Université de Ngaoundéré
Institut Universitaire de Technologie

Base de Données non structurées (NoSQL)

Chapitre 5 Cohérence

Pr. Dayang Paul
Maître de conférence

Année académique
2022/2023

INTRODUCTION

L'un des plus grands changements d'une base de données relationnelle centralisée à une base de données NoSQL orientée cluster est la façon dont vous pensez à la cohérence. Les bases de données relationnelles essaient de faire preuve d'une forte cohérence en évitant les diverses incohérences dont nous allons bientôt parler.

Dès que vous commencez à vous intéresser au monde NoSQL, des expressions telles que "théorème CAP" et "cohérence éventuelle" apparaissent, et dès que vous commencez à construire quelque chose, vous devez réfléchir au type de cohérence dont vous avez besoin pour votre système

INTRODUCTION

La cohérence se présente sous différentes formes, et ce seul mot couvre une myriade de façons dont les erreurs peuvent se glisser dans votre vie. Nous allons donc commencer par parler des différentes formes que peut prendre la cohérence.

Ensuite, nous verrons pourquoi vous pourriez vouloir relâcher la cohérence (et sa grande sœur, la durabilité).

1. Cohérence des mises à jour

Nous allons commencer par envisager la mise à jour d'un numéro de téléphone. Par coïncidence, Martin et Pramod consultent le site Web de l'entreprise et remarquent que le numéro de téléphone est obsolète. De façon peu vraisemblable, ils ont tous les deux un accès à la mise à jour, et ils y vont donc en même temps pour mettre à jour le numéro.

Pour rendre l'exemple intéressant, nous supposerons qu'ils le mettent à jour de manière légèrement différente, car chacun utilise un format légèrement différent. Ce problème s'appelle un conflit d'écriture : deux personnes mettent à jour le même élément de données en même temps.

1. Cohérence des mises à jour

Lorsque les écritures atteignent le serveur, celui-ci les **sérialise** et décide d'en appliquer une, puis l'autre. Supposons qu'il utilise l'ordre alphabétique et choisit d'abord la mise à jour de Martin, puis celle de Pramod. Sans aucun contrôle de la concurrence, la mise à jour de Martin serait appliquée et immédiatement remplacée par celle de Pramod. Dans ce cas, la mise à jour de Martin est **une mise à jour perdue**

Les approches visant à maintenir la cohérence face à la concurrence sont souvent décrites comme **pessimistes** ou **optimistes**.

- **Une approche pessimiste** fonctionne en empêchant les conflits de se produire ;
- **une approche optimiste** laisse les conflits se produire, mais les détecte et prend des mesures pour les résoudre.

1. Cohérence des mises à jour

Pour les conflits de mise à jour, l'**approche pessimiste** la plus courante consiste à mettre en place des verrous d'écriture, de sorte que pour modifier une valeur, il faut acquérir un verrou, et le système veille à ce qu'un seul client puisse obtenir un verrou à la fois.

Ainsi, Martin et Pramod tentent tous deux d'acquérir le verrou d'écriture, mais seul Martin (le premier) y parvient. Pramod verrait alors le résultat de l'écriture de Martin avant de décider d'effectuer sa propre mise à jour.

Une **approche optimiste** courante est une mise à jour conditionnelle où tout client qui effectue une mise à jour teste la valeur juste avant de la mettre à jour pour voir si elle a changé depuis sa dernière lecture. Dans ce cas, la mise à jour de Martin réussirait mais celle de Pramod échouerait.

1. Cohérence des mises à jour

L'erreur indiquerait à Pramod qu'il doit réexaminer la valeur et décider s'il doit tenter une nouvelle mise à jour. Les approches pessimiste et optimiste que nous venons de décrire reposent toutes deux sur une sérialisation cohérente des mises à jour.

Avec un seul serveur, c'est évident : il doit choisir l'un, puis l'autre. Mais s'il y a plus d'un serveur, comme dans le cas de la réplication poste à poste, deux nœuds peuvent appliquer les mises à jour dans un ordre différent, ce qui se traduit par une valeur différente pour le numéro de téléphone sur chaque poste

Souvent, lorsqu'on parle de concurrence dans les systèmes distribués, on parle de cohérence séquentielle, c'est-à-dire la garantie que tous les nœuds appliquent les opérations dans le même ordre.

1. Cohérence des mises à jour

Il existe une autre façon optimiste de gérer un conflit d'écriture. **Sauvegarder les deux mises à jour** et enregistrer qu'elles sont en **conflit**. Cette approche est familière à de nombreux programmeurs grâce aux systèmes de contrôle de version, en particulier les systèmes de contrôle de version distribués qui, par nature, ont souvent des commits conflictuels.

Vous devez fusionner les deux mises à jour d'une manière ou d'une autre. Vous pouvez montrer les deux valeurs à l'utilisateur et lui demander de faire le tri. C'est ce qui se passe si vous mettez à jour le même contact sur votre téléphone et votre ordinateur.

1. Cohérence des mises à jour

Les approches pessimistes dégradent souvent fortement la réactivité d'un système, au point de le rendre impropre à son usage. Ce problème est aggravé par le danger d'erreurs - la concurrence pessimiste conduit souvent à des blocages, qui sont difficiles à prévenir et à déboguer.

La réplication rend les conflits d'écriture beaucoup plus probables. Si différents nœuds possèdent différentes copies de certaines données qui peuvent être mises à jour indépendamment, vous obtiendrez des conflits, à moins que vous ne preniez des mesures spécifiques pour les éviter. L'utilisation d'un seul nœud comme cible de toutes les écritures pour certaines données facilite grandement le maintien de la cohérence des mises à jour.

2. Cohérence de lecture

Avoir un modèle de données qui maintient la cohérence des mises à jour est une chose, mais cela ne garantit pas que les lectures de ce données obtiendront toujours des réponses cohérentes à leurs demandes.

Imaginons que nous ayons une commande avec des articles et des frais d'expédition. Les frais d'expédition sont calculés en fonction des articles de la commande. Si nous ajoutons un article, nous devons donc également recalculer et mettre à jour les frais d'expédition.

Dans une base de données relationnelle, les frais d'expédition et les articles se trouvent dans des tables distinctes. Le risque d'incohérence est que Martin ajoute un poste à sa commande, que Pramod lise ensuite les postes et les frais d'expédition, puis que Martin mette à jour les frais d'expédition.

2. Cohérence de lecture

Il s'agit d'une lecture incohérente ou d'un conflit de lecture-écriture. Pramod a effectué une lecture au milieu de l'écriture de Martin.

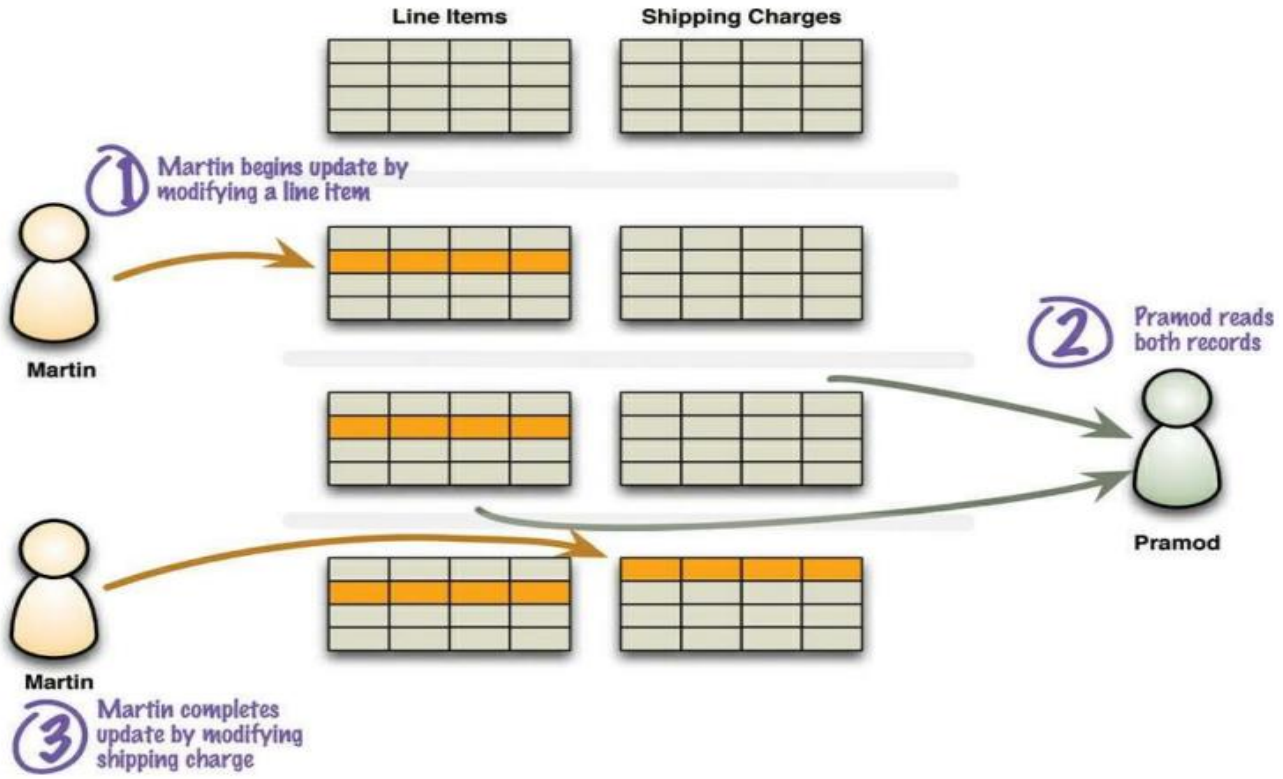


Figure 5.1. Un conflit de lecture-écriture en cohérence logique

2. Cohérence de lecture

Nous appelons ce type de cohérence la cohérence logique : il s'agit de garantir que différents éléments de données ont un sens ensemble. Pour éviter un conflit lecture-écriture logiquement incohérent, les bases de données relationnelles prennent en charge la notion de transaction.

Si Martin intègre ses deux écritures dans une transaction, le système garantit que Pramod lira les deux éléments de données avant la mise à jour ou les deux après la mise à jour.

On entend souvent dire que les bases de données NoSQL ne prennent pas en charge les transactions et ne peuvent donc pas être cohérentes. Cette affirmation est généralement fausse car elle passe sous silence de nombreux détails importants.

2. Cohérence de lecture

Notre première clarification est que toute déclaration sur l'absence de transactions ne s'applique généralement qu'à certaines bases de données NoSQL, en particulier celles qui sont orientées vers les agrégats.

En revanche, les bases de données de graphes ont tendance à supporter les transactions ACID tout comme les bases de données relationnelles.

Deuxièmement, les bases de données orientées agrégats prennent en charge les mises à jour atomiques, mais uniquement au sein d'un seul agrégat. Cela signifie que vous aurez une cohérence logique au sein d'un agrégat mais pas entre agrégats. Ainsi, dans l'exemple, vous pouvez éviter cette incohérence si la commande, les frais de livraison et les postes font tous partie d'un seul agrégat de commande.

2. Cohérence de lecture

Bien sûr, toutes les données ne peuvent pas être placées dans le même agrégat, de sorte que toute mise à jour qui affecte plusieurs agrégats laisse ouvert un moment où les clients pourraient effectuer une lecture incohérente.

Cet exemple de lecture logiquement incohérente est l'exemple classique que vous verrez dans tous les livres traitant de la programmation des bases de données. Cependant, dès que vous introduisez la réplication, vous obtenez un tout nouveau type d'incohérence. Imaginons qu'il reste une dernière chambre d'hôtel pour un événement souhaité. Le système de réservation de l'hôtel fonctionne sur de nombreux nœuds.

2. Cohérence de lecture

Martin et Cindy sont un couple qui envisage cette chambre, mais ils en discutent au téléphone car Martin est à Londres et Cindy à Boston. Pendant ce temps, Pramod, qui se trouve à Mumbai, va réserver cette dernière chambre. Cela met à jour la disponibilité de la chambre répliquée, mais la mise à jour arrive à Boston plus vite qu'à Londres.

Lorsque Martin et Cindy ouvrent leur navigateur pour voir si la chambre est disponible, Cindy la voit réservée et Martin la voit libre. Il s'agit d'une autre lecture incohérente, mais c'est une violation d'une autre forme de cohérence que nous appelons **cohérence de réplication**.

2. Cohérence de lecture

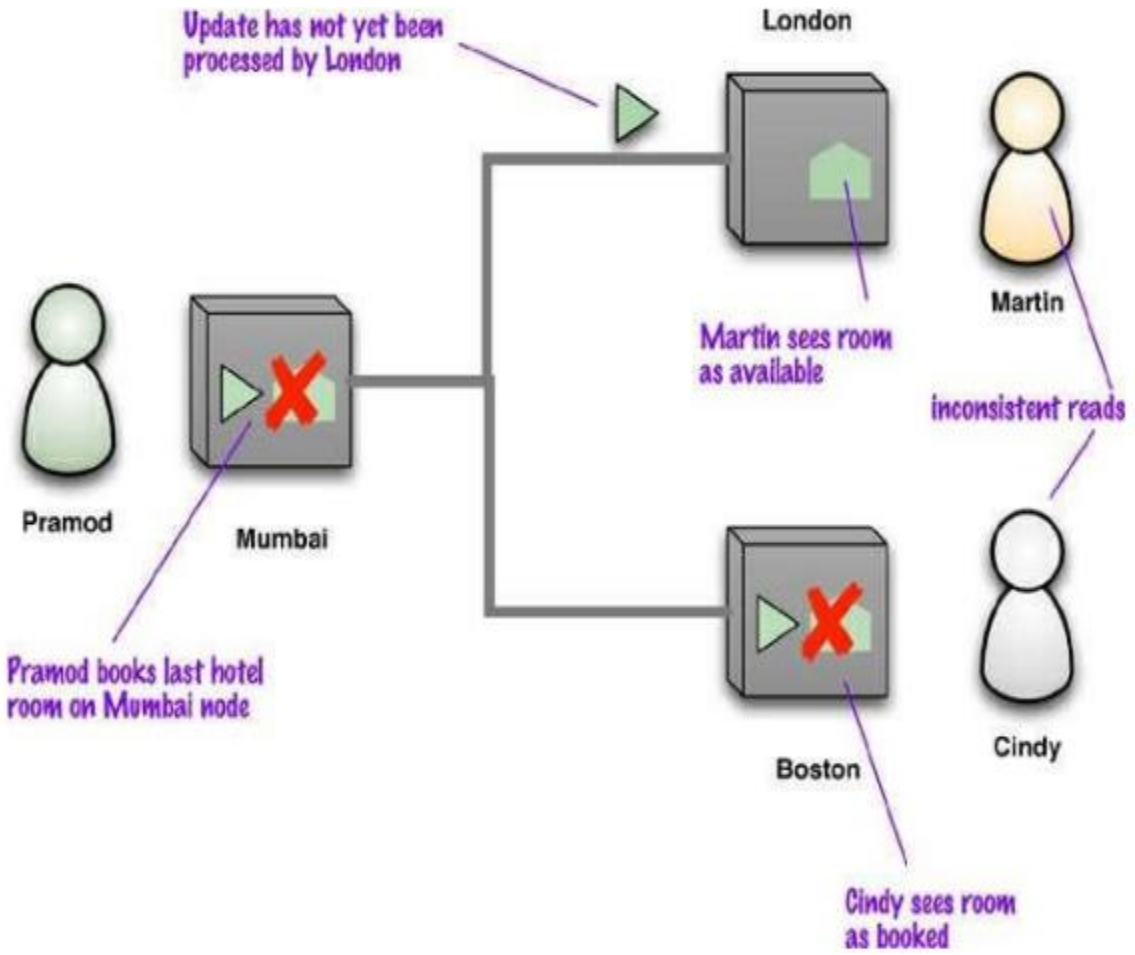


Figure 5.2. Un exemple d'incohérence de réplication

2. Cohérence de lecture

En fin de compte, bien sûr, les mises à jour se propageront complètement, et Martin verra que la chambre est entièrement réservée. Par conséquent, cette situation est généralement qualifiée d'éventuellement cohérente, ce qui signifie qu'à tout moment, les nœuds peuvent présenter des incohérences de réplication mais, s'il n'y a pas d'autres mises à jour, tous les nœuds seront finalement mis à jour avec la même valeur.

Les données qui ne sont pas à jour sont généralement qualifiées de périmées, ce qui nous rappelle qu'un cache est une autre forme de réplication, qui suit essentiellement le modèle de distribution maître-esclave.

Bien que la cohérence de la réplication soit indépendante de la cohérence logique, la réplication peut exacerber une incohérence logique en allongeant sa fenêtre d'incohérence.

2. Cohérence de lecture

Deux mises à jour différentes sur le maître peuvent être effectuées en succession rapide, laissant une fenêtre d'incohérence de quelques millisecondes. Mais des retards dans la mise en réseau peuvent faire que la même fenêtre d'incohérence dure beaucoup plus longtemps sur un esclave.

Les garanties de cohérence ne sont pas quelque chose de global pour une application. Vous pouvez généralement spécifier le niveau de cohérence que vous souhaitez pour les demandes individuelles. Cela vous permet d'utiliser une cohérence faible la plupart du temps lorsque ce n'est pas un problème, mais de demander une cohérence forte lorsque c'est le cas.

2. Cohérence de lecture

La présence d'une fenêtre d'incohérence signifie que différentes personnes verront des choses différentes au même moment. Si Martin et Cindy regardent des chambres pendant un appel transatlantique, cela peut entraîner une confusion.

Souvent, les systèmes gèrent la charge de ces sites en fonctionnant en grappe et en répartissant la charge des requêtes entrantes sur différents nœuds. C'est là que réside un danger : vous pouvez poster un message en utilisant un nœud, puis rafraîchir votre navigateur, mais le rafraîchissement va vers un nœud différent qui n'a pas encore reçu votre message - et il semble que votre message ait été perdu.

2. Cohérence de lecture

Dans des situations comme celle-ci, vous pouvez tolérer des fenêtres d'incohérence raisonnablement longues, mais vous avez besoin d'une **cohérence en lecture et en écriture**, ce qui signifie qu'une fois que vous avez effectué une mise à jour, vous avez la garantie de continuer à voir cette mise à jour. Une façon d'obtenir cela dans un système par ailleurs cohérent à terme est de fournir une **cohérence de session** : Dans la session d'un utilisateur, il y a une cohérence "**read-your-writes**".

Cela signifie que l'utilisateur peut perdre cette cohérence si sa session se termine pour une raison quelconque ou si l'utilisateur accède au même système simultanément à partir de différents ordinateurs, mais ces cas sont relativement rares.

2. Cohérence de lecture

Il existe plusieurs techniques pour assurer la cohérence des sessions. L'une des plus courantes, et souvent la plus simple, consiste à avoir une **session collante** : une session liée à un nœud (on parle également d'**affinité de session**). Une session collante vous permet de vous assurer que tant que vous conservez la cohérence lecture-écriture sur un nœud, vous l'obtiendrez également pour les sessions.

L'inconvénient est que les sessions collantes réduisent la capacité de l'équilibreur de charge à faire son travail.

Une autre approche de la cohérence des sessions consiste à utiliser des **timbres de version** et à s'assurer que chaque interaction avec le magasin de données inclut le dernier timbre de version vu par une session. Le noeud serveur doit alors s'assurer qu'il dispose des mises à jour incluant ce cachet de version avant de répondre à une demande.

2. Cohérence de lecture

Le maintien de la cohérence de la session avec les sessions collantes et la réplication maître-esclave peut s'avérer délicat si vous souhaitez lire à partir des esclaves pour améliorer les performances de lecture, mais que vous devez toujours écrire au maître. Une façon de gérer cela est d'envoyer les écritures à l'esclave, qui se charge ensuite de les transmettre au maître tout en maintenant la cohérence de la session pour son client.

Une autre approche consiste à basculer temporairement la session vers le maître lors d'une écriture, juste assez longtemps pour que les lectures soient effectuées depuis le maître jusqu'à ce que les esclaves aient rattrapé la mise à jour.

3. Cohérence relaxante

La cohérence est une bonne chose - mais, malheureusement, nous devons parfois la sacrifier. Il est toujours possible de concevoir un système pour éviter les incohérences, mais souvent impossible de le faire sans rendre insupportables des sacrifices dans d'autres caractéristiques du système.

Par conséquent, nous devons souvent sacrifier la cohérence pour autre chose. Alors que certains architectes considèrent cela comme un désastre, nous le voyons comme faisant partie des compromis inévitables dans la conception d'un système.

En outre, différents domaines ont des tolérances différentes pour l'incohérence, et nous devons tenir compte de cette tolérance lorsque nous prenons nos décisions.

3. Cohérence relaxante

Ici, notre principal outil pour faire respecter la cohérence est la transaction, et les transactions peuvent fournir de solides garanties de cohérence. Cependant, les systèmes transactionnels offrent généralement la possibilité de relâcher les niveaux d'isolation, ce qui permet aux requêtes de lire des données qui n'ont pas encore été validées.

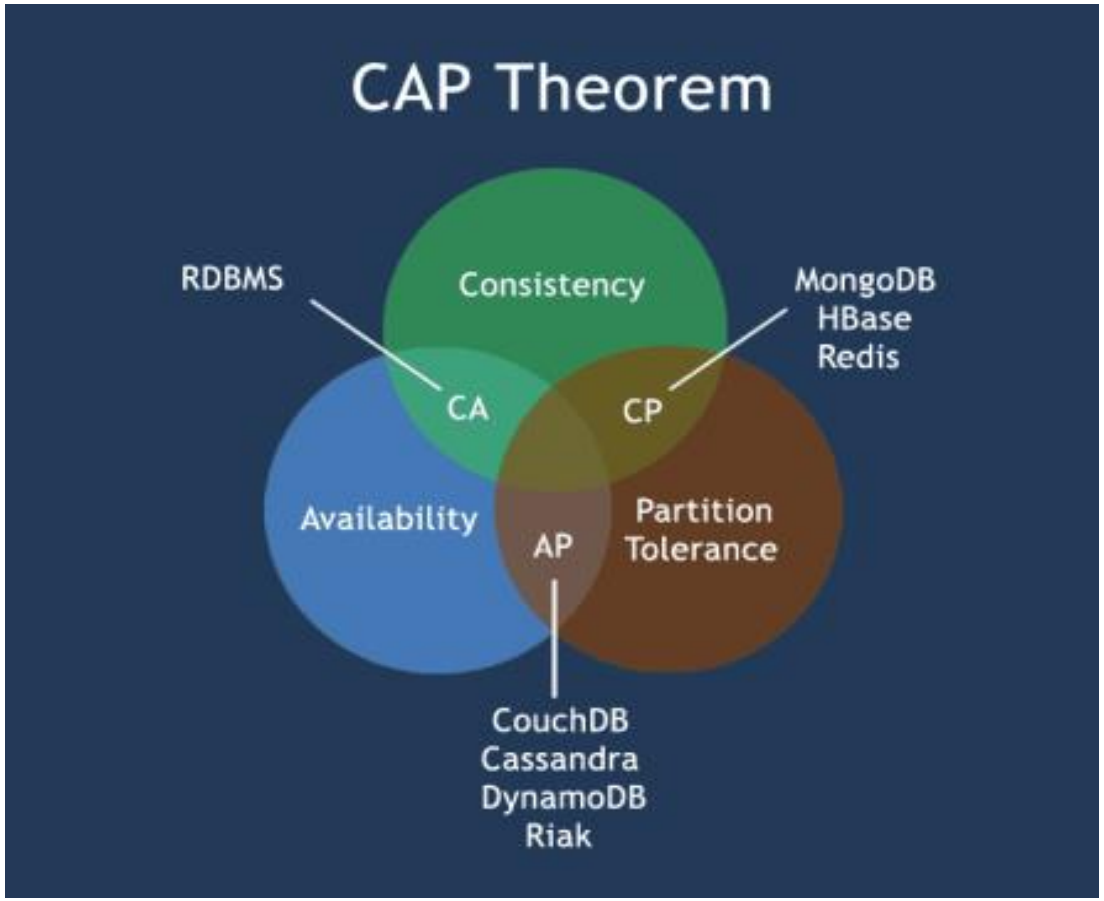
3.1. Le théorème de CAP

Dans le monde NoSQL, il est courant de faire référence au théorème CAP comme la raison pour laquelle vous pouvez avoir besoin de relâcher la cohérence.

Le théorème CAP nous dit qu'un système distribué ne peut fournir que deux des trois caractéristiques souhaitées : cohérence (Consistency), disponibilité (Availability) et tolérance au partitionnement (Partition Tolérance).

3. Cohérence relaxante

3.1. Le théorème de CAP



3.1. Le théorème de CAP

- La cohérence signifie que tous les clients voient les mêmes données en même temps, quel que soit le nœud auquel ils se connectent.

- La disponibilité signifie que tout client qui fait une requête obtient une réponse, même si un ou plusieurs nœuds sont en panne.

- La tolérance au partitionnement signifie que le cluster doit continuer à fonctionner malgré un nombre quelconque de pannes de communication entre les nœuds du système.

3.1. Le théorème de CAP

- **Base de données CP :** Une base de données CP assure la cohérence et la tolérance au partitionnement au détriment de la disponibilité. Lorsqu'un partitionnement se produit entre deux nœuds, le système doit arrêter le nœud incohérent (c'est-à-dire le rendre indisponible) jusqu'à ce que le partitionnement soit résolu.

- **Base de données AP** : Une base de données AP assure la disponibilité et la tolérance au partitionnement au détriment de la cohérence. Lorsqu'un partitionnement se produit, tous les nœuds restent disponibles, mais ceux qui se trouvent à l'autre bout du partitionnement peuvent renvoyer une version plus ancienne des données que les autres. (Lorsque le partitionnement est résolu, les bases de données AP resynchronisent généralement les nœuds pour réparer toutes les incohérences du système.)

3.1. Le théorème de CAP

- **Base de données CA :** Une base de données CA assure la cohérence et la disponibilité sur tous les nœuds. Cependant, elle ne peut pas le faire s'il existe un partitionnement entre deux nœuds du système, et elle ne peut donc pas assurer la tolérance au partitionnement.

Les défenseurs de NoSQL disent souvent qu'au lieu de suivre les propriétés **ACID** des transactions relationnelles, les systèmes NoSQL suivent les propriétés **BASE** (Basically Available, Soft state, Eventual consistency)(Brewer)

Bien que nous pensions devoir mentionner l'acronyme BASE ici, nous ne pensons pas qu'il soit très utile. L'acronyme est encore plus artificiel que ACID, et ni "**basically available**" ni "soft state" n'ont été bien définis. Nous devons également souligner que lorsque Brewer a introduit la notion de BASE, il considérait le compromis entre ACID et BASE comme un spectre, et non comme un choix binaire.

3.1. Le théorème de CAP

Propriété BASE

- **Basically Available** : quelle que soit la charge de la base de données (données/requêtes), le système garantie un taux de disponibilité de la donnée;
- **Soft-state** : La base peut changer lors des mises à jour ou lors d'ajout/suppression de serveurs. La base NoSQL n'a pas à être cohérente à tout instant;
- **Eventually consistent** : À terme, la base atteindra un état cohérent

4. Durabilité relaxante

Jusqu'à présent, nous avons parlé de cohérence, ce qui correspond en grande partie à ce que les gens entendent lorsqu'ils parlent des propriétés ACID des transactions de base de données. La clé de la cohérence consiste à sérialiser les demandes en formant des unités de travail atomiques et isolées. Mais la plupart des gens se moqueraient de la durabilité relaxante - après tout, quel est l'intérêt d'un système de stockage de données s'il peut perdre des mises à jour ?

Il s'avère qu'il existe des cas où l'on peut vouloir sacrifier une certaine durabilité pour obtenir de meilleures performances. Si une base de données peut fonctionner principalement en mémoire, appliquer des mises à jour à sa représentation en mémoire et extraire périodiquement les modifications sur le disque, elle peut être en mesure d'offrir une réactivité nettement supérieure aux demandes.

4. Durabilité relaxant

Le stockage de l'état de session de l'utilisateur est un exemple de cas où ce compromis peut être utile. Un grand site Web peut avoir de nombreux utilisateurs et conserver des informations temporaires sur ce que fait chaque utilisateur dans une sorte d'état de session. Il y a beaucoup d'activité sur cet état, créant beaucoup de demande, ce qui affecte la réactivité du site web.

Le point essentiel est que la perte des données de session n'est pas trop tragique - elle créera un certain ennui, mais peut-être moins que le ralentissement du site Web. Cela en fait un bon candidat pour les écritures non durables.

4. Durabilité relaxant

- Un autre exemple de relâchement de la durabilité est la capture de données télémétriques à partir de dispositifs physiques. Il se peut que vous préfériez capturer les données à un rythme plus rapide, au prix de manquer les dernières mises à jour si le serveur tombe en panne
- Une autre catégorie de compromis de durabilité se présente avec les données répliquées. Un échec de la durabilité de la réplication se produit lorsqu'un nœud traite une mise à jour mais échoue avant que cette mise à jour ne soit répliquée vers les autres nœuds.

Un cas simple de cette situation peut se produire si vous avez un modèle de distribution maître-esclave où les esclaves désignent automatiquement un nouveau maître en cas de défaillance du maître existant.

4. Durabilité relaxant

Si un maître tombe en panne, toutes les écritures non transmises aux répliques seront effectivement perdues. Si le maître revient en ligne, ces mises à jour seront en conflit avec les mises à jour effectuées depuis. Nous considérons cela comme un problème de durabilité, car vous pensez que votre mise à jour a réussi puisque le maître l'a reconnue, mais une défaillance du nœud maître l'a perdue.

Si vous êtes suffisamment sûr de pouvoir remettre le maître en ligne rapidement, c'est une raison de ne pas faire de basculement automatique vers un esclave. Sinon, vous pouvez améliorer la durabilité de la réplication en faisant en sorte que le maître attende que certaines répliques accusent réception de la mise à jour avant d'en accuser réception auprès du client.

5. Quorums

Lorsqu'il s'agit de choisir entre la cohérence et la durabilité, ce n'est pas une question de tout ou rien. Plus le nombre de nœuds impliqués dans une requête est élevé, plus les chances d'éviter une incohérence sont grandes. Cela conduit naturellement à la question suivante : ***Combien de nœuds doivent être impliqués pour obtenir une cohérence forte ?***

Imaginez des données répliquées sur trois nœuds. Il n'est pas nécessaire que tous les noeuds accusent réception d'une écriture pour garantir une cohérence forte ; il suffit de deux d'entre eux - une majorité. Si vous avez des écritures conflictuelles, un seul peut obtenir la majorité. C'est ce qu'on appelle le **quorum d'écriture**, exprimé par l'inégalité légèrement prétentieuse W > N/2, ce qui signifie que le nombre de noeuds participant à l'écriture (W) doit être supérieur à la moitié du nombre de noeuds impliqués dans la réplication (N). Le nombre de répliques est souvent appelé le **facteur de réplication**.

5. Quorums

De même que le quorum d'écriture, il existe la notion de quorum de lecture : Le nombre de nœuds qu'il faut contacter pour être sûr d'avoir la modification la plus récente. Le quorum de lecture est un peu plus compliqué car il dépend du nombre de nœuds qui doivent confirmer une écriture.

Considérons un facteur de réplication de 3. Si toutes les écritures doivent être confirmées par deux nœuds (W = 2), nous devons contacter au moins deux nœuds pour être sûrs d'obtenir les dernières données. Si, par contre, les écritures ne sont confirmées que par un seul noeud (W = 1), nous devons parler aux trois noeuds pour être sûrs d'avoir les dernières mises à jour.

Dans ce cas, comme nous n'avons pas de quorum d'écriture, nous pouvons avoir un conflit de mise à jour, mais en contactant suffisamment de lecteurs nous pouvons être sûrs de le détecter.

5. Quorums

Cette relation entre le nombre de nœuds que vous devez contacter pour une lecture (R), ceux confirmant une écriture (W) et le facteur de réplication (N) peut être capturée dans une inégalité : Vous pouvez avoir une lecture fortement cohérente si R + W > N.

Ces inégalités sont écrites en tenant compte d'un modèle de distribution pair-à-pair. Si vous avez une distribution maître-esclave, il suffit d'écrire vers le maître pour éviter les conflits d'écriture-écriture, et de la même manière, il suffit de lire depuis le maître pour éviter les conflits de lecture-écriture. Avec cette notation, il est courant de confondre le nombre de nœuds dans le cluster avec le facteur de réplication, mais ils sont souvent différents. Je peux avoir 100 nœuds dans mon cluster, mais n'avoir qu'un facteur de réplication de 3, la majeure partie de la distribution étant due au sharding.

5. Quorums

L'intérêt de tout cela est que vous disposez d'un éventail d'options avec lesquelles travailler et que vous pouvez choisir quelle combinaison de problèmes et d'avantages vous préférez. Certains auteurs sur le NoSQL parlent d'un simple compromis entre cohérence et disponibilité ; nous espérons que vous réalisez maintenant que c'est plus flexible - et plus compliqué - que cela.