

CLOUD COMPUTING, CM4

STOCKAGE DE DONNÉES DANS LE CLOUD

Pépin Rémi, Ensai, 2023

remi.pepin@ensai.fr

LE STOCKAGE DE DONNÉES DANS LE CLOUD

LES DIFFÉRENTS SERVICES DE STOCKAGE



LES DIFFÉRENTS SERVICES DE STOCKAGE 1/2



- **Stockage block** : données directement sur un disque dur (SSH/HDD). Pour stocker des données pour une VM. AWS EBS, GCP Persistent Disk
- **Stockage réseau** : données sur un stockage partagé entre plusieurs VM. Pensez aux dossiers partagés de l'Ensaï. AWS EFS, GCP Filestore, Azure Files
- **Stockage objet** : données stockées sur des serveurs distants. Pas de point de montage sur la machine. Fichiers accessibles via API REST. AWS S3, GCP Cloud Storage, Azure Blob Storage

LES DIFFÉRENTS SERVICES DE STOCKAGE 2/2

- **Base de données relationnelle** : pour des données tabulaires. On ne stocke plus un fichier mais des données structurées et on utilise SQL pour les requêter. Possibilité de faire des jointures. AWS RDS, GCP Cloud SQL
- **Base de données no-SQL** : pour des données semi structurées voir non structurées. Interaction avec les données possibles, par contre plus de jointure. AWS DynamoDB, GCP Cloud Big Table
- **Cache** : les données sont stockées en RAM et plus sur disque. Assure une grande performance, par contre les données peuvent être perdues. Uniquement pour accélérer l'accès à des données. AWS ElastiCache, GCP Memorystore, Azure Cache

AMAZON S3

IN A NUTSHELL

- Stockage *illimité* ( )
- Stockage non structuré (image, vidéo, texte ...)
- 5TB max par objet
- Objet accessible via API Rest (identifiant unique par objet)
- Possibilité de versionner et chiffrer les objets
- Gestion fine des accès
- Différents niveaux de stockage (de fréquent à archive) pour différents cas d'utilisation

LES AVANTAGES

- **Durabilité** : 99,9999999999 (onze 9).
- **Disponibilité** : les données sont toujours accessible via Internet. < 60 minutes de downtime par an
- **Passage à l'échelle** : stockage *infini*
- **Sécurité** : chiffrement, gestion des droits fin
- **Performance** : peut gérer un grand nombre de requêtes, s'insère parfaitement dans l'écosystème AWS
- **Traitement des données** : avec AWS GLue et Amazon Athena, S3 devient un data lake (on peut requêter les données)
- **Cycle de vie** : les données peuvent être supprimée automatiquement, passer dans un autre niveau de stockage et versionnées si besoin.

LES NIVEAUX DE STOCKAGE

1. **S3 standard** : meilleures performances, mais le plus cher
2. **S3 standard IA** : pour données accédées rarement. Les mêmes perfs que S3 standard, moins cher en terme de Go/mois mais l'accès aux données est plus cher
3. **S3 One Zone IA** : les données ne sont stockées que dans un data center. Moins cher que standard IA, mais les perfs sont inférieures
4. **S3 Glacier, deep archive** : les solutions les moins chères, mais l'accès aux données est cher et prend du temps

LES NIVEAUX DE STOCKAGE

	S3 Standard	S3 Standard - IA	S3 One Zone-IA	S3 Glacier Instant Retrieval	S3 Glacier Flexible Retrieval	S3 Glacier Deep Archive
Conçu pour la durabilité	99,999999999 % (11 9s)	99,999999999 % (11 9s)	99,999999999 % (11 9s)	99,999999999 % (11 9s)	99,999999999 % (11 9s)	99,999999999 % (11 9s)
Conçu pour la disponibilité	99,99 %	99,9 %	99,5 %	99,9 %	99,99 %	99,99 %
Disponibilité SLA	99,9 %	99 %	99 %	99 %	9,9 %	99,9 %
Zones de disponibilité	≥3	≥3	1	≥3	≥3	≥3
Frais de capacité minimale par objet	N/A	128 Ko	128 Ko	128 Ko	40 Ko	40 Ko
Frais minimum de durée de stockage	N/A	30 jours	30 jours	90 jours	90 jours	180 jours
Frais d'extraction	N/A	par Go extrait	par Go extrait	par Go extrait	par Go extrait	par Go extrait
Latence du premier octet	millisecondes	millisecondes	millisecondes	millisecondes	minutes ou heures	heures
Type de stockage	Objet	Objet	Objet	Objet	Objet	Objet
Transitions du cycle de vie	Oui	Oui	Oui	Oui	Oui	Oui

CAS D'UTILISATION

- Stockage pour application : photos pour un réseau social par exemple
- Data Lake : espace de stockage illimité peu onéreux. On connecte des outils pour traiter les données
- Sauvegarde pour données critiques : chiffrement, disponibilité, durabilité, droit
- Archivage de données : S3 Glacier, deep archive
- Migration : Snowcone, Snowball, Snowmobile

S3 ET CDKTS

```
from constructs import Construct
from cdktf import App, TerraformStack
from cdktf_cdktf_provider_aws.provider import AwsProvider
from cdktf_cdktf_provider_aws.s3_bucket import S3Bucket

class S3Stack(TerraformStack):
    def __init__(self, scope: Construct, id: str):
        super().__init__(scope, id)

        AwsProvider(self, "AWS", region="us-east-1")

        bucket = S3Bucket(
            self, "s3_bucket",
            bucket_prefix = "my-cdtf-test-bucket",
            acl="private",
            force_destroy=True
        )

app = App()
S3Stack(app, "S3")










app.synth()
```

S3 ET PYTHON

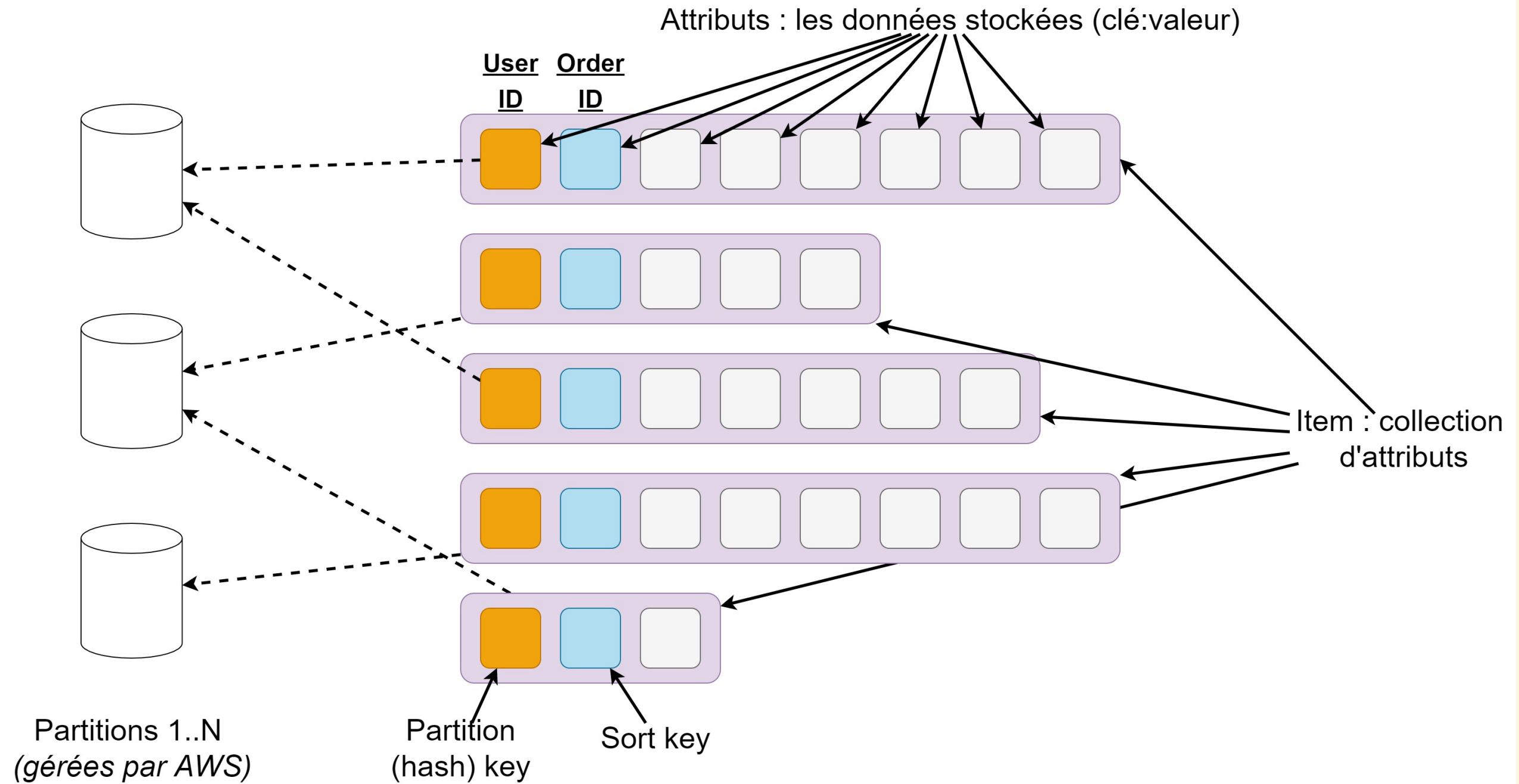
```
import boto3
# Create an S3 resource
s3 = boto3.resource('s3')
# Create a bucket
s3.create_bucket(Bucket='mybucket')
# Upload file
with open('/tmp/hello.txt', 'rb') as file:
    s3.Object('mybucket', 'hello_s3.txt').put(Body=file)
# Download file
s3.Bucket('mybucket').download_file('hello_s3.txt', 'hello.txt')
# Delete file
s3 = boto3.resource('s3')
s3.Object('mybucket', 'hello_s3.txt').delete()
```

DYNAMODB

IN A NUTSHELL

- Base de données No-SQL  **clé:valeur**
- Serverless : pas d'infra à gérer côté utilisateur
- Stockage *illimité* (       )
- Paiement du stockage, des opérations de lecture et écriture (RCU, WCU)
- Passage à l'échelle automatique
- Gestion fine des accès (uniquement certaines lignes/attributs)
- Réplication entre régions possible
- Peut faire des opérations en mode ACID (Atomicité, Cohérence, Isolation, Durabilité)

MODÈLE DE DONNÉES



MODÈLE DE DONNÉES

- Pas de schéma stricte pour les données. Uniquement partition key d'obligatoire et la sort key si définie
- **Partition key** : sert à déterminer où l'objet sera stocké (clé hashage dans un table de hachage). Si pas de sorted key, sert de **primary key**
- **Sorted key** : Permet de trier les données avec la même partition key. Avec elle, elles définissent la **composite primary key**.
- GET/PUT sur les données en utilisant la primary key UNIQUEMENT.
- Possibilité d'avoir des indexes secondaires pour requête sur autre attributs (mais 💰💰)
- AWS gère le stockage sur différentes partitions et augmente leur nombres si nécessaire. Attention aux déséquilibres.

Même si DynamoDB est serverless, le service n'est pas brainless. Définir une bonne clef primaire permet de meilleurs performance et de réduire les coûts

CONCEVOIR UNE BASE NO SQL

No SQL : pas de schéma, pas de jointure : ne pas penser en model relationnel

Concevoir sa base pour répondre à des besoins spécifiques (*access pattern*)

- S'assurer que l'on peut identifier une entité (PK)
- Ne pas avoir peur de mettre plusieurs type d'entité par table
- Limiter les requêtes pour répondre à un besoin
- Placer des indexes secondaires si besoin

Au lieu de faire une base "généraliste" on va faire une/des bases spécialisées

CAS D'UTILISATION

- Vente en lignes : Amazon utilise DynamoDB pour les paniers
- Cache : stockage des états d'un programme
- lot : états des objets
- Jeux vidéo : leaderboard en temps réel

CDKTF ET DYNAMODB

```
from constructs import Construct
from cdktf import App, TerraformStack
from cdktf_cdktf_provider_aws.provider import AwsProvider
from cdktf_cdktf_provider_aws.dynamodb_table import DynamodbTable, DynamodbTableAttribute

class DynamoDBStack(TerraformStack):
    def __init__(self, scope: Construct, id: str):
        super().__init__(scope, id)

        AwsProvider(self, "AWS", region="us-east-1")

        bucket = DynamodbTable(
            self, "DynamoDB-table",
            name= "user_score",
            hash_key="username",
            range_key="lastname",
            attribute=[
                DynamodbTableAttribute(name="username", type="S" ),
                DynamodbTableAttribute(name="lastname", type="S" )
            ],
            billing_mode="PROVISIONED",
            read_capacity=5,
            write_capacity=5
        )

app = App()
DynamoDBStack(app, "DynamoDBStack")
app.synth()
```

DYNAMODB ET PYTHON : CRÉATION TABLE

```
import boto3
# Get the service resource.
dynamodb = boto3.resource('dynamodb')
# Create the DynamoDB table.
table = dynamodb.create_table(
    TableName='users',
    KeySchema=[
        {'AttributeName': 'username', 'KeyType': 'HASH'},
        {'AttributeName': 'last_name', 'KeyType': 'RANGE'}
    ],
    AttributeDefinitions=[
        {'AttributeName': 'username', 'AttributeType': 'S'},
        {'AttributeName': 'last_name', 'AttributeType': 'S'},
    ],
    ProvisionedThroughput={'ReadCapacityUnits': 5, 'WriteCapacityUnits': 5}
)
# Wait until the table exists.
table.meta.client.get_waiter('table_exists').wait(TableName='users')
```

DYNAMODB ET PYTHON : AJOUT/SUPPRESSION ÉLÉMENT

```
import boto3
# Get the service resource.
dynamodb = boto3.resource('dynamodb')
# Get the table.
table = dynamodb.Table('users')
# Put item to the table.
table.put_item(
    Item={
        'username': 'janedoe',
        'first_name': 'Jane',
        'last_name': 'Doe',
        'age': 25,
        'account_type': 'standard_user',
    }
)
# Delete item from the table.
table.delete_item(
    Key={
        'username': 'janedoe',
        'last_name': 'Doe'
    }
)
```

DYNAMODB ET PYTHON : AJOUT EN BATCH

```
import boto3
# Get the service resource.
dynamodb = boto3.resource('dynamodb')
# Get the table.
table = dynamodb.Table('users')
# Batch writing item. Only one big query, cost less and it's quicker
with table.batch_writer() as batch:
    for i in range(50):
        batch.put_item(
            Item={
                'account_type': 'anonymous',
                'username': 'user' + str(i),
                'first_name': 'unknown',
                'last_name': 'unknown'
            }
        )
```

DYNAMODB ET PYTHON : RÉCUPÉRATION D'UN OBJET

```
import boto3
# Get the service resource.
dynamodb = boto3.resource('dynamodb')
# Get the table.
table = dynamodb.Table('users')
# Get item from the table.
response = table.get_item(
    Key={
        'username': 'janedoe',
        'last_name': 'Doe'
    }
)
item = response['Item']
print(item)
```


DYNAMODB ET PYTHON : MODIFICATION D'UN OBJET

```
import boto3
# Get the service resource.
dynamodb = boto3.resource('dynamodb')
# Get the table.
table = dynamodb.Table('users')
# Update item from the table.
table.update_item(
    Key={
        'username': 'janedoe',
        'last_name': 'Doe'
    },
    UpdateExpression='SET age = :val1',
    ExpressionAttributeValues={
        ':val1': 26
    }
)
```

TP S3 - DYNAMODB

Objectif

- Créer un bucket avec Terraform et interagir avec via Python
- Créer une table DynamoDB avec Terraform et interagir avec via Python

