passé*

La restauration à un instant dans le passé permet de protéger les tables DynamoDB contre les opérations d'écriture ou de suppression accidentelles. Grâce à la restauration à un instant dans le passé, l'utilisateur n'a plus à se soucier de la création, de la maintenance ou de la planification des sauvegardes à la demande.

### 2.1.4 Gestion de la cohérence, de la disponibilité et de la tolérance au partitionnement

**Cohérence**

Amazon DynamoDB est disponible dans plusieurs régions AWS du monde. Chaque région est indépendante et isolée des autres régions AWS. Par exemple, si il y a une table nommée People dans la région us-east-2, et une autre table nommée People dans la région us-west-2, celles-ci sont considérées comme deux tables entièrement séparées.

Chaque région AWS se compose de plusieurs emplacements distincts appelés zones de disponibilité.

Chaque zone de disponibilité est isolée des pannes dans d'autres zones de disponibilité et fournit une connectivité réseau peu coûteuse et à faible latence vers d'autres zones de disponibilité de la même région. Cela permet une réplication des données entre plusieurs zones de disponibilité d'une région.

Lorsqu'une application écrit des données dans une table DynamoDB et reçoit une réponse HTTP 200 (OK), cela signifie que l'écriture a eu lieu et est durable. Les données seront éventuellement cohérentes sur tous les emplacements de stockage, généralement en une seconde ou moins. DynamoDB prend en charge les lectures éventuellement cohérentes et fortement cohérentes.

Cependant, à retenir pour les lectures à cohérence forte :

- Une lecture cohérente forte peut ne pas être disponible en cas de retard ou de panne réseau. Dans ce cas, il se peut que DynamoDB renvoie une erreur de serveur (HTTP 500).
- Les lectures cohérente forte peuvent avoir une latence plus importante que les lectures éventuellement cohérentes.

**Disponibilité**

DynamoDB a fait le choix d'une cohérence forte, et par conséquent, fait un compromis avec la disponibilité (selon le théorème CAP). En effet lorsqu'on requête en mode cohérence forte, DynamoDB va retourner la plus récente des réponses (la donnée selon la dernière mise à jour sur le système). En cas de panne de réseau, ou retard, une réponse cohérente forte peut donc réduire la disponibilité.

**Partitonnement**

Amazon DynamoDB stocke les données dans des partitions. Une partition est une allocation de stockage pour une table, basée sur des disques SSD, et automatiquement répliquée dans plusieurs zones de disponibilité au sein d'une région AWS. La gestion des partitions étant entièrement effectuée par DynamoDB.

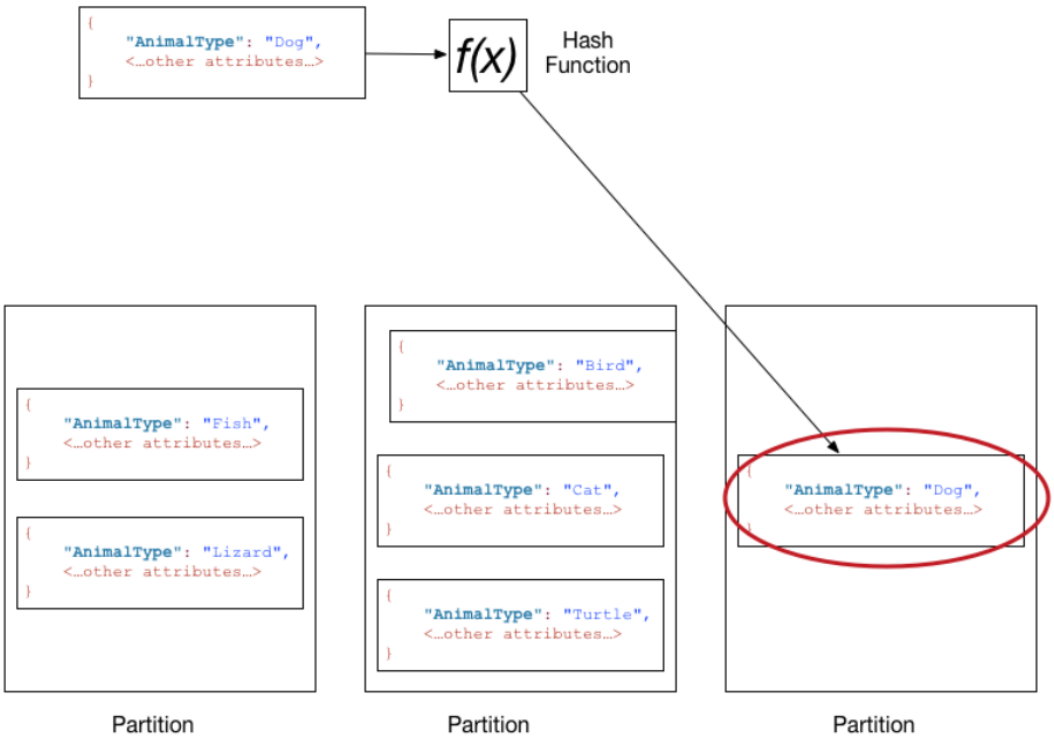
Si une table possède une clé primaire simple (clé de partition uniquement), DynamoDB stocke et récupère chaque élément en fonction de sa valeur de clé de partition.

Pour écrire un élément dans la table, DynamoDB utilise la valeur de la clé de partition comme entrée pour une fonction de hachage interne. La valeur de sortie de la fonction de hachage détermine la partition dans laquelle l'élément sera stocké.

Pour lire un élément de la table, on doit spécifier sa valeur de clé de partition. DynamoDB utilise cette valeur comme entrée pour sa fonction de hachage, donnant la partition dans laquelle se trouve l'élément.

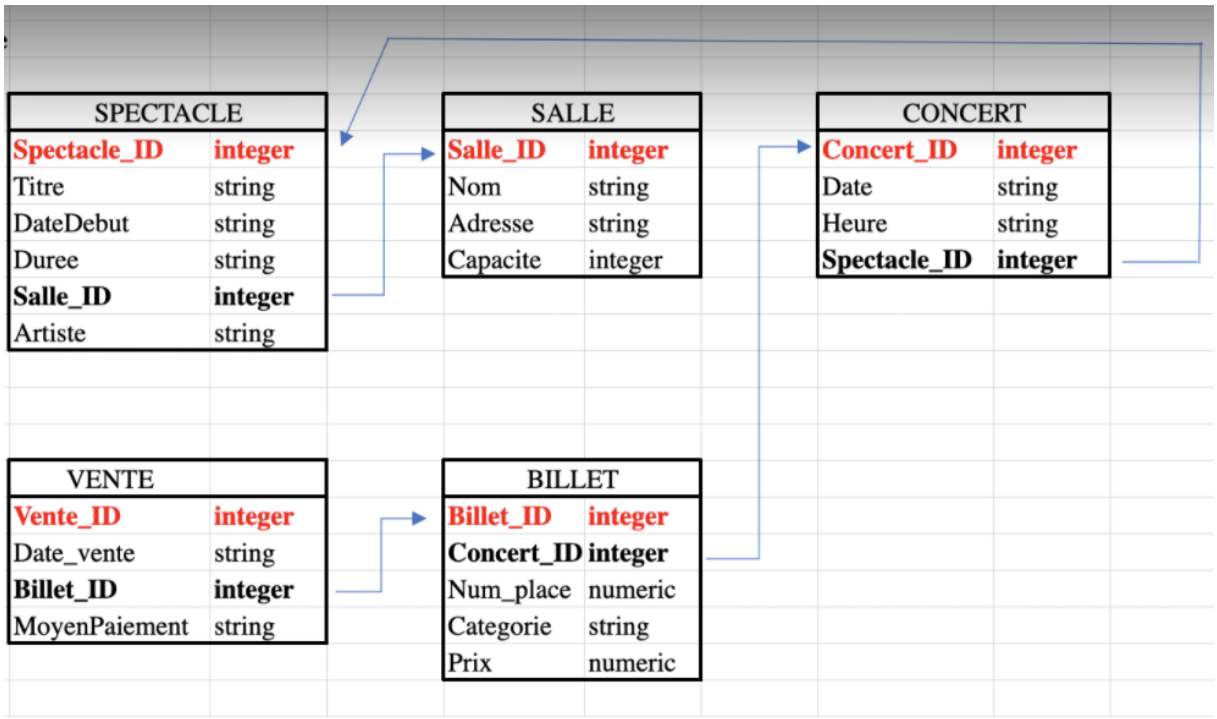
**Exemple**

Le schéma suivant montre une table nommée Animaux de compagnie, qui s'étend sur plusieurs partitions. La clé primaire de la table est AnimalType (seul cet attribut de clé est affiché). DynamoDB utilise sa fonction de hachage pour déterminer où stocker un nouvel élément, en l'occurrence, en fonction de la valeur de hachage de la chaîne Dog. Notez que les éléments ne sont pas stockés selon un ordre trié. L'emplacement de chaque élément est déterminé par la valeur de hachage de sa clé de partition.




## 2.2 Modélisation

Les platesformes traditionnelles du système de gestion de bases de données relationnelles (SGBDR) stockent les données dans une structure relationnelle normalisée. Cette structure réduit les structures de données hiérarchiques à un ensemble d'éléments communs stockés sur plusieurs tables. Le schéma suivant est une représentation en base de données relationnelles de notre problème :



Ce genre de base de données nécessite un volume de traitement important. On doit souvent interroger les données de plusieurs emplacements, et les résultats doivent être assemblés pour être présentés.

Un autre facteur qui peut ralentir les SGBDR est la nécessité de prendre en charge une infrastructure de transaction conforme à ACID. Les structures de données hiérarchiques utilisées par la plupart des applications de traitement de transaction en ligne (OLTP) doivent être décomposées et réparties dans plusieurs tables logiques lorsqu'elles sont stockées dans un SGBDR. Par conséquent, une infrastructure de transaction conforme à ACID est nécessaire pour éviter les conditions de concurrence pouvant avoir lieu si une application tente de lire un objet qui est en cours d'écriture. Une telle infrastructure de transaction ajoute nécessairement un traitement supplémentaire important au processus d'écriture.

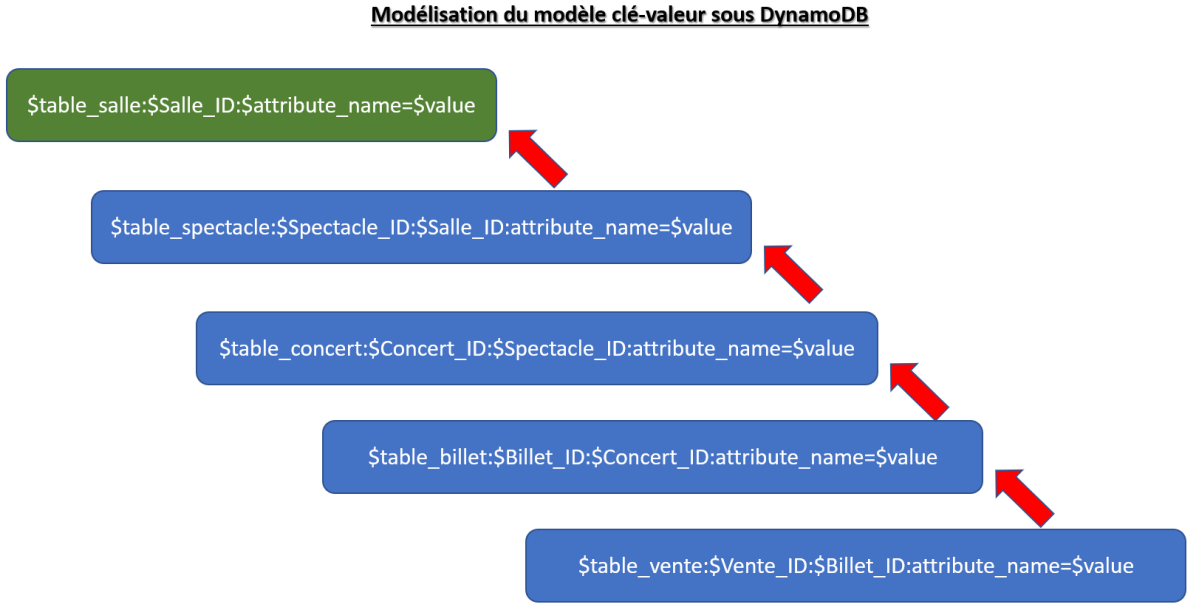
Ainsi pour une activité exigeant une réponse à faible latence pour des requêtes à trafic élevé, il est judicieux de tirer parti des avantages d'un système NoSQL tel que DynamoDB.

DynamoDB se met à l'échelle correctement pour plusieurs raisons :

- Avec son schéma flexible, DynamoDB peut stocker des données hiérarchiques complexes au sein d'un seul élément. ‘
- Une conception de clé composite permet à ce service de stocker les éléments liés proches les uns des autres dans la même table.

### 2.2.1 Création du modèle

Dans un premier temps, nous avons cherché à modéliser notre problème selon les codes du modèle clé-valeur.




En effet, la base SQL étant constitué de différentes jointures, il nous fallait modéliser des clés étrangères.

Par exemple, pour la table Concert, la modélisation clé-valeur ressemblerait à :

**Table_concert:$Concert_ID:$Spectacle_ID:attribute_name=$value**

Soit, de manière plus concrète :

- Exemple 1 : Table_concert:123:78097:Date=12/09/2023

- Exemple 2 : Table_concert:123:78097:Heure=17:30

De la même manière pour la table Billet, nous obtenons :

**Table_billet:$Billet_ID:$Concert_ID:attribute_name=$value**

Soit:

- Exemple 1: Table_billet:1333:123:Num_Place=192B
- Exemple 2: Table_billet:1423:123:Categorie=B7
- Exemple 3: Table_billet:7123:123:Prix=125€

Dans un second temps, nous nous sommes penchés sur la modélisation sous format clé-valeur avec DynamoDB. Notre base de données relationnelle contient de nombreuses jointures. Cependant, ces dernières n'existent pas sous DynamoDB. Cela implique donc que nous devons effectuer plusieurs requêtes pour récupérer à la fois les noms des artistes et les salles dans lesquelles ils ont joué par exemple. Cela peut poser problème car nous devons alors effectuer plusieurs requêtes en cascade. Au fur et à mesure que l'application évolue, on peut perdre en performance.

Toutefois, pour pallier à ce problème, on peut utiliser les collections d'éléments de DynamoDB.

Une collection d'éléments dans DynamoDB fait référence à tous les éléments d'une table ou d'un index qui partagent une clé de partition. Dans l'exemple ci-dessous, nous avons une table DynamoDB qui contient les artistes et les salles dans lesquelles ils ont joué.

| Primary Key | | Attributes | |
|---|---|---|---|
| Artiste (Partition Key) | Salle_ID (Sort Key) | | |
| | | Spectacles | Concerts |
| | La Girouete | Olé | Ola |
| | | Spectacles | Concerts |
| Tom Hanks | Le Pacha Club | Holé | Hola |
| | | Spectacles | Concerts |
| Tim Allen | Big Mama | Ayaya | Ayoyo |

### 2.2.2 Les requêtes

Pour faire nos différentes requêtes, nous avons utilisé l'API Python de DynamoDB, BOTO3.

**Création de table**

```
Access_key ="AKIAY4YQEYSPXGY5IV4M"
Secret_Access_key = "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"
client = boto3.client(
    'dynamodb',
    "eu-west-3",
    aws_access_key_id=Access_key,
    aws_secret_access_key=Secret_Access_key
    )
dynamodb = boto3.resource(
   'dynamodb',
    "eu-west-3",
    aws_access_key_id=Access_key,
    aws_secret_access_key=Secret_Access_key
)
```

```
billet_Table = dynamodb.Table("Billet")
vente_Table = dynamodb.Table("Vente")
spectacle_Table = dynamodb.Table("Spectacle")
salle_Table = dynamodb.Table("Salle")
concert_Table = dynamodb.Table("Concert")

billets = pd.read_csv("billet.csv", sep =';')
spectacle = pd.read_csv("specatacle.csv", sep =';')
salle = pd.read_csv("salle.csv", sep =';')
concert = pd.read_csv("concert.csv", sep =';')
vente =  pd.read_csv("vente.csv", sep =';')

csv = [billets, spectacle, salle, concert, vente]
tables= [billet_Table, spectacle_Table, salle_Table, concert_Table, vente_Table]

for i in range(len(csv)):
    for j in range(len(csv[i])):
        doc = dict(csv[i].loc[j,:])
        for value in doc:
            if not isinstance(doc[value],str):
                doc[value] = int(doc[value])
        tables[i].put_item(Item= doc)
```

**Requête 1**

```
#a) Quels sont les dates du concert d'un artiste donné ?

Artiste = "artiste_0"

response = spectacle_Table.scan()
data = response['Items']
for spec in data:
    if spec["artist"] == Artiste:
        print(spec["date_debut"])
```

**Requête 2**

```
#b) Quels sont les noms des salles ayant la plus grande capacité ?
import numpy as np

response = salle_Table.scan()
data = response['Items']
capacite = []
for salle in data:
    capacite.append(salle["capacite"])
index = np.argmax(capacite)
data[index]["nom"]
```

**Requête 3**

```
response = salle_Table.scan()
data = response['Items']
capacite = []
for salle in data:
```

```
    capacite.append(salle["capacite"])
index = np.argmax(capacite)
data[index]["nom"]
```

**Requête 3**

```
salle_nom = "salle_0"

response = spectacle_Table.scan()
spectacle_data = response['Items']
response = salle_Table.scan()
salle_data = response['Items']

salle_id = [salle for salle in salle_data if salle["nom"] == salle_nom][0]["Salle_ID"]
artists = list(set([spectacle["artist"] for spectacle in spectacle_data]))

res = []
for artist in artists:
    spectacles = [spec for spec in spectacle_data if spec["artist"]==artist]
    salles = [spec["salle_id"] for spec in spectacles]
    if salle_id not in salles:
        res.append(artist)
res
```

**Requête 4**

```
#d) Quel sont les chanteurs ayant réalisé au moins un concert dans toutes les salles ?

response = spectacle_Table.scan()
data = response['Items']
salles = list(set([data[i]["salle_id"] for i in range(len(data))]))
artistes = list(set([data[i]["artist"] for i in range(len(data))]))
res = []
for artiste in artistes:
    tmp = salles
    for spec in data:
        if spec["salle_id"] in tmp:
            tmp.pop(tmp.index(spec["salle_id"]))
    if len(tmp) == 0:
        res.append(artiste)
print(artiste)
```

**Requête 5**

```
#e) Quels sont les chanteurs et les identificateurs des concerts pour lesquels il ne
↪  reste aucun billet invendu ?

response = vente_Table.scan()
vente_data = response['Items']
response = billet_Table.scan()
billets_data = response['Items']
response = spectacle_Table.scan()
spectacle_data = response['Items']

all_id = [billets_data[i]["Billet_ID"] for i in range(len(billets_data))]
```

```
id_vendus = [vente_data[i]["Billet_ID"] for i in range(len(vente_data))]

res = []
for spe in spectacle_data:
    id = spe["Spectacle_ID"]
    if all_id.count(id)==id_vendus.count(id):
        res.append((id, spe["artist"]))

print(res)
```

**Requête 6**

```
#f) Combien de billets d'une catégorie donnée ont été vendus par spectacles à une date
↪  donnée ?
date = "25/11/2019"
cat = 3


response = spectacle_Table.scan()
spectacle_data = response['Items']
response = vente_Table.scan()
vente_data = response['Items']
response = billet_Table.scan()
billets_data = response['Items']
response = concert_Table.scan()
concerts_data = response['Items']


for spec in spectacle_data:
    concerts = [concert for concert in concerts_data if concert["spectacle_id"] ==
    ↪  spec["Spectacle_ID"] and concert["date"]==date]
    concerts_id= [concert["Concert_ID"] for concert in concerts]
    billets = [i for i in billets_data if i["Concert_ID"] in concerts_id and
    ↪  i["categorie"] == cat]
    billets_id = [i["Billet_ID"] for i in billets]
    billets_vendus = [i for i in vente_data if i["Billet_ID"] in billets_id]
    print(spec["Spectacle_ID"], len(billets_vendus))
```

### 2.2.3 Avantage du modèle

Ce type d'architecture en monotable vous permet :

[breaklines=True]d'accéder aux données plus rapidement, en une ou deux requêtes, de minimiser le nombre de tables que vous devez gérer. d'éviter des surcoûts opérationnels (monitoring, backup, etc ...) comme on peut le constater dans un design classique comprenant plusieurs tables. de réaliser des économies : si vous avez un ou deux types d'entités dans votre monotable qui sont consultés beaucoup plus fréquemment que les autres, vous pouvez masquer une partie de la capacité supplémentaire pour les éléments moins fréquemment consultés dans la mémoire tampon des autres éléments. de profiter d'une mise à l'échelle quasi infinie grâce à DynamoDB. d'améliorer les performances de votre solution en effectuant une seule demande pour récupérer tous les éléments nécessaires.

### 2.2.4 Inconvénients du modèle

Le modèle DynamoDB présente quelques inconvénients à prendre en compte :

- Une seule table DynamoDB surchargée peut sembler vraiment étrange à première vue par rapport aux tables propres et normalisées de votre base de données relationnelle. Il est difficile de désapprendre toutes les leçons apprises au fil des années de la modélisation de données relationnelles.
- Une deuxième difficulté concernant DynamoDB est celle d'adapter de nouveaux modèles d'accès dans une conception monotable.
- DynamoDB est conçu pour les cas d'utilisation OLTP — accès aux données à grande vitesse où vous travaillez sur quelques enregistrements à la fois. Mais les utilisateurs ont également besoin de modèles d'accès OLAP – de grandes requêtes analytiques sur l'ensemble de données pour trouver des éléments populaires, ou le nombre de commandes par jour, ou d'autres informationK. DynamoDB n'est pas bon pour les requêtes OLAP. C'est intentionnel. DynamoDB se concentre sur l'ultra-performance des requêtes OLTP.

# 3 Modèle colonne - Scylla

## 3.1 Analyse du modèle

Scylla est une base de données grosse colonne distribuée NoSQL en pair-à-pair, conçue pour traiter des données volumineuses sans avoir de point de défaillance unique[1]. Même en cas de défaillance du matériel ou du réseau, le système est capable d'assurer une haute disponibilité. Ecrit en C++, Scylla a été créé en remplacement d'Apache Cassandra, et a donc adopté une grande partie de sa conception distribuée à grande échelle à partir de Cassandra.

### 3.1.1 Architecture

Dans Scylla, la donnée est répartie dans un cluster, composé d'un ensemble de noeuds, dont chacun est divisé en plusieurs shards indépendants (avec un CPU, RAM et un stockage permanent). Un noeud contient donc une portion de l'ensemble des données stocké dans un cluster.

Scylla stocke les informations dans des tables, comme un ensemble de lignes et de colonnes. Une ligne est un ensemble de colonnes, identifiée par une clé primaire. Dans un modèle grosse colonne, comme Scylla, la clé primaire est composée au minimum d'une clé de partitionnement et en option, d'une clé de clustering. La clé de partionnement est responsable de la distribution des données dans les noeuds, et permet donc de déterminer le noeud sur lequel est stockée une ligne donnée. La clé de clustering est responsable, quant à elle, de l'ordonnancement des lignes au sein d'une partition – sous-ensemble de lignes stockées dans un noeud et répliqué sur d'autres noeuds. Dans l'exemple ci-dessous, nous avons 3 partitions avec comme clé de partionnement 101, 103 et 104, chacune composée d'une ligne avec un nombre de colonnes différents. On retrouve ici l'une des différences majeures avec une base de données relationnelles : dans un modèle grosse colonne, une ligne possède un nombre flexible de colonnes, alors que dans un modèle relationnel, une ligne aura un nombre fixe de colonnes.

Pour retrouver une ligne à l'aide de la clé de partitionnement, une fonction de hachage est utilisée. Celle-ci renvoie un token, qui permet de trouver l'ensemble de lignes qui compose une partition, d'identifier le noeud du cluster sur lequel est stockée la partition, mais aussi de distribuer la donnée au sein du cluster. Un token prend sa valeur dans une plage de valeurs définis par rapport au nombre de noeuds dans un cluster. La figure ci-dessus illustre un exemple de plage de tokens de 0 à 1200 répartis de manière égale entre un cluster de trois nœuds.

Les noeuds sont distribués en forme d'anneau, ainsi un cluster est souvent désigné sous le nom d'architecture en anneau. Plus précisément, Scylla implémente une architecture de noeuds virtuels — ils représentent chacun un ensemble de plages contiguë de tokens.

[1] Point d'un système informatique dont le reste du système est dépendant et dont une partie entraîne l'arrêt complet du système https://en.wikipedia.org/wiki/Single_point_of_failure

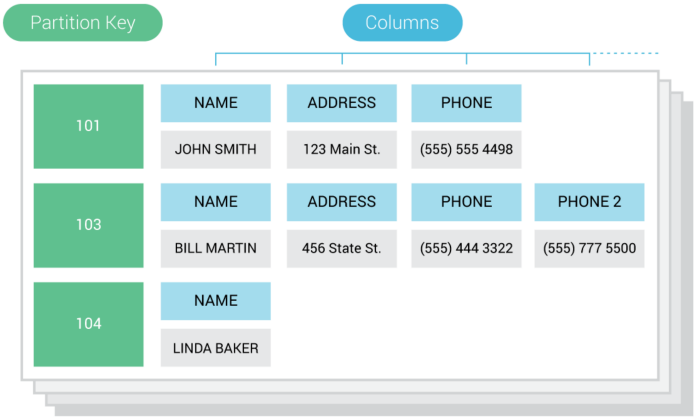



Figure 2: Une table en Scylla

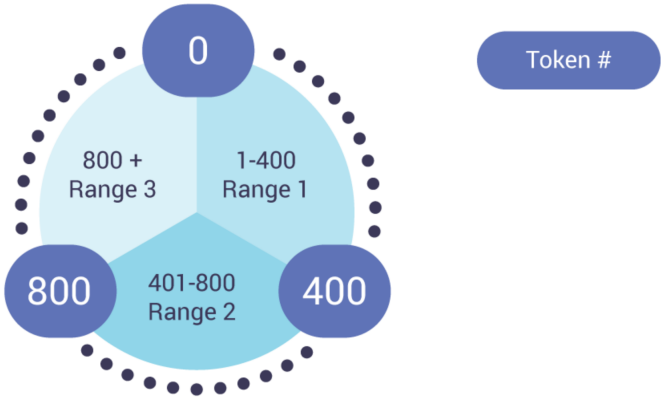



Figure 3: Exemple de plage de tokens de 0 à 1200 répartis de manière égale entre un cluster de trois nœuds

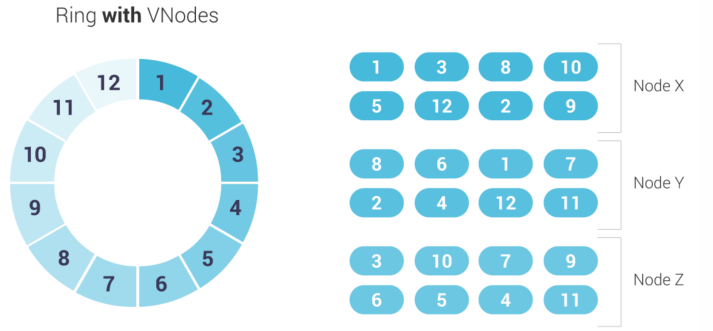



Figure 4: Cluster de trois noeuds, avec 12 noeuds virtuels

Il est ainsi très simple d'ajouter des noeuds (pour agrandir la capacité de stockage et de traitement) ou d'en retirer. Un cluster de Scylla est capable de se reconfigurer automatiquement et de rééquilibrer les données qu'il contient. Au sein d'un cluster, les noeuds communiquent en pair-à-pair, évitant les points de défaillance unique. Tous les noeuds sont égaux : il n'y pas de maître ou de noeuds esclaves, et chaque noeud peut gérer une demande de lecture ou d'écriture. A chaque requête, un noeud coordinateur (*coordinator node*) est désigné. A l'aide d'une fonction de hachage qui permet de déterminer la localisation de la donnée dans un noeud, et de la stratégie de réplication, il envoie la requête aux noeuds concernés. Le niveau de cohérence détermine le nombre de nœuds dont le coordinateur doit recevoir une réponse pour que la demande aboutisse.

### 3.1.2 Gestion de la réplication

La réplication est automatique et est gérée par le facteur de réplication RF. Il correspond au nombre de noeuds où les données seront répliquées. Si RF = 1, cela implique qu'une seule copie d'une ligne est présente dans le cluster, et il n'y aura aucun moyen de récupérer cette donnée si le noeud concerné est compromis. Si RF = 2, il y a alors deux copies d'une même ligne dans un cluster. Un RF d'au moins 3 est utilisé dans de nombreux systèmes. Dans l'exemple ci-dessous, un client envoie une requête d'écriture 1 au noeud V. Celui-ci devient le noeud coordinateur, et envoie ainsi la requête à trois noeuds (car RF = 3), en l'occurrence W, X et Z.

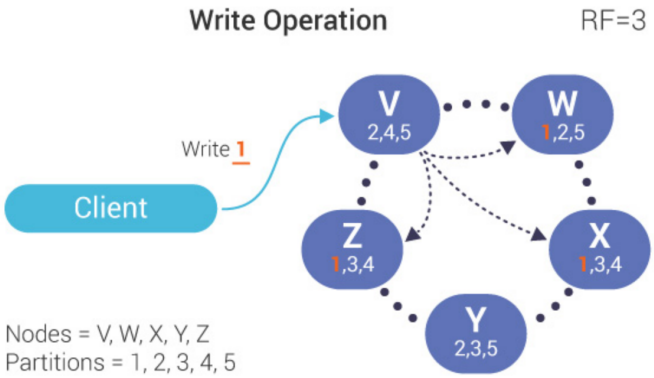



Figure 5: Opération d'écriture avec RF = 3

### 3.1.3 Gestion des transactions et reprise sur panne

#### Gestion des transactions

A l'instar de Cassandra, une transaction dans Scylla ne satisfait pas les propriétés ACID comme dans les bases de données relationnelles. Cependant, les instructions `UPDATE`, `INSERT` et `DELETE` sont effectuées de manière atomique et isolée si les modifications appartiennent à la même partition.

Dans Scylla, les transactions sont limitées à une seule instruction conditionnelle. Plus précisement, le système vérfie si une condition est vraie, et si c'est le cas, la transaction est effectuée, si la condition n'est pas remplie, la transaction n'a pas lieu. C'est ainsi qu'elles sont qualifiées de "légères" car elles ne vérrouillent pas la base de données pour la transaction. Pour gérer les possibles conflits, un protocle de consensus (en l'occurence Paxos[2] est utilisée pour s'assurer qu'un accord est établi entre noeuds pour valider une modification.

#### Gestion de reprise sur panne

Scylla utilise, à l'instar de Cassandra, un paramètre de cohérence pour gérer la cohérence des données. Le niveau de cohérence (*consistency level*, CL) en lecture détermine le nombre de noeuds devant être interrogés par le cluster avant que celui-ci ne retourne la donnée suite à la demande de lecture du client. Le niveau de cohérence en écriture détermine le nombre de noeuds sur lesquels doit être écrit la donnée avant de valider l'écriture faite par le client[3]. A ne pas confondre avec le facteur de réplication. Par exemple, si RF = 3 et CL = 1, les trois copies devront être mises à jour (RF = 3) mais le noeud coordinateur doit seulement attendre qu'un noeud confirme l'écriture (CL = 1). La confirmation d'écriture des deux autres noeuds peut intervenir après celle du coordinateur au client (voir figure 6).

Le niveau de cohérence et le facteur de réplication ont tous deux un impact sur les performances. Plus le niveau de cohérence et/ou le facteur de réplication sont faibles, plus l'opération de lecture ou d'écriture

[2] https://en.wikipedia.org/wiki/Paxos_(computer_science)
[3] https://stph.scenari-community.org/contribs/nos/Cassandra3/co/3_2_1-Consistency-Level.html

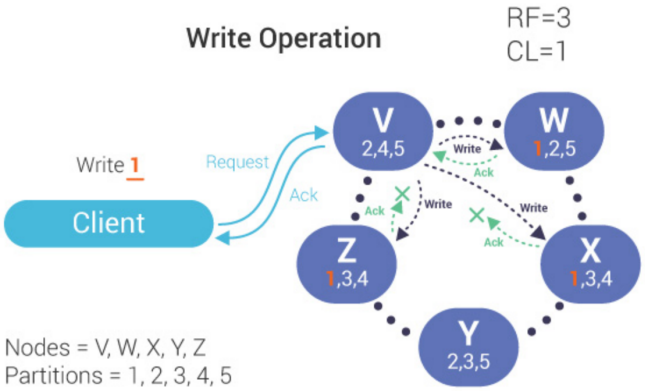



Figure 6: Opération d'écriture avec RF = 3 et CL = 1

est rapide. En effet, si CF est faible, il faut alors très peu de temps avant la confirmation en lecture ou en écriture par CF noeuds au noeud coordinateur. Si RF est faible, alors le temps de latence pour écrire sur RF noeuds sera également court. Cependant, la tolérance aux pannes sera moindre si un nœud tombe en panne puisqu'il y aura moins de copies.

### 3.1.4 Gestion de la cohérence, de la disponibilité et de la tolérance au partitionnement

Scylla, comme de nombreux systèmes de bases de données distribuées, adhère au théorème CAP. Le théorème CAP est la notion que la cohérence, la disponibilité et la tolérance au partitionnement des données sont mutuellement dépendantes dans un système distribué. En augmentant deux de ces propriétés, on réduit le troisième.

Scylla privilégie la disponibilité et la tolérance au partitionnement au détriment de la cohérence. D'après le théorème CAP, la troisième propriété qui est la cohérence, ne pourra être assurée.

## 3.2 Modélisation

Dans Scylla, contrairement aux bases de données relationnelles, le modèle est construit à partir des requêtes : on construit une table par requête. Un des objectifs de la modélisation dans Scylla (et plus largement pour les modèles grosses colonnes) est de retourner une réponse rapide aux requêtes de lecture[4]. Pour cela :

- Les données sont réparties uniformément sur les noeuds du cluster, c'est-à-dire que chaque noeud contiendra approximativement la même quantité de données.
- Idéalement, toutes les données nécessaires pour répondre à une requête de lecture doivent être stockées dans une même table. La lecture est ainsi plus rapide puisque le nombre de partitions auxquelles on accède lors de la requête est minimisé.

Que faire lorsqu'une requête nécessite des données provenant de plus de deux tables ? Dans une modélisation relationnelle, on utiliserait un *join* afin de combiner les lignes de plusieurs tables en rejoingnant des colonnes communes. Dans une modélisation grosse colonne (Scylla ou Cassandra), les jonctions ne sont pas supportées. Une solution possible est alors de dénormaliser les données. Ce processus consiste à créer une nouvelle table avec des données redondantes afin de répondre à des requêtes spécifiques, où une jonction aurait été utilisée dans une base de données relationnelle[5]. Dans Scylla, en

[4] https://university.scylladb.com/courses/data-modeling/lessons/basic-data-modeling-2/topic/introduction-3/

[5] https://university.scylladb.com/courses/data-modeling/lessons/advanced-data-modeling/topic/denormalization/

privilégiant une lecture rapide et efficiente, les données sont ainsi amenées à être redondantes au sein d'un même noeud.

Dans la suite, nous créons une table (quand cela est possible) par requête. La sous-section ci-dessous est donc organisée par requête. Le langage de Scylla est le même que celui de Cassandra, CQL.

### 3.2.1 Création du modèle et réponses aux requêtes

**Création du keyspace.** Nous créons tout d'abord un *keyspace*, l'équivalent d'une base de données dans le monde relationnel. Le *keyspace* stocke l'ensemble des tables, et à sa création, permet de définir la stratégie de réplication. La documentation recommande de fixer la stratégie suivante `NetworkTopologyStrategy` au lieu de `SimpleStrategy`, peu adaptée à la mise en production[6]. Néanmoins, ce n'est pas très important pour notre exemple. La valeur du facteur de réplication sera ici arbitraire. Ce dernier est à fixer selon le nombre de noeuds disponibles dans un réel cluster.

```
CREATE KEYSPACE Spectacle_Keyspace WITH replication = {'class':
↪  'NetworkTopologyStrategy', 'replication_factor': 3};
USE Spectacle_Keyspace;
```

**Comment choisir une clé de partionnement ?** Rappelons qu'une ligne (ensemble de colonnes) est identifiée dans une table par une clé primaire, composée d'une clé de partionnement et optionnellement d'une clé de clustering. La clé de partionnement est responsable de la distribution des données dans les noeuds, et permet donc de déterminer le noeud sur lequel est stockée une ligne donnée. La clé de clustering est responsable, quant à elle, de l'ordonnancement des lignes au sein d'une partition.

Une clé de partitionnement peut être qualifiée de pertinente lorsqu'elle permet de tirer parti de la manière dont Scylla stocke les données, et donc d'exécuter rapidement les requêtes de lecture. Pour cela, il suffit d'inclure la ou les colonnes ciblées par la requête dans la clé de partitionnement. Scylla saura ainsi déterminer les noeuds sur lesquels se trouvent la donnée en hachant la clé de partionnement grâce à une fonction de hâchage[7].

**a. Quelles sont les dates du concert de Mireille à la Cigale ?** Etant donné le nom d'un artiste et la salle dans laquelle cet artiste va se produire, on recherche les dates de ses concerts. Les informations pertinentes à mettre dans la table sont le nom de l'artiste, l'identifiant, la salle et la date du concert. Une ligne peut être identifiée par le nom de l'artiste. A priori, nous pouvons donc utiliser le nom de l'artiste comme clé de partitionnement. Cependant, un artiste peut se produire dans une même salle plusieurs fois à des dates différentes, il faudrait donc utiliser une clé de partitionnement composée à la fois du nom de l'artiste et de la salle de concert. Enfin, pour ordonner les lignes au sein d'une partition, la date du concert est utilisée comme clé de clustering.

| **Concert_par_Artiste** | | | |
|---|---|---|---|
| Artiste | Nom_Salle | #Concert_ID | Date_Concert |
| *TEXT* | *TEXT* | *INT* | *DATE* |

Table 1: Table *Concert_par_Artiste* pour répondre à la requête "Quelles sont les dates d'un concert pour un artiste donné ?" La clé de partionnement (souligné) est (Artiste, Nom_Salle) et la clé de clustering (précédé d'un #) est (Concert_ID).

**CQL**

[6] https://docs.scylladb.com/getting-started/ddl/#create-keyspace-statement

[7] https://university.scylladb.com/courses/data-modeling/lessons/basic-data-modeling-2/topic/importance-of-primary-key-selection/

```
/* Creation de la table */
CREATE TABLE Concert_par_Artiste(
    Artiste TEXT,
    Nom_Salle TEXT,
    Concert_ID INT,
    Date_Concert DATE, /* Format yyyy-mm-dd */
    PRIMARY KEY((Artiste, Nom_Salle), Date_Concert)
);

/* Insertion des donnees */
INSERT INTO Concert_par_Artiste(Artiste, Nom_Salle, Concert_ID, Date_Concert)
↪  VALUES('Mireille', 'Cigale', 1, '2021-01-05');
INSERT INTO Concert_par_Artiste(Artiste, Nom_Salle, Concert_ID, Date_Concert)
↪  VALUES('Mireille', 'Cigale', 2, '2021-02-05');
INSERT INTO Concert_par_Artiste(Artiste, Nom_Salle, Concert_ID, Date_Concert)
↪  VALUES('Yves', 'Animatis', 3, '2022-01-27');
INSERT INTO Concert_par_Artiste(Artiste, Nom_Salle, Concert_ID, Date_Concert)
↪  VALUES('Jeanine', 'ZenithAuvergne', 4, '2021-12-15');
INSERT INTO Concert_par_Artiste(Artiste, Nom_Salle, Concert_ID, Date_Concert)
↪  VALUES('Jeanine', 'ZenithAuvergne', 5, '2021-12-16');
INSERT INTO Concert_par_Artiste(Artiste, Nom_Salle, Concert_ID, Date_Concert)
↪  VALUES('Jeanine', 'ZenithAuvergne', 6, '2021-12-17');
INSERT INTO Concert_par_Artiste(Artiste, Nom_Salle, Concert_ID, Date_Concert)
↪  VALUES('Paulette', 'Bikini', 7, '2022-02-06');
INSERT INTO Concert_par_Artiste(Artiste, Nom_Salle, Concert_ID, Date_Concert)
↪  VALUES('Yves', 'Astrada', 8, '2021-01-05');
INSERT INTO Concert_par_Artiste(Artiste, Nom_Salle, Concert_ID, Date_Concert)
↪  VALUES('Yves', 'Cigale', 9, '2021-03-25');
INSERT INTO Concert_par_Artiste(Artiste, Nom_Salle, Concert_ID, Date_Concert)
↪  VALUES('Jeanine', 'AccorHotelArena', 10, '2021-03-10');

/* Contenu de la table */
SELECT * FROM Concert_par_Artiste;
```

| artiste | nom_salle | date_concert | concert_id |
|---|---|---|---|
| Yves | Astrada | 2021-01-05 | 8 |
| Jeanine | AccorHotelArena | 2021-03-10 | 10 |
| Paulette | Bikini | 2022-02-06 | 7 |
| Yves | Animatis | 2022-01-27 | 3 |
| Yves | Cigale | 2021-03-25 | 9 |
| Mireille | Cigale | 2021-01-05 | 1 |
| Mireille | Cigale | 2021-02-05 | 2 |
| Jeanine | ZenithAuvergne | 2021-12-15 | 4 |
| Jeanine | ZenithAuvergne | 2021-12-16 | 5 |
| Jeanine | ZenithAuvergne | 2021-12-17 | 6 |

Figure 7: Table Concert_par_Artiste

```
/* Requete */
SELECT Date_Concert FROM Concert_par_Artiste
WHERE Artiste = 'Mireille'
AND Nom_Salle = 'Cigale';
```

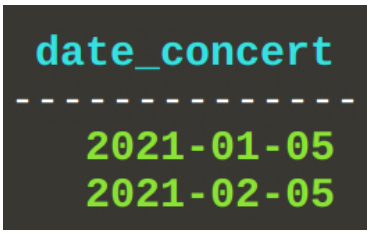



Figure 8: Résultat de la requête "Quelles sont les dates du concert de Mireille à la Cigale ?"

**b. Quels sont les noms des salles ayant la plus grande capacité ?** Cette requête concerne seulement les données d'une salle. On peut donc créer une table `Salle`, de la même manière que dans la section 1. `Salle_ID` sera la clé de partionnement. En choisissant `Capacite` comme clé de clustering, nous pouvons ordonner les lignes au sein du noeud par ordre décroissant de valeur grâce à la clause d'ordonnancement `WITH CLUSTERING ORDER BY (Capacite DESC)`.

| **Salle** | | | |
|---|---|---|---|
| Salle_ID | Nom | Adresse | #Capacite |
| *TEXT* | *TEXT* | *TEXT* | *INT* |

Table 2: Table *Salle* pour répondre à la requête "Quels sont les noms des salles ayant la plus grande capacité ?" La clé de partionnement (souligné) est (Salle_ID) et la clé de clustering (précédé d'un #) est (Capacite).

**CQL**

```
/* Creation de la table */
CREATE TABLE Salle(
    Salle_ID INT,
    Nom TEXT,
    Adresse TEXT,
    Capacite INT,
    PRIMARY KEY (Salle_ID, Capacite)
) WITH CLUSTERING ORDER BY (Capacite DESC);

/* Insertion des donnees */
INSERT INTO Salle(Salle_ID, Nom, Adresse, Capacite) VALUES(1, 'AccorHotalArena',
↪  'Paris', 20000);
INSERT INTO Salle(Salle_ID, Nom, Adresse, Capacite) VALUES(2, 'ZenithAuvergne',
↪  'Cournon', 10000);
INSERT INTO Salle(Salle_ID, Nom, Adresse, Capacite) VALUES(3, 'Cigale', 'Paris',
↪  2000);
INSERT INTO Salle(Salle_ID, Nom, Adresse, Capacite) VALUES(4, 'Bataclan', 'Paris',
↪  3000);
INSERT INTO Salle(Salle_ID, Nom, Adresse, Capacite) VALUES(5, 'Bikini', 'Toulouse',
↪  1000);
INSERT INTO Salle(Salle_ID, Nom, Adresse, Capacite) VALUES(6, 'Astrada', 'Marsiac',
↪  50000);
INSERT INTO Salle(Salle_ID, Nom, Adresse, Capacite) VALUES(7, 'LePlan', 'Ris', 2000);
INSERT INTO Salle(Salle_ID, Nom, Adresse, Capacite) VALUES(8, 'Ninkasi', 'Lyon', 500);
INSERT INTO Salle(Salle_ID, Nom, Adresse, Capacite) VALUES(9, 'Animatis', 'Issoire',
↪  3);
INSERT INTO Salle(Salle_ID, Nom, Adresse, Capacite) VALUES(10, 'ZenithToulouse',
↪  'Toulouse', 25000);

/* Contenu de la table */
SELECT * FROM Salle;
```

| salle_id | capacite | adresse | nom |
|---|---|---|---|
| 5 | 1000 | Toulouse | Bikini |
| 10 | 25000 | Toulouse | ZenithToulouse |
| 1 | 20000 | Paris | AccorHotalArena |
| 8 | 500 | Lyon | Ninkasi |
| 2 | 10000 | Cournon | ZenithAuvergne |
| 4 | 3000 | Paris | Bataclan |
| 7 | 2000 | Ris | LePlan |
| 6 | 50000 | Marsiac | Astrada |
| 9 | 3 | Issoire | Animatis |
| 3 | 2000 | Paris | Cigale |

Figure 9: Table Salle

Il suffit alors de récupérer la première ligne de la table avec `LIMIT 1` qui a été ordonnée par ordre décroissant de capacité.

```
SELECT Nom FROM Salle LIMIT 1;
```

| nom |
|---|
| Bikini |

Figure 10: Résultat de la requête "Quels sont les noms des salles ayant la plus grande capacité ?"

**c. Quels sont les artistes n'ayant jamais réalisé de concert au Zénith d'Auvergne ?** Cette requête implique une négation sur la valeur du nom d'une salle. Or en CQL, il n'est pas possible d'appliquer l'opérateur "not equals" dans la clause `WHERE`[8]. Donc le modèle grosse colonne ne permet pas de répondre à cette requête.

**d. Quels sont les chanteurs ayant réalisé au moins un concert dans toutes les salles ?** Notre première idée fut de créer une table regroupant les artistes, leurs concerts et les salles. Avec le nom de l'artiste comme clé de partionnement et le identificateur d'un concert comme clé de clustering, nous pourrions récuperer pour un artiste donné la liste des salles de ses concerts. Cependant, comment comparer cette liste à l'ensemble de toutes les salles ? Une option serait d'effectuer une sous-requête comme en SQL pour récupérer dans un second temps la liste de toutes les salles depuis la table `Salle`. Or en CQL, il n'est pas possible d'effectuer des sous-requêtes.

Notre seconde idée fut d'associer à chaque artiste la liste des salles de ses concerts. Le type de donnée de la colonne serait alors `SET<TEXT>`. Dans une sous-requête, on détermine le nombre total de salles, en calculant le nombre de lignes dans la table `Salle` (possible avec un `Count(*)`) puis de comparer ce nombre avec la longueur de la liste des salles d'un artiste. Cependant, pour la même raison que précédemment, il n'est pas possible de réaliser des requêtes imbriquées, et par ailleurs d'obtenir le nombre d'éléments d'un `SET<TEXT>`. Pour ces raisons, le modèle grosse colonne ne permet pas de répondre à cette requête.

[8] https://cassandra.apache.org/doc/latest/cassandra/cql/dml.html#select

**e. Quels sont les chanteurs et les identificateurs des concerts pour lesquels il ne reste aucun billet invendu ?** Le modèle grosse colonne ne permet pas de répondre à cette requête, pour les mêmes raisons citées aux requêtes 3 et 4.

**f. Combien de billets de catégorie 1 ont été vendus par spectacles le '2020-30-11' ?** La question porte sur la date de vente d'un billet d'une catégorie particulière (1 ici) pour un spectacle donné. Le table devra donc au minimum contenir ces trois informations (date de vente, catégorie du billet, identificateur du spectacle) et les colonnes associées formeront la clé de partionnement, par anticipation de la requête. Puisque plusieurs billets peuvent être vendus pour un spectacle à la même date, il est nécessaire d'avoir une colonne supplémentaire pour différencier chaque vente : il suffit d'ajouter la colonne `Vente_ID` en tant que clé de clustering.

| **Billet_Vendu_Par_Date** | | | |
|---|---|---|---|
| Date_Vente | Spectacle_ID | Categorie_Billet | #Vente_ID |
| *DATE* | *INT* | *INT* | *INT* |

Table 3: Table *Billet_Vendu_Par_Date* pour répondre à la requête "Combien de billets de catégorie 1 ont été vendus par spectacles le '2020-30-11' ?" La clé de partionnement (souligné) est (Date_Vente, Spectacle_ID, Categorie_Billet) et la clé de clustering (précédé par un #) est (Vente_ID).

**CQL**

```
/* Creation de la table */
CREATE TABLE Billet_Vendu_par_Date(
        Vente_ID INT,
        Date_Vente DATE,
        Categorie_Billet INT,
        Spectacle_ID INT,
        PRIMARY KEY ((Date_Vente, Spectacle_ID, Categorie_Billet), Vente_IDs)
);

/* Insertion des donnees */
INSERT INTO Billet_Vendu_par_Date(Vente_ID, Date_Vente, Categorie_Billet,
↪  Spectacle_ID) VALUES (1, '2020-12-12', 1, 8);
INSERT INTO Billet_Vendu_par_Date(Vente_ID, Date_Vente, Categorie_Billet,
↪  Spectacle_ID) VALUES (2, '2020-10-07', 1, 8);
INSERT INTO Billet_Vendu_par_Date(Vente_ID, Date_Vente, Categorie_Billet,
↪  Spectacle_ID) VALUES (3, '2020-11-30', 1, 8);
INSERT INTO Billet_Vendu_par_Date(Vente_ID, Date_Vente, Categorie_Billet,
↪  Spectacle_ID) VALUES (4, '2020-11-30', 1, 4);
INSERT INTO Billet_Vendu_par_Date(Vente_ID, Date_Vente, Categorie_Billet,
↪  Spectacle_ID) VALUES (5, '2020-08-07', 2, 4);
INSERT INTO Billet_Vendu_par_Date(Vente_ID, Date_Vente, Categorie_Billet,
↪  Spectacle_ID) VALUES (6, '2020-06-17', 1, 9);
INSERT INTO Billet_Vendu_par_Date(Vente_ID, Date_Vente, Categorie_Billet,
↪  Spectacle_ID) VALUES (7, '2020-11-09', 3, 4);
INSERT INTO Billet_Vendu_par_Date(Vente_ID, Date_Vente, Categorie_Billet,
↪  Spectacle_ID) VALUES (8, '2020-12-09', 2, 7);
INSERT INTO Billet_Vendu_par_Date(Vente_ID, Date_Vente, Categorie_Billet,
↪  Spectacle_ID) VALUES (9, '2020-12-25', 4, 2);
INSERT INTO Billet_Vendu_par_Date(Vente_ID, Date_Vente, Categorie_Billet,
↪  Spectacle_ID) VALUES (10, '2020-12-12', 1, 2);

/* Contenu de la table */
SELECT * FROM Billet_Vendu_par_Date;
```

| date_vente | spectacle_id | categorie_billet | vente_id |
|---|---|---|---|
| 2020-12-12 | 2 | 1 | 10 |
| 2020-11-09 | 4 | 3 | 7 |
| 2020-11-30 | 8 | 1 | 3 |
| 2020-11-30 | 4 | 1 | 4 |
| 2020-12-09 | 7 | 2 | 8 |
| 2020-06-17 | 9 | 1 | 6 |
| 2020-10-07 | 8 | 1 | 2 |
| 2020-08-07 | 4 | 2 | 5 |
| 2020-12-12 | 8 | 1 | 1 |
| 2020-12-25 | 2 | 4 | 9 |

Figure 11: Table Billet_Vendu_par_Date

Pour un spectacle donné, on compte le nombre de lignes qui correspond à une vente de billets de catégorie 1 en date du '2020-11-30'.

```
/* Requete */
SELECT Spectacle_ID, Count(*) FROM Billet_Vendu_par_Date WHERE Spectacle_ID = 8 AND
↪  Date_Vente = '2020-11-30' AND Categorie_Billet = 1;
```

| spectacle_id | count |
|---|---|
| 8 | 1 |

Figure 12: Résultat de la requête "Combien de billets de catégorie 1 ont été vendus par spectacles le '2020-30-11' "?

### 3.2.2 Avantages du modèle

Les avantages majeurs du modèle sont possibles grâce à l'architecture intrinsèque de ce dernier.

- Lecture rapide permise par la modélisation adaptée aux requêtes. En effet l'accès aux données est très rapide puisque les tables sont créés en fonction des requêtes.
- Requête simple

### 3.2.3 Inconvénients du modèle

- Pas de requêtage complexe : impossible par exemple de répondre aux requêtes 3, 4 et 5 avec le modèle grosse colonne. Pas de `count` sur les colonnes, pas de sous-requête possible, pas de négation (not equals ne peut être utilisé dans la clause WHERE par exemple) etc.
- Jointure déléguée au développeur : risque d'erreurs lors de l'entrée des données si cela n'est pas automatisé.

# 4 Modèle document

## 4.1 Analyse du modèle

ElasticSearch peut être assimilé à un moteur de recherche capable de stocker une grande quantité de documents et que l'on peut interroger en temps réel. C'est un "Document Store" distribué dans lequel chaque document est indéxé, et requêtable. ElasticSearch est capable de gérer des centaines de serveurs et des petabytes de données (structurées ou non).

### 4.1.1 Architecture

On peut voir les unités de ElasticSearch de manière hiérarchique comme suit :

- Un cluster est la plus grande instance d'ElasticSearch. C'est un ensemble de noeuds. Il peut en contenir un ou plusieurs qui vont travailler ensemble en partageant leurs données et partager leur "workload".
- Un noeud est une instance ElasticSearch (chaque noeud lance un processus ElasticSearch). Elastic search utilise une architecture **maître esclave**. Un noeud du cluster sera élu pour être le maître et sera en charge de la gestion des changements dans les noeuds de son cluster (que ce soit concernant les modifications au niveau des données ou bein au niveau des noeuds). N'importe quel noeud peut être le noeud maître. Le client peut communiquer avec n'importe quel noeud du cluster, et puisque tous les noeuds connaissent la localisation des documents du cluster, n'importe quel noeud peut transferer la requête au noeud concerné. Le noeud auquel on s'est adressé (donc pas forcément le master node) va ensuite se charger de rassembler les réponses des noeuds concernés et envoyer la réponse finale à l'utilisateur.
- Le modèle employé est celui du **sharding**. Les données sont distribuées par bloc. Les shards peuvent être stockés sur un seul noeud ou sur plusieurs. Les shards seront répartis sur les noeuds pour équilibrer l'indexation et la charge de travail entre les noeuds. Ce travail est fait automatiquement lorsque nos clusters grandissent ou rétrécissent. Un shard contient une partie des données d'un index. Un shard peut être primaire ou être une réplique, mais chaque document de l'index n'appartient qu'à un seul shard primaire.
- Un index est l'équivalent d'une table. C'est dans un index qu'on va placer des documents. Lors de la création d'un index, on peut choisir le nombre de shards primaires et le nombre de répliques. En réalité, un index est une entité logique qui va pointer vers un ou plusieurs shards physiques (qui donc contiennent concrètement les données). Plutôt que de s'adresser aux documents ou aux shards directement, l'application s'adresse aux index.

### 4.1.2 Gestion de la réplication

Comme évoqué précédemment, le modèle employé est celui du sharding. Cela consiste à distribuer les données par bloc entre les différents noeuds du cluster. Un index est décomposé en shards qui contiennent les données.

Lors de la création d'un index, on peut choisir plusieurs paramètres (les settings). On peut notamment choisir "number_of_shards" qui correspond au nombre de shards primaires pour notre index ; et "number_of_replicas" qui correspond au nombre de répliques. Par défaut, le nombre de shards primaires est 5 et le nombre de répliques est 1. Donc si on dispose d'un index avec les paramètres par défaut, nous aurons 5 shards primaires et une réplique pour chacun, soit un total de 10 shards.

Un index avec plusieurs shards primaires peut profiter d'avoir plusieurs noeuds et répartir les shards dans les différents noeuds. Si par exemple on n'avait qu'un seul noeud au début et un index avec 2 shards primaires, les 2 seraient sur le même noeud. Si on ajoute par la suite un noeud, les shards vont être répartis un sur chaque noeud.

Si on ne dispose que d'un seul noeud, on ne peut donc pas avoir de redondance. En effet, cela n'a pas de sens de mettre des répliques dans le même noeuds que le shard primaire.

Lorsqu'un nouveau document est ajouté, il est d'abord indéxé dans le shard primaire, et ensuite dans les répliques.

Les répliques n'augmentent pas la capacité d'un index (ce sont seulement les primaires qui le font). Cependant, les répliques peuvent servir à faire des lectures. Si de très nombreuses requêtes sont faites à chaque instant sur notre index, on peut augmenter les performances en augmentant le nombre de répliques (s'il y a suffisamment de noeuds). Augmenter le nombre de réplicats améliore également la disponibilité.

L'état d'un cluster vis à vis des réplications est géré avec des couleurs.

Vert Si tous les shards primaires et répliques sont actifs. Tous les shards ont bien été alloués, le cluster est tout à fait opérationnel.

Jaune Si tous les shards primaires sont actifs, mais que tous les replicas ne le sont pas (c'est-à-dire que certains replicas sont sur le même noeuds, ce qui n'a pas de sens). Le cluster est toujours fonctionnel puisque nous avons bien accès à toutes les données. Néanmoins, certains shards sont inutiles.

Rouge Si tous les shards primaires ne sont pas actifs. Cela signifie qu'il manque quelque part des données.

Cet indicateur est très utile pour gérer la santé de la réplication.

### 4.1.3 Gestion des transactions et reprise sur panne

Changer les données d'un seul document satisfait les propriétés ACID. En revanche, effectuer une transaction impliquant plusieurs documents ne les satisfait plus (à moins que les données soient relationelles et non pas aplaties mais ça n'est pas vraiment l'intérêt de ElasticSearch). En effet, il n'y a par exemple pas atomicité puisque il n'y a pas de réinitialisation l'index à son état initial s'il y a une panne dans une partie de la transaction. Pour gérer la concurrence, il existe diverses stratégies telles que le Global Locking, Document Locking, ou Tree Locking.

L'objectif principal des Replicas shard est de gérer les pannes. En effet, si l'on dispose de plusieurs noeuds, on pourra répartir les shards sur les noeuds et ainsi prévenir les pannes.

Un cluster doit avoir obligatoirement un master node pour fonctionner correctement. Donc la première chose faite lorsqu'un noeud maître tombe en panne est d'élire un nouveau master node. Lorsqu'un noeud sur lequel vit un primary shard tombe en panne (qu'il soit maître ou non), une des réplique du shard en question est promue au rôle de shard primaire. C'est bien sûr le (nouveau si c'etait le master qui etait en panne) noeud maître qui va se charger de cette promotion.

### 4.1.4 Gestion de la cohérence, de la disponibilité et de la tolérance au partitionnement

ElasticSearch est tout d'abord construit pour être toujours disponible et de passer à l'echelle selon les besoins de ses utilisateurs. C'est un système distribué par nature qui sait de lui même comment gérer la réplication et la haute disponibilité (l'application n'aura pas besoin de s'en soucier).

Puisqu'ElasticSearch privilégie le partionnement et la disponibilité, le théorème CAP nous dit que la troisième propriété ne pourra être assurée. En effet le passage à l'échelle (bien sûr horizontal) et la haute disponibilité se fait au détriment de la cohérence : ElasticSearch va toujours répondre (relativement) rapidement aux utilisateurs, même en cas de partition du réseau. En revanche, l'utilisateur n'aura pas toujours la version la plus récente (c'est une cohérence à termes).

Cela répond en effet aux besoins d'un moteur de recherche.

## 4.2 Requêtes

### 4.2.1 Création du modèle

Il existe deux manières de créer un index dans ElasticSearch. On peut soit insérer les données, et un schéma (mapping) sera déduit automatiquement, soit définir le schéma nous-même. Néanmoins, il est recommandé d'avoir le moins possible recours au mapping automatique. Nous avons donc créé notre propre schéma pour la création du modèle, que nous avons orienté pour répondre le plus efficacement aux requêtes concernant les concerts.

```
PUT concert
{
  "mappings": {
    "dynamic":"strict",
    "properties": {
      "Jour": {
        "type": "date",
        "format": "date"
      },
      "Heure": {
        "type": "date",
        "format": "hour_minute"
      },
      "Artiste": {
        "type": "text"
      },
      "Salle": {
        "type": "object",
        "properties":{
          "Nom": {
            "type": "text"
          },
          "Adresse": {
            "type": "text"
          },
          "Capacité": {
            "type": "integer"
          }
        }
      },
      "Spectacle": {
        "type": "object",
        "properties":{
          "Titre": {
            "type": "text"
          },
          "DateDebut": {
            "type": "date",
            "format": "date"
          },
          "Durée": {
            "type": "integer"
          }
        }
      },
      "Billet": {
        "type": "object",
```

```
        "properties":{
          "Num_place": {
            "type": "integer"
          },
          "Vendu":{
            "type":"boolean"
          },
          "Date_Vente": {
            "type": "date",
            "format": "date"
          },
          "Catégorie": {
            "type": "integer"
          },
          "Prix":{
            "type": "integer"
          },
          "MoyenPaiement":{
            "type":"text"
          }
        }
      }
    }
  }
}
```

La commande pour insérer des données est alors :

```
POST concert/_doc/1
{"Artiste":"Mireille",
"Billet":{
  "Catégorie":1,
  "Vendu":true,
  "Num_place":1
  },
"Heure":"18:00",
"Jour":"21-01-05",
"Salle":{
  "Nom":"Cigale",
  "Adresse":"Paris",
  "Capacité":2000
  },
"Spectacle":{
  "DateDebut":"22-02-20",
  "Durée":2,
  "Titre":"Tournée"
  }
}


POST concert/_doc/3
{"Artiste":"Yves",
"Billet":{
  "Catégorie":1,
  "Vendu":true,
  "Date_Vente":"20-11-30",
  "Num_place":3
  },
```

```
"Heure":"20:00",
"Jour":"22-01-27",
"Salle":{
  "Nom":"Animatis",
  "Adresse":"Issoire",
  "Capacité":3
  },
"Spectacle":{
  "DateDebut":"22-01-27",
  "Durée":1,
  "Titre":"Anniv"
  }
}
```

Nous n'avons inséré ici que les concerts 1 et 3, mais c'est exactement la même chose pour les 8 autres concerts. Ces documents suffisaient pour fournir une réponse aux requêtes suivantes.

### 4.2.2 Réponses aux requêtes

Le modèle met en avant les attributs d'un concert et permet donc de répondre à la majorité des requêtes. Toutes les requêtes ci-dessous ont été testées sur une console elastic, et ont rendu le résultat attendu.

**a. Quelles sont les dates du concert de Mireille à la Cigale ?**

```
 GET concert/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {
          "Artiste": "Mireille"
        }},
        {
          "match": {
            "Salle.Nom":"Cigale"
          }
        }
        ]
    }
  },
  "fields" : [
        "Jour"
    ],
    "_source":false
}
```

**b. Quels sont les noms des salles ayant la plus grande capacité ?**

```
      GET concert/_search
{
  "query": {
    "match_all": {}
  },
```

```
  "sort":[
    {"Salle.Capacité": {
      "order":"desc"
    }}],
  "size": 1,
  "fields" : [
        "Salle.Nom"
    ],
    "_source":false
}
```

**c. Quels sont les artistes n'ayant jamais réalisé de concert à la Cigale?**

```
    GET concert/_search
{
  "query": {
    "bool": {
      "must_not": [
        {"match": {
          "Salle.Nom": "Cigale"
        }}
        ]
    }
  },
  "fields" : [
        "Artiste"
    ],
    "_source":false
}
```

**e. Quels sont les chanteurs et les identificateurs des concerts pour lesquels il ne reste aucun billet invendu ?**

```
    GET concert/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {
          "Billet.Vendu": true
        }}
        ]
    }
  },
  "fields" : [
        "id",
        "Artiste"
    ],
    "_source":false
}
```

**f. Combien de billets de catégorie 1 ont été vendus le '11-30-20' ?**

```
    GET concert/_doc/_count
```

```
{
  "query": {
    "bool": {
      "must": [
        {"match": {
          "Billet.Catégorie": 1
        }},
        {
          "match": {
            "Billet.Date_Vente":"20-11-30"
          }
        }
        ]
    }
  }
}
```

Pour la dernière requête, le fait de ne plus avoir le modèle relationnel rend le group by très fastidieux. Nous avons simplement affiché le nombre de billets catégorie 1 vendus ce jour.

## 4.3 Choix de modélisation

ElasticSearch (comme la plupart des systèmes NoSQL) traite les données de manière applatie (opposé à normalisé). Un index est donc une collection applatie de documents indépendants entre eux. Selon les consignes d'ElasticSearch, un document doit contenir toutes les informations requises pour décider si ce document matche une requête de recherche.

Nous avons donc décidé d'orienter le document vers les concerts et d'insérer dans un document concert toutes les informations le concernant.

Lors de la définition du schéma, nous avons choisi "dynamic" : "strict", ce qui signifie qu'on ne pourra indexer dans concert que des documents qui correspondent exactement au schema défini, dans le sens où on ne peut pas mettre de champs en plus. En revanche, on n'est pas obligés de renseigner tous les champs.

L'id du document ne fait pas partie du mapping mais on le définit directement lorsqu'on insert un document (Par exemple POST concert/_doc/3 insère un document avec l'id 3).

Un document concert dispose des attributs suivants :

- Jour : correspond à la date du concert, et sera de type "date" (c'est à dire 'yy-mm-dd' ce qui est un type simple et correspond parfaitement à nos besoins).
- Heure : correspond à l'heure du concert et sera de type "hour_minute" (c'est à dire "hh:mm" type qui correspond egalement parfaitement a nos besoins)
- Artiste : le ou les artistes qui vont faire le concert. Cette donnée sera simplement de type texte (on aurait aussi pu mettre keywords). Il aurait été possible de mettre un objet artiste avec plus de details (peut être nom prénom date naissance etc.)
- Salle : renseigne sur la salle du concert. Type "object", c'est à dire qu'une salle est en fait un document, lui même contenant des attributs, qui sont le nom (text), l'adresse (text) et la capacité(integer). Cet objet contient en fait les informations de la ligne de la table salle qui aurait été jointe avec la ligne de la table concert correspondante.
- Spectacle : Renseigne sur le spectacle dont fait partie le concert. C'est également un objet. Ses attributs sont titre (text), DateDébut (date) et durée (integer).

- Billet : Rensigne sur tous les billets concernant un concert. Puisqu'il y a une relation one to one entre un billet et une vente (on ne peut pas vendre un billet plusieurs fois, et on ne vend qu'un billet à la fois ici), nous avons voulu rassembler ces informations dans le même objet. Le champs Billet peut contenir plusieurs valeurs (mais qui doivent avoir le même type, donc dans notre cas représenter le même objet billet). En effet, dans ElasticSearch, les arrays ne nécéssitent pas de type dédié puisque chaque champs peut contenir 0 ou plusieurs valeurs par défaut (mais toutes les valeurs de l'array doivent être du même type). On peut donc mettre ici tous les billets disponibles à la vente, et donner plus de renseignements s'ils sont vendus. En effet, un billet contiendra des informations de base, et ne contiendra les informations de vente que s'il est vendu. Les attributs sont :
  - Num_place : l'emplacement acheté de type integer.
  - Vendu : indique si le billet est vendu ou non et est de type 'boolean'. Cet attribut n'est pas obligatoire mais j'ai trouvé ça plus clair et plus rapide pour certaines requêtes.
  - Date_Vente : la date de la vente, ce champs ne sera rempli que si le billet a été vendu.
  - Catégorie : la catégorie du billet de type integer
  - Prix : prix du billet de type integer
  - MoyenPaiement : le moyen pour payer le billet, ne sera rempli que si le billet a été vendu également.

### 4.3.1 Avantages et inconvénients

Ce modèle permet d'avoir accès à de nombreuses informations concernant les concerts sans avoir à faire de jointure. Nous avons pu répondre à la majorité des requêtes de manière assez directe. Le fait d'avoir ajouté un champs booléen "Vendu" dans les billets est d'ailleurs très pratique pour répondre à quelques requêtes concernant les billets. Néanmoins, lorsque l'on veut faire des requêtes sur les sous-document, ça n'est pas toujours l'idéal, même si le format json aide à cela. Enfin, les group by sont rendus assez compliqués.