

# Résumé CLD - TE2

## CLD

PaaS, Storage as a Service, NoSQL, Container cluster management, IaaS and labs

## Résumé du document

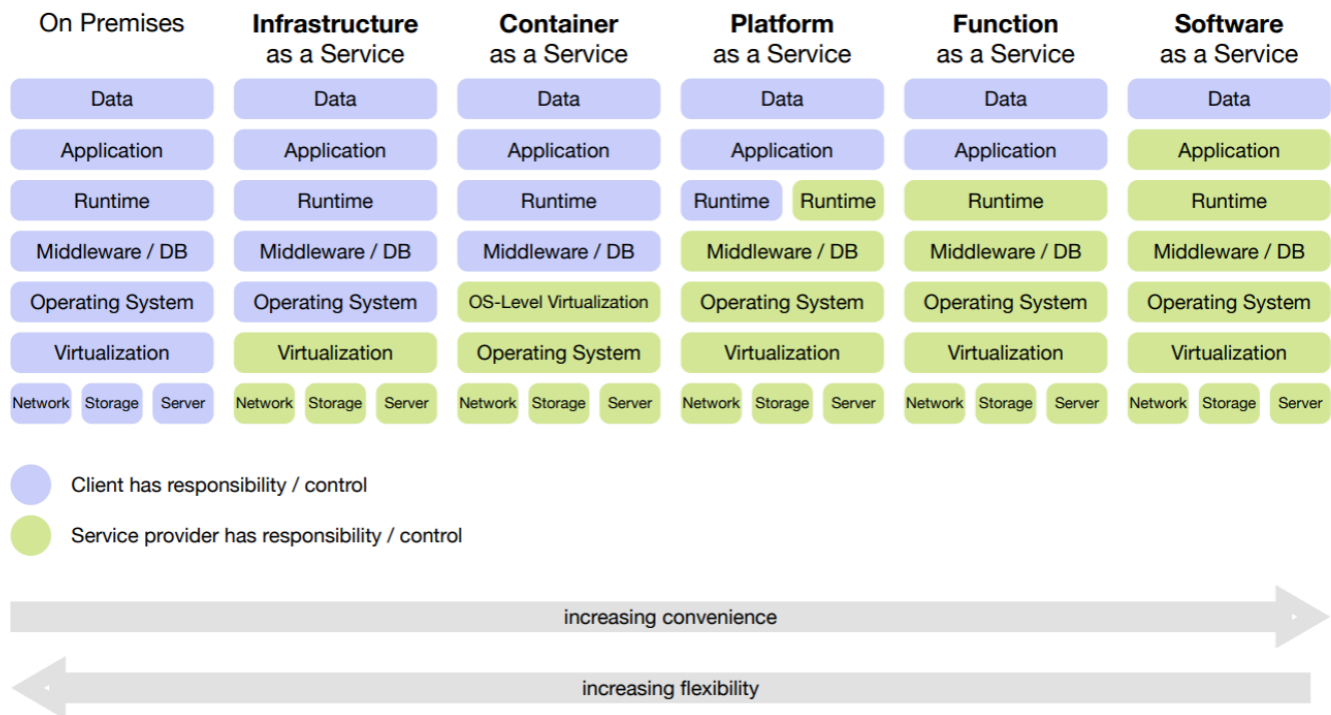
### Table des matières

|  |           |
|--|-----------|
| <b>1. PaaS</b>   | <b>3</b>  |
| 1.1. Risque de verrouillage des fournisseurs                         | 3         |
| 1.2. Google App Engine   | 3         |
| 1.2.1. Architecture  | 4         |
| 1.2.2. Request handler   | 4         |
| 1.2.3. App instance  | 5         |
| 1.2.4. Type de scaling   | 5         |
| <b>2. Storage as a Service</b>                                       | <b>6</b>  |
| 2.1. Base de données relationnelle                                   | 6         |
| 2.1.1. Problèmes des bases de données relationnelles                 | 6         |
| 2.1.2. Problèmes de scalabilité                                      | 6         |
| 2.1.3. Pourquoi les bases relationnelles continuent d'être utilisées | 7         |
| 2.2. Object-oriented databases                                       | 7         |
| <b>3. NoSQL</b>  | <b>8</b>  |
| 3.1. Caractéristiques  | 8         |
| 3.2. Modèle clé-valeur   | 8         |
| 3.3. Modèle de colonnes  | 8         |
| 3.4. Modèle de documents   | 9         |
| 3.5. Modèle de graphe  | 9         |
| 3.6. Comment utiliser NoSQL  | 10        |
| 3.6.1. Utilisation recommandée                                       | 10        |
| <b>4. Base de données distribuée</b>                                 | <b>11</b> |
| 4.1. Sharding vs Replication   | 11        |
| 4.1.1. Repliation  | 12        |
| 4.1.2. Sharding  | 12        |
| 4.1.2.1. Tables de hachage   | 13        |
| 4.1.2.2. Consistant hashing  | 13        |
| 4.1.2.3. Ecritures concurrentes                                      | 13        |
| 4.1.2.3.1. Transactions lecture-écriture                             | 14        |
| 4.1.2.3.2. Transactions écriture                                     | 15        |
| 4.1.2.4. Consistance avec de la réplication                          | 15        |
| 4.1.2.4.1. Tolerance à la réplication                                | 16        |
| 4.1.2.5. Consistance forte avec commit à deux phases                 | 16        |
| 4.1.2.6. Disponibilité sans consistance forte                        | 16        |
| <b>5. Conteneurs logiciels</b>                                       | <b>17</b> |
| 5.1. Conteneurs vs Machines virtuelles                               | 17        |
| 5.2. Création et téléchargement d'images de conteneurs               | 17        |
| <b>6. Gestion des clusters de conteneurs</b>                         | <b>18</b> |
| 6.1. Introduction  | 18        |

|  |           |
|--|-----------|
| 6.2. Orchestration des conteneurs .....                  | 18        |
| 6.3. YAML (Yet Another Markup Language) .....            | 18        |
| 6.3.1. Structure .....                                   | 18        |
| 6.3.2. Exemple de YAML .....                             | 18        |
| <b>7. Kubernetes .....</b>                               | <b>19</b> |
| 7.1. Introduction .....                                  | 19        |
| 7.2. Anatomie d'un cluster .....                         | 20        |
| 7.2.1. Composants du nœud maître .....                   | 20        |
| 7.2.2. Composants du nœud de travail .....               | 20        |
| 7.3. Concepts principaux .....                           | 20        |
| 7.4. Concepts communs .....                              | 20        |
| 7.5. Déployer une application : IaaS vs Kubernetes ..... | 20        |
| 7.6. Exemple de YAML Kubernetes .....                    | 21        |
| 7.7. Pods .....  | 21        |
| 7.7.1. Loosely vs Tightly Coupled Containers .....       | 21        |
| 7.8. Volumes .....                                       | 22        |
| 7.8.1. Monter un volume persistant .....                 | 23        |
| 7.9. Replication Controller .....                        | 23        |
| 7.10. Services .....                                     | 23        |
| 7.10.1. Type de services .....                           | 23        |

# 1. PaaS

Platform-as-a-Service (PaaS) permet aux développeurs de déployer des applications scalables dans le cloud, en se concentrant sur le code de l'application et en déléguant le reste au fournisseur de services cloud.



Dans un PaaS nous avons donc la main uniquement sur l'application et les données, le reste est géré par le fournisseur de service cloud. Le PaaS existe aussi pour l'IoT ainsi que l'IA.

Le cloud provider gère donc tout ce qui concerne :

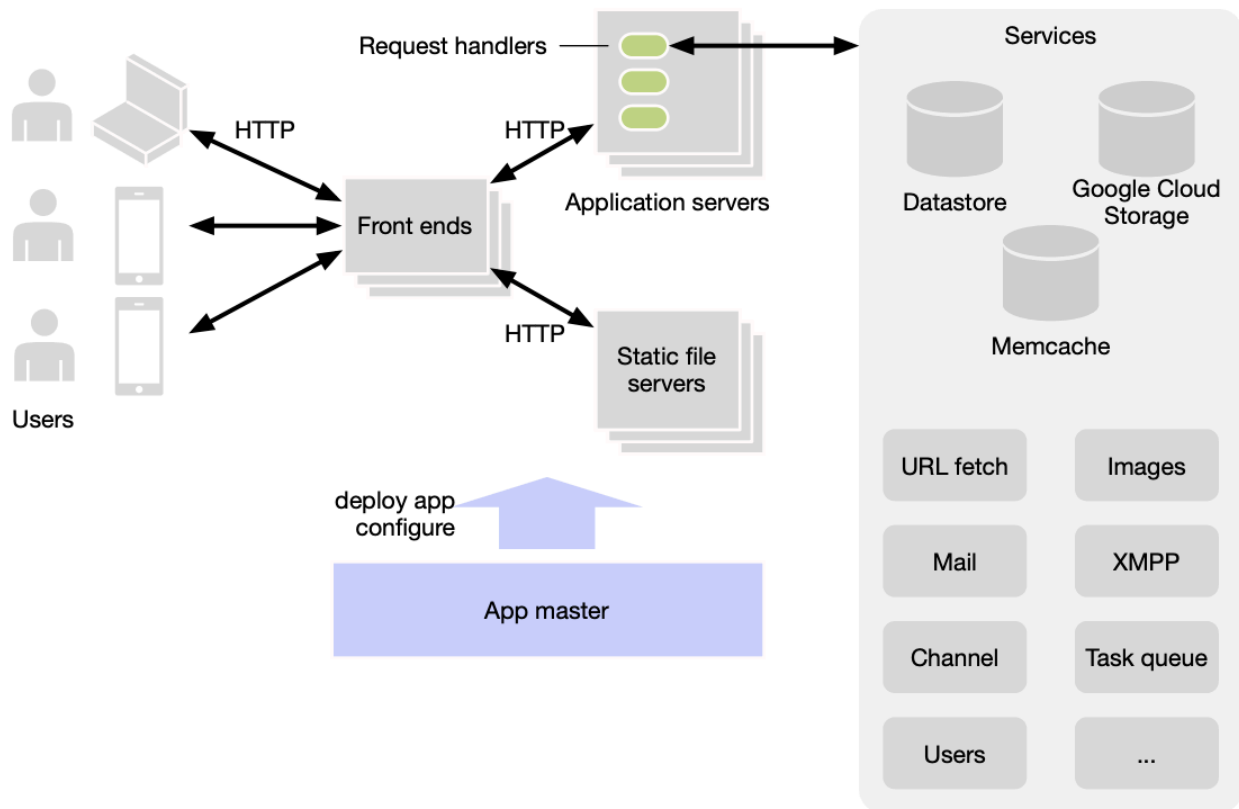
- provisionnement automatique des machines virtuelles (VM)
- maintenance du système d'exploitation
- installation et maintenance du serveur d'applications web
- installation de l'application et déploiement des mises à jour de l'application
- provisionnement des load-balancers, vérification de l'état des VM
- mise à l'échelle automatique avec ajustement adaptatif
- bases de données et data stores gérés

## 1.1. Risque de verrouillage des fournisseurs

Dans de nombreux cas les solutions de PaaS sont propriétaires (architectures et API) et ne sont pas compatibles entre elles. Il est donc difficile de migrer d'un fournisseur à un autre. Il est donc important de bien choisir son fournisseur de PaaS ou d'opter pour une solution open-source.

## 1.2. Google App Engine

Google App Engine est un PaaS qui permet de déployer des applications web et mobiles. Il est possible de déployer des applications web et mobiles sans se soucier de la gestion des serveurs.

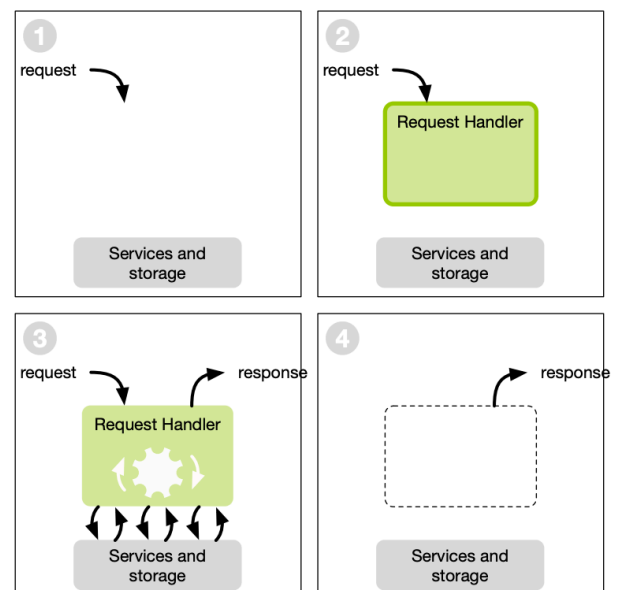


### 1.2.1. Architecture

- **Multi-tenant** : La plateforme de Google déploie automatiquement le code de votre application et ajuste l'échelle pour s'adapter à la charge.
  - Pas de contrôle des serveurs, équilibreur de charge ou groupes de mise à l'échelle automatique.
  - La plateforme contrôle et partage les ressources entre tous les locataires.
- **Frontend** : Reçoit les requêtes HTTP, identifie le locataire et l'application, et route la requête.
  - Effectue des vérifications de l'état des instances d'application et assure l'équilibrage de la charge.
- **Serveurs d'application** : Exécutent le code de l'application dans des instances conteneurs ou machines virtuelles.
  - Fournit le système d'exploitation et l'environnement d'exécution (Java, Python, etc.).
- **App master** : Gère les instances d'application, déploie le code, remplace les instances défectueuses et ajuste automatiquement le nombre d'instances.

### 1.2.2. Request handler

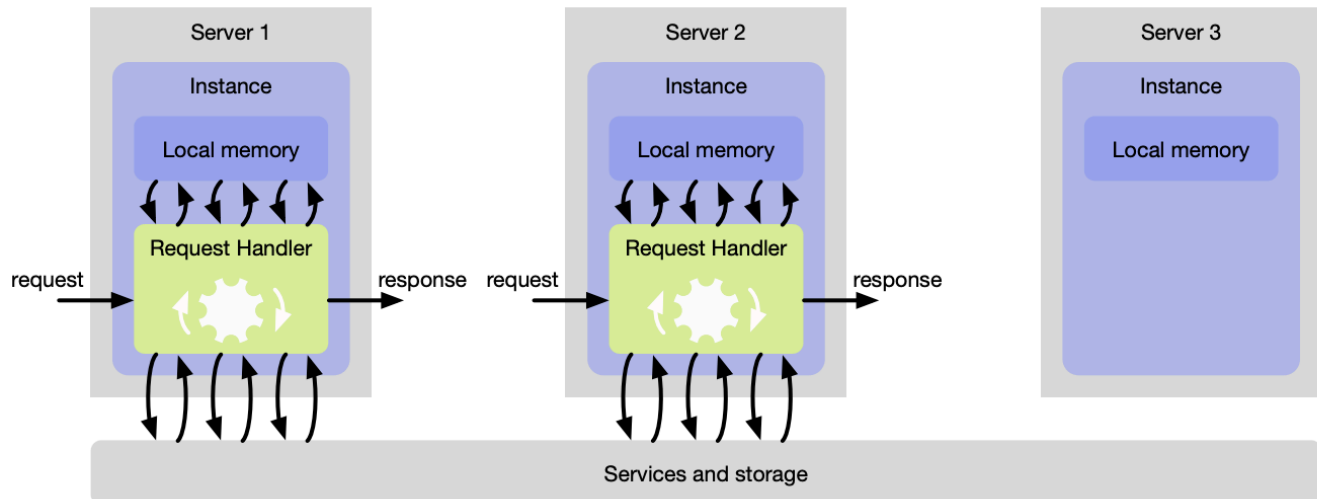
- **Request Handler** : Code responsable de créer une réponse HTTP à une requête HTTP.
- Exemple : Java servlet.
- Cycle de vie dans App Engine :
  1. Une requête arrive.
  2. Le Request Handler est créé et reçoit la requête.
  3. Le Request Handler crée la réponse, en utilisant éventuellement d'autres services cloud et stockage cloud.
  4. Une fois la réponse envoyée, App Engine peut supprimer le Request Handler de la mémoire.
- Le Request Handler doit être sans état pour que ce cycle de vie fonctionne (sauf en cas de mise à l'échelle manuelle).



- Lorsque le nombre de requêtes augmente, App Engine alloue des Request Handlers supplémentaires.
- Tous les Request Handlers traitent les requêtes en parallèle (mise à l'échelle automatique).

### 1.2.3. App instance

- App Instance : Conteneur ou machine virtuelle qui exécute le code de l'application.



- Le cycle de vie du Request Handler peut être satisfaisant, mais il n'est pas pratique d'instancier et de détruire un programme pour chaque requête.
  - L'initialisation d'un programme est coûteuse en temps, surtout pour la mémoire locale.
- Les instances sont des conteneurs dans lesquels vivent les Request Handlers.
  - Une instance conserve la mémoire locale initialisée pour les requêtes suivantes.
- Une application a un certain nombre d'instances allouées pour traiter les requêtes.
  - Le front-end distribue les requêtes parmi les instances disponibles.
  - Si nécessaire, App Engine alloue de nouvelles instances.
  - Si une instance n'est pas utilisée pendant un certain temps, App Engine la libère.

On peut voir les app instances dans le tableau de bord de Google Cloud Platform.

**Prix:** vous êtes facturé en fonction du temps d'allocation d'une instance (parmi d'autres frais).

### 1.2.4. Type de scaling

Lors du téléchargement d'une application, vous spécifiez le type de mise à l'échelle dans un fichier de configuration. Le type de mise à l'échelle contrôle comment la plateforme crée les instances.

- **Mise à l'échelle manuelle** : Vous spécifiez manuellement le nombre d'instances à instancier. Les instances fonctionnent en continu, permettant à l'application d'effectuer une initialisation complexe et de s'appuyer sur l'état de sa mémoire au fil du temps (elles peuvent **stateful**).
- **Mise à l'échelle basique** : La plateforme commence avec zéro instance et crée une instance lorsqu'une requête est reçue. L'instance est arrêtée lorsque l'application devient inactive. L'application doit être **stateless**. Le développeur contrôle directement deux paramètres : le **nombre maximum d'instances** et le **délai d'inactivité** pour supprimer les instances.
- **Mise à l'échelle automatique** : La plateforme décide quand créer et supprimer des instances en utilisant des algorithmes prédictifs basés sur le taux de requêtes, les latences de réponse, et d'autres métriques de l'application. Le développeur n'a qu'un contrôle indirect en ajustant certains paramètres. L'application doit être stateless. Ce type de mise à l'échelle est celui par défaut.

## 2. Storage as a Service

Le stockage en tant que service (STaaS) est un modèle de stockage de données dans le cloud, où les données sont stockées sur des serveurs distants accessibles via Internet. Les services de stockage en tant que service peuvent être utilisés pour stocker des données structurées, non structurées ou semi-structurées.

Il existe 3 catégories bien distinctes de STaaS :

- **Block storage** : Stockage de blocs, utilisé pour stocker des données brutes, généralement utilisé pour les bases de données.
- **Object storage** : Stockage d'objets, utilisé pour stocker des objets (fichiers, images, vidéos, etc.), généralement utilisé pour les applications web.
- **Database as a Service** : Stockage de données structurées, utilisé pour stocker des données structurées, généralement utilisé pour les applications web.

### 2.1. Base de données relationnelle

- **Stockage de données persistantes** :
  - Permet de stocker de grandes quantités de données sur le disque, tout en permettant aux applications d'accéder aux données nécessaires via des requêtes.
- **Intégration d'applications** :
  - De nombreuses applications au sein d'une entreprise ont besoin de partager des informations.
  - En utilisant la base de données commune, on assure que toutes ces applications disposent de données cohérentes et à jour.
- **Principalement standardisé** :
  - Le modèle relationnel est largement utilisé et compris.
  - L'interaction avec la base de données se fait avec SQL, un langage (principalement) standard.
  - Ce degré de standardisation permet de maintenir une familiarité pour éviter d'apprendre de nouvelles choses.
- **Contrôle de la concurrence** :
  - De nombreux utilisateurs accèdent aux mêmes informations en même temps.
  - Gérer cette concurrence est difficile à programmer, donc les bases de données fournissent des transactions pour garantir une interaction cohérente.
- **Reporting** :
  - Le modèle de données simple et la standardisation de SQL en font une base pour de nombreux outils de reporting.

#### 2.1.1. Problèmes des bases de données relationnelles

- **Impedance mismatch** : La différence entre les structures de données en mémoire (objets) du programme et le modèle relationnel de la base de données.
  - **Modèle relationnel** : Ensemble de tuples avec des valeurs simples.
  - **Structure de données du programme** : Hiérarchie d'objets.
- Source de frustration pour les développeurs.

#### 2.1.2. Problèmes de scalabilité

- Les bases de données relationnelles sont conçues pour fonctionner sur une seule machine, donc pour les mettre à l'échelle, vous devez acheter une machine plus puissante.
- Cependant, il est moins cher et plus efficace de mettre à l'échelle horizontalement en achetant de nombreuses machines.
- Les machines dans ces grands clusters sont individuellement peu fiables, mais le cluster dans son ensemble continue de fonctionner même lorsque des machines tombent en panne, donc le cluster global est fiable.
- **Les bases de données relationnelles ne fonctionnent pas bien sur des clusters.**

### 2.1.3. Pourquoi les bases relationnelles continuent d'être utilisées

- **Le modèle relationnel reste pertinent** : le modèle tabulaire convient à de nombreux types de données, notamment lorsque vous devez analyser les données et les réassembler de différentes manières pour différents usages.
- **Transactions ACID** : Pour fonctionner efficacement sur un cluster, la plupart des bases de données NoSQL ont une capacité transactionnelle limitée. Souvent, cela suffit... mais pas toujours.
- **Outils** : La longue domination de SQL signifie que de nombreux outils ont été développés pour fonctionner avec les bases de données SQL. Les outils pour les systèmes de stockage de données alternatifs sont beaucoup plus limités.
- **Familiarité** : Les systèmes NoSQL sont encore récents, donc les gens ne sont pas familiers avec leur utilisation. Par conséquent, nous ne devrions pas les utiliser dans des projets utilitaires où leurs avantages auraient moins d'impact.

## 2.2. Object-oriented databases

- Invention au milieu des années 1990, les bases de données orientées objet promettaient de résoudre le problème du déséquilibre d'impédance.
- Contrairement aux attentes, elles n'ont pas connu beaucoup de succès.
- Les bases de données SQL sont restées dominantes.
  - La principale raison : elles sont utilisées comme bases de données d'intégration entre différentes applications.

### 3. NoSQL

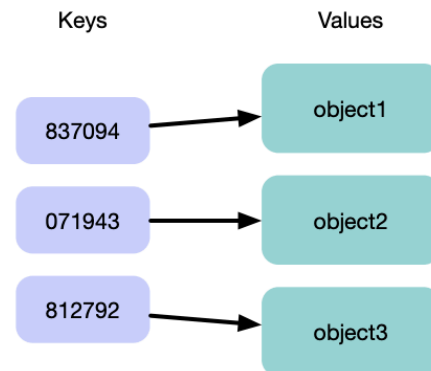
NoSQL signifie “Not Only SQL”. Il s’agit d’une catégorie de bases de données qui ne suivent pas le modèle de base de données relationnelle traditionnel. Les bases de données NoSQL sont conçues pour les applications web modernes, qui nécessitent une scalabilité horizontale, une faible latence et une disponibilité élevée.

#### 3.1. Caractéristiques

- Ils n’utilisent pas le modèle de données relationnel, et par conséquent, n’utilisent pas le langage SQL.
- Ils sont généralement conçus pour fonctionner sur un cluster.
- Ils n’ont pas de schéma fixe, ce qui vous permet de stocker n’importe quelle donnée dans n’importe quel enregistrement.
- Ils ont tendance à être open source (lorsqu’ils sont proposés en tant que logiciel).

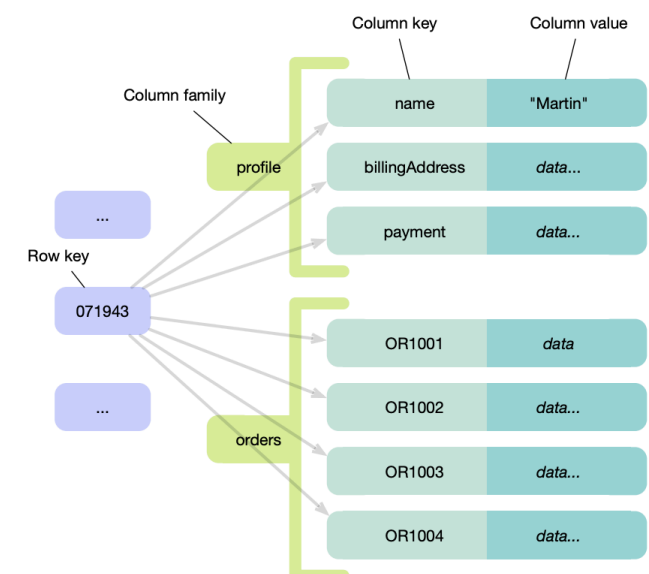
#### 3.2. Modèle clé-valeur

- La base de données permet de stocker des objets arbitraires (un nom, une image, un document, ...) et de les récupérer avec une clé.
- C’est le principe d’une table de hachage, mais stockée de manière persistante sur un disque.



#### 3.3. Modèle de colonnes

- **Clé de ligne :**
  - Clé unique pour chaque ligne.
  - Une ligne contient plusieurs familles de colonnes.
  - Accessible via la clé.
- **Famille de colonnes :**
  - Une combinaison de colonnes qui vont ensemble.
  - A un nom.
  - Contient plusieurs paires de clés de colonne / valeurs de colonne.
- **Clé de colonne / Valeur de colonne :**
  - Paire clé/valeur contenant les données.





### 3.4. Modèle de documents

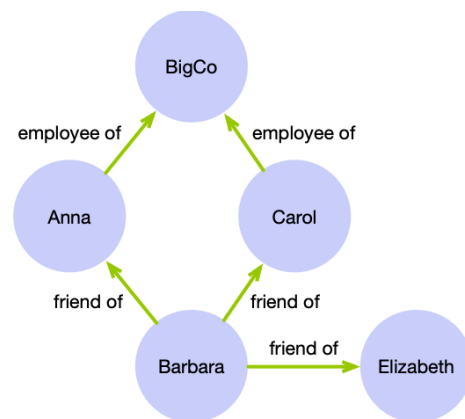
Ce modèle de base de données permet de stocker des documents, la ou chaque document peut avoir sa propre structure. La structure est souvent représentée en JSON. Cela permet au développeur de faire des requêtes directement sur la structure de données.

```
{ "id": 1001,
  "customer_id": 7231,
  "line8items": [
    { "product_id": 4555, "quantity": 8 },
    { "product_id": 7655, "quantity": 4 },
  ],
  "discount8code": Y
}
```

```
{ "id": "1002",
  "customer_id": 9831,
  "line8items": [
    { "product_id": 4555, "quantity": 3 },
    { "product_id": 2155, "quantity": 4 },
    { "product_id": 6384, "quantity": 1 },
  ],
}
```

### 3.5. Modèle de graphe

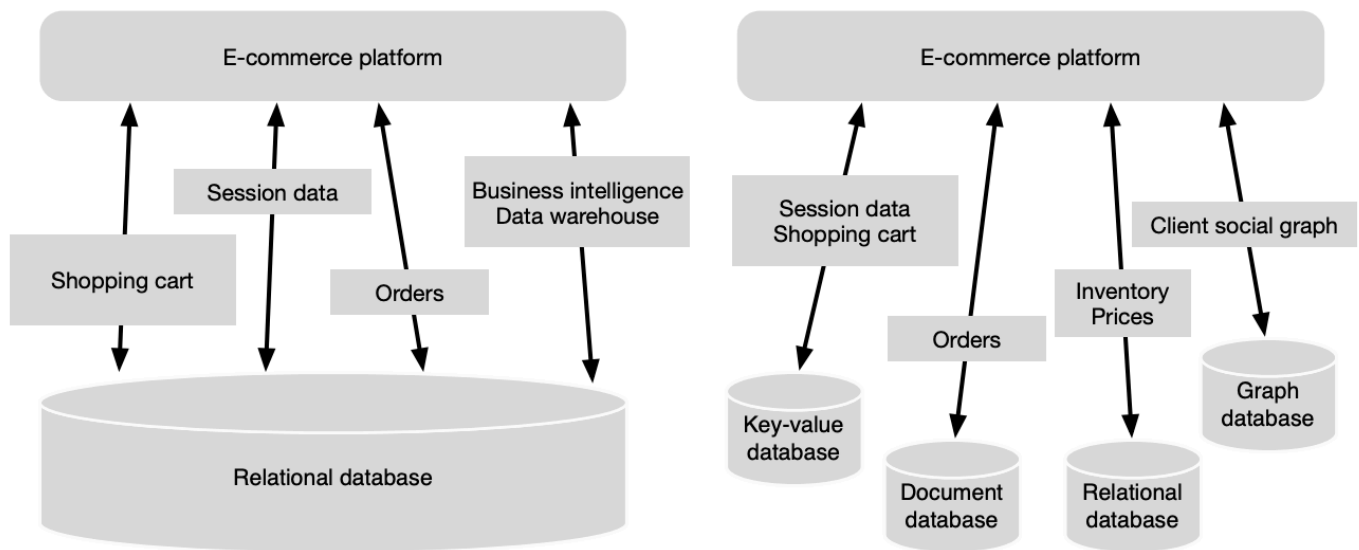
- **Structure d'un graphe avec des sommets et des arêtes :**
  - Peut être orienté ou non.
  - Bien adapté pour suivre les relations entre les objets.
  - Les bases de données relationnelles ne fonctionnent pas bien dans ce cas. Il faut faire des jointures qui peuvent devenir très complexes.
  - Le terme "relationnel" vient de la théorie des ensembles.
- **Langage de requête adapté à la structure du graphe.**



```
START barbara = node:nodeIndex(name = "Barbara")
MATCH (barbara)-[:FRIEND]->(friend_node)
RETURN friend_node.name, friend_node.location
```

### 3.6. Comment utiliser NoSQL

Prenons l'exemple d'une plateforme de E-Commerce, traditionnellement nous aurions opté pour une base de donnée relationnelle. L'avantage du NoSQL est qu'il peut s'adapter au différents types de données que nous avons à stocker.



#### 3.6.1. Utilisation recommandée

D'accord, voici la correction :

- **Modèle de données clé-valeur**
  - Stockage des données de session web
  - Profils et préférences utilisateur
  - Données du panier d'achat
- **Modèle de données document**
  - Journalisation des événements
  - Gestion de contenu d'entreprise, plateformes de blogs
  - Collecte de données pour l'analyse web
- **Modèle de données en famille de colonnes**
  - Gestion de contenu d'entreprise, plateformes de blogs
  - Compteurs
- **Modèle de données graphique**
  - Réseaux sociaux
  - Applications de livraison et de routage basées sur la géolocalisation
  - Moteurs de recommandation

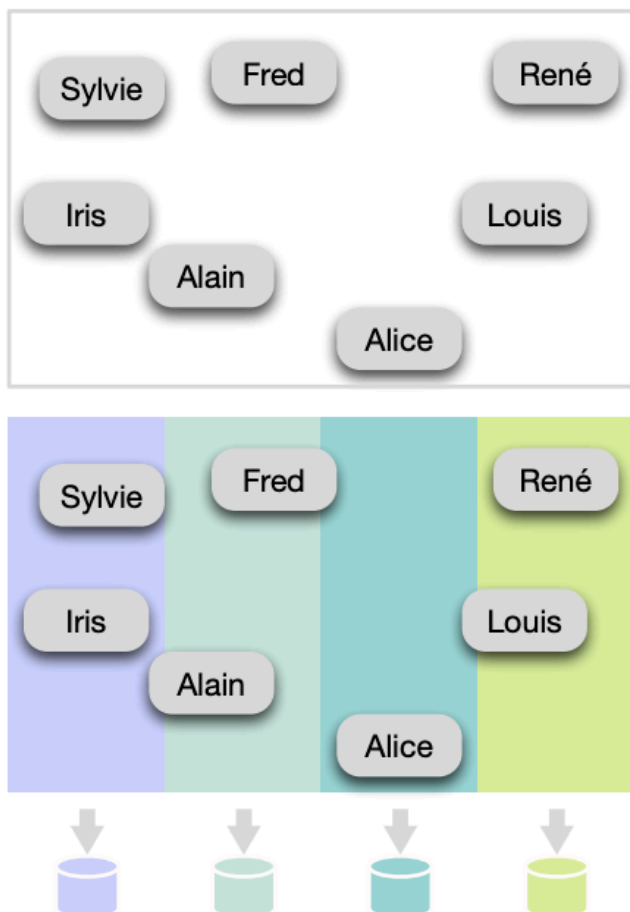
## 4. Base de données distribuée

Dimensions selon lesquelles une base de données peut avoir besoin de s'étendre :

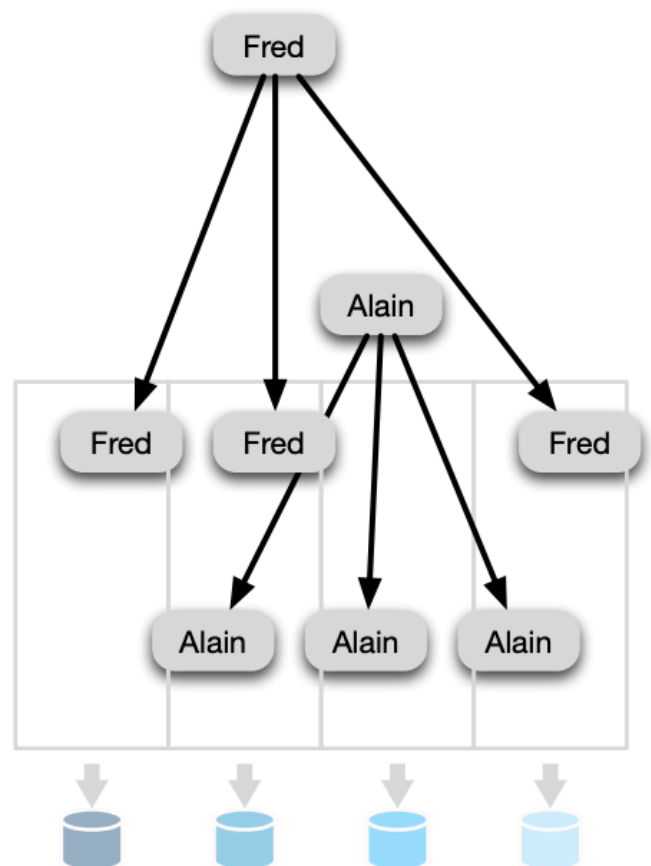
- **Capacité de stockage**
  - Nombre d'objets stockés
    - Moteur de recherche : métadonnées de 2 milliards de pages du World Wide Web
    - Réseaux sociaux : profils d'utilisateurs de 1 milliard d'utilisateurs
    - Suivi des comportements
    - Internet des objets
- **Capacité de débit des requêtes de lecture**
  - Nombre de requêtes de lecture par seconde
    - Commerce électronique
    - Jeux en ligne : jusqu'à 100 000 lectures par seconde
- **Capacité de débit des requêtes d'écriture**
  - Nombre de requêtes d'écriture par seconde
    - Jeux en ligne : jusqu'à 100 000 écritures par seconde
    - Internet des objets : jusqu'à 1 million d'écritures par seconde

### 4.1. Sharding vs Replication

- **Sharding** : Partitionnement des données sur plusieurs machines.



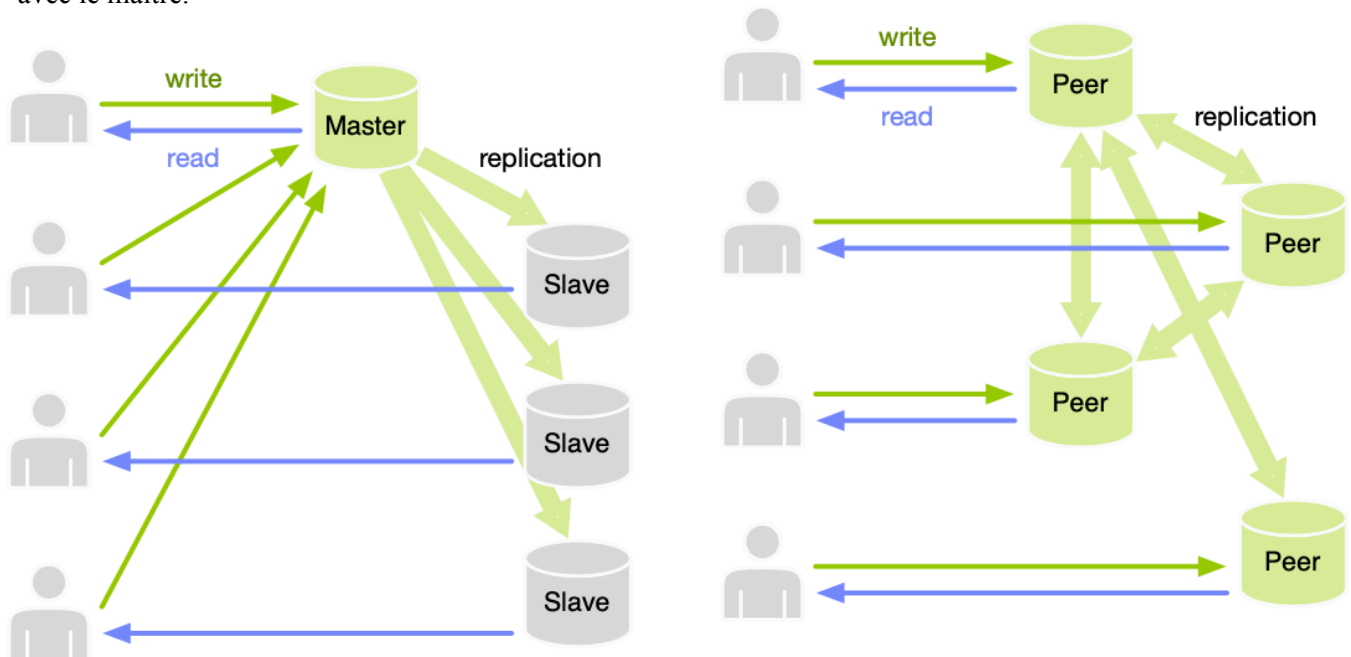
- **Replication** : Duplication des données sur plusieurs machines.



### 4.1.1. Replication

Lors ce que les données sont répliquées il existe deux modèles de réplication :

- **Master-slave** : Un serveur est le maître et les autres sont des esclaves. Les esclaves sont synchronisés avec le maître.
- **Peer-to-peer** : Chaque serveur est un pair et les données sont synchronisées entre les pairs.

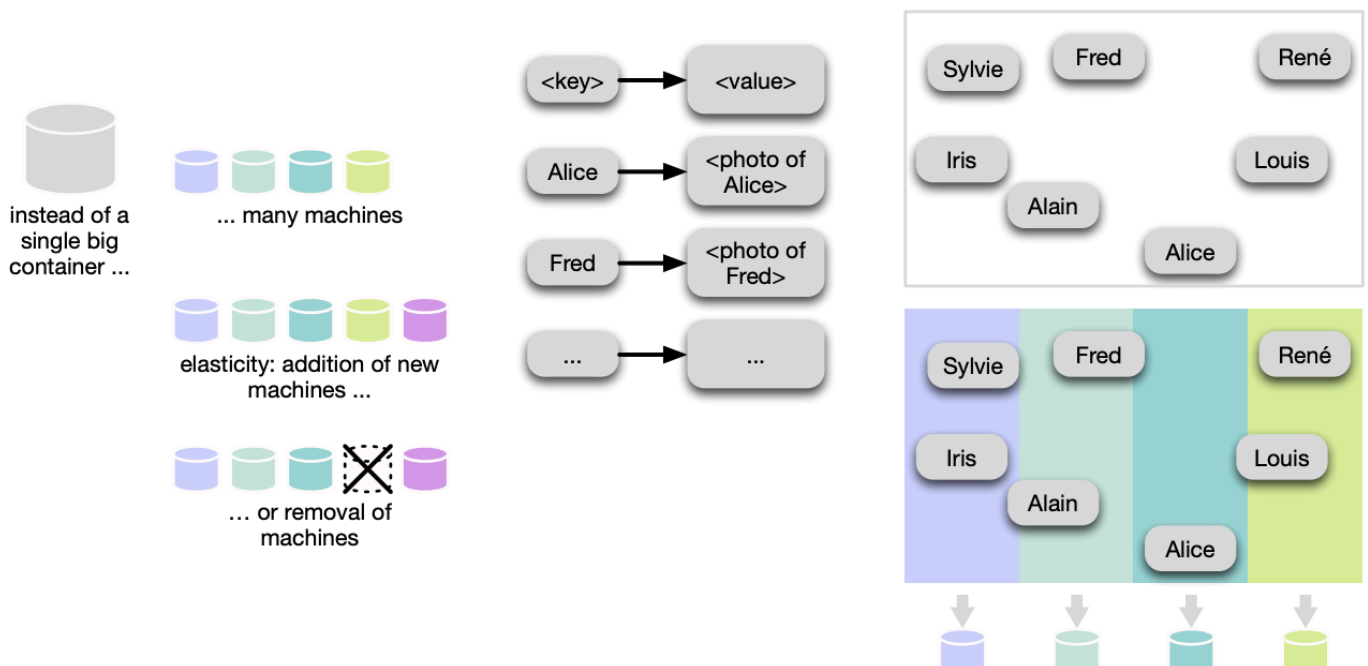


### 4.1.2. Sharding

Au lieu de traiter la base de données comme un conteneur monolithique, elle est divisée en fragments (shards).

Supposons que les données soient organisées sous forme de paires clé-valeur, par exemple des photos d'utilisateur (valeur) identifiées par leur nom (clé).

La question est de savoir comment subdiviser l'espace clé entre les machines pour répartir uniformément la charge.



Les **load balancers** sont utilisés pour rediriger les requêtes vers les bons serveurs. Ils doivent prendre une décision très rapidement, donc ils utilisent souvent des tables de hachage.

#### 4.1.2.1. Tables de hachage

- Une table de hachage distribue des objets dans une table à l'aide d'une fonction de hachage qui est calculée à partir de la clé.
- Une bonne fonction de hachage distribue les objets de manière plus ou moins uniforme pour minimiser les collisions.
- **Seul problème** : Lorsqu'une machine est ajoutée, les positions de presque tous les objets changent. Cela entraînerait un trafic réseau inacceptable pour migrer les objets !

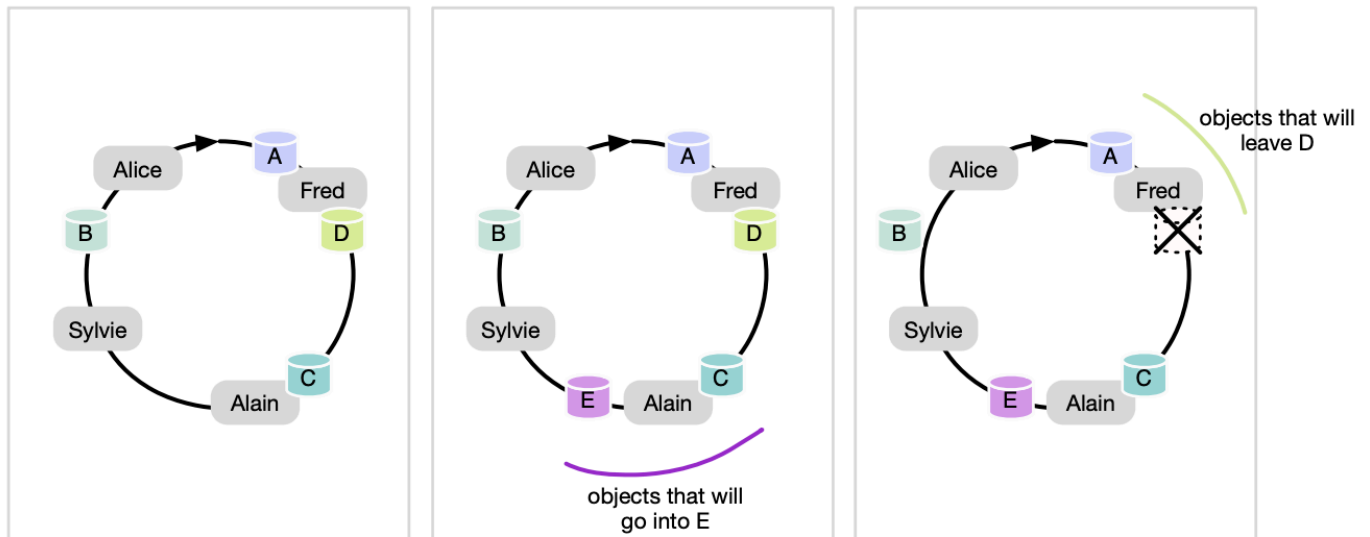
#### 4.1.2.2. Consistent hashing

- Le hachage cohérent évite de déplacer les objets lors de l'ajout ou de la suppression de machines.
- Les clés sont mappées par la même fonction de hachage, puis les valeurs de hachage sont mappées sur un cercle.
- Les machines sont également mappées dans le cercle en utilisant leur nom comme clé.
- Une convention est établie : chaque objet est attribué à la machine suivante dans le cercle dans le sens horaire.

Situation initiale.

Ajouter une machine n'affecte que les objets se trouvant entre la nouvelle machine et la précédente.

Retirer une machine n'affecte que les objets se trouvant entre la machine retirée.

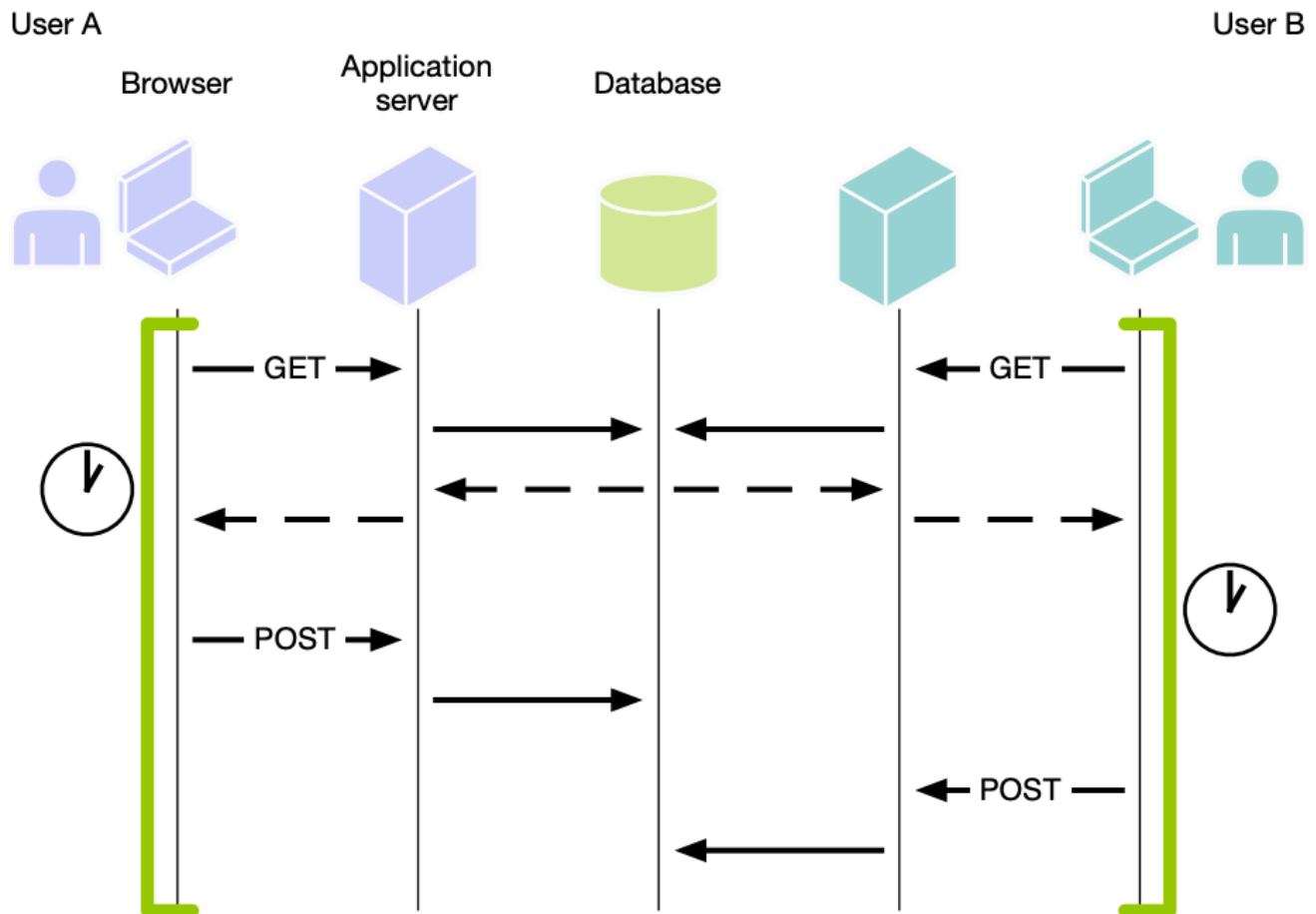


#### 4.1.2.3. Ecritures concurrentes

Dans un système distribué, les écritures concurrentes peuvent poser problème. Par exemple, si deux utilisateurs modifient le même objet en même temps, comment résoudre le conflit ?

#### 4.1.2.3.1. Transactions lecture-écriture

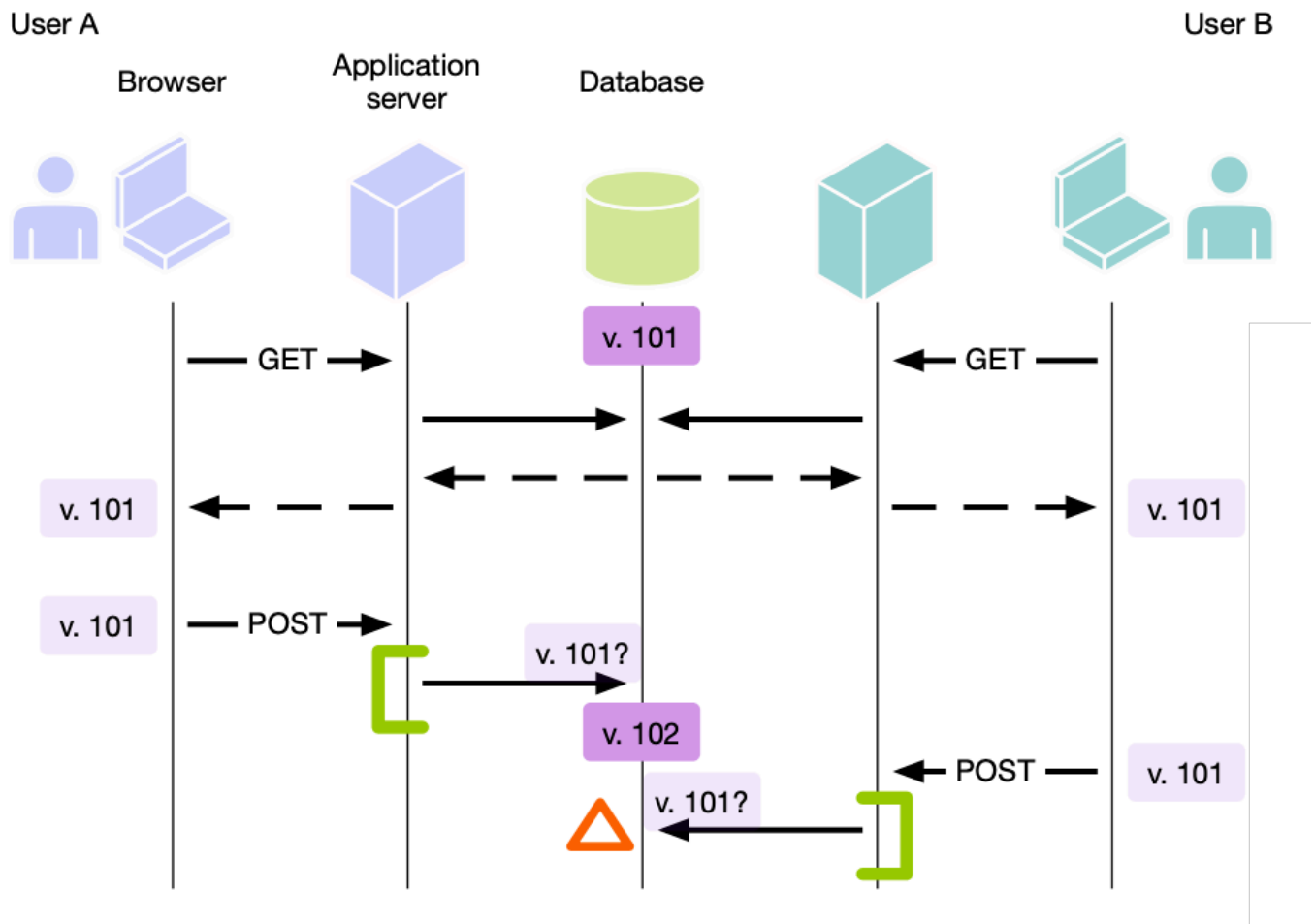
La première solution consiste à envelopper la lecture et l'écriture de chaque utilisateur dans une transaction. La base de données détectera le conflit, exécutera avec succès l'une des deux transactions et annulera l'autre. Cependant, un problème majeur est que le **maintien d'une transaction** sur une aussi longue période peut entraîner une **dégradation des performances**.



#### 4.1.2.3.2. Transactions écriture

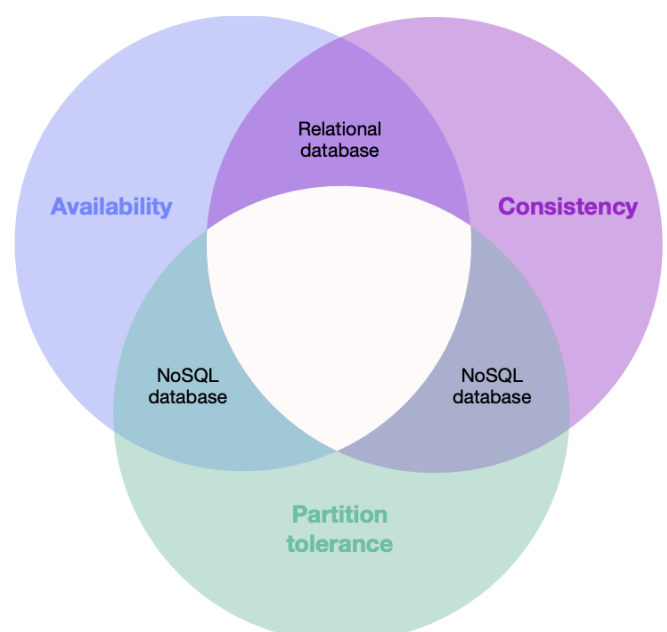
La deuxième solution consiste à envelopper uniquement l'écriture dans une transaction. Cela garantira que l'écriture est exécutée complètement ou pas du tout (pas d'écriture à moitié). Cependant, ni la **base de données** ni l'**application** ne seront conscients d'un **conflit entre les utilisateurs**.

De plus l'ajout d'une **version** sur le champ de données permet de vérifier si la donnée a été modifiée. Avant de modifier la donnée, on vérifie si la version est la même que celle que l'on a en mémoire. Nous pouvons donc détecter s'il y a eu un problème de concurrence.



#### 4.1.2.4. Consistance avec de la réplication

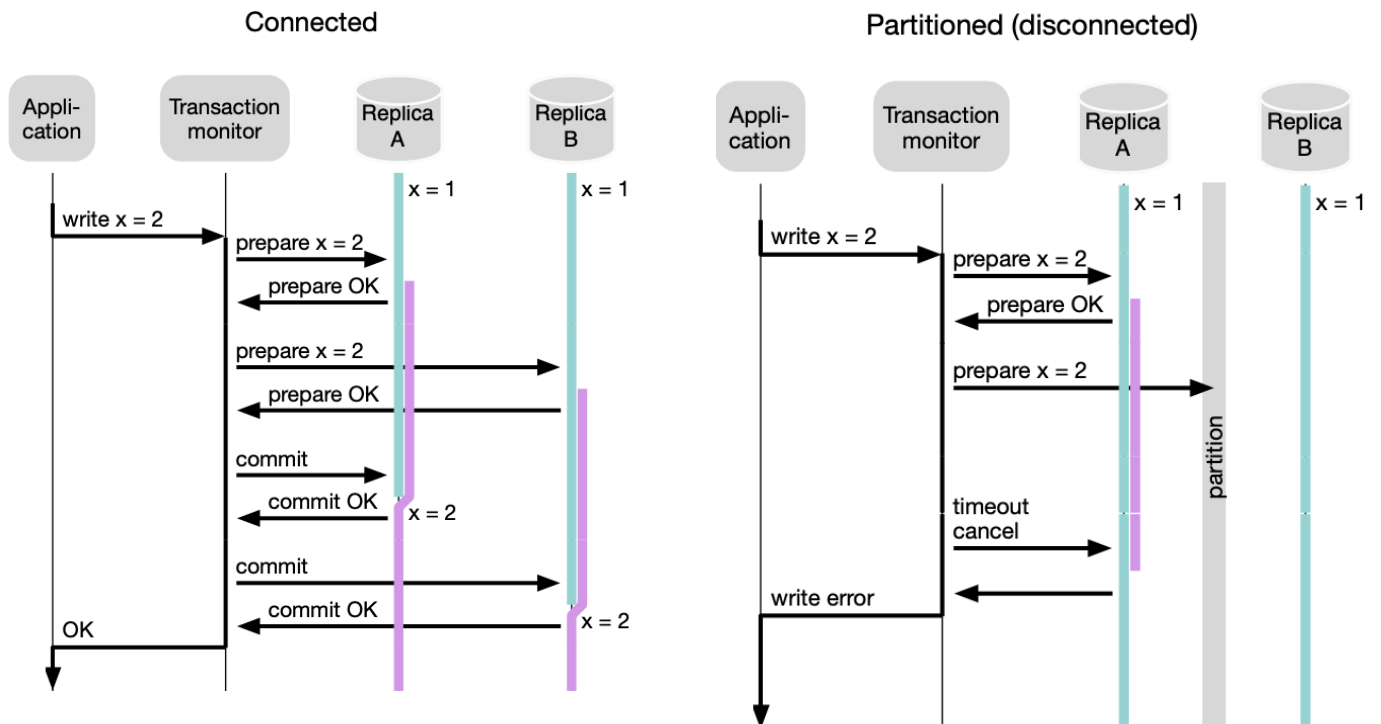
- **Le théorème CAP** aborde les compromis dans les bases de données distribuées. Il stipule que parmi les objectifs de Cohérence (Consistency - C), Disponibilité (Availability - A) et Tolérance aux partitions réseau (Partitions - P), seuls deux peuvent être atteints simultanément.
- Conjecturé par Eric Brewer en 2000, preuve formelle par Nancy Lynch et Seth Gilbert en 2002.



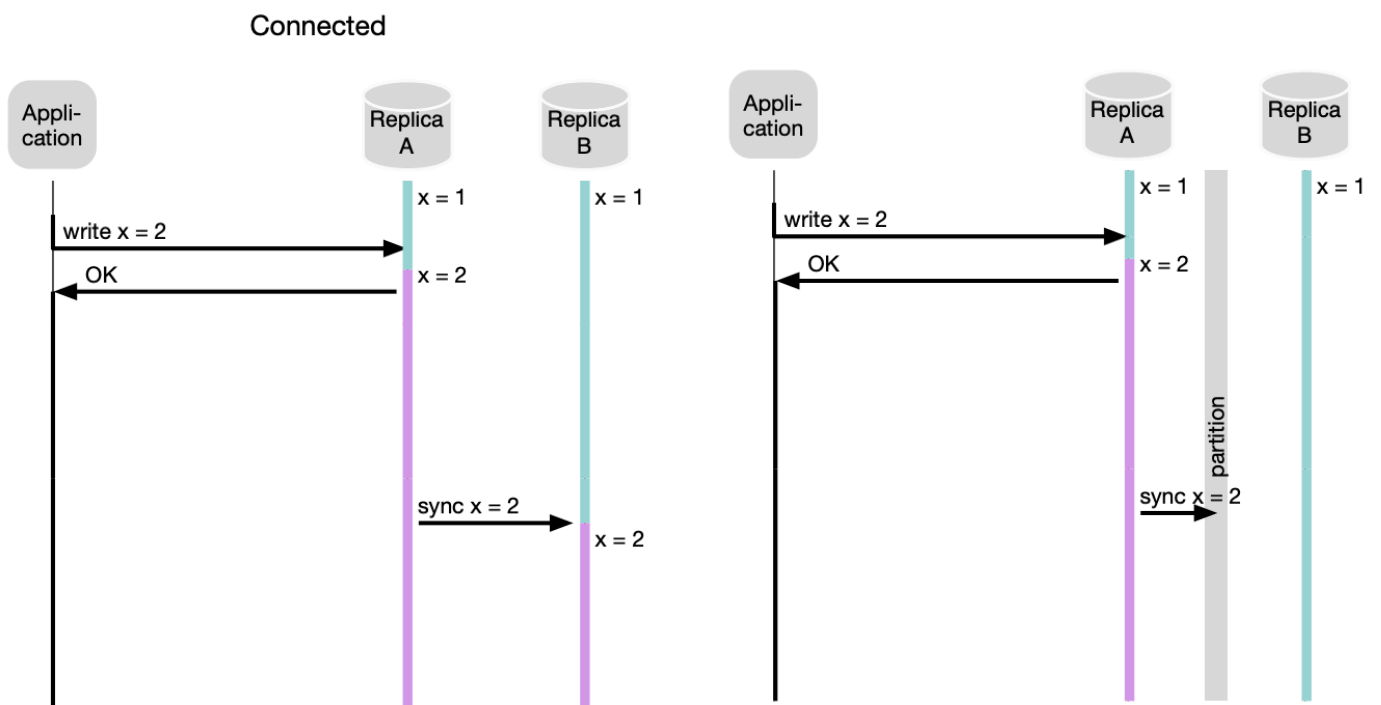
#### 4.1.2.4.1. Tolerance à la réplication

- **Problèmes de réseau** : peuvent mener à des partitions dites partitions réseau.
  - Avec deux ruptures dans les lignes de communication, le cluster de la base de données se partitionne en deux groupes.
- Un système est tolérant aux partitions s'il continue de fonctionner en présence d'une partition.

#### 4.1.2.5. Consistance forte avec commit à deux phases



#### 4.1.2.6. Disponibilité sans consistance forte

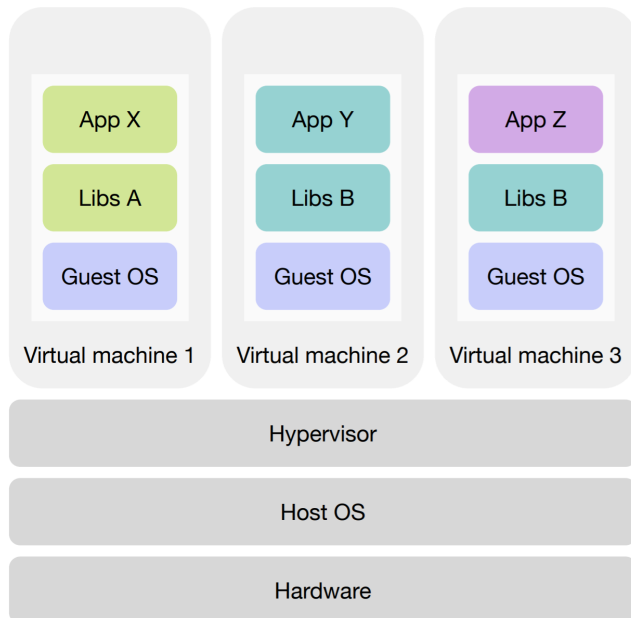




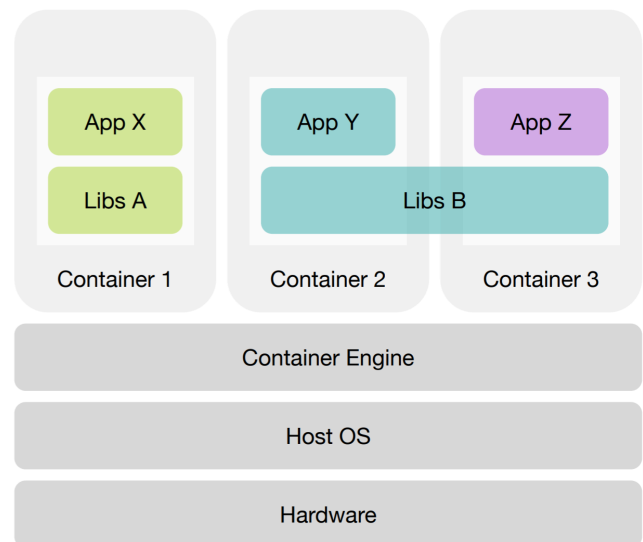
## 5. Conteneurs logiciels

### 5.1. Conteneurs vs Machines virtuelles

- Les conteneurs offrent une alternative légère aux machines virtuelles en partageant le noyau du système d'exploitation hôte tout en maintenant des espaces utilisateurs isolés.
- Les conteneurs sont plus efficaces en termes d'utilisation des ressources comparés aux machines virtuelles traditionnelles.



Three VMs running on a single host



Three containers running on a single host

### 5.2. Création et téléchargement d'images de conteneurs

- Le processus implique la création d'une image de conteneur, généralement avec Docker, et son téléchargement dans un registre de conteneurs.
- Les registres populaires incluent Docker Hub, GitHub Container Registry, Amazon Elastic Container Registry, Azure Container Registry et Google Artifact Registry.

## 6. Gestion des clusters de conteneurs

### 6.1. Introduction

- La gestion des clusters de conteneurs est essentielle pour déployer des applications sur plusieurs hôtes afin d'assurer la robustesse et la continuité du service.
- Les besoins clés incluent la surveillance de la santé des conteneurs, le placement optimal des conteneurs et la gestion efficace des pannes.

### 6.2. Orchestration des conteneurs

- L'orchestration détermine le placement des conteneurs d'application sur les nœuds du cluster en fonction des besoins en ressources et des contraintes comme l'affinité et l'anti-affinité.
- Les objectifs sont d'augmenter l'utilisation du cluster tout en répondant aux exigences des applications.

### 6.3. YAML (Yet Another Markup Language)

- L'opérateur peut créer des objets K8s avec la ligne de commande ou décrire les objets dans des fichiers manifestes.
  - `kubectl create -f file.yaml`
  - Le format de fichier est JSON, qui peut également être écrit en YAML

#### 6.3.1. Structure

- Seulement deux structures de données de base : tableaux et dictionnaires, qui peuvent être imbriqués
- YAML est un sur-ensemble de JSON
- Plus facile à lire et écrire pour les humains que JSON
- L'indentation est significative
- Spécification à <http://yaml.org/>

#### 6.3.2. Exemple de YAML

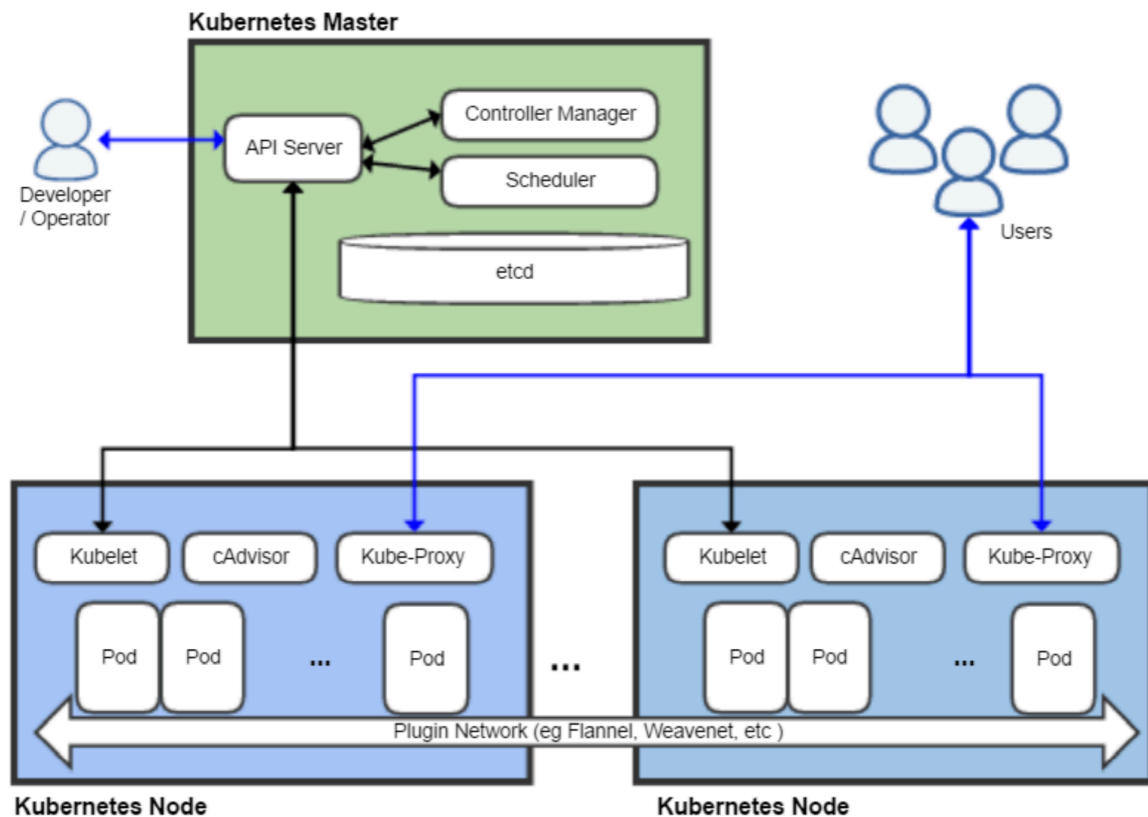
```
apiVersion: v1
kind: Pod
metadata:
  name: redis
  labels:
    component: redis
    app: todo
spec:
  containers:
  - name: redis
    image: redis
    ports:
  - containerPort: 6379
    resources:
      limits:
        cpu: 100m
  args:
  - redis-server
  - --requirepass ccp2
  - --appendonly yes
```

## **7. Kubernetes**

### **7.1. Introduction**

- Kubernetes est une plateforme open-source pour automatiser le déploiement, la mise à l'échelle et la gestion des applications conteneurisées.
- Développé à l'origine par Google, il est maintenant maintenu par la Cloud Native Computing Foundation (CNCF).

## 7.2. Anatomie d'un cluster



### 7.2.1. Composants du nœud maître

- **etcd** : Un magasin clé/valeur pour les données de configuration du cluster.
- **API Server** : Sert l'API de Kubernetes.
- **Scheduler** : Décide des nœuds sur lesquels les pods doivent fonctionner.
- **Controller Manager** : Exécute les contrôleurs principaux comme le Replication Controller.

### 7.2.2. Composants du nœud de travail

- **Kubelet** : Gère l'état des conteneurs sur un nœud.
- **Kube-proxy** : Gère le routage réseau et l'équilibrage de charge.
- **cAdvisor** : Surveille l'utilisation des ressources et la performance.
- **Réseau superposé** : Connecte les conteneurs entre les nœuds.

## 7.3. Concepts principaux

- **Cluster** : Un ensemble de machines (nœuds) où les pods sont déployés et gérés.
- **Pod** : La plus petite unité déployable, composée d'un ou plusieurs conteneurs.
- **Controller** : Gère l'état du cluster.
- **Service** : Définit un ensemble de pods et facilite la découverte de services et l'équilibrage de charge.
- **Label** : Paires clé-valeur attachées aux objets pour la gestion et la sélection.

## 7.4. Concepts communs

- Les objets Kubernetes peuvent être créés et gérés en utilisant des fichiers YAML ou JSON.
- YAML est un format lisible par l'homme utilisé pour décrire les objets Kubernetes dans les fichiers de configuration.

## 7.5. Déployer une application : IaaS vs Kubernetes

- L'IaaS traditionnel implique des étapes manuelles comme le lancement de VMs, leur configuration et la mise en place d'équilibreurs de charge.
- Kubernetes simplifie ce processus avec des images de conteneur et des manifestes, permettant un déploiement et une mise à l'échelle automatisés.

## 7.6. Exemple de YAML Kubernetes

- Chaque description d'objet Kubernetes commence par deux champs :
  - kind** : une chaîne qui identifie le schéma que cet objet doit avoir
  - apiVersion** : une chaîne qui identifie la version du schéma que l'objet doit avoir
- Chaque objet a deux structures de base : Métadonnées de l'objet et Spécification (ou Spec).
- La structure des Métadonnées de l'objet est la même pour tous les objets dans le système
  - name** : identifie de manière unique cet objet dans l'espace de noms actuel
  - labels** : une carte de clés et de valeurs de chaîne qui peut être utilisée pour organiser et catégoriser les objets
- Spec** est utilisé pour décrire l'état désiré de l'objet

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
  labels:
    component: redis
    app: todo
spec:
  containers:
  - name: redis
    image: redis
    ports:
    - containerPort: 6379
    resources:
      limits:
        cpu: 100m
    args:
    - redis-server
    - --requirepass ccp2
    - --appendonly yes
```

## 7.7. Pods

**Le pod est l'unité atomique de déploiement dans Kubernetes.**

- L'unité atomique de déploiement dans Kubernetes est le Pod.
- Un Pod contient un ou plusieurs conteneurs, le cas le plus courant étant un seul conteneur.

**Si un Pod a plusieurs conteneurs :**

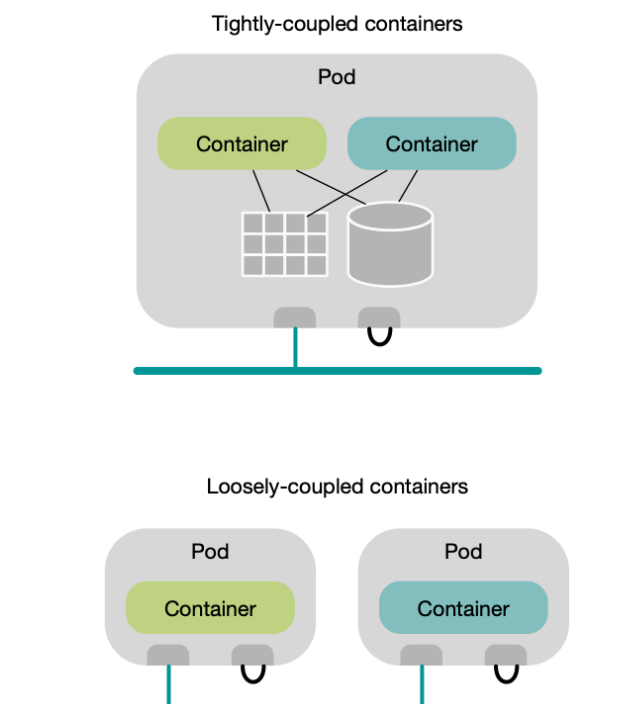
- Kubernetes garantit qu'ils sont programmés sur le même nœud du cluster.
- Les conteneurs partagent le même environnement du Pod :
  - Espace de noms IPC, mémoire partagée, volumes de stockage, pile réseau, etc.
  - Adresse IP.
- Si les conteneurs doivent communiquer entre eux au sein du Pod, ils peuvent simplement utiliser l'interface **localhost**.

### 7.7.1. Loosely vs Tightly Coupled Containers

#### Loosely Coupled Containers

**Définition :**

- Les conteneurs faiblement couplés fonctionnent de manière indépendante, avec peu ou pas de dépendances directes entre eux. Ils sont conçus pour effectuer des tâches autonomes et ne nécessitent pas une interaction continue avec d'autres conteneurs pour fonctionner correctement.



**Caractéristiques :**

- **Indépendance** : Chaque conteneur peut être géré, mis à l'échelle et redéployé indépendamment des autres.
- **Isolation** : Moins de partage de ressources, comme la mémoire et le stockage, ce qui minimise l'impact des erreurs d'un conteneur sur les autres.
- **Flexibilité** : Adapté aux microservices où chaque service peut être développé, déployé et mis à jour indépendamment.
- **Communication** : Les conteneurs communiquent souvent via des réseaux internes ou des API externes.

**Exemples d'utilisation :**

- Services de microservices où chaque service a sa propre base de code, cycle de déploiement et peut évoluer indépendamment des autres services.
- Applications où des composants indépendants traitent des tâches distinctes, comme un service de facturation séparé d'un service de gestion des utilisateurs.

**Tightly Coupled Containers****Définition :**

- Les conteneurs étroitement couplés travaillent ensemble de manière plus intégrée et partagent souvent des ressources communes. Ils sont conçus pour collaborer étroitement, où l'un peut dépendre directement de l'autre pour fournir une fonctionnalité complète.

**Caractéristiques :**

- **Interdépendance** : Les conteneurs sont fortement dépendants les uns des autres pour accomplir leurs tâches.
- **Partage de ressources** : Les conteneurs partagent des ressources telles que l'espace de noms IPC, la mémoire, les volumes de stockage et la pile réseau.
- **Coordination** : Souvent utilisés dans des scénarios où une application principale est assistée par des conteneurs secondaires qui fournissent des services supplémentaires comme le traitement des journaux ou la mise à jour du contenu.
- **Communication locale** : Les conteneurs peuvent utiliser l'interface localhost pour une communication rapide et directe.

**Exemples d'utilisation :**

- Un serveur web conteneurisé qui dépend d'un conteneur d'assistance pour mettre à jour dynamiquement le contenu ou gérer les journaux.
- Une application où un conteneur exécute l'application principale tandis qu'un autre conteneur gère des tâches auxiliaires telles que la surveillance ou la sauvegarde des données en temps réel.

| Caractéristique            | Faiblement couplés (Loosely Coupled) | Étroitement couplés (Tightly Coupled) |
|----------------------------|--------------------------------------|---------------------------------------|
| Indépendance               | Haute                                | Faible                                |
| Partage de ressources      | Minimum                              | Élevé                                 |
| Flexibilité de déploiement | Haute                                | Moyenne                               |
| Communication              | Réseau ou API externe                | Interface localhost                   |
| Utilisation typique        | Microservices                        | Applications complexes et intégrées   |

**7.8. Volumes**

Un pod peut avoir besoin de stockage persistant ou pas pour stocker des données. Kubernetes fournit des volumes pour gérer le stockage.

**Stockage supportés**

- **emptyDir**: an empty directory the app can put files into, erased at Pod deletion
- **hostPath**: path from host machine, persisted with machine lifetime
- **gcePersistentDisk**, **awsElasticBlockStore**, **azureFileVolume**: cloud vendor volumes, independent lifecycle

- **secret**: used to pass sensitive information, such as passwords, to Pods. You can store secrets in the Kubernetes API and mount them as files for use by Pods without coupling to Kubernetes directly. secret volumes are backed by tmpfs (a RAM-backed filesystem) so they are never written to nonvolatile storage
- **nfs**: network file system, persisted
- Many more: iscsi, flocker, glusterfs, rbd, gitRepo ...

### 7.8.1. Monter un volume persistant

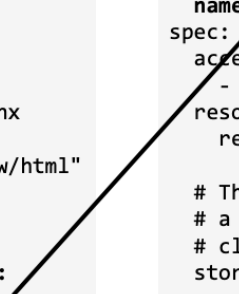
K8s propose 2 manières de monter un volume persistant :

- **Directement**
- Avec **PersistentVolumeClaim (PVC)**

```
apiVersion: v1
kind: Pod
metadata:
  name: test-efs
spec:
  containers:
    - image: icclab/test-webserver
      name: test-container
      volumeMounts:
        - mountPath: /test-efs
          name: test-volume
  volumes:
    - name: test-volume
      # This AWS EBS volume
      # must already exist.
      awsElasticBlockStore:
        volumeID: <volume-id>
        fsType: ext4
```

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: dockerfile/nginx
      volumeMounts:
        - mountPath: "/var/www/html"
          name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: myclaim
```

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
  # The cluster must have
  # a configured storage
  # class
  storageClassName: default
```



## 7.9. Replication Controller

Permet de déployer automatiquement un nombre spécifique de réplicas identiques d'un pod. Si un pod échoue, le Replication Controller en crée un nouveau pour le remplacer.

Cependant les **Replication Controllers** sont gentiment remplacés par les déploiements.

## 7.10. Services

Un service Kubernetes est une abstraction qui définit un ensemble de pods et une politique d'accès à ces pods. Les services permettent de définir un ensemble de pods et de les exposer au sein du cluster.

- L'interface utilisateur communique avec l'adresse IP fiable du Service.
- Le Service dispose également d'un nom de domaine identique à son nom.
- Le Service répartit la charge de toutes les requêtes sur les Pods de l'arrière-plan qui le composent.
- Le Service garde une trace des Pods qui se trouvent derrière lui.

### 7.10.1. Type de services

- **ClusterIP** : Expose le Service sur une adresse IP interne du cluster. C'est le type de Service par défaut.
- **NodePort** : Expose le Service sur un port statique sur chaque nœud du cluster.
- **LoadBalancer** : Expose le Service via un équilibreur de charge externe.