Architecture distribuée de gestion de données

Nathan Eudeline
nathan.eudeline@student.junia.com

Cyprien Kelma
Chef de groupe
cyprien.kelma@student.junia.com

Paul Pousset
paul.pousset@student.junia.com

Nolan Cacheux
nolan.cacheux@student.junia.com

Mamoun Kabbaj
mamoun.kabbaj@student.junia.com

***Abstract*—Cette recherche s'effectue dans le cadre du projet de 1er semestre de notre année de M1 lors de notre master en Big-Data à Junia ISEN, Lille (2024). Elle porte sur la conception d'une architecture de données unique, scalable et résiliente, utilisée pour un service sous forme d'application confrontée à une hausse croissante de débit et de volumétrie et répond à la problématique suivante : « Comment développer l'architecture de données d'un système de stockage pour assurer sa scalabilité et sa résilience face à une augmentation croissante de la charge et de la volumétrie des données ? » Elle aborde les défis de l'extensibilité au-delà des solutions verticales, en passant par la disponibilité et la synchronisation des données, ainsi que la prise en charge de l'analyse et des prédictions avancées à partir des données stockées. La solution proposée dans ce document porte sur l'adoption de l'extensibilité horizontale par le sharding et la réplication, mais également sur des stratégies de stockage polyglottes en adoptant des modèles non relationnels, ainsi que des pipelines de traitement de données optimisés.**

***Index Terms*—Big Data, Scalability, Reliability, ACID, BASE, CAP theorem, SQL, NoSQL, Sharding, Replication, Cache, Data Lake, Data Warehouse**

I. Introduction

A. Summer-Trip

« Summer-Trip » est une application web conçu dans le cadre d'un projet étudiant de 2ème de cycle préparatoire informatique et réseau à l'ISEN Lille. Elle permet à des groupes (amis, famille, etc.) de simplifier l'organisation de leurs voyages grâce à plusieurs fonctionnalités attractives, telles que la création de groupes avec accès à un canal de messagerie, un planning commun généré automatiquement et modifiable à souhait, ou encore le choix d'activités interactif, reflétant les préférences de l'ensemble des membres du groupe.

a) *Les évolutions du service bridé par son architecture actuelle:*

Le service repose actuellement sur une base de données MySQL classique contenant l'ensemble des tables utiles à son fonctionnement (Fig. 1). Cependant, cette structure de données initiale ne répond qu'aux besoins essentiels à la réalisation du projet, c'est-à-dire être fonctionnelle pour relativement peu d'utilisateurs et ne proposer que des fonctionnalités de base.

Bien que Summer-Trip soit un projet informatique intéressant, le transformer en un produit commercial réellement fonctionnel soulève de nombreux défis concernant son système de stockage, auxquels nous n'avions pas été confrontés initialement.

En premier lieu, nous ne prenons actuellement pas en compte le risque d'un usage trop intensif de la base de données par un grand nombre d'utilisateurs potentiels. Nous n'assurons pas non plus la résilience de notre infrastructure en cas de panne. Si la base de données crashait, nous ne disposerions pas de réplique de secours. Enfin, nous n'avons pas encore considéré l'implémentation de certaines fonctionnalités avancées mais indispensables pour une application de réseau social moderne telle que Summer-Trip. On pourrait citer notamment l'analyse des tendances et des statistiques de l'application, qui demanderait une structure plus avancée qu'une base de données relationnelle classique, comme un lac ou un entrepôt de données, par exemple.

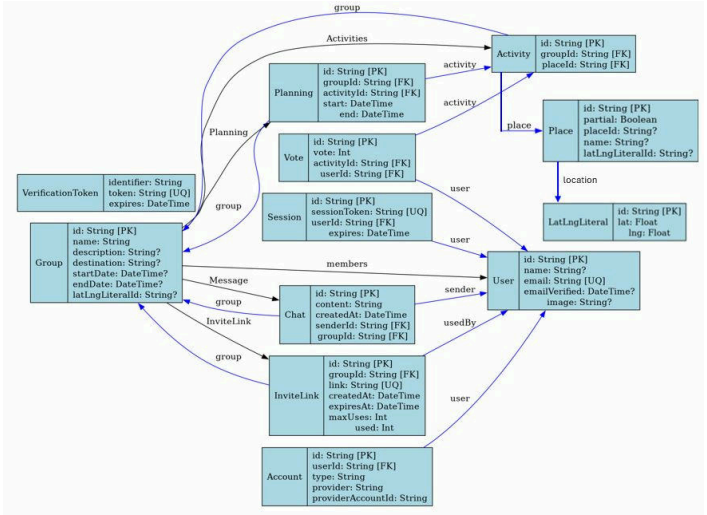

Fig. 1: Schéma UML de la base de données Summer-Trip

À terme, nous pourrions également vouloir créer des modèles de suggestion personnalisés pour chaque utilisateur ou groupe à l'aide de Machine Learning, ce qui demande bien plus qu'une simple base de données unique. Mais aussi des *Test A/B* pour améliorer à terme la qualité du service afin d'assurer une meilleure expérience sur l'application. De tels besoins imposent de mettre en place de nouveaux moyens de stockage du comportement des utilisateurs.

B. Plan

Nous allons concentrer cette recherche sur l'exploration de nombreux concepts liés à l'architecture scalable et résiliente,

ainsi que les nombreuses options de stockage qui s'offrent à nous afin d'améliorer la structure de données de Summer-Trip.

Pour se faire, dans un premier lieu, nous allons exposer les limites du scaling verticale et évoquerons des techniques de scaling horizontale, le sharding et la réplication de données. Nous allons ensuite évoquer les bienfaits d'utiliser une architecture de stockage polyglotte, en évoquant les faiblesses d'une architecture uniquement en SQL et donc l'utilisation de NoSQL pour certaines données et l'ajout d'un système de caching. Dans un troisième temps, nous allons évoquer les nouveaux besoins en stockage de données qui feront suite au développement de Summer-Trip et l'ajout d'un modèle de recommandation prédictif, donc une analyse massive de données d'utilisateurs. Pour finir, nous conclurons en résumant à quoi pourrait ressembler l'architecture finale de l'application.

II. Scaling et réplication des données

Avec une quantité accrue d'utilisateurs sur le service et un besoin de stocker et accéder aux données de manière rapide et fiable, cela nous amène à nous poser des questions sur les capacités de notre système à s'agrandir et être disponible en tout temps.

A. Les avantage et limite du scaling vertical

Le Scaling vertical consiste à augmenter les ressources (CPU, RAM, stockage) d'une machine existante pour y augmenter les performances et capacités.

Elle a l'avantage de garder toutes les données sur un seul nœud, ce qui facilite leur gestion. L'objectif est donc d'ajouter des ressources à une machine unique pour gérer une charge de travail plus importante. [1]

a) *Les limites du scaling vertical:* La principale limite est ce que l'on appelle la loi des rendements décroissants. Ce phénomène est observé lorsque l'on atteint un certain niveau d'augmentation des ressources. Cette augmentation ne suit pas linéairement l'augmentation des performances. Cet ajout peut doubler le prix sans pour autant doubler la capacité de traitement [2]. On remarque aussi que certains systèmes d'exploitation ne sont pas forcément toujours optimisés pour tirer de façon avantageuse de telles configurations matérielles, ce qui, à terme, limite l'efficacité du scaling vertical. Plus grave encore, une panne matérielle peut affecter toute une application étant donné que toutes les données sont présentes sur un seul nœud.

b) *Cas d'usage du scaling vertical:* L'usage de cette pratique peut s'avérer être une bonne option dans un contexte où une application doit supporter une charge de travail croissante sans pour autant apporter des modifications à son architecture. Il faut aussi prendre en compte qu'elle est idéale pour des bases de données monolithiques qui ne supportent pas facilement le partitionnement ou la distribution.

Le scaling vertical est bien la solution la plus simple dans de nombreux contextes, mais il comprend de majeures limitations qui ne peuvent être résolues qu'en utilisant d'autres méthodes.

Dans notre cas de Summer-Trip, nous avons une application orientée utilisateurs, ce qui entraîne une quantité de données qui s'agrandit avec celle-ci. Nous devons donc avoir la capacité de prendre en charge un flux qui va évoluer au-delà des limites d'un scaling vertical. Cela nous amène donc à explorer les deux principales solutions : le scaling horizontal et le sharding.

B. La répartitions des données à l'aide du sharding

Sharding Il s'agit du principe de partitionnement des données dans plusieurs nœuds à l'aide de l'utilisation d'une base de données distribuée.

Les différents nœuds peuvent être répartis sur plusieurs serveurs, ce qui permet un scaling horizontal de notre infrastructure. Cela peut avoir de nombreux avantages tels qu'une augmentation des performances du système ou même une meilleure disponibilité du service. [1]

Le sharding répartit les données dans les nœuds via trois stratégies principales : par plages, où les données sont divisées selon des intervalles (dates, valeurs numériques), mais risquent une distribution inégale ; par hachage, assurant une répartition uniforme via une fonction de hachage, bien que complexe pour les requêtes sur plages ; et géographique, adaptée aux accès spécifiques par région [1]. L'utilisation d'une stratégie va fortement dépendre de l'implémentation utilisée (Cassandra utilise le hachage par exemple), mais aussi du contexte dans lequel on va utiliser notre base de données (du sharding géographique peut être facilement mis en place dans un contexte de service international).

a) *Les avantages du sharding des bases de données:*

- Scalabilité horizontale : Le sharding permet de répartir la charge de travail sur plusieurs serveurs. Cette action a pour but d'améliorer la puissance sans pour autant devoir augmenter les performances d'un seul serveur (scaling vertical). Il devient donc facile de rajouter des partitions en cas de hausse du volume de données.
- Amélioration des performances : Dans le cas de bases de données très sollicitées, avec un grand nombre de requêtes, le fait de pouvoir répartir les données en plusieurs partitions permet un meilleur temps de réponse. Les plateformes de streaming qui diffusent des flux vidéos peuvent bénéficier de cet aspect du sharding.
- Meilleure disponibilité : Les bases de données sont parfois soumises à des pannes. Le sharding, bien qu'il permette d'augmenter les performances, peut s'avérer très utile pour pallier des pannes de serveurs. Dans le cas d'une panne qui concerne un *shard* ou nœud, les autres ne sont pas touchés et peuvent continuer d'assurer les requêtes.
- Réduction de latence géographique : Le sharding géographique englobe le processus de rapprochement géographique des utilisateurs avec des nœuds plus proches. Ce processus permet de réduire la latence et de se conformer aux différentes législations.

[2]

b) *Défis du sharding:* Le principal inconvénient du sharding est que son implémentation peut vite devenir un véritable défi technique [2]. En effet, le sharding tel qu'on le connaît est une méthode assez complexe à paramétrer, principalement pour définir la méthode de répartition des données et garantir un équilibre des charges. [1] La nature distribuée du sharding

peut augmenter la complexité des transactions lorsqu'elles impliquent plusieurs nœuds. Il faut aussi noter que la maintenance d'une base de données shardée est plus complexe et donc coûteuse qu'une base de données classique.

c) *La mise en place du sharding:* On peut retrouver la possibilité d'utiliser cette méthode sur un nombre limité de bases de données. Par exemple, Cassandra ou PostgreSQL implémentent nativement cette technologie, ce qui nous amène à réfléchir dans quel contexte nous voulons l'utiliser [3]. Comme on le verra plus loin dans Chapitre III.B, le choix de base de données que nous allons faire ne nous permet pas forcément de l'implémenter dans tout notre système, mais seulement dans les points critiques qui auront besoin d'une grande capacité de traitement et de stockage, comme la gestion des messages et des activités de Summer-Trip.

Cependant, suite à une augmentation du nombre de serveurs ou de bases de données dans notre système, due à la mise en place d'un sharding, cela entraîne une hausse de problèmes potentiels tels que la corruption, des crashs ou des soucis au niveau du matériel [2]. Ce qui entraîne un risque accru de perte de données critiques. Il devient alors impératif d'utiliser des solutions de réplication. [4] Chapitre II.C

C. Réduire le risque de pannes et de pertes de données grâce aux réplications

Réplication de base de données c'est le processus de copier et de maintenir des copies synchronisées d'une base de données sur plusieurs serveurs ou emplacements.

Elle permet d'assurer la disponibilité, la redondance et une meilleure tolérance aux pannes des données dans un système. Ces mécanismes de réplication sont souvent utilisés sur des systèmes sensibles ou très volumineux qui pourraient être fortement affectés par des pertes de données. [5]

Contrairement à une sauvegarde, qui est une représentation figée à un instant donné, la réplication se fait elle en continu. La problématique liée à la réplication est d'avoir, avec le moins de latence possible, des copies des différentes bases de données en temps réel. La forme la plus simple de réplication utilise une base de données de référence dite "maître" et une base de données esclave. Le maître tient compte des opérations, et la base de données esclave, quant à elle, récupère ces opérations pour les répercuter de son côté, comme le montre le Fig. 2. Cependant, il est important de faire la différence entre la réplication de données et la réplication de base de données. La réplication de bases de données englobe le processus sur l'ensemble d'une base de données, ce processus est souvent utilisé pour protéger et améliorer la disponibilité et la fiabilité de l'entièreté d'une base de données. La réplication de données, quant à elle, a pour but de répliquer des fragments de données vers des data warehouses. Un cas d'usage serait de répliquer les données relatives aux transactions de clients vers un data warehouse pour pouvoir, par la suite, les analyser.

Fig. 2: Processus de réplication de la data

a) *Les différents types de réplications de base de données:* La réplication de bases de données peut être réalisée localement ou sur le cloud, de manière synchrone ou asynchrone. Chacune de ces méthodes a des avantages et des inconvénients, il est donc crucial d'adapter la méthode au cas d'usage et aux différentes contraintes (financières, disponibilité). La réplication synchronisée sous-entend que les données seront toujours répliquées depuis le serveur principal vers des serveurs de réplication en même temps. Quant à la réplication asynchrone, les données sont copiées depuis le serveur principal vers des serveurs de réplication par lots de données. Cette dernière méthode est moins coûteuse et nécessite moins de moyens. Néanmoins, la réplication synchrone permet de perdre le moins de données possible en cas de panne. Il existe trois grands types de réplication de bases de données, chaque type étant adapté à un certain contexte.

Réplication transactionnelle: Cette méthode consiste en la capture des transactions effectuées sur la base de données principale, elle réplique les différentes actions effectuées vers des bases de données secondaires au fur et à mesure. La synchronisation des données est quasi en temps réel, car elle effectue les transactions dès que les précédentes sont validées. Elle s'avère très efficace pour des systèmes à forte écriture et pour des systèmes qui doivent être mis à jour fréquemment. Cependant, sa configuration est complexe et elle n'est pas la méthode la plus performante. [6]

Réplication par fusion: La réplication par fusion est une méthode qui permet à deux bases de données, principale et secondaire, de pouvoir être modifiées indépendamment. Les modifications sont fusionnées de manière périodique et les conflits sont gérés par des règles définies en amont du processus. Elle s'avère particulièrement performante dans des contextes de bases distribuées où différents utilisateurs peuvent modifier les données. La gestion des conflits peut être assez compliquée lorsque beaucoup de modifications sont effectuées et elle reste moins rapide que la réplication transactionnelle à cause du processus de fusion, qui peut engendrer des ralentissements. [7] souligne que, malgré la complexité associée à la réplication par fusion, elle reste très efficace dans des environnements où plusieurs utilisateurs peuvent modifier les données.

Réplication d'instantanées: La réplication d'instantanées, ou encore réplication par snapshot, consiste en une copie complète à des intervalles réguliers que l'on appelle snapshot. Elle est particulièrement facile à mettre en place et convient aux bases de données qui ne sont pas souvent mises à jour. Elle est particulièrement utile dans des contextes de reporting ou d'analyse de données. Elle ne convient pas pour des bases de données avec une volumétrie importante, car des copies

complètes peuvent engendrer des coûts supplémentaires en raison de leur taille. [6]

b) *Notre cas d'utilisation de la réplication:* La réplication de base de données peut s'avérer être une technique polyvalente et utile dans divers secteurs. Elle peut garantir une haute disponibilité et une forte tolérance aux pannes, et peut de ce fait éviter des pertes de données. Voici certains cas d'usage de la réplication.

Dans le cadre de la mise en œuvre de Summer-Trip, beaucoup de contraintes sont à prendre en compte. Notre application implique une forte croissance du volume de données à mesure que le nombre d'utilisateurs augmente. Notre infrastructure doit être capable de gérer un fort volume de données. La réplication s'avère être une solution tout à fait réaliste.

Pour mettre en place la réplication au sein de notre projet, nous allons utiliser PostgreSQL, qui convient parfaitement à des applications nécessitant une forte disponibilité. PostgreSQL offre une réplication synchrone, permettant de garantir que les données critiques, comme les messages et les activités planifiées, soient immédiatement répliquées sur des serveurs secondaires. De plus, l'architecture maître-esclave permet de minimiser les interruptions de service. Dans le cas d'une panne sur le serveur principal, les données restent accessibles via les serveurs secondaires. La configuration inclut un serveur maître pour les écritures et plusieurs serveurs esclaves pour les requêtes en lecture, réduisant ainsi la charge sur le serveur principal tout en améliorant les performances globales. [8]

III. L'utilisation d'une architecture de stockage polyglotte

A. SQL : Une structure trop figé face à des données aussi variés

Historiquement, les bases de données relationnelles classiques (utilisant SQL) constituent *de facto* le système standard pour gérer des données d'entreprise du fait qu'elles adhèrent à ce qu'on appelle le modèle ACID. Il s'agit d'une sorte de modèle d'exigence que les bases de données modernes respectent absolument. Il signifie :

- Atomicité : une opération est une unité indivisible. Ce qui signifie que soit toutes les opérations d'une transaction sont effectuées, soit aucune ne l'est. Par exemple, quand on transfère de l'argent entre deux comptes, l'argent doit être débité du premier et crédité sur le second, ou sur aucun des deux.
- Cohérence : une transaction maintient l'intégrité des données. Autrement dit, elle commence dans un état cohérent et se termine aussi dans un état cohérent. Une fois le transfert effectué, les totaux des comptes doivent rester identiques.
- Isolation : Les transactions faites simultanément ne doivent pas interférer les unes avec les autres. Chaque transaction est donc dite isolée. Si l'on prend deux utilisateurs qui modifient le même groupe de sortie, les données modifiées ne doivent pas se mélanger.
- Durabilité : Une fois qu'une transaction a été confirmée, ses effets sont permanents, même en cas de défaillance du système. Si on achète un abonnement, le paiement est enregistré même si le serveur tombe en panne juste après.

Il garantit une forte cohérence des données, ce qui est essentiel pour certaines fonctionnalités comme les systèmes bancaires ou de gestion d'inventaire [9]. Pour ce type d'usage, les bases de données relationnelles classiques sont tout à fait fiables. Si l'on règle les soucis de scalabilité grâce au sharding et au réplications (Chapitre II.B), de tels systèmes semblent suffisants, même pour Summer-Trip, et pour cause :

En premier lieu, les bases de données relationnelles classiques ont l'avantage de bien s'adapter à de nombreux types de données différents [9], par exemple le type de données JSON est également pris en charge par plusieurs bases de données (dans leur version les plus récente). Notamment IBM DB2, MySQL et PostgreSQL [2].

Plus encore, ils permettent surtout d'assurer de bonnes relations entre les entités [2], [9]. En effet, de tels systèmes s'adaptent extrêmement bien lors de relations *One To Many* (une entité reliée à plusieurs autres, comme un panier associé à plusieurs articles qu'il contient) et même plutôt bien dans les plus complexes *Many To Many* (plusieurs entités reliées à plusieurs autres, comme des joueurs de foot qui sont dans plusieurs équipes) [2]. Les databases SQL classiques sont ainsi un très bon choix dans la plupart des cas. Et même dans le pire des cas, comme des données à structure complexes (XML, JSON, HashMap, Array, etc..), sans relation avec d'autres, elles restent "acceptables" [2].

Cependant, bien qu'un système qui soit acceptable dans la plupart des cas soit suffisant pour des usages basiques, à mesure que l'infrastructure d'une application se consolide et que les métriques d'usages sont toujours plus exigeantes, il convient de réfléchir à optimiser l'architecture globale là où cela est possible.

En particulier, si une partie des données est structurée d'une façon tout juste acceptable, ne pourrait-on pas essayer de changer cette partie pour une architecture plus "naturelle" [10] ?

Par exemple, un changement simple (et qui devient vite indispensable pour toute les architecture de stockage moderne) serait de transiter le stockage des médias "lourds" (en particulier des images, fichiers, photos et vidéos vers ce qu'on appelle un *Bucket*. Après réflexion, notre choix portera sur Amazon S3 [11], qui est très réputé pour ses fonctionnalités avancées comme le S3 Glacier (qui permet de faire de l'archivage), les Buckets intelligents (optimisation des coûts) ou encore son scaling infini [12].

Par ailleurs, si notre application cherche à représenter des liens entre les individus, à la manière d'un réseau social, pourquoi ne pas représenter ces utilisateurs par une structure en graphe ? Nous répondrons à ces questions de choix de structure de données dans la partie suivante. Car les bases de données relationnelles ont avant tout un dernier problème : le fait de prioriser systématiquement la cohérence à la disponibilité [13].

En *system design* informatique, il existe un théorème fondamental appelé le théorème CAP, découvert par Eric Brewer, qui dit qu'un système distribué ne peut pas garantir pleinement les trois propriétés suivantes en même temps [14], [15] :

- Cohérence (C) : Chaque lecture renvoie toujours la version la plus récente d'une donnée.
- Disponibilité (A) : Chaque demande (lecture/écriture) reçoit une réponse, même en cas de défaillance d'un nœud.
- Tolérance de partition (P) : Le système continue de fonctionner correctement même si les nœuds sont isolés les uns des autres en raison de défaillances du réseau.

En cas de partitionnement du réseau (c'est-à-dire une défaillance du réseau (le 3) ou tout autre scénario fréquent et inévitable dans les systèmes distribués), un système doit choisir entre :

- La cohérence (C) : Opérations en bloc pour garantir la synchronisation des données. On parle alors de système "CP", pour priorité à la consistance entre les données et la tolérance aux partitions.
- Disponibilité (A) : Autoriser les opérations sur des données potentiellement non synchronisées. Ici on dira d'un système qu'il est "AP", qu'il priorise l'accessibilité absolue des données même si elles ne sont pas strictement à jour.

Et bien que l'on puisse discuter théoriquement des bases de données réparties CA (qui négligent (P) donc), elles ne peuvent pas exister dans la pratique car, dans un système distribué, les partitions sont inévitables [14], [15].

Par définition, et selon le modèle ACID, les bases de données relationnelles sont toujours des systèmes CP. Cela implique que ces systèmes, en plus d'être tolérants aux partitions, sont optimisés pour apporter en lecture les données les plus à jour possibles, même en cas de sharding à travers différents nœuds. Cela signifie également que l'on néglige pour cela la disponibilité absolue (A), toujours en accord avec le théorème CAP.

Toutefois il peut arriver qu'une application nécessite pour certaines fonctionnalités une grande disponibilité plutôt qu'une grande cohérence entre les données [13]. De fait, nous verrons dans la partie suivante que certains systèmes favorisant la disponibilité (A) au détriment de la cohérence (C) ont pour conséquence de grandement faciliter la scalabilité globale d'un système de stockage.

En définitive, on comprend alors que si les bases de données relationnelles classiques ont beau être un choix préférentiel pour la plupart des usages, il existe des situations où celles-ci ne sont pas forcément les plus judicieuses. Dans notre cas, cela dépendra des usages et des priorités des différentes parties de Summer-Trip. Nous allons maintenant explorer les systèmes permettant d'optimiser notre architecture de données, afin d'aller plus loin que le scaling horizontal (Chapitre II.B) de base de données relationnelles classiques.

B. L'utilisation de NoSQL pour certaines données

L'arrivée de nouveaux besoins de stockage distribué a conduit à l'émergence de nouvelles formes de base de données. Ces nouvelles technologies que l'on appelle NoSQL (pour *Not Only* SQL) se distingue de celles vues précédemment (au modèle ACID) en suivant le modèle BASE [16], qui signifie :

- *Basically Available* (Disponible en principe) : Le système garantit que chaque demande reçoit une réponse, même si celle-ci ne reflète pas l'état le plus récent des données. (Par exemple, si un site de vente en ligne affiche un stock légèrement pas à jour pour garantir la rapidité).
- *Soft-Stat* (État non strict): Les données peuvent être dans un état intermédiaire ou en cours de synchronisation. L'état d'une donnée peut changer sans intervention explicite, car les mises à jour se propagent progressivement.
- *Eventual consistency* (Cohérence éventuelle) : Les données finiront par devenir cohérentes sur tous les nœuds si aucune nouvelle écriture n'est effectuée. (Dans un réseau social, un message publié sur un serveur peut ne pas apparaître immédiatement sur tous les serveurs, mais il sera visible partout après un certain délai).

Les technologies NoSQL qui suivent ce modèle priorisent donc la disponibilité absolue (le A du théorème CAP vu précédemment) plutôt que la synchronisation instantanée (C). On compte aujourd'hui 4 grande familles de NoSQL : les clé-valeur, les graphs, en stockage par colonne, et par document [9], dont leurs propriétés distributive sont présenté Fig. 3. Les technologies respectives les plus connues et les plus avancées sont : Redis, Neo4j, Cassandra et MongoDB. Après les avoir longuement analysés et comparés, il apparaît que :

Tout d'abord, les clé-valeur (comme Redis) serait idéale pour des données simples qui peuvent être représentées facilement sous la forme simple d'une clé et d'une valeur (i.e sans dépendance liée à des relation complexes entre les données), à la manière d'une HashMap en Java ou d'un Dictionnaire en Python [2]. En effet, elles offrent de hautes performances, une grande évolutivité et une grande flexibilité. Ce type de NoSQL excelle dans les systèmes de caching, une technique d'amélioration essentielle de la scalabilité qui sera plus détaillé dans le cadre de Summer-Trip dans Chapitre III.C

Ensuite, les NoSQL de type document (comme MongoDB) permettent quant à elles de stocker des informations, non pas sous la forme de table mais plutôt sous la forme d'un document dont l'encodage/sérialisation rend le transfert de données plus efficace que sur les SQL classique [2] comme le JSON ou le BSON propre à MongoDB par exemple. Elles jouissent d'une partition tolérante avec une haute cohérence (CP), ce qui les rend judicieuses pour stocker des données demandant une cohérence forte mais sans agrégation composite, jointures complexes (si on a des relations *Many To Many* par exemple) ou les opérations de table dérivée [9]. Dans notre cas, nous n'allons pas utiliser ce type de base. D'une part car les informations "délicates" (comme les infos de connexion ou de paiement) qui nécessitent une forte cohérence seront toujours incluses dans une base de données relationnelles classiques (suivant le scaling horizontal vu Chapitre II.B). D'autres part car celles qui vont suivre sont, après études, beaucoup plus judicieuses pour le cas précis de Summer-Trip.

En effet, dans le cas d'une application permettant de relier les gens entre eux comme Summer-Trip, une base de données

en graph (comme Neo4j, la plus aboutie de ce type à ce jour [9]) serait idéal pour représenter les relations entre les groupes de sorties, les individus et les suggestions d'activités. Elle utilise un langage de query simple et puissant, Cypher [17], qui permet d'exprimer des requêtes complexes de manière concise et lisible. Si l'on voulait trouver tous les amis d'un utilisateur qui ont visité une certaine ville, Cypher rendrait la tâche trivial comparé à une requête SQL classique. Plus encore, il s'agit de la façon la plus naturelle de traiter les relations *Many To Many* entres les entités et elle permet une grande évolutivité, granularité et souplesse dans l'accès aux données [2].

Feature	Wide Column Store (Cassandra)	Document Store (MongoDB)	Key Value pair Store (Redis)	Graph Database (Neo4j)
Sharding and Partitioning	Auto sharding and order preserving	Built in and order preserving	Auto sharding and no order	Supports sharding but should be avoided
Scaling	Horizontal	Horizontal	Horizontal	Horizontal
Replication	Selectable Replication Factor	Master slave	Relaxed Master slave	Causal Clustering using Raft protocol (master slave)

Fig. 3: Analyse des bases de données NOSQL sur la base des propriétés distributives.

Néanmoins, pour construire une architecture de stockage à forte scalabilité, les bases de données orientées colonnes sont également un excellent choix. La plus utilisée de ce type à l'heure actuelle est Cassandra [13]. Le principe fondamental de Cassandra est de répartir uniformément les données sur des nœuds, plutôt que sur une seule unité centrale. Cela permet d'appliquer une scalabilité linéaire à l'infini [18], [19]. Elle a été créée en 2008 par Facebook spécialement pour traiter de très grandes quantités de données réparties sur de nombreux serveurs de base tout en fournissant un service hautement disponible sans point de défaillance unique. En outre, son élasticité permet un débit de lecture et d'écriture qui augmente linéairement (contrairement aux autres bases de données) au fur et à mesure que de nouvelles machines sont ajoutées, sans temps d'arrêt ni interruption des applications [19]. En revanche, elle ne peut ni faire de jointures, ni de sous-requêtes, car elle privilégie la dénormalisation des données [2]. C'est pour cette raison que Cassandra serait idéale pour certaines fonctionnalités de Summer Trip qui nécessitent d'effectuer des requêtes ultra-rapides avec des accès directs, mais qui ne font pas appel à des données trop liées entre elles nécessitant des jointures. Il pourrait s'agir des messages de groupes, des notifications en temps réel et surtout des activités proposées aux utilisateurs, puisqu'il est essentiel de proposer un flux d'activités défilants constamment fonctionnel.

En définitif, le point commun de ces bases de données NoSQL est qu'elles sont très pertinentes dès lors que la consistance absolue (C) n'est pas une priorité comparé à l'accès systématique (A) des informations [9]. De telles technologies pourraient s'avérer extrêmement pertinentes pour améliorer l'architecture de stockage de Summer-Trip. Pour leur mise en production, le fait d'utiliser Docker [20] pour conteneuriser chaque base de données puis de gérer leurs orchestrations grâce à Kubernetes [21] dans des *pods* (groupe de conteneurs qui fonctionnent ensemble) semble vivement améliorer les performances [22], comparés à des machines virtuelles VMware. Pour Cassandra par exemple, utiliser Docker offre des performances supérieures de 16 à 29 % en fonction de la charge (lecture seule, écriture seule, charges mixtes) comparé à VMware [22].

C. Ajout d'un système de caching pour optimiser le temps de lecture

Toujours dans l'idée d'améliorer la récupération de données de la base de données, nous allons nous appuyer sur des stratégies de data caching pour réduire la latence, limiter la charge sur la base de données principale et optimiser les performances globales.

Data caching est une technique qui permet de stocker temporairement des données souvent consultées dans une mémoire rapide afin de réduire le temps d'accès et la charge sur la base de données principale.

Un système de cache va écrire les données dans la mémoire vive ce qui permet d'avoir des performances plus rapides. [23]

Le fonctionnement du caching repose sur un système de clés-valeurs où chaque donnée est associée à une clé unique. Lorsqu'une application demande une donnée, le processus suit généralement ces étapes :

1) **Requête au cache :** L'application vérifie si la donnée est déjà disponible dans le cache .
2) **Cache Hit ou Miss :** Si la donnée est trouvée ce que l'on nomme un cache hit, elle est renvoyée directement, ce qui réduit considérablement la latence (Ci dessous seulement la flèche 1). Si elle est absente, un cache miss, l'application récupère la donnée depuis la base de données principale, la stocke dans le cache pour les prochaines requêtes, et la renvoie.(Ci dessous les trois flèches)

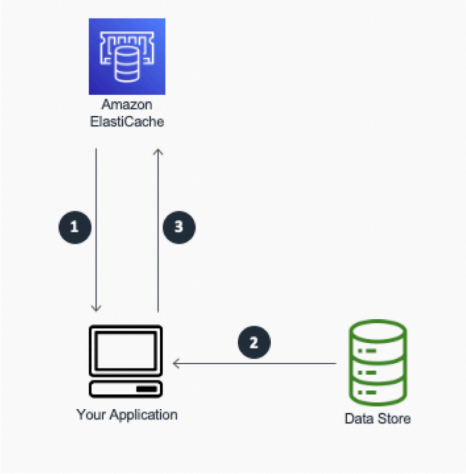

Fig. 4: [24]

1) **Expiration et Synchronisation :** Les données dans le cache peuvent être configurées avec un TTL (Time To Live), c'est-à-dire une durée d'expiration après laquelle

elles sont supprimées. Cela garantit que les données restent à jour tout en limitant la consommation de mémoire.

Dans notre cas, certaines entités comme User, Group, et Activity sont accédées fréquemment pour des opérations de lecture:

- User : les informations utilisateur (nom, email, image) peuvent être mises en cache, car elles sont souvent consultées et rarement modifiées.
- Group : des données sur les groupes et les membres sont fréquemment sollicitées, ce qui peut générer une forte charge si les accès sont directs à la base.
- Activity : les activités planifiées ou liées aux utilisateurs peuvent être sujet à des requêtes répétitives.

—

IV. De nouveaux besoins suite au développement de l'entreprise

A. Modèle de recommandation prédictif :

Avec la croissance rapide de l'application Summer Trip, les besoins en infrastructures capables de gérer et analyser de larges volumes de données se sont accrus de manière significative. La conception d'un modèle de recommandation prédictif basé sur les logs d'activités des utilisateurs représente un élément essentiel pour améliorer l'expérience utilisateur et optimiser les activités affichées sur la plateforme.

a) *Étape 1 : Récupération des logs dans un Data Lake:*

L'intégration des logs d'activités utilisateurs dans un Data Lake constitue la première étape essentielle pour collecter des données brutes variées et en grande quantité. Ces logs incluent des interactions utilisateur sur les activités proposées (e.g., « swipe à droite » ou « swipe à gauche ») et les statistiques des groupes formés pour les voyages [25].

Les Data Lakes se démarquent par leur flexibilité et leur capacité à stocker des données structurées, semi-structurées et non structurées (JSON, vidéos, logs, etc.). Pour Summer Trip, un Data Lake basé sur des technologies comme Hadoop Distributed File System (HDFS) ou Amazon S3 peut être utilisé pour une ingestion rapide et fiable [26]. L'utilisation de flux en temps réel via Apache Kafka ou Amazon Kinesis renforce cette architecture en permettant une mise à jour instantanée des données collectées [27].

b) *Étape 2 : Transformation des données via un pipeline ELT:*

Les données brutes collectées dans le Data Lake nécessitent une transformation pour être exploitables par des modèles de Machine Learning. Cette étape s'effectue grâce à des pipelines ELT (Extract, Load, Transform) modernes utilisant des outils comme Apache Spark pour garantir un traitement distribué et scalable. Dans le contexte de Summer Trip, les transformations incluent :

- **Nettoyage des données** : Suppression des doublons, gestion des valeurs manquantes, et détection d'anomalies (par exemple, logs d'activités incohérents).
- **Normalisation des formats** : Conversion des données en tables Parquet ou ORC pour optimiser leur accès ultérieur [28].
- **Enrichissement des données** : Croisement des logs d'activités avec des informations sur les utilisateurs, les groupes et les préférences exprimées.

Ces étapes garantissent une structuration optimale des données pour l'analyse prédictive.

c) *Étape 3 : Gestion des métadonnées et gouvernance:*

Un défi majeur des Data Lakes réside dans leur capacité à éviter de devenir des « Data Swamps » (fouillis de données inutilisables). Pour Summer Trip, l'utilisation d'un outil de gestion des métadonnées comme AWS Glue Data Catalog ou Apache Atlas permet de cataloguer les données en suivant leur origine, leur transformation, et leur destination [29]. Ces solutions garantissent également une gouvernance stricte en conformité avec des réglementations comme le GDPR, ce qui est essentiel pour le traitement des données utilisateur sensibles.

En outre, des stratégies de gouvernance sont mises en place pour sécuriser l'accès au Data Lake, incluant le chiffrement des données sensibles et une gestion granulaire des droits d'accès [30].

d) *Étape 4 : Préparation des données pour le Machine Learning:*

Une fois nettoyées et structurées, les données sont utilisées pour entraîner des modèles prédictifs. Ces modèles exploitent des frameworks tels que TensorFlow, PyTorch ou Scikit-Learn pour analyser les logs utilisateurs et identifier des patterns dans leurs interactions [31].

Les étapes clés comprennent ::

- **Sélection des caractéristiques** : Extraction des attributs pertinents (e.g., temps passé à évaluer une activité, préférences historiques).
- **Division des données** : Séparation en ensembles d'entraînment, de validation et de test pour évaluer la performance des modèles.
- **Automatisation des flux de données** : L'utilisation de frameworks comme Apache Airflow permet d'orchestrer le passage des données transformées dans les modèles ML, garantissant une réactivité en temps réel.

Applications pratiques:

Le modèle prédictif conçu dans ce cadre pour Summer Trip permet de réaliser des avancées notables :

- **Recommandations personnalisées** : Proposer des activités en fonction des interactions passées des utilisateurs (e.g., préférences exprimées via le swipe ou le temps passé sur des activités similaires) [32].
- **Optimisation des calendriers** : Organiser les activités les plus appréciées par le groupe dans des plages horaires qui maximisent leur satisfaction.
- **Analyse prédictive des tendances** : Identifier des activités qui pourraient devenir populaires à l'avenir, facilitant la stratégie de partenariats avec des fournisseurs locaux [33].
- **Réduction des abandons** : Anticiper les utilisateurs susceptibles de se désengager en raison d'un manque de contenu pertinent et les cibler avec des suggestions plus adaptées ou des offres promotionnelles [34].

Impact sur Summer Trip:

La mise en œuvre de ce modèle améliore significativement l'expérience utilisateur en proposant des recommandations plus pertinentes, ce qui renforce l'engagement et la satisfaction. En exploitant les logs d'activités dans un Data Lake, la plateforme se dote également d'une infrastructure évolutive et résiliente, capable de soutenir la croissance continue de son audience et de ses services.

B. Analyse massive de données d'utilisateurs

La capacité de traiter et d'analyser des données utilisateur à grande échelle est un élément clé pour soutenir la croissance de Summer Trip. Afin de permettre une exploration efficace et rapide des données, le Data Warehouse (DW) est la solution idéale grâce à son architecture optimisée pour les requêtes analytiques complexes. Cette approche permet d'extraire des insights stratégiques pour améliorer l'expérience utilisateur, affiner les offres et anticiper les tendances.

a) *Étape 1 : Migration des données nettoyées vers un Data Warehouse:*

Les données collectées et transformées dans le Data Lake sont ensuite migrées vers un Data Warehouse pour permettre des analyses OLAP (Online Analytical Processing) [25]. Le modèle de stockage privilégié est le modèle en étoile (Star Schema) [2], qui organise les données en une table centrale de faits entourée de tables de dimensions.

- **Table de faits** : Contient les mesures quantitatives comme les interactions des utilisateurs avec les activités (e.g., clics, temps passé, swipes effectués) ou les conversions (e.g., activités ajoutées au calendrier).
- **Tables de dimensions** : Incluent des détails sur les utilisateurs (âge, localisation), les catégories d'activités (culturelles, sportives, gastronomiques), et les périodes temporelles (jour, semaine, mois) [26].

Exemple d'application dans Summer Trip : Pour une analyse des préférences d'activités à Madrid, une table de faits peut inclure les « likes » des utilisateurs pour chaque activité, et une table de dimensions peut classifier ces activités par type (e.g., musées, bars). Cela permet de comprendre les préférences spécifiques selon des groupes d'utilisateurs.

b) *Étape 2 : Optimisation des requêtes analytiques:*

Une fois les données intégrées dans le Data Warehouse, elles sont accessibles pour des analyses en temps quasi réel à l'aide de solutions comme Amazon Redshift, Snowflake ou Google BigQuery [27]. Ces plateformes adoptent une architecture en colonnes et un traitement parallèle massivement distribué, réduisant significativement les temps de réponse des requêtes.

Des techniques avancées sont utilisées pour optimiser davantage les performances :

- **Ordonnancement des données (Z-ordering)** : Permet de regrouper les données associées à des clés de tri (e.g., région, période) pour accélérer les requêtes multidimensionnelles [28].
- **Filtres Bloom** : Réduisent le volume de données scannées en excluant rapidement les valeurs qui ne correspondent pas aux critères [29].
- **Découplage stockage-calcul** : Avec Snowflake, cette technique permet d'allouer dynamiquement des ressources pour répondre à des charges de travail fluctuantes, un avantage crucial pour Summer Trip lors des périodes de forte utilisation, comme les vacances estivales [30].

c) *Étape 3 : Construction des tableaux OLAP pour analyses:*

Le Data Warehouse organise les données en tableaux OLAP multidimensionnels pour une exploration analytique approfondie. Ces tableaux permettent d'analyser les données selon plusieurs axes, offrant une vue complète des tendances et comportements des utilisateurs.

Dimensions d'analyse pour Summer Trip :

- Temps : Identifier les pics d'activité utilisateur, comme les périodes où les utilisateurs swipent le plus pour choisir des activités.
- Localisation : Comparer les préférences d'activités selon les villes (e.g., activités sportives populaires à Barcelone vs activités culturelles à Florence).
- Profil utilisateur : Déterminer les tendances spécifiques à des groupes démographiques (e.g., jeunes adultes vs familles) [31].

Les tableaux OLAP permettent aussi d'exécuter des calculs complexes en une seule requête SQL, comme des moyennes pondérées ou des agrégations par région.

Exemple concret : Une requête OLAP peut analyser les « top 5 » activités les plus aimées à Madrid pour des groupes d'amis planifiant un voyage entre le 1er et le 10 décembre. Ces insights peuvent ensuite être utilisés pour ajuster les suggestions d'activités dans l'application.

d) *Étape 4 : Outils de visualisation et reporting:*

Les résultats des analyses OLAP sont traduits en tableaux de bord interactifs pour les équipes de décision. Des outils comme Tableau, Power BI ou Amazon QuickSight permettent de visualiser en temps réel les indicateurs clés de performance (KPI) de l'application [32]. Ces tableaux de bord peuvent inclure des graphiques interactifs montrant :

- Le taux de conversion : Proportion d'activités aimées par rapport à celles ajoutées au calendrier.
- Les activités les plus populaires : Par ville et par groupe démographique.
- Les périodes d'activité maximale : Heures ou jours où l'engagement utilisateur est le plus élevé.

Application stratégique pour Summer Trip : Grâce à ces visualisations, l'équipe peut identifier les tendances émergentes et ajuster les recommandations ou les partenariats locaux pour maximiser la satisfaction utilisateur et les revenus.

e) *Applications pratiques:*

Les analyses fournies par le Data Warehouse ouvrent la voie à des initiatives stratégiques pour Summer Trip :

- Segmentation avancée des utilisateurs : Identifier les segments les plus engagés pour des campagnes marketing ciblées. Par exemple, promouvoir des activités gastronomiques pour les groupes ayant montré un intérêt élevé pour ce type d'activité.
- Prévision de la demande : Anticiper les activités populaires dans des destinations spécifiques en fonction des tendances historiques, permettant d'optimiser les partenariats avec les fournisseurs locaux.

- Optimisation des offres personnalisées : Proposer des réductions sur les activités sous-représentées pour équilibrer les calendriers des utilisateurs, augmentant ainsi leur satisfaction et l'utilisation de la plateforme.

f) *Impact global sur Summer Trip:*

L'intégration d'un Data Warehouse bien conçu permet non seulement de répondre aux besoins analytiques, mais également d'améliorer l'agilité stratégique de l'entreprise. En combinant les analyses OLAP avec les résultats des modèles prédictifs du Data Lake, Summer Trip est en mesure d'offrir des recommandations personnalisées et de renforcer la fidélité des utilisateurs. Cette infrastructure analytique complète soutient également la planification à long terme en fournissant des insights exploitables à tous les niveaux de l'entreprise.

V. Conclusion : L'architecture finale de notre application

En définitive, nous avons trouvé un ensemble de bonnes pratiques et de technologies qui vont nous permettre de mettre en place une architecture solide de stockage et de traitement de données pour Summer-Trip lors de la deuxième phase d'exécution de ce projet. En premier lieu, nous allons conserver les données les plus critiques de l'infrastructure, comme les informations lié à l'entreprise, aux employés, ou encore au information sensible des utilisateurs (comme les tables d'authentification), dans une base de données relationnelles classiques de type PostgreSQL ainsi que Citus, l'extension permettant du sharding sur plusieurs nœuds. Le tout combiné à des réplications des différentes instances pour s'assurer d'avoir un système rapide, fiable et résilient. Pour ainsi prêter à la fois attention à réduire la latence en répartissant les bases de données selon différentes régions du globe, ainsi qu'à prévenir les risques de plantage de serveur et s'assurer que le système fonctionne toujours grâce à des bases de données de secours.

Par ailleurs, nous profiterons des avantages octroyés par les technologies plus récentes que sont les bases de données NoSQL. Tout d'abord, il sera particulièrement utile de stocker les médias photo, vidéo, etc..) sur un Bucket S3. Puis nous pourrons profiter du scaling horizontal naturel et de la disponibilité absolue de Cassandra pour le contenu lié aux activités, aux suggestions et au messages de groupes. Par ailleurs, nous allons structurer nos utilisateurs et nos groupes grâce à Neo4j, pour laquelle sa structure en graphe et sa capacité d'adaptation et d'évolution vont grandement simplifier l'expansion du nombre d'utilisateurs sur la plateforme. Enfin, nous profiterons des capacités offertes par Redis pour afin d'implémenter un système de cache sur les informations les plus souvent récupérés par les utilisateurs.

L'orchestration de tous ces différents systèmes de stockages se fera avec Kubernetes. Grâce à la conteneurisation de chaque instances de stockages, ils sera possible d'automatiser leurs déploiement lorsque nous pourrions en avoir besoin (en cas de panne ou d'usage intensif du système). Pour finir, nous répondrons aux nouvelles problématiques complexes de l'entreprise grâce à un Data Lake basé sur Hadoop pour les pipelines ELT, qui stockera les logs bruts des activités utilisateurs afin de pouvoir alimenter les modèles de machine learning. Plus encore, nous utiliserons un Data Warehouse Snowflake pour les flux analytiques, avec un schéma en étoile pour les opérations OLAP. Ainsi, on garantit un stockage scalable et une extraction rapide d'insights, répondant aux besoins nouveaux de modélisation prédictive et d'analyse massive des données de Summer-Trip.

VI. Réalisation concrète : Gestion et planification de projet

La seconde partie de ce projet porte sur l'implémentation concrète de la solution présenté dans ce rapport, à réaliser en 4 semaines. Nous allons nous organiser de la façon suivante :

A. Avant tout : Les outils et bonnes pratiques pour s'assurer d'une bonne organisation

Si ce projet à beau pouvoir sembler *intimidant* au premier regard, de part l'ampleur du travail à réaliser, de la quantité d'information et de compétence que l'on va devoir acquérir ou même de la grande autonomie qui nous ait accordé. Nous ne somme pas pour autant sous-préparé face à l'ampleur de la tache. Nous avons déjà réalisé un certain nombres de projet d'informatique en groupe et nous prévoyons d'utiliser de multiples outils et pratiques utiles lors de ce projet.

D'abord nous procéderons systématiquement selon la *méthode AGILE*, qui consiste à privilégier les courtes implémentations et évaluations régulières d'avancement plutôt de constater le résultats juste avant la fin de la *deadline*. Ensuite, que ce soit au travers d'IDE générique comme VSCode ou plus spécialisé pour l'ingénierie de données tel que DataGrip (qui est plus propice à la gestion de stockage multiple comme ici), nous porterons une grande attention à notre usage de *Git* et *GitHub*. D'une part, parce que c'est essentiel pour avancer correctement et garder l'historique en cas de problème. D'autre part, car nous pourrions avoir besoin de mettre en place des plateforme de déploiement automatique de code ou de *migration* (mise à jour *post-déploiement*) de base de données. Auquel cas des platformes de CI/CD offerte par GitHub Action pourront s'avérer très utile.

Pour ce qui est de l'organisation, nous allons utiliser un tableau *Kanban* grâce au logiciel Trello. Celui-ci permet de marquer clairement ce qui *est à faire*, ce qui est *en cours* (et quel(s) membres s'en charge(s)) et ce qui est *terminé*. Nous l'avions déjà utilisé par le passé et il a grandement fait s'est preuves, en particulier lorsqu'on le combine avec nos réunions journalières que l'on va effectuer chaque matin pour faire le point l'avancement.

B. La Question du budget

Pour ce qui est du budget nécessaire au bon déroulement de ce projet, nous avons pris la décision de travailler quasi exclusivement avec des outils soit en usage local, soit sur des serveurs, mais **pas** sur du Cloud. Hormis éventuellement pour le Bucket de stockage des médias, de part les avantages évoqués précédemment et la facilités à le mettre en place.

Cela signifie que notre infrastructure globale ne devrait pas nous coûter trop chère. Même pour un le coût lié à la location

d'un serveur, nous ne devrions normalement pas dépasser les 100 euros octroyés avec l'école pour assurer sa réalisation.

C. Semaine 1 : La mise en place du scaling horizontal et de l'orchestrateur Kubernetes

Nous allons commencer par travailler pleinement sur la base de donnée relationnelle classique et sur le *scaling* de celle-ci. Car il s'agit du pilier fondamental de notre application. Le simple fait qu'elle va contenir les informations sensibles et critiques des utilisateurs dès la création de leurs compte implique que les interactions vont ce faire sur celles-ci avant même de parler de NoSQL ou de lacs et d'entrepôts de données. Nous allons donc créer notre base de données PostgreSQL et mettre en place le sharding offert par *Citus* pour obtenir différents nœuds de notre instance PostgreSQL, reliés ensemble. Ensuite nous allons la conteneuriser grâce à Docker, chose que nous avons appris durant nos précédents années d'études en cours de DevOps. Puis nous allons apprendre à gérer Kubernetes. Cet orchestrateur va nous permettre de créer et gérer à souhait plusieurs instance de notre base de données conteneurisé. Nous sommes déjà à l'aise avec l'utilisation de PostgreSQL et de Docker, l'essentiel de notre apprentissage ici concernera Citus et Kubernetes. En nous focalisant tous sur cette partie essentiel de notre projet, nous devrions amplement réussir à implémenter cette mise en place en une semaine.

D. Semaine 2 : Ajout de la réplication de la DB et mise en place des NoSQL

Pour cette deuxième semaine, nous allons commencer à nous diviser les tâches :

Deux membres du groupes travaillerons sur la mise en place de la réplication de la base de données SQL principale. La difficulté ici ne sera pas tant de mettre en place les réplications en elle même, mais plutôt d'assurer une bonne synchronisations en les nœuds partagés et leurs *replica* respectifs. Deux membres, quand à eux travaillerons sur la création des bases de données NoSQL et réfléchirons à l'ajustement du format et de la structure des données contenues dans chaque bases de données. Un dernier membre commencera quand à lui à déjà travailler sur l'implémentation des pipelines de données pour le Data Lake (pour le Machine Learning) et le Data Warehouse (pour l'analyse), car ces étapes prendrons sans aucun doute plus de temps.

E. Semaine 3 : Mise en relation de toutes les base de données (relationelles ou non) et avancée des pipelines.

Pour cette troisième semaine, nous finirons la liaisons entre les différentes bases de données et assurerons le paramétrages complet de la scalabilité de celles-ci (toujours par du scaling horizontal).

Nous ajusterons aussi l'orchestrateur Kubernetes pour réagir correctement en cas de variations des charges de débit et de volumétrie. Grâce à cela, nous allons pouvoir réaliser des Benchmarks pour tester la fiabilité du système et feront en sorte de conserver les résultats pour les présenter lors de la présentation final (fin mars 2024).

Enfin nous finirons la première pipeline de données à destinations du Data Lake, en prenant soins de sauvegarder les formats, code et schéma finaux de celle-ci pour la présentation.

F. Semaine 4 : Ajustement globale, dernière pipeline et préparation à la présentation devant le jury.

Pour cette dernière semaine, nous allons finir la pipeline lié au traitement de données à destination d'un Data Warehouse. nous testerons alors les deux pipelines en poussant le débit et la quantité de données dans le système de stockage globale pour procéder à un test générale de toute l'infrastructure de ce projet. Nous conserverons, bien sûr, les résultats et préparerons la conclusion sur ce qui a marché, ou non, suite aux tests.

Enfin nous allons préparer la présentation devant le jury, à savoir écrire le *pitch*, faire la diapositive et synthétiser les résultats.

References

[1] A. Zulkifli, « Accelerating Database Efficiency in Complex IT Infrastructures: Advanced Techniques for Optimizing Performance, Scalability, and Data Management in Distributed Systems ».

[2] M. Kleppmann, *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems (Book)*. " O'Reilly Media, Inc.", 2017.

[3] Wikipedia contributors, « Shard (database architecture) — Wikipedia, The Free Encyclopedia ». [En ligne]. Disponible sur: https://en.wikipedia.org/w/index.php?title=Shard_(database_architecture)&oldid=1260915144

[4] S. Solat, « Sharding distributed databases: A critical review », *arXiv preprint arXiv:2404.04384*, 2024.

[5] A. Shakarami, M. Ghobaei-Arani, A. Shahidinejad, M. Masdari, et H. Shakarami, « Data replication schemes in cloud computing: a survey », *Cluster Computing*, vol. 24, p. 2545-2579, 2021.

[6] T. Pohanka et V. Pechanec, « Evaluation of replication mechanisms on selected database systems », *ISPRS International Journal of Geo-Information*, vol. 9, nº 4, p. 249, 2020.

[7] P. Rantanen, « Database replication: an overview of replication techniques in common database systems », 2010.

[8] PostgreSQL Global Development Group, « PostgreSQL Documentation: Runtime Configuration - Replication ». [En ligne]. Disponible sur: https://www.postgresql.org/docs/current/runtime-config-replication.html

[9] A. Gupta, S. Tyagi, N. Panwar, S. Sachdeva, et U. Saxena, « NoSQL databases: Critical analysis and comparison », p. 293-299, 2017.

[10] P. P. Khine et Z. Wang, « A review of polyglot persistence in the big data world », *Information*, vol. 10, nº 4, p. 141, 2019.

[11] Amazon, « AWS S3 Bucket official documentation », 2024.

[12] M. R. Palankar, A. Iamnitchi, M. Ripeanu, et S. Garfinkel, « Amazon S3 for science grids: a viable solution? », 2008, p. 55-64. [En ligne]. Disponible sur: https://doi.org/10.1145/1383519.1383526

[13] M. Diogo, B. Cabral, et J. Bernardino, « Consistency models of NoSQL databases », *Future Internet*, vol. 11, nº 2, p. 43, 2019.

[14] A. IBM Official, « What is the CAP theorem? », 2024.

[15] M. Kleppmann, « A Critique of the CAP Theorem », *arXiv preprint arXiv:1509.05393*, 2015.

[16] D. Ganesh Chandra, « BASE analysis of NoSQL database », *Future Generation Computer Systems*, 2015.

[17] Neo4j, « Official Neo4j's documentation », 2024.

[18] Apache, « Official Apache Cassandra's documentation », 2024.

[19] T. Rabl, M. Sadoghi, H.-A. Jacobsen, S. Gómez-Villamor, V. Muntés-Mulero, et S. Mankowskii, « Solving big data challenges for enterprise application performance management », *arXiv preprint arXiv:1208.4167*, 2012.

[20] Docker, « Official Docker's documentation », 2024.

[21] T. L. Fundation, « Official Kubernetes's documentation », 2024.

[22] S. Shirinbab, L. Lundberg, et E. Casalicchio, « Performance evaluation of containers and virtual machines when running Cassandra workload

concurrently », *Concurrency and Computation: Practice and Experience*, vol. 32, nº 17, p. e5693, 2020.
[23] ByteByteGo, « Cache Systems Every Developer Should Know ». [En ligne]. Disponible sur: https://www.youtube.com/watch?v=dGAgxozNWFE&t=196s
[24] A. web service, « (Database Caching Strategies Using Redis) », vol. 65, p. 6, 2021.
[25] E. Saddad, A. El-Bastawissy, H. Mokhtar, et M. Hazman, « Lake Data Warehouse Architecture for Big Data Solutions », *International Journal of Advanced Computer Science and Applications*, 2020.
[26] A. Nambiar et D. Mundra, « An Overview of Data Warehouse and Data Lake in Modern Enterprise Data Management », *Big Data and Cognitive Computing*, 2022.
[27] M. Armbrust, A. Ghodsi, R. Xin, et M. Zaharia, « Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics », 2021.
[28] N. Miryala et D. Gupta, « Big Data Analytics in Cloud – Comparative Study », *International Journal of Computer Trends and Technology*, 2023.
[29] A. Nuthalapati, « Architecting Data Lakehouses in the Cloud: Best Practices and Future Directions », *International Journal of Science and Research Archive*, 2024.
[30] V. Mandala, « Exploring the Potential of Snowflake Analytics for Real-Time Predictive Analytics », *ESP International Journal of Advancements in Computational Technology*, 2024.
[31] S. Shukla, « Developing Pragmatic Data Pipelines Using Apache Airflow on Google Cloud Platform », *International Journal of Computer Sciences and Engineering*, 2022.
[32] J. George, « Build a Realtime Data Pipeline: Scalable Application Data Analytics on Amazon Web Services (AWS) », *Journal of Emerging Technologies and Innovative Research*, 2024.
[33] J. Wang, Y. Yang, T. Wang, R. Sherratt, et J. Zhang, « Big Data Service Architecture: A Survey », *Journal of Internet Technology*, 2020.
[34] R. Goss, « Journey to a Big Data Analysis Platform: Are We There Yet? », in *2021 32nd Annual SEMI Advanced Semiconductor Manufacturing Conference (ASMC)*, IEEE Xplore, 2021.