BE SOFTWARE ARCHITECTURE WISE · CHAPITRE 7

Architecture des données

Les données survivent au code, et dans les systèmes distribués elles sont la partie la plus difficile à réussir. Ce chapitre traite de la question de savoir à qui appartient chaque donnée et pourquoi une base de données partagée couple les services, de la manière de choisir entre bases relationnelles et autres bases, de ce que les transactions et les niveaux d'isolation garantissent réellement, de la façon dont la réplication et le partitionnement fonctionnent et échouent, de CQRS et de l'event sourcing, de la mise en cache, de l'évolution des schémas sans interruption de service, ainsi que de l'analytique, du streaming et du data mesh. Il s'adresse aux développeurs et aux architectes qui connaissent déjà les styles distribués du chapitre 6 et doivent prendre des décisions solides et défendables sur les données.

bewiseapps.com/fr/software-architecture-wise/data-architecture/ · Version 2026-11-03. Information pédagogique uniquement, pas un conseil professionnel.

EN UN COUP D'ŒIL

Vous apprendrez à

- Expliquer pourquoi une base de données partagée couple les services, et attribuer la propriété des données, y compris lorsque plusieurs services écrivent les mêmes données.
- Choisir une base de données pour chaque ensemble de données selon la forme des données, les modes d'accès, les besoins de cohérence, l'échelle, l'exploitation et les compétences de l'équipe, et justifier le coût de chaque stockage supplémentaire.
- Distinguer les niveaux d'isolation et les anomalies qu'ils permettent, dont les mises à jour perdues et le write skew, et protéger un chemin lecture-modification-écriture par verrouillage, vérification de version ou transactions sérialisables.
- Expliquer la réplication leader-suiveur, multi-leader et sans leader, les anomalies causées par le retard de réplication, et les risques du failover et du split brain.
- Choisir un schéma de partitionnement et une clé de sharding à partir de volumes et de requêtes mesurés, et expliquer le hachage cohérent, les clés très sollicitées et les index secondaires.
- Décider quand CQRS, l'event sourcing et la mise en cache valent leur coût, et éviter leurs défaillances courantes.
- Planifier les modifications de schéma avec des règles de compatibilité et le patron expansion-contraction, afin qu'elles n'exigent aucune interruption de service.
- Distinguer les architectures de données opérationnelles et analytiques : entrepôts de données, data lakes, lakehouses, traitement de flux et data mesh.

Avant de commencer

Le chapitre 6, en particulier les sections sur les microservices, l'architecture pilotée par les événements, la cohérence, CAP et PACELC, et les sagas. La section du chapitre 0 sur le stockage des données couvre les bases des bases de données supposées acquises ici.

Durée

Environ 3 heures et demie de lecture ; 45 minutes pour le test.

Termes clés

système de référence · base de données par service · persistance polyglotte · niveau d'isolation · isolation par instantané · write skew · retard de réplication · hachage cohérent · CQRS · event sourcing · cache-aside · expansion-contraction · lakehouse · data mesh

Pourquoi les données sont la partie difficile

Le code peut être réécrit en quelques mois ; les données vivent pendant des décennies. Les comptes clients, les grands livres, les commandes et les dossiers médicaux survivent à chaque framework et à chaque service qui les touche, et les déplacer est lent, risqué et coûteux. C'est pourquoi les décisions de ce chapitre comptent parmi les plus importantes que prend un

architecte, au sens du chapitre 1 : elles sont difficiles à défaire, larges dans leur portée et déterminantes pour des attributs de qualité comme la disponibilité, les performances et la modifiabilité.

Le chapitre 6 a découpé les systèmes en services qui communiquent par le réseau. Découper le code est la moitié facile. Dès qu'il y a plusieurs services, quelqu'un doit décider qui peut modifier chaque donnée, comment les autres la voient, quelles garanties tiennent quand de nombreux utilisateurs agissent en même temps, et ce qui se passe quand les copies des données divergent. Le chapitre suit deux exemples récurrents : une plateforme de livraison de repas, avec son coup de feu du soir, la position des livreurs et les cartes des restaurants, et une banque en ligne, où perdre ou compter deux fois un seul paiement est inacceptable.

Posséder, stocker et protéger les données

Les quatre premières sections portent sur les données opérationnelles qui font tourner les systèmes. Propriété des données établit la règle la plus importante : chaque fait important doit avoir un seul système de référence et un seul rédacteur. Elle explique pourquoi une base de données d'intégration partagée couple les services, aussi proprement que leur code soit séparé, comment résoudre les cas où plusieurs services veulent écrire les mêmes données, comment lire des données dont vous n'êtes pas propriétaire, et comment les contrats de données rendent fiables les données partagées.

Choisir une base de données passe en revue les bases relationnelles, orientées documents, clé-valeur, orientées colonnes larges, orientées graphe, de séries temporelles, de recherche, vectorielles et SQL distribuées, et soutient que la base relationnelle reste le bon choix par défaut pour les systèmes de référence, les bases spécialisées ne méritant leur place qu'en satisfaisant une exigence que le choix par défaut ne peut pas satisfaire. Transactions et isolation va au-delà du sigle bien connu ACID pour examiner les anomalies que provoque la concurrence, comme les mises à jour perdues et le write skew, ce que les niveaux d'isolation garantissent réellement dans PostgreSQL, MySQL, Oracle et SQL Server, et quand utiliser des verrous, des vérifications de version ou des transactions sérialisables. Réplication et partitionnement explique comment les bases de données copient et répartissent les données entre plusieurs machines, ce que le retard de réplication et le failover font à l'exactitude, et comment choisir une clé de sharding à partir des chiffres.

Dériver, mettre en cache et faire évoluer les données

La seconde moitié porte sur les données dérivées des systèmes de référence, et sur le changement dans le temps. CQRS et event sourcing sépare le modèle qui accepte les changements des modèles qui répondent aux requêtes, et montre comment stocker un historique d'événements plutôt que l'état courant permet l'audit et les requêtes temporelles, au

prix d'une complexité bien réelle. Mise en cache couvre les patrons, les niveaux et les défaillances de la conservation de copies au plus près de l'endroit où elles sont nécessaires, du navigateur à la base de données, y compris les ruées sur le cache qui peuvent faire tomber un système quand une entrée très demandée expire.

Évolution des schémas part du constat que tout schéma, qu'il s'agisse de tables de base de données, de charges utiles d'API ou d'événements, changera pendant que le système fonctionne, et montre comment les règles de compatibilité et le patron expansion-contraction permettent ce changement sans interruption de service. Enfin, analytique et streaming passe de la conduite de l'activité à sa compréhension : entrepôts de données, data lakes et lakehouses, traitement de flux, et le data mesh comme manière d'organiser les données analytiques autour des domaines.

Comment les sections s'articulent

Les sections forment une chaîne. La propriété décide où chaque fait est écrit. Le choix de la base de données et ses garanties transactionnelles décident quelles règles peuvent être appliquées à moindre coût chez ce propriétaire. La réplication et le partitionnement décident comment les données du propriétaire survivent aux défaillances et grandissent au-delà d'une seule machine. Tout le reste du chapitre est une copie délibérée : modèles de lecture, caches, index de recherche, entrepôts de données et flux. Chaque copie est plus rapide ou plus pratique pour un certain usage, et chacune est périmée dans une certaine mesure. L'évolution des schémas permet au propriétaire et à toutes ses copies de changer sans se casser mutuellement.

Le chapitre suppose acquis le vocabulaire des systèmes distribués du chapitre 6, en particulier les modèles de cohérence, l'architecture pilotée par les événements et les sagas et le patron outbox. Il passe ensuite le relais au chapitre 8, qui protège ces données, et au chapitre 9, qui garde rapides et disponibles les systèmes qui les détiennent.

L'idée à retenir

Pour chaque donnée importante, vous devez pouvoir répondre à trois questions : qui est son seul rédacteur, où ses règles sont-elles appliquées, et à quel point chacune de ses copies peut-elle être périmée ? La plupart des problèmes de données des systèmes réels, comme des commandes dont l'état recule, des rapports en désaccord avec le grand livre, des doubles réservations ou des clients qui ne voient pas leurs propres modifications, viennent de ce qu'on n'a pas répondu délibérément à l'une de ces questions. Les techniques de ce chapitre sont des moyens d'y répondre correctement, et chacune d'elles est un compromis entre cohérence, disponibilité, performances, coût et effort d'exploitation du système.

SECTION 7.1 SUR 7.8 · INTERMÉDIAIRE · 26 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Propriété des données et base de données par service

AVANT DE LIRE

Cette section s'appuie sur :

- 0.5 Stocker les données
- 6.3 Microservices
- 6.4 Pilotée par les événements

Termes déjà vus qu'elle utilise : Saga, Cohérence à terme (eventual consistency), Schéma, Transaction

L'IDÉE EN BREF

Imaginez un tableau d'affichage commun dans une cuisine, où n'importe qui peut effacer et réécrire les commandes du jour. Cela fonctionne avec deux cuisiniers. Avec dix, des commandes sont écrasées, personne ne sait qui a changé quoi, et repenser le tableau suppose de demander la permission à tout le monde. Une cuisine mieux organisée donne à chaque poste son propre tableau et sa propre responsabilité : le grill écrit les commandes du grill, et les autres postes les lisent ou demandent au grill de les modifier. La propriété des données consiste à décider, pour chaque information, quelle partie d'un système est la seule autorisée à la modifier.

CE QU'IL FAUT SAVOIR

- Un système de référence est la source qui fait autorité pour une donnée ; la propriété des données signifie qu'un seul composant l'écrit, applique ses règles, contrôle son schéma et la publie pour les autres.
- Une base de données partagée (d'intégration) couple les services par le schéma, le déploiement, la sémantique et la charge à l'exécution ; Fowler (2004) et la plupart des architectes la déconseillent entre applications développées indépendamment.
- La base de données par service garde privées les données de chaque service (tables, schéma ou serveur privés), au prix des jointures, des transactions et de l'intégrité référentielle entre services, qui doivent être remplacées par des API, des événements, des sagas et des pipelines de reporting.
- La propriété conjointe peut se résoudre par une scission de table, un petit domaine de données partagé, une délégation à un seul propriétaire ou une consolidation des services, chacun avec un compromis distinct (The Hard Parts, 2021).

- Les données dont on n'est pas propriétaire peuvent être lues par des appels synchrones, une réplication de colonnes, des caches répliqués ou des événements ; toute copie est en lecture seule et peut être périmée.
- Les contrats de données précisent le schéma, la signification, la qualité, les niveaux de service et la propriété des données partagées entre équipes, et sont surtout utiles lorsqu'ils sont vérifiés automatiquement.
- Pour chaque fait important, une bonne conception sait nommer son seul rédacteur, l'endroit où ses règles sont appliquées et à quel point chaque copie peut être périmée.

Pourquoi la propriété est la première question sur les données

Le chapitre 6 a découpé les systèmes en services déployés, mis à l'échelle et modifiés indépendamment. Le code se découpe assez facilement ; les données, non. Une commande fait référence à un client, à un plat de la carte, à un restaurant et à un livreur, et chaque équipe veut lire, et souvent modifier, une partie de ces informations. La question la plus importante d'une architecture de données distribuée n'est donc pas de savoir quelle base de données acheter, mais **à qui appartient chaque donnée** : quel composant a le droit de la modifier, lequel applique ses règles, et comment tous les autres peuvent la voir.

La propriété compte parce que les données survivent au code. Les services sont réécrits tous les quelques ans ; les comptes clients, les grands livres et les dossiers médicaux sont conservés pendant des décennies. Se tromper sur la propriété produit le type coûteux de dette architecturale : deux services qui ne peuvent pas être mis en production séparément parce qu'ils partagent des tables, un champ de statut que trois équipes écrasent tour à tour, ou un rapport mensuel que personne ne peut reproduire parce que les chiffres viennent de cinq copies discordantes des mêmes données.

Cette section utilise deux exemples récurrents qui reviennent tout au long du chapitre : une plateforme de livraison de repas (clients, restaurants, livreurs, commandes) et une banque en ligne (clients, comptes, paiements, grand livre).

Système de référence et propriété des données

Un système de référence (system of record) est la source qui fait autorité pour une donnée : l'endroit dont la valeur l'emporte quand les copies divergent. Dans la banque, le grand livre central est le système de référence des soldes des comptes ; le solde affiché dans l'application mobile, celui du cache du moteur antifraude et celui du chargement de la nuit dernière dans l'entrepôt de données sont tous des copies. Dans la plateforme de livraison de repas, le service

de carte propre au restaurant est le système de référence des prix, même si l'index de recherche, le cache hors ligne de l'application mobile et les lignes de commande en détiennent tous des copies.

La propriété des données transforme cela en responsabilité. Le composant propriétaire (et l'équipe qui en a la charge) est le seul à écrire les données, à faire respecter leurs *invariants* (des règles qui doivent toujours être vraies, comme «une écriture du grand livre n'est jamais modifiée, seulement contrepassée»), à décider de leur schéma et à les publier pour les autres au moyen d'une interface qu'il contrôle. La propriété est une décision de conception, pas une propriété de la base de données : deux services peuvent résider sur le même serveur de base de données et avoir malgré tout une propriété claire, et deux services dotés de bases séparées peuvent encore se disputer les mêmes faits si la frontière entre eux est mal tracée.

Une règle utile tirée de *Software Architecture: The Hard Parts* (Neal Ford, Mark Richards, Pramod Sadalage et Zhamak Dehghani, 2021) veut que le service qui écrit dans une table en soit le propriétaire. La lecture se partage à peu de frais ; l'écriture, non, car celui qui écrit doit faire respecter les règles.

La base de données partagée, et pourquoi elle couple les services

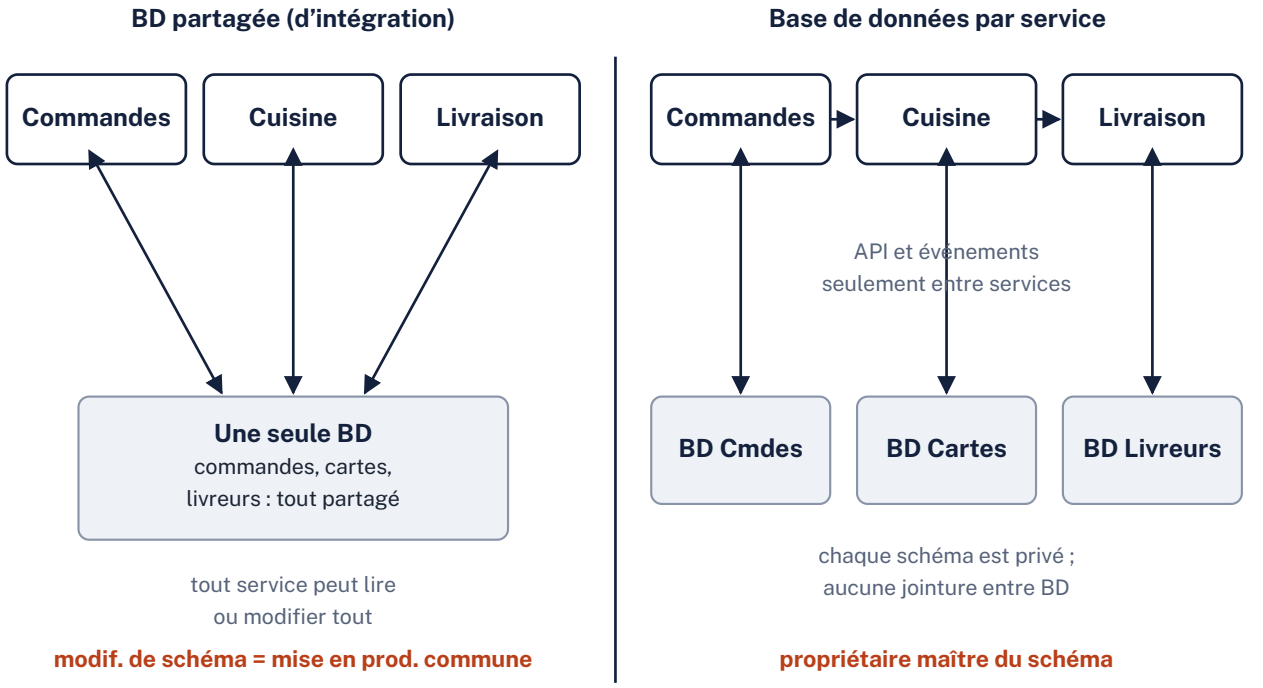
La façon la plus simple de permettre à plusieurs applications de travailler sur les mêmes données est de leur donner à toutes accès à une seule base de données. Martin Fowler a appelé cela une base de données d'intégration (2004) : une base qui constitue le point d'intégration entre des applications, par opposition à une *base de données d'application* qui ne sert qu'une seule application. Elle semble séduisante : aucune API à construire, les jointures fonctionnent sur tout, et une seule transaction peut mettre à jour ensemble les commandes et les paiements. Le verdict de Fowler, partagé par la plupart des architectes, est qu'il faut éviter les bases de données d'intégration, car le couplage qu'elles créent est profond et en grande partie invisible.

Considérez les formes de couplage qu'elle introduit. **Couplage de schéma** : chaque table et chaque colonne est de fait une API publique. Renommer `orders.state` en `orders.status` casse tout service qui la lit, et personne ne peut savoir avec certitude lesquels le font, car les requêtes SQL sont dispersées dans de nombreuses bases de code, rapports et tableurs. **Couplage de déploiement** : une modification de schéma doit être coordonnée entre toutes les équipes qui touchent la table, si bien que des services censés être mis en production indépendamment le sont ensemble. **Couplage sémantique** : le schéma doit servir tous les clients, il devient donc plus général et plus confus (Fowler note qu'il finit par tenter d'unifier des concepts qui relèvent de contextes délimités distincts), et les règles métier d'une table finissent dupliquées, de façon incohérente, dans chaque service qui l'écrit. **Couplage à**

l'exécution : la requête de reporting lente ou le traitement par lots gourmand en verrous d'un service dégrade tous les autres services, et la base de données devient un point de défaillance unique partagé et un plafond pour le nombre de connexions.

Rien de tout cela n'est fatal pour une application unique ou pour un monolithe modulaire dont les modules respectent les tables les uns des autres, ni pour l'architecture à base de services à gros grain, dans laquelle une poignée de services de domaine partagent souvent délibérément une base de données. Cela devient un problème quand les équipes attendent une déployabilité indépendante mais conservent les tables partagées, ce qui est une voie classique vers un *monolithe distribué*.

Base de données partagée ou base de données par service



Avec une base de données partagée, chaque table est une interface publique ; avec une base de données par service, chaque schéma est privé et les données ne franchissent les frontières que par des API et des événements.

Base de données par service

L'alternative, associée aux microservices, est le patron base de données par service : les données persistantes de chaque service lui sont privées, et les autres services ne peuvent y accéder que par l'API du service ou les événements qu'il publie. Le catalogue de patrons de Chris Richardson (microservices.io) note que cela ne signifie pas forcément un serveur distinct par service. Il existe trois niveaux de séparation, et la ligne importante est la première.

Trois façons de garder privées les données d'un service

Niveau	Ce qui est séparé	Solidité de la frontière	Usage typique
Tables privées par service	Chaque service a son propre ensemble de tables dans une base de données partagée, avec des identifiants distincts qui ne donnent accès qu'à ces tables.	La plus faible : repose sur les permissions et la discipline ; partage encore la capacité et les défaillances.	Au début d'une migration ; petits systèmes ; monolithes modulaires.
Schéma par service	Chaque service a son propre schéma (espace de noms) sur un serveur partagé.	Propriété et contrôle d'accès clairs ; partage encore les performances et la disponibilité du serveur.	La plupart des équipes qui débutent avec plusieurs services ; moins coûteux à exploiter.
Serveur de base de données par service	Chaque service a sa propre instance de base de données, éventuellement d'un type différent.	La plus forte : mise à l'échelle, défaillances et choix technologiques séparés.	Services dont la charge, la forme des données ou les besoins de disponibilité sont très différents.

Les avantages sont le reflet des coûts de la base partagée : les propriétaires modifient librement leurs schémas, mettent à l'échelle et règlent leurs bases indépendamment, choisissent le bon type de base de données et circonscrivent les défaillances. Les coûts sont réels et doivent être indiqués dans tout enregistrement de décision. Les jointures entre services disparaissent et doivent être remplacées par une composition d'API ou des données répliquées. Une seule transaction ACID ne peut plus couvrir à la fois les commandes et les paiements ; les changements entre services deviennent des sagas avec des étapes de compensation. L'intégrité référentielle entre services (chaque commande pointe vers un client réel) n'est plus garantie par des clés étrangères. Le reporting exige un pipeline distinct. Et il y a davantage de bases de données à sauvegarder, à mettre à jour et à superviser.

The Hard Parts présente la décision comme un équilibre de forces. Les **désintégrateurs** poussent à séparer les données : le contrôle des changements (combien de services cassent quand une table change), les limites de connexions, la scalabilité, la tolérance aux pannes, le souhait d'unités déployables indépendamment et le besoin de types de bases de données différents. Les **intégrateurs** rassemblent les données : les relations qui exigent l'intégrité référentielle, et les opérations qui ont réellement besoin d'une seule transaction atomique. Si deux tables sont toujours modifiées ensemble dans une même transaction et n'ont pas de vie indépendante, c'est un indice fort qu'elles appartiennent au même propriétaire.

Quand plusieurs services écrivent les mêmes données

Les questions de propriété deviennent difficiles quand plus d'un service modifie légitimement la même entité. *The Hard Parts* distingue trois situations. En **propriété unique**, un seul service écrit dans la table, et il en est simplement le propriétaire. En **propriété commune**, la plupart des services écrivent dans la même table (un journal d'audit ou un ensemble de paramètres de configuration, par exemple) ; la réponse habituelle est de créer un service dédié qui possède les données et de donner une API à tous les autres, souvent asynchrone pour ne pas ralentir les rédacteurs. En propriété conjointe, un petit nombre de services, généralement deux, écrivent dans la même table, le plus souvent au sein du même domaine. Le livre propose quatre techniques pour la propriété conjointe.

Résoudre la propriété conjointe

Technique	Comment ça marche	Principal compromis
Scission de table	Scinder la table pour que chaque service possède, dans sa propre table, les colonnes qu'il écrit. Dans la plateforme de livraison de repas, la ligne `orders` devient une commande possédée par Commandes et un bon de cuisine possédé par Cuisine.	Propriété nette, mais la création ou la suppression de l'entité parente doit désormais être synchronisée entre services, et toute vue d'ensemble a besoin des deux.
Domaine de données	Garder la table partagée entre les quelques services concernés, dans un petit schéma partagé qu'ils contrôlent conjointement.	Simple et rapide, et les transactions restent possibles, mais c'est une base de données partagée volontairement limitée : les services sont couplés lors des changements de schéma et la frontière entre eux s'affaiblit.
Délégation	Choisir un seul propriétaire ; l'autre service lui envoie des demandes (appels synchrones ou commandes asynchrones) au lieu d'écrire. Le propriétaire est choisi selon le domaine principal (à qui appartient ce concept ?) ou selon les caractéristiques opérationnelles (quel service a besoin des écritures les plus rapides et les plus disponibles ?).	Propriété et règles claires, en un seul endroit, au prix d'un saut réseau supplémentaire, d'un nouveau mode de défaillance et éventuellement d'une cohérence à terme pour le service qui délègue.
Consolidation des services	Fusionner en un seul service les services qui partagent les données.	Supprime le problème et rétablit les transactions locales, mais crée un service plus gros, dont le déploiement, la mise à l'échelle et l'isolation des pannes sont plus grossiers.

Aucune technique n'est gratuite, et la bonne dépend de la fréquence à laquelle les données changent, de la rigueur de leurs règles et du degré d'indépendance nécessaire aux équipes. En pratique, la délégation est la réponse la plus courante pour les entités métier dotées d'un cycle de vie (commandes, paiements, réservations), la scission de table fonctionne bien quand les colonnes écrites conjointement sont en réalité des faits différents qui se trouvaient stockés ensemble, et la consolidation est la réponse honnête quand les deux services ont été découpés trop finement dès le départ.

Lire des données dont vous n'êtes pas propriétaire

Une fois la propriété établie, tous les autres services ont encore besoin de *lire* les données. L'écran de la cuisine a besoin des instructions de livraison du client ; le moteur antifraude a besoin des paiements récents du compte. Il existe quatre grandes options, et chacune arbitre entre fraîcheur et couplage. Notez que les conceptions pilotées par les événements, traitées au chapitre 6, fournissent la troisième et la quatrième.

Façons de lire les données d'un autre service

Option	Comment ça marche	Fraîcheur	Couplage et coût
Appel synchrone	Interroger l'API du propriétaire au moment où les données sont nécessaires.	À jour, au moment de l'appel.	Couplage à l'exécution : si le propriétaire est lent ou en panne, l'appelant l'est aussi ; les latences s'additionnent le long d'une chaîne.
Réplication de colonnes	Copier les quelques colonnes nécessaires (par exemple le nom et l'adresse du restaurant) dans la propre base de données du lecteur, tenues à jour par des événements ou par capture des données modifiées.	Cohérence à terme : périmé du délai de réplication.	Lectures et jointures locales rapides ; le lecteur doit gérer les mises à jour tardives ou manquées et ne doit jamais écrire dans la copie.
Cache répliqué	Une grille de données en mémoire (comme Hazelcast ou Apache Ignite) conserve une copie des données du propriétaire dans chaque service consommateur, synchronisée par la grille.	Quasi temps réel.	Lectures très rapides ; coût mémoire, chargement au démarrage et complexité opérationnelle de la grille.
Événements (transfert d'état porté par les événements)	Le propriétaire publie des événements qui transportent les données modifiées ; les lecteurs conservent ce dont ils ont besoin.	Cohérence à terme.	Couplage temporel faible ; les lecteurs doivent gérer les doublons, l'ordre et l'évolution des schémas.

Fowler appelle la quatrième option transfert d'état porté par les événements (event-carried state transfer) : des événements comme `MenuItemPriceChanged` transportent les nouvelles valeurs, si bien que les consommateurs n'ont pas besoin de rappeler le propriétaire. Un mécanisme apparenté, la capture des données modifiées (CDC), lit le journal de la base de données du propriétaire et transforme chaque changement validé en événement, ce qui évite le *problème de la double écriture* (écrire dans la base de données et publier un événement séparément, si bien que l'un peut réussir et l'autre échouer). L'outbox transactionnelle, qui résout le même problème depuis l'intérieur de l'application, est traitée avec les sagas.

ATTENTION

Une copie est un cache, pas un second maître

Les données répliquées ne sont sûres que si elles sont traitées comme étant en lecture seule et potentiellement périmées. Dès qu'un consommateur commence à corriger sa copie (en rectifiant localement l'adresse d'un restaurant parce que la mise à jour du propriétaire tardait), il existe deux systèmes de référence, et ils vont diverger. Décidez pour chaque copie à quel point elle peut être périmée, ce qui se passe quand elle est fausse, et comment elle est reconstruite à partir du propriétaire quand elle est perdue.

SI VOUS DÉBUTEZ

Pas à pas : interroger le propriétaire, ou garder une copie

L'idée la plus difficile de cette section est la manière dont un service doit utiliser des données dont il n'est pas propriétaire. Il peut interroger le propriétaire à chaque fois, ou garder sa propre copie, et chaque choix arbitre entre fraîcheur et dépendance envers le propriétaire.

- Pour envoyer les livreurs à la bonne porte, Livraison doit savoir où se trouve chaque restaurant et comment il s'appelle. Ces données appartiennent au service Restaurants, qui en est le système de référence.
- Première option : Livraison appelle l'API de Restaurants chaque fois qu'il affecte un livreur. La réponse est toujours à jour, mais si Restaurants est lent ou en panne, les affectations de livreurs ralentissent ou se bloquent aussi, alors que Livraison lui-même n'a aucun problème.
- Deuxième option : Livraison copie uniquement ces deux colonnes dans sa propre base de données, ce que le tableau ci-dessus appelle la réplication de colonnes. Chaque fois que les informations d'un restaurant changent, Restaurants publie un événement qui transporte les nouvelles valeurs et Livraison met à jour sa copie : c'est le transfert d'état porté par les événements du chapitre 6. Les lectures sont désormais locales et rapides, et elles continuent de fonctionner quand Restaurants est en panne.
- Le prix à payer, c'est l'obsolescence. Quand un restaurant corrige son adresse, Livraison conserve l'ancienne jusqu'à l'arrivée de l'événement. Si l'événement est perdu, la copie reste fausse jusqu'à ce qu'elle soit rafraîchie à partir du propriétaire.
- La copie est donc en lecture seule. Si Livraison repère une adresse erronée, il le signale à Restaurants, qui la corrige et publie la correction ; modifier la copie créerait un second maître, comme l'explique l'avertissement ci-dessus.
- **La leçon :** interroger le propriétaire donne des données fraîches mais vous lie à lui pendant l'exécution ; une copie vous libère mais elle est en lecture seule et potentiellement périmée. Pour chaque copie, décidez à quel point elle peut être périmée et comment elle est reconstruite à partir du propriétaire.

Le reporting transversal aux services

La base de données par service complique une tâche ancienne : le rapport transversal. «Le chiffre d'affaires par restaurant et par semaine, net des remboursements, ventilé par type de livreur» exige les commandes, les paiements, les restaurants et les livraisons. Le produire en appelant quatre API page par page est lent et fait peser la charge du reporting sur les services opérationnels. Interroger directement leurs bases de données réintroduit par la petite porte le couplage de la base partagée.

La réponse habituelle consiste à séparer le reporting des opérations. Fowler a décrit une *base de données de reporting* : une base distincte, conçue pour les requêtes et alimentée par les systèmes opérationnels, que les utilisateurs du reporting peuvent lire librement. Aujourd'hui, il s'agit généralement d'un entrepôt de données ou d'un lakehouse alimenté par CDC ou par des événements, traité dans analytique et streaming. Pour de petites vues en quasi temps réel au sein du produit (le tableau de bord des commandes du jour d'un restaurant), un service peut construire son propre modèle de lecture à partir d'événements, l'idée à la base de CQRS. Dans les deux cas, les pipelines qui alimentent le reporting dépendent des structures de données des propriétaires, et c'est pourquoi ils ont besoin de contrats.

Contrats de données

Un contrat de données est un accord explicite et versionné entre le propriétaire d'un jeu de données et ses consommateurs. Il précise le schéma (champs, types, lesquels peuvent être nuls), la signification de chaque champ (`amount` est-il en pence ou en livres, TVA comprise ou non ?), les attentes de qualité (aucun identifiant de commande en double, au plus 0,1 % de lignes sans restaurant), les niveaux de service (fraîcheur des données, durée de conservation), le propriétaire et la manière dont les changements sont annoncés. Il applique l'idée d'un contrat d'API aux données qui circulent entre équipes au moyen d'événements, de flux CDC et de tables d'entrepôt de données.

Les contrats sont surtout utiles lorsqu'ils sont lisibles par une machine et vérifiés automatiquement : un pipeline qui rejette une modification de schéma qui casserait des consommateurs enregistrés, ou un test qui échoue quand une règle de qualité est enfreinte. L'Open Data Contract Standard (ODCS), maintenu par le projet Bitol au sein de la Linux Foundation AI & Data, définit un format YAML couvrant le schéma, les règles de qualité, les niveaux de service, les rôles et les canaux d'assistance. Les contrats de données ne sont pas gratuits : quelqu'un doit les rédiger et les maintenir, et des contrats trop stricts ralentissent le changement. *The Hard Parts* oppose les contrats stricts (précis, vérifiables, fortement couplants) aux contrats souples (plus faciles à faire évoluer, mais demandant davantage de tests), et propose comme voie médiane les contrats pilotés par le consommateur, dans lesquels les consommateurs indiquent ce qu'ils utilisent réellement. La manière de modifier un schéma sans casser les consommateurs fait l'objet de l'évolution des schémas.

EXEMPLE COMMENTÉ

Exemple commenté : à qui appartient le statut des commandes dans la plateforme de livraison de repas ?

La plateforme compte trois services qui s'intéressent à l'avancement d'une commande : Commandes (prend les commandes et les fait payer), Restaurants (la tablette de la cuisine : accepte les commandes, signale que les plats sont prêts) et Livraison (affecte les livreurs, suit la récupération et la remise). Aujourd'hui, les trois écrivent la colonne `status` d'une table `orders` partagée. Des clients ont vu des commandes repasser de «Récupérée» à «En préparation», et personne ne sait dire où se trouvent les règles d'annulation.

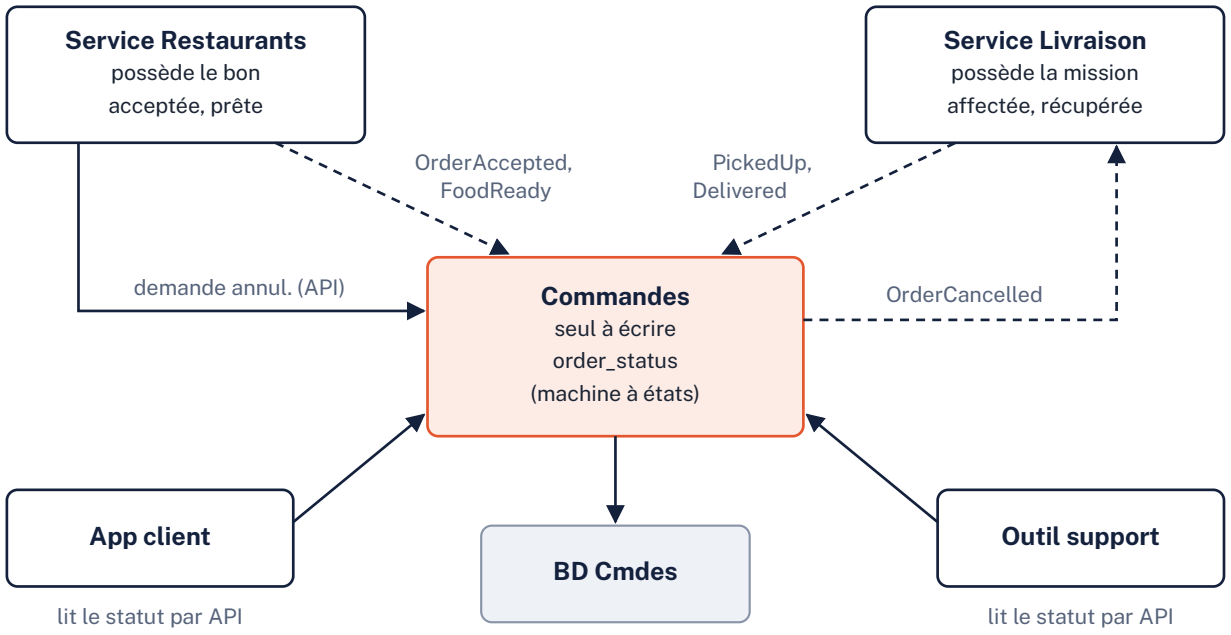
Diagnostic	Propriété conjointe d'une colonne par trois rédacteurs. Chacun écrit ce qu'il sait, la dernière écriture l'emporte, et des messages tardifs font reculer le statut. La règle «une commande ne peut être annulée qu'avant que la cuisine commence» est vérifiée dans deux services, différemment.
Séparer les faits	Il y a en réalité trois faits : l'état du bon de cuisine (possédé par Restaurants), l'état de la mission de livraison (possédé par Livraison) et le statut de la commande présenté au client, qui combine les deux et porte les règles métier (annulation, remboursements, dédommagement).
Option : scission de table	Donner à chaque service sa propre colonne de statut et laisser les lecteurs les combiner. Rejetée : chaque lecteur (app, outil de support, notifications) réimplémenterait la combinaison, et la règle d'annulation n'aurait toujours pas de place.
Option : consolidation des services	Fusionner les trois en un seul service. Rejetée : trois équipes, des charges très différentes (les mises à jour de position des livreurs arrivent toutes les quelques secondes) et des cycles de mise en production différents.
Décision : délégation	Commandes est le seul rédacteur de `order_status`. Restaurants et Livraison possèdent leurs propres faits et publient des événements (`OrderAccepted`, `FoodReady`, `PickedUp`, `Delivered`). Commandes les consomme et applique une machine à états. Une annulation depuis la tablette du restaurant est une demande adressée à l'API de Commandes, qui l'accepte ou la refuse.
Machine à états	PLACED, ACCEPTED, PREPARING, READY, PICKED_UP, DELIVERED, avec CANCELLED autorisé seulement depuis PLACED ou ACCEPTED. Les transitions ne vont que vers l'avant : un `FoodReady` tardif qui arrive après `PickedUp` est enregistré mais ne change pas le statut ; les doublons sont ignorés.

Conséquences	Un seul endroit pour les règles ; plus de retours en arrière ; chaque équipe met en production indépendamment. Coûts : le statut est en retard sur la cuisine du délai des événements (généralement bien moins d'une seconde), Commandes doit être disponible pour accepter les annulations, et les événements deviennent un contrat qui doit être versionné.

Le geste décisif n'a pas été de choisir une technique dans le tableau, mais de remarquer qu'une seule colonne cachait trois faits différents, avec trois propriétaires naturels.

Consignez la décision dans un enregistrement de décision d'architecture (ADR), en incluant les options rejetées ; la prochaine équipe qui demandera «pourquoi ne pas simplement mettre à jour le statut depuis l'app du livreur ?» en aura besoin.

Le statut des commandes après la décision



Chaque fait a un seul rédacteur ; les autres lisent ou demandent

Chaque fait a un seul rédacteur : Restaurants et Livraison publient ce qu'ils possèdent, et le service Commandes est le seul à transformer ces événements en statut présenté au client.

Erreurs courantes

Découper les services mais garder les tables partagées. Les équipes subissent le coût opérationnel de la distribution sans l'indépendance qu'elle devait apporter. Si des services ne peuvent pas être déployés sans coordonner une modification de schéma, ils ne sont pas indépendants. **Les services d'entité.** Un `CustomerService` qui n'est rien d'autre que des opérations de création, lecture, mise à jour et suppression sur une table, appelé par tous les

autres services, est une base de données partagée avec de la latence en plus ; la propriété doit suivre les capacités métier et leurs règles, pas les noms communs. **Des copies qui deviennent discrètement des maîtres.** Voir l'avertissement ci-dessus. **Ignorer le reporting jusqu'à la fin.** Le premier rapport financier après le découpage révèle souvent que personne n'a prévu comment les données seraient rassemblées. **Traiter la propriété comme une question purement technique.** La propriété est aussi organisationnelle : quelqu'un doit répondre de la qualité des données et, selon la loi de Conway, les frontières de données qui traversent les frontières d'équipes ont tendance à ne pas durer.

Un bon test de toute conception de données consiste à poser, pour chaque fait important, trois questions : qui est le seul rédacteur, où ses règles sont-elles appliquées, et à quel point chaque copie peut-elle être périmée ? Si l'une des réponses est «ça dépend» ou «plusieurs services», la conception n'est pas terminée.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Deux services mettent chacun à jour la colonne `status` de la même table. De quel type de propriété s'agit-il ? Citez deux façons de la résoudre.**

Propriété conjointe. On peut la résoudre en scindant la table, en déléguant les écritures à un seul propriétaire, en gardant un petit domaine de données partagé ou en consolidant les services.

▼ **Pourquoi une base de données partagée compromet-elle le déploiement indépendant ?**

Chaque table est de fait une interface publique : une modification de schéma doit donc être coordonnée avec chaque service qui la lit ou l'écrit, ce qui impose des mises en production conjointes.

▼ **Quelle est la différence entre la réplication de colonnes et un appel synchrone pour lire les données d'un autre service ?**

La réplication conserve une copie locale, cohérente à terme, rapide et disponible même si le propriétaire est en panne ; un appel synchrone renvoie des données à jour mais lie la disponibilité et la latence de l'appelant à celles du propriétaire.

▼ **Qu'apporte un contrat de données au-delà d'un schéma ?**

La signification des champs, les attentes de qualité, les niveaux de fraîcheur et de conservation, la propriété, et la manière dont les changements sont annoncés et versionnés.

▼ **À vous : Réservation, Paiements et la Billetterie mettent tous à jour la colonne status d'une table partagée tickets, et des billets repassent parfois de Refunded à Paid. Qui devrait posséder status, et comment les autres devraient-ils le modifier ou le lire ?**

Désignez un seul auteur des écritures. Réservation doit être propriétaire du statut du billet et faire respecter les changements autorisés en un seul endroit, y compris une règle selon laquelle Refunded ne peut pas devenir Paid. Paiements et la Billetterie ne l'écrivent pas ; ils envoient une commande ou publient un événement, tel que RefundCompleted, et Réservation décide. Les autres lisent via l'API de Réservation ou conservent des copies en lecture seule, en acceptant une certaine péremption. Le détail propre à chaque équipe, comme l'état du paiement, reste dans son propre magasin. Quitter la table partagée vient en dernier.

SOURCES

- Martin Fowler: IntegrationDatabase (2004): https://martinfowler.com/bliki/IntegrationDatabase.html
- Martin Fowler: ReportingDatabase: https://martinfowler.com/bliki/ReportingDatabase.html
- Chris Richardson, microservices.io: Pattern: Database per service: https://microservices.io/patterns/data/database-per-service.html
- Chris Richardson, microservices.io: Pattern: Shared database: https://microservices.io/patterns/data/shared-database.html
- Bitol (Linux Foundation AI & Data): Open Data Contract Standard: https://bitol-io.github.io/open-data-contract-standard/latest/

Choisir une base de données : relationnelle, documents, clé-valeur, colonnes, graphe et séries temporelles

AVANT DE LIRE

Cette section s'appuie sur :

- 0.5 Stocker les données
- 6.6 CAP et cohérence
- 7.1 Propriété des données

Termes déjà vus qu'elle utilise : Système de référence (system of record), Transaction, SQL (Structured Query Language), Capture des données modifiées (CDC), Enregistrement de décision d'architecture (ADR)

L'IDÉE EN BREF

Choisir une base de données, c'est comme choisir des rangements pour une maison. Un classeur à dossiers étiquetés (une base de données relationnelle) convient aux papiers que vous devez recouper. Une penderie de tenues complètes (une base orientée documents) convient aux choses que vous sortez toujours ensemble. Une rangée de casiers numérotés (une base clé-valeur) est le plus rapide quand vous connaissez le numéro. Un tableau de liège avec des fils tendus entre des punaises (un graphe) montre qui connaît qui. La plupart des maisons ont besoin d'un classeur et d'un ou deux meubles spécialisés, pas d'un exemplaire de chaque, car chaque meuble supplémentaire demande de la place, de l'entretien et quelqu'un qui sait s'en servir.

CE QU'IL FAUT SAVOIR

- Les bases de données relationnelles restent le choix par défaut pour les données métier, car elles offrent des transactions, des contraintes, des requêtes flexibles, des outils matures et des compétences largement disponibles, et gèrent désormais aussi les données JSON, textuelles, géospatiales et vectorielles à une échelle modérée.
- Les bases orientées documents, clé-valeur, colonnes larges, graphe, séries temporelles, recherche et vecteurs rendent chacune certains schémas d'accès peu coûteux et d'autres coûteux ; elles excellent comme stockages spécialisés ou dérivés.
- Le SQL distribué (Spanner, CockroachDB, YugabyteDB, TiDB) conserve SQL et les transactions tout en s'étendant sur plusieurs nœuds et régions, au prix de la latence d'écriture, de l'argent et de la complexité opérationnelle.

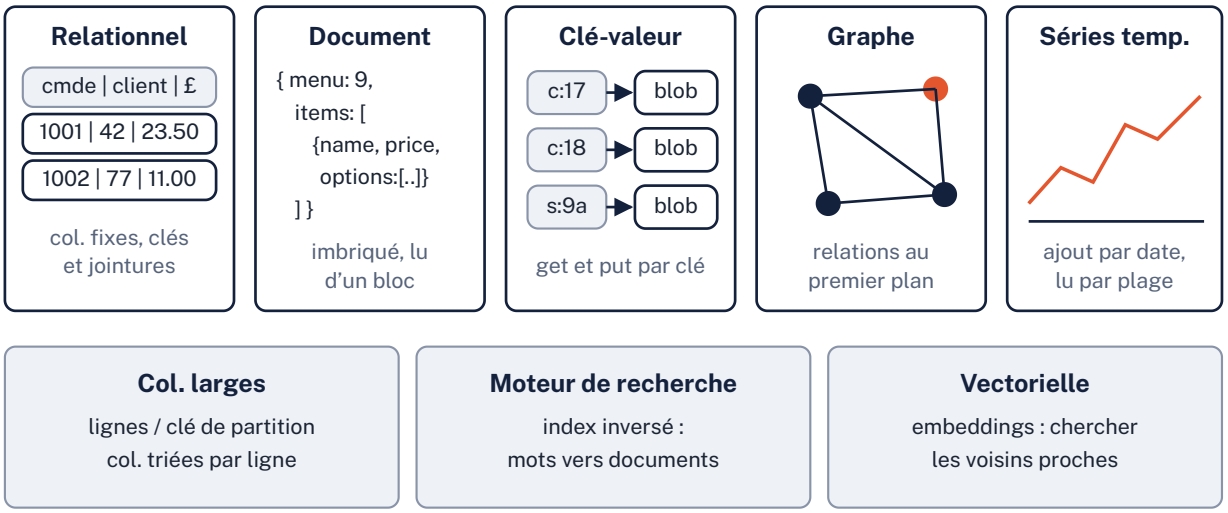
- Choisissez par jeu de données en vous fondant sur des preuves : forme des données, schémas d'accès, besoins de cohérence, échelle mesurée, charge d'exploitation et compétences de l'équipe ; notez d'abord les requêtes clés et les volumes.
- Séparez les systèmes de référence, qui exigent la technologie la plus prudente, des stockages dérivés comme les index de recherche et les caches, qui peuvent être spécialisés parce qu'ils peuvent être reconstruits.
- La persistance polyglotte permet à chaque jeu de données d'utiliser un stockage adapté, mais chaque technologie supplémentaire ajoute un coût opérationnel et cognitif durable ; l'essai de McKinley « Choose Boring Technology » (2015) plaide pour dépenser l'innovation avec parcimonie.

Pourquoi ce choix mérite de l'attention

Une base de données est l'un des composants les plus difficiles à remplacer. Les applications sont réécrites autour de leurs données, et non l'inverse ; migrer des téraoctets de données vivantes pendant que l'activité continue peut prendre un an. Le choix du stockage est donc une porte à sens unique qui mérite une décision explicite, alors que beaucoup d'équipes le font par habitude ou par effet de mode.

Le chapitre 0 a présenté les bases de données relationnelles et les quatre familles NoSQL. Cette section va plus loin : ce que chaque famille fait vraiment bien, les catégories plus récentes (séries temporelles, recherche, vectorielle et SQL distribué), comment choisir sur la base de preuves plutôt que d'enthousiasme, et ce que coûte l'exploitation de plusieurs types à la fois. La réponse est rarement une seule base de données pour tout, et presque jamais une base différente pour chaque service.

Les modèles de données en un coup d'œil



Chaque famille organise les données différemment, et cette forme décide quelles requêtes sont peu coûteuses et lesquelles sont impossibles.

Le relationnel : toujours le choix par défaut, pour de bonnes raisons

Une base de données relationnelle stocke les données dans des tables dotées de types, de clés et de contraintes déclarés, et répond à des requêtes SQL déclaratives, une idée issue de l'article d'Edgar Codd de 1970. Cinquante ans plus tard, elle reste le choix par défaut raisonnable pour les données métier, pour quatre raisons. D'abord, **les transactions et les contraintes** : la base de données garantit elle-même qu'un paiement et ses écritures au grand livre sont validés ensemble et qu'aucune commande ne pointe vers un client inexistant, si bien que ces règles n'ont pas à être réimplémentées, imparfaitement, dans chaque service. Ensuite, **la flexibilité des requêtes** : vous n'avez pas besoin de connaître toutes les questions futures quand vous concevez le schéma ; un nouveau rapport est une nouvelle requête, pas une nouvelle table. Troisièmement, **la maturité** : des décennies d'outils de sauvegarde, de réplication, de supervision, de migration et de diagnostic des performances, et des ingénieurs qui la connaissent déjà. Quatrièmement, **l'étendue** : les systèmes relationnels modernes stockent et indexent du JSON (le type `jsonb` de PostgreSQL), prennent en charge la recherche en texte intégral, les données géospatiales (PostGIS), le partitionnement temporel et la similarité vectorielle (l'extension pgvector), si bien qu'une seule base bien exploitée peut couvrir de nombreux besoins à une échelle modérée.

Les vraies limites sont précises. Les écritures sur un primaire unique sont bornées par une seule machine ; les très grandes tables exigent un partitionnement soigneux ; les modifications de schéma sur des tables énormes demandent de la technique (évolution des schémas) ; et certains schémas d'accès, comme les parcours de graphe profonds ou la recherche textuelle

classée par pertinence, sont bien mieux servis par des moteurs spécialisés. Un seul serveur moderne doté de disques SSD rapides peut traiter des milliers de transactions par seconde et des téraoctets de données, ce qui couvre les besoins de la plupart des organisations.

Choisir des identifiants : séquences, UUIDv7 et ULID

Les clés sont difficiles à changer plus tard. Une **séquence** est compacte et ordonnée, mais exige un allocateur unique, et le numéro de commande 1042 révèle votre volume à des tiers. Un **UUIDv4** aléatoire peut être créé n'importe où, mais la RFC 9562 met en garde contre sa mauvaise localité d'index : les insertions tombent à des endroits aléatoires d'un arbre B, et l'effet « peut être spectaculaire ». **UUIDv7** place d'abord un horodatage de 48 bits en millisecondes, puis des bits aléatoires, de sorte que les nouvelles valeurs restent proches les unes des autres ; PostgreSQL 18 inclut `uuidv7()`, et la RFC conseille de stocker les UUID en binaire. **ULID** est similaire : 48 bits de temps, 80 d'aléa et 26 caractères de texte. Les identifiants ordonnés dans le temps révèlent quand une ligne a été créée et peuvent créer une plage chaude avec le partitionnement par plage de clés. La RFC avertit aussi de ne pas supposer que les UUID sont difficiles à deviner : l'autorisation ne doit donc pas reposer sur eux.

Les familles non relationnelles

Une base de données orientée documents stocke chaque enregistrement sous forme de document autonome, généralement en JSON ou dans une forme binaire de celui-ci. Elle convient aux *agrégats* au sens de la conception pilotée par le domaine : des choses lues et écrites d'un bloc, comme la carte d'un restaurant avec ses rubriques, ses plats et ses groupes d'options. Lire un document, c'est une seule recherche, sans jointure. Le compromis apparaît quand les mêmes données doivent être interrogées sous une autre forme (tous les plats contenant des arachides, sur toutes les cartes) ou modifiées dans de nombreux documents à la fois ; les jointures et les transactions multidocuments existent dans les produits modernes, mais elles sont plus coûteuses et moins naturelles qu'en SQL.

Une base clé-valeur est un dictionnaire distribué : on range une valeur sous une clé, on la récupère. Comme la base ne regarde jamais à l'intérieur des valeurs, elle peut être extrêmement rapide et facile à partitionner. Redis (sous licences source-available depuis 2024, avec une option open source AGPL depuis Redis 8) et Valkey, son fork sous licence BSD, conservent les données en mémoire avec une persistance optionnelle, et ajoutent des structures de données (ensembles triés, flux, index géospatiaux) et l'expiration ; Amazon DynamoDB est une base clé-valeur et documents managée et partitionnée. Elles conviennent aux sessions, aux paniers, aux compteurs de limitation de débit, aux feature flags, aux caches et à la dernière position de chaque livreur.

Une base orientée colonnes larges (wide-column store) (Bigtable, Apache Cassandra, ScyllaDB, HBase) regroupe les lignes par clé de partition et les stocke triées par une clé de clustering au sein de chaque partition. Elle est conçue pour des volumes d'écriture massifs répartis sur de nombreux nœuds et centres de données. La discipline qu'elle exige est la *modélisation orientée requêtes* : vous concevez une table par requête, en dupliquant librement les données, car les requêtes qui ne partent pas de la clé de partition sont coûteuses, voire impossibles. Cassandra descend en partie de l'article Dynamo d'Amazon de 2007, dont les idées de réplication apparaissent dans deux sections plus loin.

Une base de données orientée graphe stocke des nœuds et des relations comme des enregistrements à part entière, et répond à des requêtes de parcours comme «les clients à trois sauts au plus de ce réseau de fraude» sans les jointures en cascade qui rendent ces requêtes lentes en SQL. Cypher, de Neo4j, est le langage de requête le plus connu, et l'ISO a publié en 2024 un langage de requête de graphe normalisé, GQL (ISO/IEC 39075). Les graphes excellent pour la détection de fraude, les recommandations et l'analyse de réseaux, et font généralement un mauvais stockage principal pour les données métier transactionnelles.

Séries temporelles, recherche et bases vectorielles

Une base de données de séries temporelles est optimisée pour des mesures horodatées qui arrivent dans l'ordre et sont rarement modifiées : métriques, relevés de capteurs, prix, historique de positions. Elle compresse fortement, partitionne par période, sous-échantillonne les données anciennes (garder chaque point pendant une semaine, des moyennes par minute pendant un an) et supprime à peu de frais les données expirées en éliminant des partitions temporelles entières. Parmi les exemples figurent InfluxDB, Prometheus (pour la supervision) et TimescaleDB, qui est une extension de PostgreSQL et montre qu'une base relationnelle peut souvent s'étendre pour convenir.

Un moteur de recherche comme Elasticsearch, OpenSearch ou Apache Solr (tous construits sur la bibliothèque Apache Lucene) maintient un *index inversé* des mots vers les documents qui les contiennent, avec une analyse du texte (racinisation, synonymes, tolérance aux fautes de frappe) et un classement par pertinence. Il répond à «ramen vegan près de chez moi» en quelques millisecondes. Il ne devrait presque jamais être un système de référence : les index sont rafraîchis en quasi temps réel plutôt que de façon transactionnelle, et la conception habituelle alimente le moteur de recherche à partir de la base de données propriétaire et reconstruit l'index à partir d'elle en cas de besoin.

Une base de données vectorielle stocke des *embeddings*, des listes de nombres qui représentent le sens de textes, d'images ou de produits, et trouve les éléments les plus proches d'un vecteur de requête. Une recherche exacte compare la requête à chaque vecteur ; ces bases utilisent donc des index de recherche approximative des plus proches voisins, comme les

graphes HNSW, qui sacrifient un peu de rappel au profit de la vitesse. Elles alimentent la recherche sémantique, les recommandations et la récupération d'informations pour les grands modèles de langage, traitée dans la génération augmentée par récupération. Des produits dédiés existent (Milvus, Qdrant, Pinecone, Weaviate), mais pour des millions plutôt que des milliards de vecteurs, une extension d'une base existante comme pgvector suffit souvent.

SQL distribué

Les bases de données SQL distribuées, parfois appelées *NewSQL* (un terme forgé par l'analyste Matthew Aslett en 2011), conservent le modèle relationnel, SQL et les transactions ACID, mais partitionnent et répliquent automatiquement les données sur de nombreux nœuds, souvent dans plusieurs régions. Google Spanner, décrit dans un article de l'OSDI de 2012, utilise des horloges étroitement synchronisées (son API TrueTime, adossée au GPS et à des horloges atomiques) pour offrir des transactions cohérentes à l'échelle mondiale. CockroachDB, YugabyteDB et TiDB proposent des modèles similaires sur du matériel ordinaire ; CockroachDB utilise par défaut l'isolation sérialisable.

Le prix à payer tient à la physique et à l'exploitation. Une transaction qui doit être approuvée par des répliques dans plusieurs régions attend un aller-retour de consensus : les écritures d'une seule ligne prennent donc de quelques millisecondes à des dizaines ou des centaines de millisecondes, au lieu de bien moins d'une milliseconde. La planification des requêtes entre partitions est plus subtile, les coûts sont plus élevés, et moins d'ingénieurs connaissent les modes de défaillance. Le SQL distribué est le bon outil quand vous avez réellement besoin de transactions relationnelles au-delà de la capacité d'un seul serveur ou entre régions (un grand livre mondial de paiements, une banque multirégion), et un outil coûteux quand un seul primaire PostgreSQL avec des répliques suffirait.

Comparaison des familles de bases de données

Famille	Modèle de données et requêtes	Points forts	Faiblesses	Exemples
Relationnel	Tables, clés, SQL avec jointures et agrégats	Transactions et contraintes, requêtes ad hoc flexibles, outils matures, largement connues	Limite de mise à l'échelle verticale pour les écritures sur un nœud ; schéma rigide qui exige des migrations	PostgreSQL, MySQL, SQL Server, Oracle
Document	Documents de type JSON, requêtes sur les champs et les valeurs imbriquées	Adaptation naturelle aux agrégats lus d'un bloc ; structure flexible	Jointures et transactions entre documents plus faibles ou plus coûteuses ; schéma imposé dans le code	MongoDB, Couchbase, Firestore
Clé-valeur	Valeur opaque par clé ; get, put, delete	Très rapides, simples, faciles à partitionner	Aucune requête portant sur les valeurs ; modélisation à la charge de l'application	Redis, Valkey, DynamoDB, etcd
Col. larges	Lignes regroupées par clé de partition, triées par clé de clustering	Débit d'écriture énorme, mise à l'échelle horizontale linéaire, multi-centres de données	Tables construites par requête ; jointures et requêtes ad hoc limitées	Apache Cassandra, ScyllaDB, HBase, Bigtable
Graphe	Nœuds et arêtes dotés de propriétés ; requêtes de parcours	Requêtes sur des relations à plusieurs sauts	Plus difficiles à partitionner ; écosystème plus restreint	Neo4j, Amazon Neptune
Séries temp.	Points horodatés, étiquettes ; requêtes par plage et agrégations	Compression, rétention, sous-échantillonnage	Peu adaptées aux mises à jour générales et aux relations	InfluxDB, TimescaleDB, Prometheus

Famille	Modèle de données et requêtes	Points forts	Faiblesses	Exemples
Moteur de recherche	Index inversé sur des documents ; requêtes textuelles classées par pertinence	Recherche en texte intégral, correspondance approximative, facettes	Pas un système de référence ; rafraîchissement en quasi temps réel	Elasticsearch, OpenSearch, Apache Solr
Vectorielle	Embeddings ; recherche approximative des plus proches voisins	Similarité sémantique pour la récupération d'informations et les recommandations	Résultats approximatifs ; coût de construction de l'index ; outils encore jeunes	pgvector, Milvus, Qdrant, Pinecone
SQL distribué	Modèle relationnel, SQL, partitionné et répliqué	Transactions et SQL avec mise à l'échelle horizontale et résistance à la perte d'une région	Latence d'écriture, coût et complexité opérationnelle plus élevés	Google Spanner, CockroachDB, YugabyteDB, TiDB

Choisir : les critères qui comptent

Choisissez par jeu de données, pas par système. Pour chaque jeu de données important, rassemblez des preuves sur six critères et consignez la décision dans un ADR.

Forme des données. Sont-elles naturellement tabulaires avec de nombreuses relations, un agrégat autonome, un réseau, une série temporelle, du texte libre ou des vecteurs ? **Schémas d'accès.** Quelles requêtes s'exécutent, à quelle fréquence, avec quels objectifs de latence, et combien sont ad hoc ? Lectures ou écritures majoritaires ? Toujours par clé, ou selon de nombreux attributs ? **Besoins de cohérence.** Les données exigent-elles des transactions sur plusieurs éléments et des contraintes (soldes, stocks, réservations), ou quelques secondes de retard sont-elles sans conséquence (mentions « j'aime », recommandations) ? Voir les modèles de cohérence. **Échelle.** Volume de données actuel et prévu, écritures par seconde et croissance, mesurés et non devinés ; une base qui s'étend jusqu'aux pétaoctets n'apporte rien pour 50 gigaoctets. **Exploitation.** Qui va l'exploiter : sauvegardes, mises à niveau, failover, supervision, correctifs de sécurité ? Existe-t-il un service managé dans votre cloud, et combien coûte-t-il à vos volumes ? **Compétences de l'équipe.** Que connaît déjà l'équipe assez bien pour déboguer à 3 heures du matin ? Une technologie mal connue comporte des modes de défaillance inconnus.

Deux habitudes améliorent le choix. Notez les cinq ou dix requêtes les plus importantes avec leurs latences cibles *avant* de choisir, et testez le candidat avec des volumes de données réalistes ; bien des déceptions viennent d'une base évaluée sur un ordinateur portable avec mille lignes. Et séparez le *système de référence* des *stockages dérivés* : le système de référence exige les garanties les plus fortes et la technologie la plus prudente, tandis que les stockages dérivés (index de recherche, caches, modèles de lecture) peuvent être spécialisés précisément parce qu'ils peuvent être reconstruits.

GUIDE DE DÉCISION

Guide de décision : quel type de stockage pour ces données ?

Posez ces questions pour chaque jeu de données (et non une seule fois pour tout le système), dans cet ordre. Arrêtez-vous au premier *oui* qui convient ; si vous arrivez au bout, le choix relationnel par défaut s'applique.

Q1 **Ces données sont-elles un système de référence qui exige des transactions sur plusieurs lignes, des contraintes ou des requêtes ad hoc (commandes, paiements, comptes, réservations) ?**

- **Oui** Utilisez une base de données relationnelle. Ne cherchez plus loin que si vous pouvez montrer qu'elle ne peut pas satisfaire une exigence mesurée d'échelle ou de latence.
- **Non** Continuez : les données sont peut-être dérivées, spécialisées ou d'un volume extrêmement élevé.

Q2 **S'agit-il d'une copie dérivée d'un système de référence pour servir une requête particulière (recherche en texte intégral, recommandations, recherche par similarité) ?**

- **Oui** Utilisez le stockage spécialisé pour cette requête (moteur de recherche, graphe ou index vectoriel), alimenté par le système de référence et reconstructible à partir de lui.
- **Non** Continuez.

Q3 **Les données sont-elles lues et écrites élément par élément, par clé, sans requête portant sur plusieurs éléments (sessions, paniers, dernières positions, feature flags) ?**

- **Oui** Une base clé-valeur convient, souvent en mémoire avec expiration. Décidez de ce qui se passe si elle est perdue.
- **Non** Continuez.

Q4 **S'agit-il d'un flux, essentiellement en ajout, de mesures horodatées interrogées par plage temporelle (métriques, relevés de capteurs, historique de positions) ?**

- **Oui** Une base de données de séries temporelles, ou des tables partitionnées par période dans une base relationnelle pour des volumes modérés.
- **Non** Continuez.

Q5 **Le volume d'écriture ou la taille des données dépassera-t-il ce qu'un seul serveur bien réglé peut gérer, avec un accès toujours par une clé connue ?**

Oui Une base orientée colonnes larges ou clé-valeur conçue pour le partitionnement (Cassandra, DynamoDB), avec des tables conçues autour des requêtes.

Non Continuez.

Q6 **Avez-vous besoin de transactions relationnelles et de SQL entre régions ou au-delà de la capacité d'un seul serveur ?**

Oui Envisagez le SQL distribué (Spanner, CockroachDB, YugabyteDB), en acceptant une latence d'écriture et un coût d'exploitation plus élevés.

Non Restez sur le choix relationnel par défaut, et réexaminez-le quand une exigence mesurée change.

SI VOUS DÉBUTEZ

Pas à pas : la seule copie, ou une copie

L'idée qui demande le plus d'attention ici est la différence entre un système de référence et un stockage dérivé, et la raison pour laquelle elle détermine le degré d'audace permis dans le choix d'une base de données. Suivez une modification de carte à travers les stockages de la plateforme de livraison de repas.

- Un restaurant modifie la description d'un plat, et la modification est enregistrée dans la base PostgreSQL des cartes. Cette base est le système de référence : chaque fois qu'une autre copie d'une carte est en désaccord avec elle, c'est sa version qui l'emporte.
- La capture des données modifiées récupère le changement validé et le transmet au moteur de recherche, qui met à jour son index quelques secondes plus tard. Pendant ces secondes, une recherche peut encore afficher l'ancienne description, un retard que l'équipe a accepté.
- Supposons maintenant que le cluster de recherche tombe en panne et que son index soit perdu. Les clients ne peuvent plus chercher pendant un moment, mais aucune information n'a disparu : l'équipe reconstruit l'index en relisant toutes les cartes depuis PostgreSQL.
- Imaginez au contraire que les modifications des restaurants soient enregistrées uniquement dans le moteur de recherche. Son index est rafraîchi en quasi temps réel plutôt qu'au sein de transactions, et il peut protéger chaque écriture moins soigneusement que la base de données : la même panne pourrait alors perdre des modifications, sans rien à partir de quoi les reconstruire.
- C'est pourquoi la plateforme peut exploiter un stockage spécialisé pour la recherche, choisi pour cette seule tâche, tandis que les commandes et les cartes restent sur PostgreSQL, la base que l'équipe exploite déjà bien.
- **La leçon :** avant de choisir un stockage, demandez-vous s'il détiendra la seule copie de quoi que ce soit. Si c'est le cas, choisissez la technologie la plus prudente possible ; s'il détient une copie qui peut être reconstruite, vous êtes libre de vous spécialiser.

La persistance polyglotte et son coût

La persistance polyglotte consiste à utiliser des technologies de stockage différentes pour des données différentes au sein d'un même système, choisies selon l'usage qui est fait des données. Martin Fowler a popularisé le terme en 2011, en attribuant à Scott Leberknight sa première utilisation. Elle découle naturellement de la base de données par service : chaque propriétaire peut choisir le bon outil.

Le coût est facile à sous-estimer. Chaque type de base de données supplémentaire exige sa propre expertise, ses procédures de sauvegarde et de restauration, sa supervision, sa planification de la capacité, son durcissement de sécurité, son cycle de mises à niveau et ses connaissances d'astreinte. Les données circulent désormais entre les stockages : il faut donc des pipelines (CDC, événements) et raisonner sur le retard et les défaillances de chacun. La cohérence entre stockages est au mieux une cohérence à terme. Pour une petite équipe, cinq bases de données peuvent demander plus d'efforts que le produit. Une règle raisonnable consiste à justifier chaque nouveau stockage par une exigence que les stockages existants ne peuvent manifestement pas satisfaire, et à préférer des services managés pour les stockages spécialisés.

D'OÙ CELA VIENT

Choose Boring Technology

Dans un essai de 2015, Dan McKinley, s'appuyant sur ses années d'ingénieur chez Etsy, a soutenu que chaque entreprise ne dispose que de quelques *jetons d'innovation*, trois environ, à dépenser en technologies peu familières. Les outils matures, «ennuyeux», ont des modes de défaillance connus ; les nouveaux apportent des inconnues inconnues. Chaque nouvelle technologie ajoute aussi un coût opérationnel et cognitif permanent à toute l'organisation. Son conseil : demandez-vous d'abord comment vous résoudriez le problème avec ce que vous exploitez déjà, et traitez l'ajout d'une technologie comme une décision de l'organisation, pas d'une seule équipe. L'essai est largement cité dans les choix de bases de données, où le coût d'exploitation d'un outil peu familier dure des années.

EXEMPLE COMMENTÉ

Exemple commenté : les stockages de la plateforme de livraison de repas

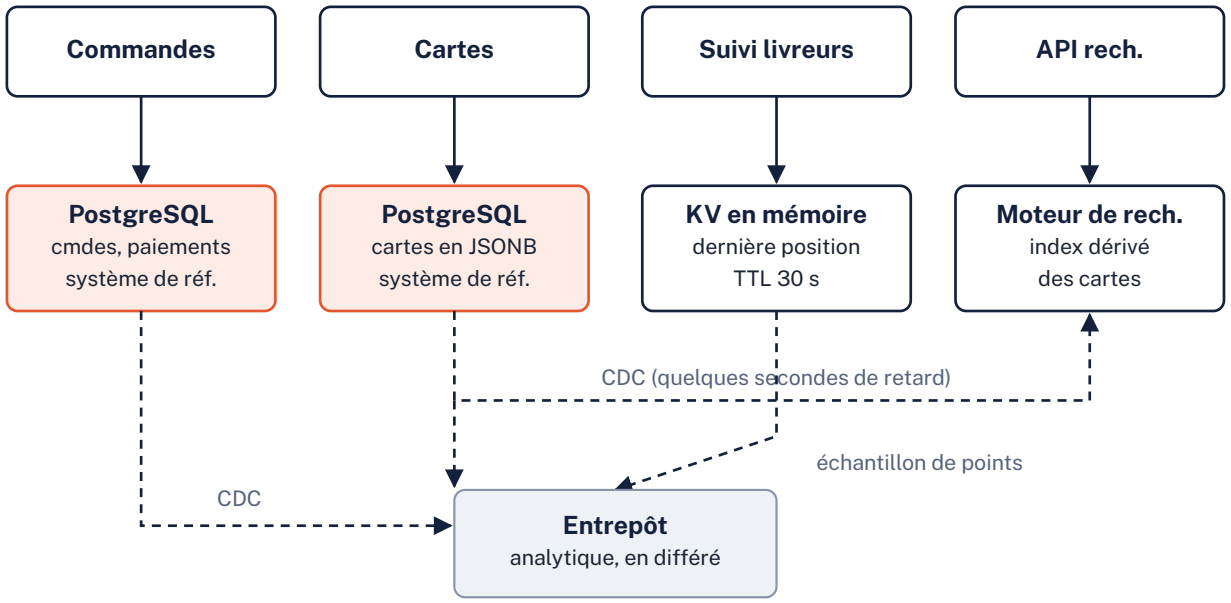
La plateforme opère dans 40 villes. Au pic du soir, elle reçoit environ 300 commandes par minute, compte 20 000 livreurs en ligne, sert 50 000 cartes de restaurants et fait l'objet d'environ 2000 recherches par seconde. L'équipe de 35 ingénieurs exploite déjà bien PostgreSQL.

Commandes et paiements	Système de référence ; transactions sur plusieurs lignes (commande, lignes, autorisation de paiement) ; règles strictes ; requêtes ad hoc du support. Environ 5 nouvelles commandes par seconde au pic (quelques dizaines d'écritures de lignes). **PostgreSQL**, avec une réplique pour le failover. Aucun argument pour autre chose.
Cartes	Lectures majoritaires, hiérarchiques (rubriques, plats, groupes d'options), modifiées par les restaurants quelques milliers de fois par jour. Les prix doivent être exactement ceux qu'enregistre la commande. **PostgreSQL**, avec un noyau relationnel (restaurant, plat, prix) et du `jsonb` pour les options variables. Une base orientée documents conviendrait à la forme des données, mais ajouterait une seconde technologie sans satisfaire aucune exigence que PostgreSQL ne satisfait pas.
Positions des livreurs	20 000 livreurs qui envoient chacun une position toutes les 4 secondes, cela fait 5000 écritures par seconde, dont seule la dernière par livreur compte pour l'affectation, et seulement pendant environ 30 secondes. **Base clé-valeur en mémoire** (Redis ou Valkey) avec requêtes géospatiales et expiration au bout de 30 secondes. La perdre au redémarrage coûte un cycle de mise à jour. Un échantillon de points (disons un toutes les 30 secondes) part vers l'entrepôt de données pour l'analytique.
Recherche de restaurants et de plats	Texte intégral, tolérance aux fautes de frappe, filtres par lieu, horaires d'ouverture et étiquettes alimentaires, classement par pertinence. **Moteur de recherche** (OpenSearch), alimenté à partir de la base des cartes par capture des données modifiées, et reconstructible à partir d'elle. Accepte quelques secondes de retard après une modification de carte.
Analytique	Chiffre d'affaires hebdomadaire, délais de livraison par ville, taux d'utilisation des livreurs : de gros parcours et agrégats sur des mois d'historique. **Entrepôt de données dans le cloud**, alimenté par CDC et par des événements, avec quelques heures de retard par conception. Voir analytique et streaming.
Rejetés	Une base orientée graphe pour les recommandations (pas encore nécessaire ; à réexaminer sur preuves), une base de séries temporelles distincte pour les positions (l'échantillon de l'entrepôt répond aux questions posées) et le SQL distribué (les commandes de toute la plateforme, environ 5 par seconde au pic, tiennent largement sur un seul primaire ; il n'y a aucune exigence de transaction multirégion).

Le résultat : deux systèmes de référence sur une technologie bien connue, plus trois stockages spécialisés et dérivés. Chaque technologie supplémentaire est justifiée par une exigence que le choix par défaut ne peut pas satisfaire : 5000 écritures éphémères par seconde, et une recherche textuelle classée par pertinence.

Noter d'abord les requêtes et les chiffres a rendu les décisions presque évidentes, et a donné à l'équipe une trace des hypothèses à réexaminer quand les volumes changeront.

Le paysage de stockage qui en résulte



Deux systèmes de référence ; tout le reste est dérivé et reconstructible

Les commandes et les cartes ont chacune un seul système de référence relationnel ; l'index de recherche, l'entrepôt de données et le stockage des positions sont dérivés ou jetables.

Erreurs courantes

Les choix dictés par le CV : choisir un stockage parce qu'il est nouveau ou parce qu'une grande entreprise l'utilise, sans l'exigence qui le justifiait chez elle. **Choisir pour une échelle imaginaire** : concevoir pour un milliard d'utilisateurs un produit qui en a mille, et payer la taxe de complexité dès le premier jour. **Utiliser un stockage dérivé comme système de référence** : un index de recherche ou un cache qui devient la seule copie des données, avec une durabilité plus faible que personne ne l'imaginait. **Ignorer le côté requêtes** : choisir une base clé-valeur ou orientée colonnes larges, puis découvrir que le métier a besoin de rapports ad hoc et de filtres qu'elle ne peut pas fournir. **Sous-estimer l'exploitation** : oublier que quelqu'un devra la restaurer à partir d'une sauvegarde, la mettre à niveau sans interruption de service et la

diagnostiquer sous charge. Enfin, **traiter le choix comme une vérité définitive** : consignez les hypothèses (volumes, requêtes, compétences de l'équipe) pour que la décision puisse être réexaminée quand elles changent.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi un moteur de recherche ne devrait-il généralement pas être le système de référence ?**

Son index est rafraîchi en quasi temps réel plutôt que de façon transactionnelle, et ses garanties de durabilité sont plus faibles ; il devrait être alimenté à partir de la base de données propriétaire et reconstruit à partir d'elle en cas de besoin.

▼ **Donnez deux signes indiquant qu'une base orientée colonnes larges peut être un bon choix.**

Un volume d'écriture très élevé, au-delà d'un seul serveur, et un accès qui part toujours d'une clé de partition connue, de sorte que les tables peuvent être conçues par requête.

▼ **Que représente un « jeton d'innovation » dans l'essai de McKinley ?**

L'une des rares occasions dont dispose une entreprise pour adopter une technologie peu familière, chacune apportant des modes de défaillance inconnus et un coût opérationnel durable.

▼ **Quand le SQL distribué vaut-il son coût ?**

Quand vous avez besoin de transactions relationnelles et de SQL au-delà de la capacité d'un seul serveur ou entre régions, comme pour un grand livre mondial ; pas quand un seul primaire avec des répliques satisfait les exigences.

▼ **À vous : Une équipe de 20 personnes en livraison de repas connaît bien PostgreSQL. Elle a besoin de stockage pour les commandes et les paiements, pour les positions en direct des livreurs mises à jour toutes les quelques secondes, et pour la recherche textuelle de restaurants. Choisissez un stockage pour chacun et dites ce qui limite le nombre que vous en ajoutez.**

Les commandes et les paiements restent dans PostgreSQL : un système de référence qui a besoin de transactions, de contraintes et de requêtes ad hoc. Les positions des livreurs conviennent à un magasin clé-valeur à expiration courte, car seule la dernière valeur compte et elle change vite. La recherche de restaurants convient à un moteur de recherche, ou d'abord à la recherche textuelle de PostgreSQL, comme magasin dérivé qui peut être reconstruit. Chaque technologie supplémentaire ajoute un coût d'exploitation durable : notez donc les requêtes et les volumes, et n'en ajoutez une que sur preuves.

SOURCES

- Dan McKinley: Choose Boring Technology (2015): https://mcfunley.com/choose-boring-technology
- Martin Fowler: PolyglotPersistence (2011): https://martinfowler.com/bliki/PolyglotPersistence.html
- Google Research: Spanner: Google's Globally-Distributed Database (OSDI 2012): https://research.google/pubs/spanner-googles-globally-distributed-database-2/
- Google Cloud: Spanner TrueTime and external consistency: https://cloud.google.com/spanner/docs/true-time-external-consistency
- DeCandia et al.: Dynamo: Amazon's Highly Available Key-value Store (SOSP 2007): https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- RFC 9562: Universally Unique IDentifiers (UUIDs): https://www.rfc-editor.org/rfc/rfc9562.txt
- ULID specification: https://github.com/ulid/spec
- PostgreSQL documentation: UUID functions: https://www.postgresql.org/docs/current/functions-uuid.html

SECTION 7.3 SUR 7.8 · AVANCÉ · 28 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Transactions, ACID et niveaux d'isolation

AVANT DE LIRE

Cette section s'appuie sur :

- 0.5 Stocker les données

Termes déjà vus qu'elle utilise : SQL (Structured Query Language), Clé étrangère, Index, Linéarisabilité

L'IDÉE EN BREF

Imaginez deux caissiers qui partagent un même registre papier pour le compte d'un client. Chacun consulte le solde, calcule le nouveau sur un bloc-notes et l'inscrit. S'ils le font au même moment, la modification de l'un des caissiers est perdue. Une transaction de base de données est une règle sur la façon dont les caissiers passent chacun leur tour : peuvent-ils jeter un œil aux écritures à moitié rédigées de l'autre, la page peut-elle changer pendant qu'ils la lisent, et l'un doit-il attendre l'autre ? Des règles plus strictes signifient moins d'erreurs mais plus d'attente, et les noms de ces règles n'ont pas tout à fait le même sens d'une banque à l'autre.

CE QU'IL FAUT SAVOIR

- ACID : l'atomicité signifie tout ou rien en cas de défaillance ; la cohérence signifie que les invariants de l'application sont respectés ; l'isolation limite les interférences entre transactions concurrentes ; la durabilité signifie que les données validées survivent aux pannes, généralement grâce à un journal d'écriture anticipée et à des répliques.
- La norme SQL définit read uncommitted, read committed, repeatable read et serializable par les lectures sales, les lectures non répétables et les fantômes qu'ils interdisent ; Berenson et al. (1995) ont montré que ces définitions sont ambiguës et incomplètes.
- Les mises à jour perdues et le write skew surviennent lorsque des transactions lisent des données, décident dans l'application et écrivent en retour ; l'isolation par instantané empêche les mises à jour perdues mais permet le write skew.
- Le MVCC conserve plusieurs versions des lignes pour que lecteurs et écrivains ne se bloquent pas mutuellement ; les transactions de longue durée font gonfler le stockage.
- Les valeurs par défaut et les noms diffèrent : PostgreSQL et Oracle utilisent par défaut read committed, MySQL InnoDB repeatable read, CockroachDB serializable ; le serializable d'Oracle est en fait de l'isolation par instantané.

> - Protégez les chemins lecture-modification-écriture par des contraintes, des mises à jour conditionnelles atomiques, SELECT … FOR UPDATE (pessimiste) ou des colonnes de version (optimiste), ou utilisez l'isolation sérialisable en relançant la transaction entière.

Ce que promet une transaction

Une transaction regroupe plusieurs lectures et écritures en une seule unité qui prend effet entièrement (elle est validée, commit) ou pas du tout (elle est abandonnée et annulée, rollback). Le chapitre 0 a présenté l'idée avec un virement bancaire. Cette section porte sur ce que la plupart des développeurs comprennent mal : ce qui se passe quand de nombreuses transactions s'exécutent en même temps, quelles garanties les bases de données donnent réellement, et comment choisir. Les exemples utilisent SQL, mais le raisonnement vaut pour tout stockage qui propose des transactions.

Les transactions comptent pour l'architecture parce qu'elles déterminent où les règles métier peuvent être appliquées à moindre coût. Au sein d'une seule base de données, une transaction peut maintenir un registre équilibré ou empêcher qu'une table soit réservée deux fois, en quelques lignes de code. Entre services, cet outil n'existe pas, et c'est pourquoi la propriété des données et les sagas sont si importantes. Savoir exactement ce que garantit une seule base de données vous dit quelles règles doivent rester à l'intérieur du périmètre d'un seul propriétaire.

ACID en détail

L'acronyme ACID a été forgé par Theo Härder et Andreas Reuter en 1983, à partir des travaux antérieurs de Jim Gray. Chaque lettre est plus étroite qu'elle n'en a l'air. L'**atomicité** concerne les défaillances, pas la concurrence : si quelque chose tourne mal en cours de route (une panne, une violation de contrainte, un interblocage, ou l'application qui décide d'abandonner), toutes les modifications faites jusque-là sont annulées, si bien que l'application peut réessayer sans risque. *Abandonnabilité* serait un nom plus exact.

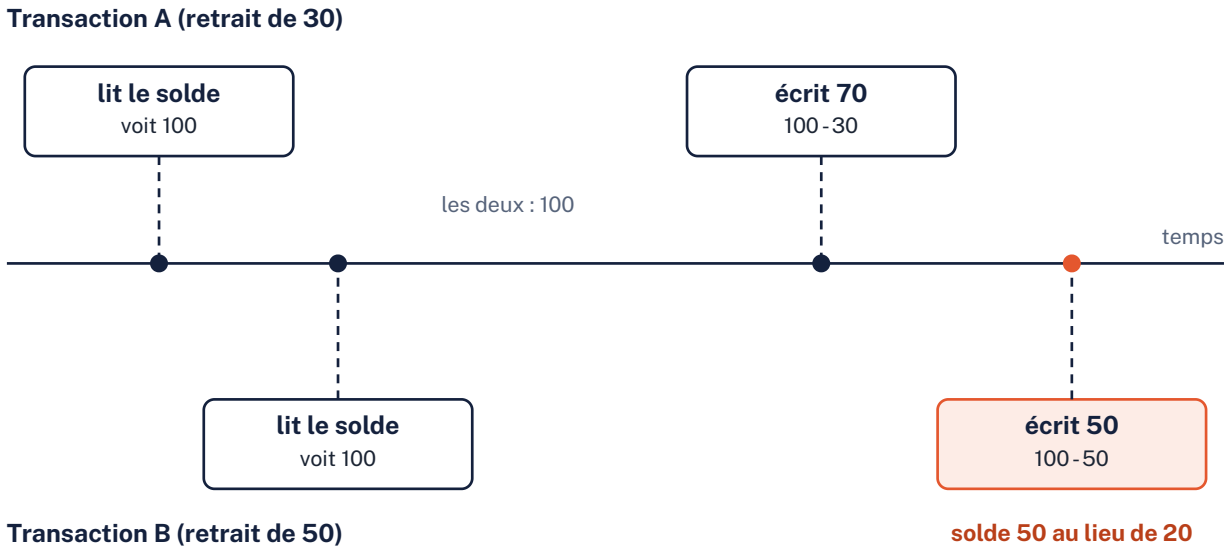
La **cohérence** signifie qu'une transaction fait passer la base de données d'un état valide à un autre, où «valide» signifie que vos invariants sont respectés : les débits égalent les crédits, le stock n'est jamais négatif, chaque commande a un client. La base de données peut appliquer elle-même certains invariants (clés, clés étrangères, contraintes `CHECK`, index uniques), mais la plupart sont des règles métier que seules les transactions de l'application maintiennent. La cohérence est donc une propriété de la façon dont l'application utilise l'atomicité et l'isolation, et non quelque chose que la base de données fournit à elle seule. Elle n'a d'ailleurs aucun rapport avec le «C» de CAP, qui désigne la linéarisabilité (chapitre 6).

L'**isolation** signifie que des transactions qui s'exécutent simultanément n'interfèrent pas entre elles. Dans sa forme la plus stricte, la sérialisabilité, le résultat de l'exécution concurrente des transactions est le même que si elles s'étaient exécutées l'une après l'autre dans un certain ordre. Les niveaux plus faibles tolèrent certaines interférences en échange de performances, et le reste de cette section explique précisément lesquelles. La **durabilité** signifie qu'une fois un commit confirmé, les données survivent aux pannes. Les bases de données à un seul nœud y parviennent grâce à un journal d'écriture anticipée écrit sur disque avant que le commit ne rende la main ; les bases de données répliquées y ajoutent des copies sur d'autres machines. La durabilité n'est jamais absolue : un disque peut mentir sur une écriture forcée, et la réplication asynchrone peut perdre les dernières écritures confirmées quand le leader tombe en panne (voir la réplication).

Ce qui tourne mal sans isolation

Les bugs de concurrence sont difficiles à trouver parce qu'ils dépendent du moment : ils passent tous les tests sur la machine tranquille d'un développeur et apparaissent au pic de l'heure du dîner. La façon la plus claire de comprendre les niveaux d'isolation est de passer par les anomalies précises (aussi appelées phénomènes) qu'ils permettent ou empêchent. La mise à jour perdue, ci-dessous, est la plus courante dans le code métier.

Une mise à jour perdue, étape par étape



Chaque transaction est correcte prise isolément ; entrelacées, la seconde écriture efface sans bruit le premier retrait.

Les anomalies, de la plus bénigne à la plus subtile

Anomalie	Ce qui se passe	Exemple de livraison de repas ou bancaire
Écriture sale (dirty write)	Une transaction écrase des données qu'une autre transaction a écrites mais pas encore validées.	Deux transactions qui affectent un livreur s'entrelacent, si bien que la ligne de la commande désigne un livreur et la ligne du livreur une autre commande. Tous les niveaux d'isolation réels empêchent cela.
Lecture sale (dirty read)	Une transaction lit des données écrites par une autre transaction qui n'a pas été validée (et qui peut être annulée).	Un rapport comptabilise un paiement de 500 qui est ensuite annulé.
Lecture non répétable (fuzzy read)	Une transaction lit deux fois la même ligne et obtient des valeurs différentes, parce qu'une autre transaction a validé une modification entre-temps.	L'étape de paiement lit le prix d'un plat à 9,50, une modification du restaurant fixant 11,00 est validée, et la seconde lecture de l'étape de paiement voit 11,00.
Read skew (biais de lecture)	Une transaction voit des parties de la base de données à des instants différents, si bien que des valeurs liées ne concordent pas.	Un client qui a 500 répartis sur deux comptes voit 450 au total, parce qu'un virement de 50 a été validé entre la lecture des deux soldes.
Lecture fantôme (phantom read)	Une transaction relance une requête avec une condition et trouve des lignes ajoutées ou supprimées par une autre transaction validée.	Compter deux fois les livraisons en cours d'un livreur renvoie 2, puis 3.
Mise à jour perdue (lost update)	Deux transactions lisent une valeur, chacune en calcule une nouvelle et l'écrit, et l'une des écritures écrase l'autre sans bruit.	Deux retraits sur le même compte (schéma ci-dessus).
Write skew (biais d'écriture)	Deux transactions lisent un ensemble de données qui se recoupent, prennent des décisions à partir de celles-ci et mettent à jour des lignes différentes, si bien qu'ensemble elles enfreignent une règle que chacune avait vérifiée.	Deux médecins de garde voient chacun que quelqu'un d'autre est encore de garde, et tous deux quittent leur garde (exemple commenté ci-dessous).

Les niveaux d'isolation de la norme SQL

La norme SQL (à partir de SQL-92) définit quatre niveaux d'isolation selon les phénomènes qu'ils interdisent parmi trois : les lectures sales, les lectures non répétables et les fantômes. La norme fait de serializable le niveau par défaut, mais presque aucun produit courant ne la suit

sur ce point ; la plupart utilisent par défaut read committed ou repeatable read, pour les performances.

Les niveaux de la norme SQL et les phénomènes qu'ils excluent

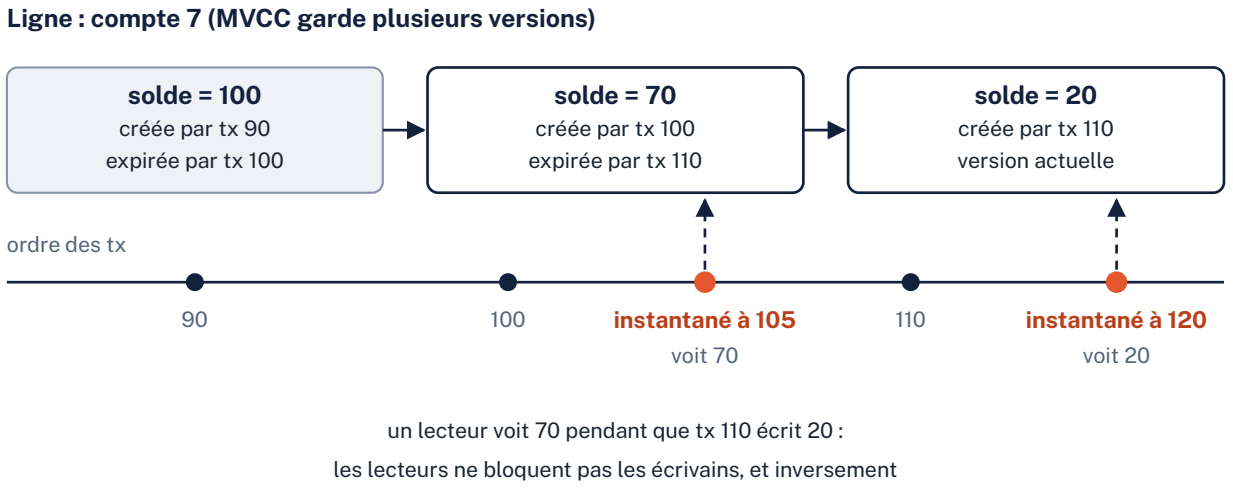
Niveau d'isolation	Lecture sale (dirty read)	Lecture non répétable	Lecture fantôme (phantom read)
Read uncommitted	possible	possible	possible
Read committed	empêchée	possible	possible
Repeatable read	empêchée	empêchée	possible
Serializable	empêchée	empêchée	empêchée, et le résultat doit équivaloir à un certain ordre d'exécution en série

En 1995, Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil et Patrick O'Neil ont publié *A Critique of ANSI SQL Isolation Levels*. Ils ont montré que les définitions de la norme, rédigées en anglais courant, étaient ambiguës, qu'interdire les trois phénomènes ne suffit pas à garantir la sérialisabilité, et que les niveaux de la norme ne décrivaient pas les niveaux d'isolation qu'utilisaient les produits réels. Ils ont ajouté des phénomènes que la norme avait oubliés, notamment l'écriture sale, la mise à jour perdue, le read skew et le write skew, et ont défini l'isolation par instantané (snapshot isolation) : chaque transaction lit dans un instantané cohérent des données validées tel qu'il était à son début, et une transaction ne peut pas valider l'écriture d'une ligne qu'une transaction concurrente a déjà modifiée et validée («le premier qui valide l'emporte»). L'isolation par instantané empêche les lectures sales, les lectures non répétables, le read skew et les mises à jour perdues, ainsi que les fantômes qui touchent les lectures, mais elle permet le write skew : elle n'est donc pas sérialisable. Elle s'insère mal entre les niveaux de la norme, et c'est pourquoi les produits l'étiquettent différemment.

MVCC : comment fonctionnent les instantanés

La plupart des bases de données modernes mettent en œuvre les lectures sur instantané avec le contrôle de concurrence multiversion (MVCC). Au lieu d'écraser une ligne sur place, une écriture crée une nouvelle version marquée de l'identifiant de la transaction qui l'écrit, et l'ancienne version est conservée jusqu'à ce qu'aucune transaction en cours ne puisse encore en avoir besoin. Chaque lecteur reçoit un instantané, c'est-à-dire en substance la liste des transactions qui avaient été validées à son démarrage, et ne voit que les versions créées par ces transactions. Le grand avantage est que les lecteurs n'attendent jamais les écrivains et que les écrivains n'attendent jamais les lecteurs ; seuls deux écrivains sur la même ligne entrent en conflit.

Les versions d'une ligne avec le MVCC



Chaque mise à jour crée une nouvelle version de la ligne ; une transaction voit la version la plus récente validée avant son instantané.

Le prix à payer, c'est l'entretien. Les anciennes versions doivent être nettoyées : PostgreSQL les garde dans la table et les supprime avec *vacuum*, tandis qu'Oracle et MySQL InnoDB conservent les images avant modification dans des journaux d'annulation (undo logs). Une transaction laissée ouverte pendant des heures retient les anciennes versions, si bien que les tables et les journaux d'annulation gonflent et que les performances se dégradent. Les transactions de longue durée sont donc un signal d'alerte opérationnel dans toute base de données MVCC, et une raison de tenir les requêtes analytiques à l'écart de la base de données opérationnelle. En **read committed** avec MVCC, chaque *instruction* reçoit un nouvel instantané ; en **isolation par instantané**, toute la *transaction* en utilise un seul.

Les valeurs par défaut réelles, et pourquoi les noms induisent en erreur

Les noms des niveaux d'isolation ne sont pas portables. Le même mot peut désigner des garanties différentes selon les produits, et les valeurs par défaut diffèrent. Le projet Hermitage de Martin Kleppmann a testé le comportement réel de nombreuses bases de données face aux anomalies ci-dessus ; la documentation des éditeurs confirme les valeurs par défaut. Considérez ce tableau comme un point de départ et vérifiez la documentation de la version que vous utilisez.

Mêmes noms, comportements différents : ce que font les bases de données réelles (résumé de la documentation des éditeurs et des tests Hermitage)

Base	Niveau par défaut	Ce que sont réellement les niveaux	Remarques
PostgreSQL	Read committed	Read uncommitted se comporte comme read committed. Repeatable read est de l'isolation par instantané. Serializable est de l'isolation par instantané sérialisable (SSI).	Repeatable read abandonne une transaction qui tente de mettre à jour une ligne modifiée depuis son instantané (*could not serialize access due to concurrent update*), si bien que les mises à jour perdues sont empêchées ; le write skew ne l'est pas.
MySQL (InnoDB)	Repeatable read	Les SELECT simples lisent un instantané pris à la première lecture ; les lectures avec verrou, UPDATE et DELETE agissent sur les dernières lignes validées, avec des verrous d'enregistrement et des verrous d'intervalle (gap locks). Serializable transforme les SELECT simples en lectures avec verrou lorsque l'autocommit est désactivé.	En repeatable read, une séquence lecture-modification-écriture écrite sous la forme d'un SELECT puis d'un UPDATE peut encore perdre une mise à jour ou produire un write skew, sauf si le SELECT utilise FOR UPDATE.
Oracle Database	Read committed	Propose read committed et serializable ; son serializable est de l'isolation par instantané.	Empêche les mises à jour perdues en serializable, mais permet le write skew.

Base	Niveau par défaut	Ce que sont réellement les niveaux	Remarques
Microsoft SQL Server	Read committed (avec verrous) sur SQL Server ; Azure SQL Database active read committed snapshot par défaut	Read committed avec verrous, read committed snapshot, repeatable read (avec verrous), snapshot (isolation par instantané) et serializable (avec verrous).	Sur SQL Server, l'isolation par instantané doit être activée base par base (ALLOW_SNAPSHOT_ISOLATION) avant d'être utilisée.
CockroachDB	Serializable	Serializable par défaut ; read committed est également disponible.	Les clients doivent relancer les transactions qui échouent avec une erreur de sérialisation.

ATTENTION

Deux pièges cachés dans les noms

D'abord, «repeatable read» dans PostgreSQL et dans MySQL désigne deux choses différentes : celui de PostgreSQL est une véritable isolation par instantané qui abandonne les mises à jour conflictuelles, tandis que celui de MySQL mélange lectures sur instantané et écritures avec verrous sur les données les plus récentes, et peut perdre des mises à jour écrites sous la forme lecture puis écriture. Ensuite, «serializable» dans Oracle est de l'isolation par instantané et permet le write skew. Un code correct sur une base de données peut être faux sur une autre, avec un paramètre portant exactement le même nom.

Obtenir la sérialisabilité

Les bases de données qui offrent une véritable sérialisabilité utilisent l'une de trois techniques. L'**exécution réellement en série** exécute les transactions une par une sur un seul thread, ce qui est rapide si chaque transaction est courte et si les données sont en mémoire (VoltDB, ainsi que les scripts et transactions Redis, fonctionnent ainsi). Le **verrouillage à deux phases (2PL)**, utilisé par les niveaux serializable de SQL Server et de MySQL, fait prendre aux lecteurs des verrous partagés et aux écrivains des verrous exclusifs, y compris des verrous sur des plages pour arrêter les fantômes, et les conserve jusqu'à la fin de la transaction ; c'est correct, mais lecteurs et écrivains se bloquent mutuellement, le débit chute en cas de contention et les

interblocages sont fréquents. L'**isolation par instantané sérialisable (SSI)**, décrite par Michael Cahill, Uwe Röhm et Alan Fekete en 2008 et utilisée dans PostgreSQL depuis la version 9.1 (CockroachDB utilise une technique optimiste apparentée), exécute les transactions sur des instantanés sans blocage, suit les dépendances lecture-écriture entre elles et abandonne une transaction quand un motif dangereux apparaît.

Quelle que soit la technique, les transactions sérialisables peuvent échouer avec une erreur de sérialisation (SQLSTATE `40001` dans PostgreSQL et CockroachDB) ou un interblocage, et l'application doit alors relancer toute la transaction. C'est le coût caché du niveau le plus fort : chaque chemin de code transactionnel a besoin d'une boucle de nouvelles tentatives qui puisse être répétée sans risque, ce qui exclut tout effet de bord, comme l'envoi d'e-mails ou l'appel de prestataires de paiement, à l'intérieur de la transaction.

Le pool de connexions

Chaque transaction occupe une connexion à la base de données, et les connexions sont limitées (section 9.1). Les applications conservent donc un **pool** et empruntent une connexion pour chaque unité de travail. Plus grand n'est pas mieux : les notes de HikariCP citent une démonstration d'Oracle dans laquelle réduire 2048 connexions à 96 a fait passer les temps de réponse d'environ 100 ms à environ 2 ms, et proposent (cœurs × 2) + disques effectifs comme point de départ à tester. Le total est la taille du pool multipliée par le nombre d'instances ; ajouter des instances peut donc épuiser la base de données, et un pooler central comme PgBouncer y aide. Son mode transaction prête une connexion pour une transaction à la fois, mais des fonctions de session comme `SET` et les verrous consultatifs de session échouent alors. Ne gardez jamais une connexion pendant l'appel à un autre service, sinon un prestataire de paiement lent videra le pool.

Contrôle de concurrence pessimiste et optimiste

En pratique, la plupart des applications s'exécutent en read committed ou en repeatable read et protègent explicitement certains chemins lecture-modification-écriture. Il existe trois outils. Le meilleur, quand c'est possible, est une **mise à jour atomique** qui laisse la base de données faire le calcul et la vérification en une seule instruction, comme `UPDATE accounts SET balance = balance - 30 WHERE id = 7 AND balance ⩾ 30`, puis vérifier qu'exactement une ligne a changé. En read committed dans PostgreSQL, une mise à jour concurrente fait attendre la seconde instruction, qui réévalue ensuite sa condition sur la nouvelle ligne : aucune mise à jour n'est perdue.

Lorsque la décision exige une logique que la base de données ne peut pas exprimer, utilisez le verrouillage pessimiste : verrouillez les lignes que vous lisez avec `SELECT ... FOR UPDATE`, afin que les transactions concurrentes qui en ont besoin attendent votre commit. C'est simple

et adapté aux opérations courtes et très disputées (affecter un livreur à une commande). Les coûts sont l'attente, le risque d'interblocages (verrouillez les lignes dans un ordre constant, par exemple par id croissant) et un débit réduit. Parmi les variantes figurent `NOWAIT`, qui échoue immédiatement au lieu d'attendre, et `SKIP LOCKED`, qui saute les lignes déjà verrouillées et constitue la façon classique de construire une file de tâches fiable sur une table relationnelle.

```
BEGIN;
-- Lock the row: other FOR UPDATE readers and writers must wait
SELECT balance FROM accounts WHERE id = 7 FOR UPDATE;
-- application checks: balance (100) ⩾ 30, so allow it
UPDATE accounts SET balance = 70 WHERE id = 7;
INSERT INTO ledger (account_id, amount, kind)
  VALUES (7, -30, 'withdrawal');
COMMIT;  -- lock released; the waiting transaction now sees 70
```

Verrouillage pessimiste : le second retrait attend au SELECT que le premier soit validé, puis lit le nouveau solde.

Le contrôle de concurrence optimiste ne prend aucun verrou pendant que l'utilisateur ou le code réfléchit. Chaque ligne porte un numéro de version (ou un horodatage, ou une empreinte). L'écrivain ne met à jour que si la version est toujours celle qu'il a lue, et l'incrémente. Si aucune ligne ne change, quelqu'un d'autre est passé avant, et l'application recharge et réessaie, ou prévient l'utilisateur. Cela convient aux données peu disputées et aux longs «temps de réflexion» : le gérant d'un restaurant qui modifie une carte dans un navigateur pendant cinq minutes ne doit pas garder un verrou de base de données tout ce temps. HTTP intègre la même idée, avec les ETags et l'en-tête `If-Match`.

```
-- Read (no lock); remember version = 12
SELECT id, name, price, version FROM menu_items WHERE id = 501;

-- Later, write only if the version is unchanged
UPDATE menu_items
   SET price = 11.00, version = version + 1
 WHERE id = 501 AND version = 12;

-- Rows affected = 1: success.
-- Rows affected = 0: a concurrent edit won; reload and retry
-- or show the user the other change.
```

Concurrence optimiste avec une colonne de version : la mise à jour ne réussit que si personne n'a modifié le plat de la carte depuis sa lecture.

EXEMPLE COMMENTÉ

Exemple commenté : le write skew avec deux médecins de garde

Le système de planning d'un hôpital applique une seule règle : au moins un médecin doit être de garde pour chaque créneau. Alice et Bob sont tous deux de garde ce soir. Tous deux se sentent mal et appuient sur «Quitter la garde» presque au même moment. Chaque demande exécute cette logique dans une transaction : compter les médecins de garde pour le créneau ; s'il y en a au moins deux, passer ma propre ligne à « pas de garde ». La base de données s'exécute en isolation par instantané (repeatable read de PostgreSQL ou serializable d'Oracle).

Début	Table `doctors` : Alice on_call = true, Bob on_call = true. Règle : count(on_call) >= 1.
Étape 1 : T1 (Alice) commence	T1 prend un instantané.
Étape 2 : T2 (Bob) commence	T2 prend un instantané. Les deux instantanés montrent deux médecins de garde.
Étape 3 : T1 lit	`SELECT count(*) FROM doctors WHERE shift = 1234 AND on_call` renvoie 2. La vérification passe.
Étape 4 : T2 lit	La même requête dans l'instantané de T2 renvoie aussi 2. La vérification passe.
Étape 5 : T1 écrit	`UPDATE doctors SET on_call = false WHERE name = 'Alice' AND shift = 1234`
Étape 6 : T2 écrit	`UPDATE doctors SET on_call = false WHERE name = 'Bob' AND shift = 1234`. C'est une autre ligne, donc aucun conflit écriture-écriture n'est détecté.
Étape 7 : les deux valident	L'isolation par instantané ne voit aucun conflit : chaque transaction a écrit une ligne que l'autre n'a pas écrite. Résultat : plus personne n'est de garde. Chaque transaction était correcte isolément ; ensemble, elles ont enfreint la règle.
En serializable (SSI)	La base de données remarque que chaque transaction a lu des données que l'autre a ensuite modifiées (un cycle dangereux de dépendances lecture-écriture). L'un des commits échoue avec SQLSTATE 40001 ; la nouvelle tentative recompte, trouve un seul médecin et refuse.

Correction à un niveau plus faible	Verrouillez les lignes dont dépend la décision : `SELECT name FROM doctors WHERE shift = 1234 AND on_call FOR UPDATE`, puis comptez les lignes renvoyées. T2 attend désormais les verrous de T1. En read committed, T2 relit ensuite, ne trouve que Bob de garde et refuse ; en repeatable read de PostgreSQL, elle échoue plutôt avec une erreur de sérialisation, et la nouvelle tentative refuse.

Le write skew est une généralisation de la mise à jour perdue : les transactions lisent les mêmes données mais écrivent des lignes *différentes*, si bien que la règle de l'isolation par instantané selon laquelle le premier qui valide l'emporte ne se déclenche jamais.

Le même motif apparaît dans les systèmes de réservation (deux personnes qui réservent la dernière place ou des créneaux de réunion qui se chevauchent), les noms d'utilisateur (deux inscriptions qui vérifient qu'un nom est libre) et la banque (deux retraits sur des comptes différents qui, ensemble, dépassent un découvert autorisé commun).

Lorsque les lignes qui prouveraient un conflit n'existent pas encore, comme pour deux réservations qui se chevauchent, `FOR UPDATE` n'a rien à verrouiller. Les options sont l'isolation sérialisable, une contrainte d'unicité ou d'exclusion (PostgreSQL peut interdire de manière déclarative les plages horaires qui se chevauchent), ou la *matérialisation du conflit* : créer des lignes, par exemple une par salle et par créneau, uniquement pour pouvoir les verrouiller.

SI VOUS DÉBUTEZ

Pas à pas : sécuriser une lecture suivie d'une écriture

Le point qui fait le plus souvent trébucher ici est que placer une lecture et une écriture dans une même transaction n'empêche pas une autre transaction de se glisser entre les deux. Deux commandes portant sur un même plat montrent pourquoi, et comment y remédier.

- Le plat du jour d'un restaurant a encore 8 portions, stockées dans une seule ligne. Les commandes X (2 portions) et Y (3) arrivent en même temps, toutes deux en read committed, le niveau par défaut de PostgreSQL et d'Oracle.
- Chacune lit 8, soustrait ses portions et enregistre : X enregistre 6, puis Y enregistre 5. Les 2 portions de X ont disparu du décompte, une mise à jour perdue, si bien que 2 portions qui n'existent pas pourraient encore être vendues. L'atomicité n'a servi à rien : elle concerne les défaillances, pas le fait de passer chacun son tour.
- Première correction, une mise à jour atomique : la base de données fait le calcul et la vérification en une seule instruction, comme « en retirer 3 s'il en reste au moins 3 ». Dans PostgreSQL, l'instruction de Y attend le commit de X, revérifie par rapport à 6 et laisse 3.
- Deuxième correction, un verrou pessimiste : chaque transaction verrouille la ligne au moment où elle la lit. X arrive la première, donc la lecture de Y attend que X soit validée, puis voit 6, et non 8.
- Troisième correction, la concurrence optimiste : aucun verrou, mais chaque écriture indique le numéro de version qu'elle a lu et ne réussit que si la ligne porte toujours ce numéro. L'écriture de X passe et incrémente la version ; celle de Y ne correspond à aucune ligne, donc Y relit et réessaie avec 6.
- L'isolation sérialisable l'empêche aussi, mais peut faire échouer l'une des transactions, qu'il faut alors relancer.
- **La leçon :** le danger se trouve dans chaque séquence lecture-décision-écriture, quel que soit le nom du niveau d'isolation. Protégez chacune par une instruction atomique, un verrou, une vérification de version ou l'isolation sérialisable avec nouvelles tentatives.

Choisir une stratégie d'isolation

Il n'existe pas un seul bon niveau ; il existe un bon niveau pour chaque élément de logique. Une approche pratique part des invariants, pas des paramètres.

Liste de contrôle : sécuriser un chemin de code transactionnel

- [ ] **Nommez l'invariant**
  Écrivez la règle que le code protège (solde jamais négatif, un seul livreur par commande, au moins un médecin de garde). S'il n'y en a pas, le niveau par défaut convient probablement.
- [ ] **Préférez les contraintes et les instructions atomiques**
  Les index uniques, les clés étrangères, les contraintes CHECK et d'exclusion, ainsi que les mises à jour conditionnelles en une seule instruction, sont appliqués par la base de données quel que soit le niveau d'isolation.
- [ ] **Repérez les lectures-modifications-écritures**
  Tout code qui lit une valeur, décide dans l'application et écrit en retour présente un risque de mise à jour perdue ou de write skew. Protégez-le avec FOR UPDATE, une colonne de version ou l'isolation sérialisable.
- [ ] **Connaissez la sémantique réelle de votre base de données**
  Vérifiez ce que signifient réellement les noms de niveaux de votre produit (repeatable read diffère entre PostgreSQL et MySQL ; le serializable d'Oracle est de l'isolation par instantané).
- [ ] **Réessayez correctement**
  Serializable et repeatable read peuvent abandonner une transaction sur une erreur de sérialisation ou un interblocage. Relancez toute la transaction avec un délai croissant entre les tentatives (back-off), et gardez les effets de bord externes en dehors d'elle.
- [ ] **Gardez les transactions courtes**
  Pas de temps de réflexion de l'utilisateur, d'appels réseau ni de gros rapports dans une transaction ; les transactions longues conservent des verrous et font gonfler le stockage MVCC.
- [ ] **Testez en situation de concurrence**
  Exécutez les chemins critiques en parallèle dans un test d'intégration sur le vrai moteur de base de données, pas sur un substitut en mémoire à la sémantique différente.

En règle générale, read committed avec verrouillage explicite ou vérifications de version est un choix par défaut solide et rapide pour la plupart des applications métier, à condition que l'équipe repère avec rigueur les chemins lecture-modification-écriture. Serializable est le choix par défaut le plus sûr quand les règles sont complexes, que l'équipe est grande ou que la violation d'un invariant coûte cher (argent, médecine, sûreté), et de nombreuses équipes l'utilisent aujourd'hui pour les chemins critiques, en acceptant les nouvelles tentatives. Les

praticiens sont réellement en désaccord sur ce point : certains soutiennent que serializable par défaut évite toute une classe de bugs que la revue de code ne détectera jamais ; d'autres estiment que ses nouvelles tentatives et sa contention coûtent trop cher et que le verrouillage explicite rend l'intention visible. Les deux positions se défendent si le choix est délibéré et testé.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi l'isolation par instantané n'est-elle pas sérialisable ?**

Elle permet le write skew : deux transactions peuvent lire des données qui se recoupent et écrire des lignes différentes, enfreignant ensemble une règle que chacune avait vérifiée, sans qu'aucun conflit écriture-écriture soit détecté.

▼ **De quoi dépend le C d'ACID ?**

Surtout de l'application : la base de données applique les contraintes déclarées, mais les invariants métier ne sont respectés que si les transactions de l'application les préservent.

▼ **Quand choisiriez-vous le contrôle de concurrence optimiste plutôt que pessimiste ?**

Quand les conflits sont rares ou que le temps de réflexion entre la lecture et l'écriture est long, comme lorsqu'un utilisateur modifie un formulaire, si bien que garder des verrous serait coûteux ou impossible.

▼ **Pourquoi un code qui utilise l'isolation sérialisable doit-il pouvoir être relancé sans risque ?**

Les transactions sérialisables peuvent être abandonnées sur des échecs de sérialisation ou des interblocages, et la bonne réponse est de réexécuter toute la transaction : elle ne doit donc avoir aucun effet de bord externe.

▼ **À vous : Deux fans achètent le dernier billet. Chaque transaction lit remaining = 1, puis écrit remaining = 0 en read committed, et les deux réussissent. Nommez le problème et donnez le correctif le plus simple que vous utiliseriez, avec une solution de rechange.**

C'est une mise à jour perdue : tous deux ont lu, décidé dans le code applicatif et réécrit. La solution la plus simple est une mise à jour conditionnelle atomique : UPDATE shows SET remaining = remaining - 1 WHERE id = ? AND remaining > 0, puis vérifiez qu'une ligne a changé ; si aucune n'a changé, le spectacle est complet. Sinon, utilisez SELECT … FOR UPDATE, ou une colonne de version avec une nouvelle tentative. Une contrainte CHECK imposant que remaining soit au moins égal à 0 ajoute un dernier filet de sécurité.

SOURCES

- Berenson, Bernstein, Gray, Melton, O'Neil and O'Neil: A Critique of ANSI SQL Isolation Levels (1995): https://www.microsoft.com/en-us/research/publication/a-critique-of-ansi-sql-isolation-levels/
- PostgreSQL documentation: Transaction Isolation: https://www.postgresql.org/docs/current/transaction-iso.html
- MySQL 8.4 Reference Manual: Transaction Isolation Levels: https://dev.mysql.com/doc/refman/8.4/en/innodb-transaction-isolation-levels.html
- Microsoft Learn: SET TRANSACTION ISOLATION LEVEL (Transact-SQL): https://learn.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql
- Jepsen: Consistency Models: https://jepsen.io/consistency
- HikariCP wiki: About Pool Sizing: https://github.com/brettwooldridge/HikariCP/wiki/About-Pool-Sizing
- PgBouncer: Features: https://www.pgbouncer.org/features.html

SECTION 7.4 SUR 7.8 · AVANCÉ · 31 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Réplication, partitionnement et sharding

AVANT DE LIRE

Cette section s'appuie sur :

- 0.5 Stocker les données
- 2.5 Disponibilité et fiabilité
- 6.6 CAP et cohérence

Termes déjà vus qu'elle utilise : Disponibilité, Latence, Index, Journal d'écriture anticipée (write-ahead log, WAL), Linéarisabilité

L'IDÉE EN BREF

Imaginez une bibliothèque très fréquentée. Pour continuer à servir ses lecteurs si l'un de ses bâtiments est inondé, elle conserve des copies identiques de son catalogue dans plusieurs annexes : c'est la réplication. Les copies sont mises à jour par coursier, si bien qu'une annexe peut avoir un peu de retard, et un lecteur qui vient de rendre un livre dans une annexe peut constater qu'une autre l'indique encore comme emprunté. Quand la collection devient trop grande pour un seul bâtiment, la bibliothèque la divise, de A à M ici et de N à Z là-bas : c'est le partitionnement. Bien diviser signifie qu'aucune annexe n'est débordée, et que les lecteurs ont rarement besoin de visiter toutes les annexes pour obtenir une réponse.

CE QU'IL FAUT SAVOIR

- La réplication conserve des copies des mêmes données sur plusieurs nœuds pour la disponibilité, le passage à l'échelle des lectures et la localité ; le partitionnement (sharding) répartit les données entre les nœuds pour le débit d'écriture et le stockage ; les systèmes réels combinent généralement les deux.
- Dans la réplication leader-suiveur, toutes les écritures vont à un seul leader ; les suiveurs synchrones protègent les écritures confirmées au prix de la latence, tandis que les suiveurs asynchrones sont rapides mais peuvent perdre des écritures récentes lors d'un failover.
- Le retard de réplication compromet la lecture de ses propres écritures, les lectures monotones et les lectures à préfixe cohérent ; les remèdes consistent notamment à lire ses propres écritures récentes depuis le leader, à attendre une position du journal et à attacher chaque utilisateur à une même réplique.

- Le failover doit détecter la défaillance, élire le nœud le plus à jour et neutraliser l'ancien leader (fencing) ; le split brain, où deux nœuds acceptent des écritures, peut entraîner des rétablissements longs et pénibles, comme l'a montré l'incident de GitHub en 2018.
- La réplication multi-leader doit résoudre les conflits d'écriture ; la réplication sans leader (de type Dynamo) utilise des quorums avec R + W > N, la réparation à la lecture (read repair) et l'anti-entropie, mais les quorums seuls ne garantissent pas la linéarisabilité.
- Le partitionnement par plage permet les parcours de plages mais crée des points chauds pour les clés ordonnées dans le temps ; le partitionnement par hachage répartit la charge mais disperse les plages ; le hachage cohérent ou des partitions logiques fixes rendent le rééquilibrage peu coûteux.
- Les clés chaudes, le déséquilibre (skew) et les index secondaires (locaux : lectures scatter-gather ; globaux : écritures d'index distantes, généralement asynchrones) doivent être pris en compte dès la conception, à partir des requêtes les plus importantes.

Deux raisons différentes de répartir les données

Un serveur de base de données unique finit toujours par manquer de quelque chose : de disponibilité (c'est une seule machine, qui peut tomber en panne), de capacité de lecture, de capacité d'écriture, de stockage, ou de proximité avec des utilisateurs situés sur d'autres continents. Il existe deux façons complémentaires de répartir les données sur plusieurs machines, et il est important de ne pas les confondre. La réplication conserve une *copie des mêmes données* sur plusieurs nœuds, pour la disponibilité, le passage à l'échelle des lectures et une latence plus faible près des utilisateurs. Le partitionnement, appelé *sharding* dans de nombreux produits, *découpe les données* pour que chaque nœud en détienne un sous-ensemble, pour un débit d'écriture et un stockage qui dépassent les capacités d'une seule machine. Les systèmes réels font généralement les deux : chaque partition est répliquée.

Les deux ont de lourdes conséquences sur l'exactitude. La réplication pose la question de ce que voit un lecteur pendant que les copies divergent, ce qui est le sujet des modèles de cohérence, de CAP et de PACELC. Le partitionnement pose la question de ce qui arrive aux requêtes, aux index et aux transactions qui s'étendent sur plusieurs partitions. Cette section explique les mécanismes de façon assez approfondie pour juger les promesses d'une base de données et concevoir une clé de sharding.

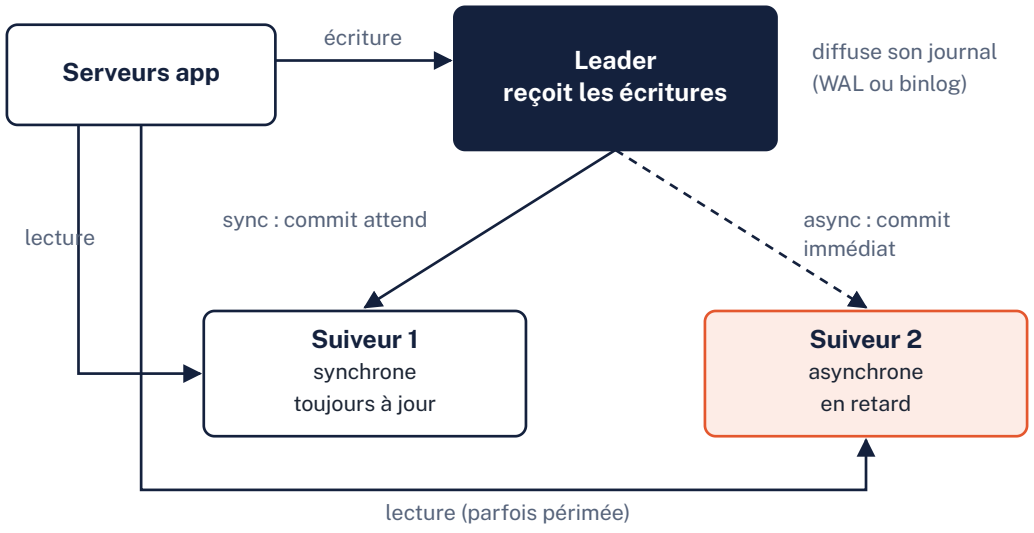
Réplication leader-suiveur

Le schéma le plus courant est la réplication leader-suiveur (aussi appelée primaire-réplique ; la documentation plus ancienne dit maître-esclave). Un nœud, le leader, accepte toutes les écritures. Il enregistre chaque modification dans son journal (le journal d'écriture anticipée de PostgreSQL, le journal binaire de MySQL) et transmet ce journal en continu aux suiveurs, qui

appliquent les modifications dans le même ordre. Les lectures peuvent aller au leader ou aux suiveurs. La réplication en continu (streaming replication) de PostgreSQL, la réplication MySQL, les groupes de disponibilité de SQL Server, les replica sets de MongoDB et la plupart des bases de données cloud managées fonctionnent ainsi.

Le choix de conception clé est le moment où le leader annonce qu'un commit a réussi. Avec la réplication **synchrone**, il attend qu'un suiveur confirme avoir reçu la modification, si bien qu'une écriture validée survit à la perte du leader ; le prix est que chaque commit inclut un aller-retour réseau, et que si le suiveur synchrone est hors service, les écritures se bloquent. Avec la réplication **asynchrone**, le leader confirme immédiatement et transmet la modification ensuite ; les commits sont rapides et les suiveurs peuvent être très éloignés, mais si le leader meurt, les dernières modifications qui n'avaient atteint aucun suiveur sont perdues, alors même que le client a été informé de leur validation. Un compromis courant, appelé semi-synchrone dans MySQL, attend un suiveur et laisse les autres être asynchrones. À la banque, perdre un paiement confirmé est inacceptable, si bien qu'au moins une copie synchrone (ou une base de données fondée sur le consensus) est la norme ; pour la base de données des cartes de la plateforme de livraison de repas, quelques secondes de perte potentielle peuvent être un prix acceptable pour la vitesse. C'est, sous une forme concrète, le compromis sur le RPO vu dans disponibilité et fiabilité.

Un leader avec un suiveur synchrone et un suiveur asynchrone

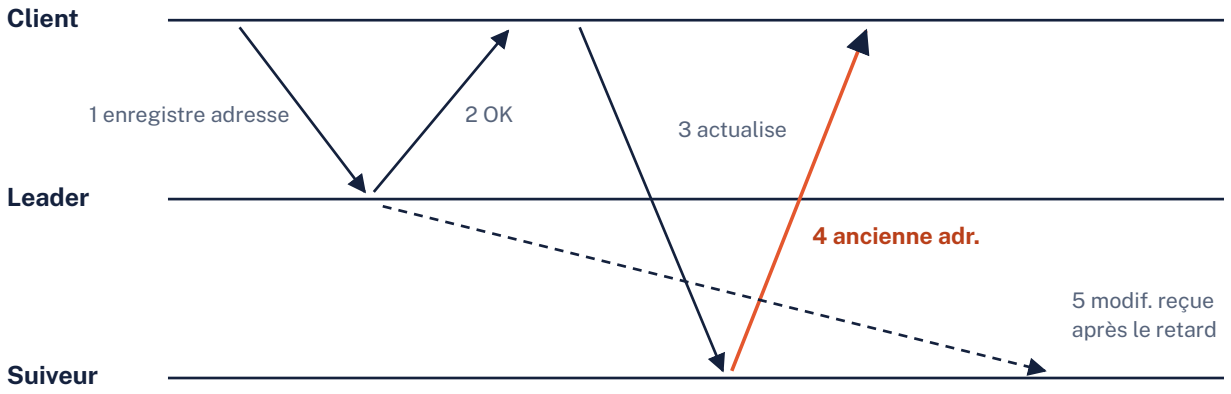


Toutes les écritures vont au leader ; son journal de modifications est transmis aux suiveurs, et le commit n'attend que le suiveur synchrone.

Le retard de réplication et les anomalies qu'il provoque

Les suiveurs asynchrones n'ont généralement que quelques millisecondes de retard, mais sous charge, pendant une grosse mise à jour par lots ou sur un réseau lent, le retard de réplication peut atteindre des secondes, voire des minutes. Lire sur les suiveurs produit alors des anomalies que les utilisateurs remarquent. Douglas Terry et ses collègues ont défini en 1994 les deux premières garanties ci-dessous, la lecture de ses propres écritures et les lectures monotones, comme des *garanties de session* ; la troisième compte surtout lorsque les données sont aussi partitionnées.

Lecture de ses propres écritures non respectée



Correctif : lire ses changements récents sur le leader, ou attendre que le suiveur soit à jour

La modification de la cliente est confirmée par le leader, mais pas encore appliquée sur le suiveur qui sert sa lecture suivante.

Les anomalies dues au retard et leurs remèdes

Garantie	Ce qui tourne mal sans elle	Remèdes courants
Lecture de ses propres écritures (read-your-writes, read-after-write)	Une cliente modifie son adresse de livraison, recharge la page et voit l'ancienne : on dirait que l'enregistrement a échoué, alors elle recommence.	Lire sur le leader les données que l'utilisateur vient peut-être de modifier (par exemple pendant une minute après sa dernière écriture) ; ou noter la position du journal du leader au moment de l'écriture et ne lire sur un suiveur qu'une fois qu'il a appliqué cette position.
Lectures monotones	L'app d'un livreur s'actualise deux fois ; la première lecture tombe sur un suiveur à jour et montre une commande comme affectée, la seconde tombe sur un suiveur en retard et la commande semble de nouveau non affectée. Le temps paraît reculer.	Acheminer les lectures de chaque utilisateur vers le même suiveur (par exemple en hachant l'id de l'utilisateur), en ne basculant que lorsqu'il est indisponible.
Lectures à préfixe cohérent	Les lecteurs voient une réponse avant la question, ou l'événement «livrée» d'une commande avant «récupérée», parce que les deux ont été écrits sur des partitions différentes qui n'ont pas le même retard.	Écrire les données liées par causalité sur la même partition, ou suivre explicitement les dépendances causales.

Aucune de ces anomalies n'existe si toutes les lectures vont au leader ; le remède le plus simple consiste donc souvent à envoyer au leader les lectures qui doivent être fraîches, et à n'utiliser les suiveurs que pour les lectures qui tolèrent des données périmées : listes de produits, rapports, flux de recherche. Concevez cela délibérément, requête par requête, au lieu d'envoyer par défaut «toutes les lectures vers les répliques».

Failover et split brain

Le failover (basculement) remplace un leader défaillant par un suiveur. Il comporte trois étapes : détecter que le leader est tombé en panne (généralement par un délai d'expiration), choisir le nouveau leader (idéalement le suiveur le plus à jour, souvent par un protocole de consensus comme Raft entre les nœuds ou par un coordinateur externe comme etcd ou ZooKeeper), et rediriger les écritures vers lui. Chaque étape peut mal tourner. Un délai d'expiration trop court déclenche un failover lors d'une brève coupure réseau ou d'un pic de charge, ce qui aggrave la situation ; un délai trop long prolonge la panne. Avec la réplication

asynchrone, les écritures que l'ancien leader n'avait pas encore transmises sont perdues, et si l'ancien leader revient en les détenant toujours, elles entrent en conflit avec les nouvelles écritures.

La défaillance la plus dangereuse est le split brain (cerveau divisé) : deux nœuds se croient tous deux leader et acceptent tous deux des écritures, si bien que les données divergent d'une manière qu'il peut être impossible de fusionner automatiquement. Il survient généralement après une partition réseau dans laquelle l'ancien leader est isolé mais toujours en vie. La protection exige un quorum (un nœud ne peut être leader que tant qu'une majorité est d'accord), le *fencing* (l'autorité du nouveau leader est imposée par un jeton toujours croissant que le stockage vérifie, ou l'ancien leader est mis hors tension de force, une technique surnommée STONITH : shoot the other node in the head, « abattre l'autre nœud »), et des clients qui suivent de façon fiable le leader actuel.

L'incident de GitHub du 21 octobre 2018 montre à quel point c'est difficile en pratique. Une panne réseau a coupé pendant 43 secondes la liaison entre son point d'interconnexion de la côte Est des États-Unis et son principal centre de données de la côte Est. Son outillage de failover, fonctionnant comme prévu, a promu des bases de données primaires de la côte Ouest. Quand la connectivité est revenue, chaque côte détenait des écritures qui manquaient à l'autre, si bien que GitHub ne pouvait pas simplement rebasculer vers la côte Est ; pour protéger l'intégrité des données, l'entreprise a choisi de restaurer puis de réconcilier, et le service a été dégradé pendant 24 heures et 11 minutes. Le failover automatique entre régions avait transformé une coupure de 43 secondes en une journée entière de rétablissement.

Les sauvegardes, et la preuve qu'elles se restaurent

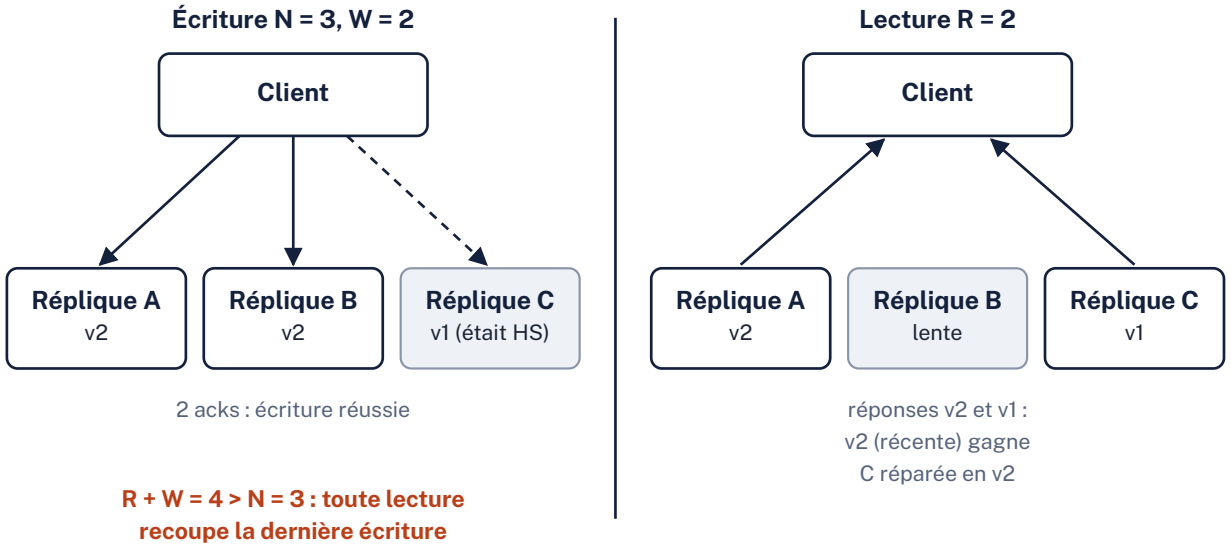
La réplication n'est pas une sauvegarde : une réplique copie fidèlement un `DELETE` erroné. La sauvegarde de base de PostgreSQL et l'archivage continu du WAL permettent de revenir à un moment choisi juste avant l'erreur, et les copies doivent se trouver là où un même incident ne peut pas atteindre les deux. Une sauvegarde ne compte qu'une fois restaurée. GitLab l'a appris en janvier 2017 : après la suppression par erreur de données de production, ses sauvegardes `pg_dump` régulières se sont révélées en échec à cause d'une incompatibilité de versions, les e-mails d'échec avaient été rejetés et le bucket de stockage était vide. GitLab a récupéré à partir d'un snapshot manuel vieux d'environ six heures. Automatisez donc des restaurations régulières dans un environnement isolé, vérifiez les données, mesurez le temps nécessaire au regard du RTO et du RPO, et déclenchez une alerte lorsqu'une sauvegarde ou un test de restauration échoue ou ne s'exécute pas.

Réplication multi-leader et sans leader

Dans la réplication multi-leader, plusieurs nœuds acceptent des écritures et se répliquent entre eux. On l'utilise pour plusieurs centres de données (chaque région écrit localement, ce qui évite aux utilisateurs une latence transocéanique), pour les clients qui travaillent hors ligne (une app d'agenda ou de notes est en fait un leader sur chaque appareil) et pour l'édition collaborative. Le problème inévitable, ce sont les **conflits d'écriture** : deux leaders modifient le même enregistrement en même temps. Les résolutions possibles sont last write wins (la dernière écriture l'emporte ; simple, mais qui écarte des données sans avertissement et dépend des horloges), la fusion des valeurs, la conservation des deux en demandant à une personne de trancher, ou l'utilisation de types de données conçus pour fusionner automatiquement (CRDT, conflict-free replicated data types). L'approche la plus fiable est d'éviter les conflits : donner à chaque enregistrement une région d'origine qui est seule à accepter ses écritures, comme le pays de résidence d'un client.

La réplication sans leader, popularisée par l'article Dynamo d'Amazon (DeCandia et ses collègues, 2007) et utilisée dans Apache Cassandra, ScyllaDB et Riak, n'a aucun leader. Le client, ou un nœud coordinateur agissant pour lui, envoie chaque écriture aux N répliques d'une clé et la considère comme réussie quand W d'entre elles l'ont confirmée ; il envoie chaque lecture aux répliques et attend R réponses, en retenant la version la plus récente. Si R + W > N, chaque ensemble de lecture recoupe chaque ensemble d'écriture réussie sur au moins une réplique, si bien qu'une lecture voit normalement la dernière écriture confirmée. Les répliques qui ont manqué des écritures sont réparées par la *réparation à la lecture* (read repair : le coordinateur réécrit la version la plus récente quand il en voit une périmée) et par une *anti-entropie* en arrière-plan qui compare les répliques. Cassandra expose ces choix requête par requête sous forme de niveaux de cohérence comme ONE, QUORUM et ALL.

Une écriture et une lecture en quorum avec N = 3



Avec W = 2 et R = 2, au moins une réplique de chaque lecture a vu la dernière écriture confirmée.

On accorde souvent trop de confiance aux quorums. R + W > N ne rend pas à lui seul le système linéarisable : les écritures concurrentes, les écritures qui ont échoué sur certaines répliques mais réussi sur d'autres, les *sloppy quorums* (quorums approximatifs, qui écrivent sur des nœuds de remplacement pendant les pannes) et last write wins avec des horloges décalées permettent tous des données périmées ou perdues. Le chapitre 6 détaille l'arithmétique des quorums et ces cas limites dans CAP et PACELC. Les stockages sans leader sont excellents pour une haute disponibilité en écriture malgré les pannes et entre centres de données ; ils conviennent mal aux invariants comme «ne jamais vendre deux fois la même place», sauf s'ils proposent une fonction de transaction légère ou de consensus pour ces opérations.

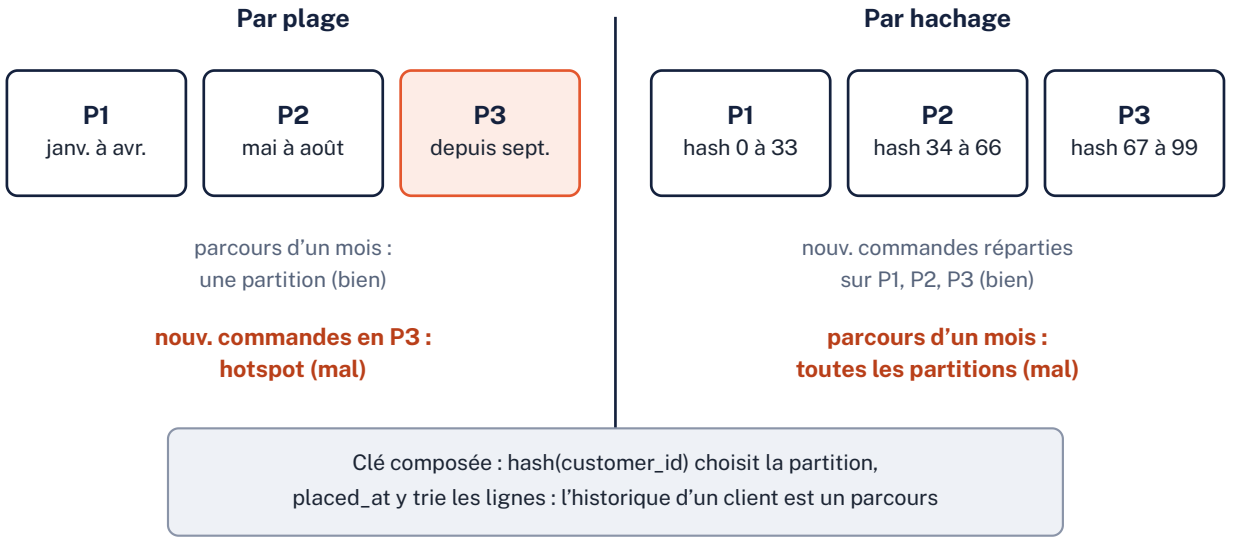
Partitionnement : par plage de clés ou par hachage

Le partitionnement affecte chaque enregistrement à une partition selon sa **clé de partition** (ou clé de sharding), et l'objectif est de répartir les données et la charge de manière égale. Une partition qui reçoit une charge disproportionnée est un point chaud, et une répartition inégale s'appelle un *déséquilibre* (skew). Il existe deux schémas de base.

Le **partitionnement par plage de clés** donne à chaque partition une plage contiguë de clés, comme les volumes d'une encyclopédie. Les parcours de plages sont efficaces parce que les clés adjacentes sont stockées ensemble (toutes les commandes d'octobre dans une ou deux partitions), et les partitions peuvent être scindées quand elles grossissent, comme dans Bigtable, HBase, Spanner et le sharding par plage de MongoDB. Le danger est une clé qui croît avec le temps : si la clé est un horodatage ou un numéro de commande séquentiel, chaque nouvelle écriture tombe sur la dernière partition, qui devient un point chaud pendant que les

autres restent inactives. Le **partitionnement par hachage** applique une fonction de hachage à la clé et affecte des plages de valeurs de hachage aux partitions, ce qui répartit uniformément même les clés séquentielles. Le prix à payer est que les requêtes par plage sur la clé doivent désormais interroger toutes les partitions. De nombreux stockages combinent les deux avec une *clé composée* : Cassandra hache la clé de partition pour choisir le nœud et trie les lignes au sein de la partition selon des colonnes de clustering, si bien que `(customer_id, placed_at)` regroupe les commandes d'un client, dans l'ordre chronologique, sur un seul nœud.

Partitionnement par plage ou par hachage pour les commandes

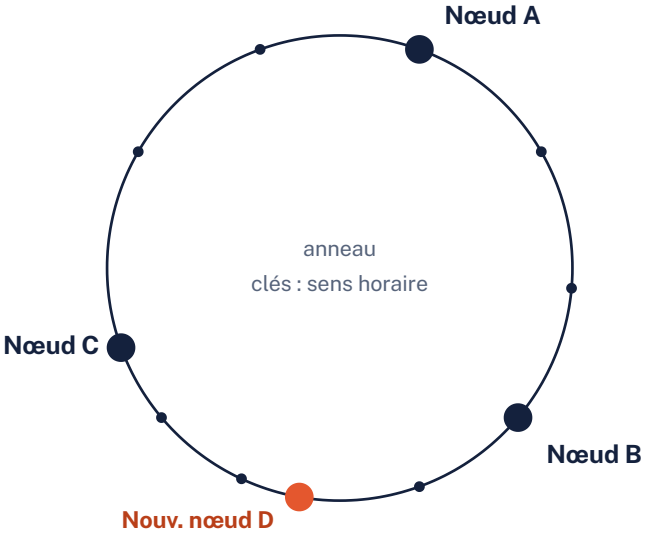


Le partitionnement par plage garde les plages ensemble mais concentre les écritures chronologiques ; le hachage répartit les écritures mais disperse les plages.

Hachage cohérent et rééquilibrage

La façon évidente d'associer les valeurs de hachage aux nœuds, `hash(key) mod N`, a un défaut rédhibitoire : quand N change, presque toutes les clés se déplacent. Passer de 4 à 5 nœuds déplace environ 80 % des clés (toute clé dont le reste diffère pour 4 et pour 5), et passer de 10 à 11 environ 91 %, ce qui revient à copier presque tout le jeu de données simplement pour ajouter une machine. Le hachage cohérent (consistent hashing), introduit par David Karger et ses collègues en 1997 pour les caches web, place à la fois les nœuds et les clés sur un espace de hachage circulaire ; chaque clé appartient au nœud suivant dans le sens des aiguilles d'une montre. Un nœud ajouté ne reprend que les clés situées entre lui et son prédécesseur, environ 1/N du total, et un nœud retiré cède ses clés à son voisin. Avec peu de nœuds, les arcs sont inégaux ; on attribue donc à chaque nœud physique de nombreux *nœuds virtuels* (tokens) en différents points de l'anneau, ce qui équilibre la charge et permet à un nouveau nœud de prendre un peu à chaque nœud existant. Dynamo et Cassandra utilisent ce schéma.

Le hachage cohérent sur un anneau



Ajout du nœud D

hash mod N, de 4 à 5 nœuds :
≈ 80 % des clés bougent

hachage cohérent :
seules les clés entre B et D
bougent (de C à D),
soit environ 1/N du total

nœuds virtuels : un serveur
a de nombreux points sur l'anneau
pour lisser la charge

Les clés appartiennent au nœud suivant dans le sens horaire : un nouveau nœud reprend un seul arc au lieu de tout redistribuer.

Deux autres stratégies de rééquilibrage sont courantes. Avec un **nombre fixe de partitions**, vous créez au départ beaucoup plus de partitions que de nœuds (par exemple 1024 pour 10 nœuds) et vous affectez des partitions entières aux nœuds ; ajouter un nœud déplace quelques partitions entières et ne rehache jamais les clés. Les shards d'Elasticsearch, les vBuckets de Couchbase et de nombreux schémas de sharding faits maison procèdent ainsi, et la décision principale est de choisir un nombre de partitions assez grand pour la croissance future. Avec le **partitionnement dynamique**, comme dans HBase et MongoDB, une partition se scinde quand elle dépasse un seuil de taille, et les deux moitiés peuvent aller sur des nœuds différents. Quel que soit le schéma, le rééquilibrage déplace de grandes quantités de données sur le réseau pendant que le système sert du trafic ; de nombreuses équipes préfèrent donc qu'une personne approuve chaque déplacement, et elles le brident pour protéger la latence.

Les requêtes doivent aussi être acheminées vers la bonne partition : soit n'importe quel nœud accepte une requête et la transmet (les nœuds Cassandra apprennent l'anneau par gossip), soit une couche de routage connaît la carte des partitions (`mongos` de MongoDB), soit la bibliothèque cliente détient la carte. La carte elle-même est généralement conservée dans un service de coordination comme etcd ou ZooKeeper, ou dans le groupe de consensus propre à la base de données.

SI VOUS DÉBUTEZ

Pas à pas : ajouter un nœud sans tout déplacer

L'idée la plus difficile de cette section est de comprendre pourquoi la façon évidente de répartir les clés entre les nœuds s'effondre quand on ajoute un nœud, et comment le hachage cohérent l'évite. Suivez les commandes de la plateforme de livraison de repas à mesure qu'un cluster grandit.

- Une fonction de hachage transforme la clé de chaque commande en nombre. Supposons, pour rester simple, que les nombres aillent de 0 à 11, et qu'une commande aille sur le nœud « hash mod 3 », le reste de la division par 3. La commande 7 se trouve sur le nœud 1.
- Un quatrième nœud est ajouté et la règle devient « hash mod 4 ». La commande 7 doit désormais aller sur le nœud 3.
- Sur les douze nombres, seuls 0, 1 et 2 donnent le même reste pour 3 et pour 4. Les neuf autres commandes, soit 75 % d'entre elles, doivent être copiées sur un autre nœud simplement pour ajouter une machine.
- Le hachage cohérent place plutôt les nœuds et les clés sur un cercle de valeurs de hachage, et chaque clé appartient au nœud suivant dans le sens des aiguilles d'une montre. Dans le cercle dessiné plus haut, le nouveau nœud D se situe entre B et C.
- Seules les clés de l'arc entre B et D changent de propriétaire, passant de C à D. Toutes les autres clés restent en place : c'est environ la part de données d'un seul nœud qui se déplace, et non la majeure partie.
- Avec peu de points, les arcs sont inégaux et D ne soulage que C ; chaque serveur reçoit donc de nombreux points sur le cercle, appelés nœuds virtuels, et un nouveau serveur prend une petite part à chacun des serveurs existants.
- **La leçon :** jugez une règle de placement à ce qui se passe quand le cluster change, et pas seulement à la régularité avec laquelle elle répartit les clés aujourd'hui. Le hachage cohérent, ou de nombreuses partitions créées dès le départ et déplacées en bloc, limite le déplacement quand un nœud arrive.

Clés chaudes et déséquilibre

Le hachage répartit uniformément les *clés*, pas la *charge*. Si une clé est bien plus active que les autres, sa partition est chaude quel que soit le schéma : le compte d'une célébrité sur un réseau social, le restaurant phare d'une chaîne nationale pendant une promotion, ou le compte de règlement propre à une banque, qui intervient dans chaque paiement. Les stockages managés explicitent les limites ; Amazon DynamoDB, par exemple, documente un plafond par partition de 3000 unités de lecture et 1000 unités d'écriture par seconde. Les parades

consistent notamment à mettre en cache les lectures chaudes, à scinder une clé chaude en lui ajoutant un petit suffixe aléatoire (écritures de `restaurant:88#0` à `restaurant:88#9`, au prix de la lecture des dix clés et de leur combinaison), à déplacer un très gros locataire vers des partitions dédiées et à agréger les compteurs en mémoire avant de les écrire. Toutes ajoutent de la complexité : mesurez donc d'abord quelles clés sont réellement chaudes.

SaaS multi-tenant : silo, pool et bridge

Supposons que la plateforme de livraison de repas vende aussi son logiciel de commande à des chaînes de restaurants, chaque chaîne étant un **locataire** (tenant). La question clé est ce que les locataires partagent. Le SaaS Lens d'AWS nomme trois modèles. Dans le modèle **silo**, chaque locataire dispose de ressources dédiées, par exemple sa propre base de données, ce qui donne l'isolation la plus forte et le coût le plus élevé, avec de nombreuses bases à provisionner et à migrer. Dans le modèle **pool**, les locataires partagent tout et chaque ligne porte un identifiant de locataire, ce qui est le moins cher et le plus simple à exploiter. Le modèle **bridge** mêle les deux par composant : un service peut être en silo pour des raisons réglementaires ou de voisin bruyant, tandis qu'un autre est en pool. Beaucoup de produits les mêlent aussi par client, en gardant les petites chaînes en pool et en donnant sa propre base de données à une chaîne nationale.

Dans un pool, un seul filtre oublié montre à une chaîne les commandes de sa rivale. La **sécurité au niveau des lignes** (row-level security) déplace le contrôle dans la base de données. Dans PostgreSQL, une politique comme `USING (tenant_id = current_setting('app.tenant_id')::uuid)` limite chaque requête sur la table. Les pièges comptent. Les superutilisateurs et les rôles `BYPASSRLS` ignorent les politiques, tout comme les propriétaires de tables sauf si `FORCE ROW LEVEL SECURITY` est défini : l'application doit donc se connecter avec un rôle ordinaire. Avec le pooling par transaction de PgBouncer, un `SET` de session n'est pas pris en charge : définissez donc le locataire avec `SET LOCAL` à l'intérieur de chaque transaction. Les contrôles d'unicité et de clé étrangère contournent les politiques, ce qui peut révéler qu'une valeur existe. Traitez la politique comme un filet de sécurité, avec des tests qui tentent de franchir la frontière entre locataires, et n'oubliez pas que les clés de cache, les files et les index de recherche ont eux aussi besoin du locataire.

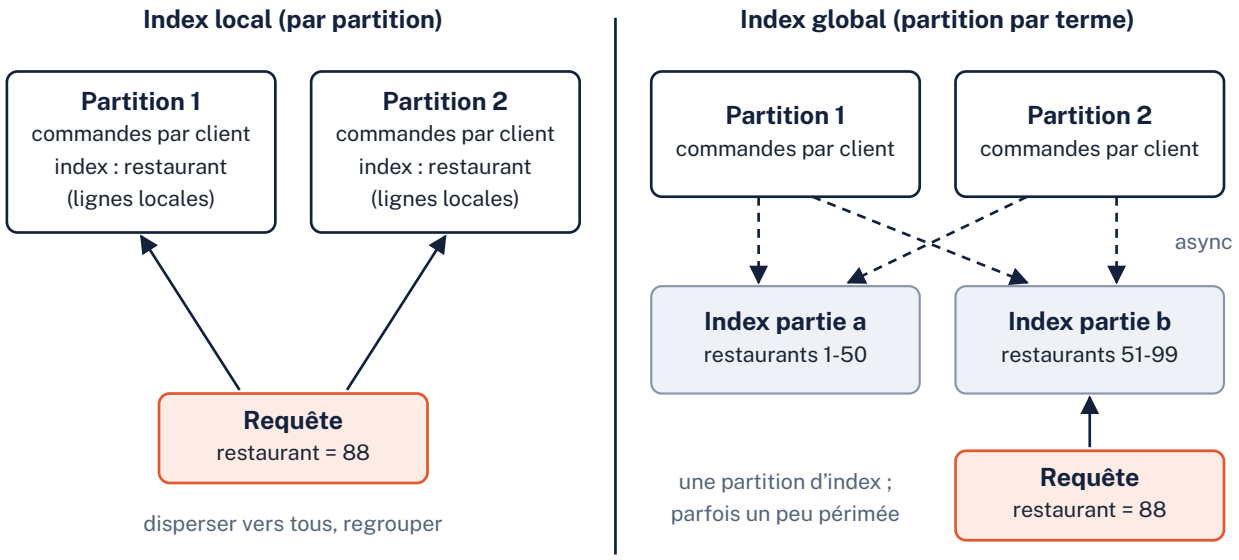
Un **voisin bruyant** (noisy neighbour) est un locataire dont l'activité nuit aux autres, comme la promotion nationale d'une chaîne qui épuise la capacité partagée. Les recommandations de Microsoft traitent cela comme une gouvernance des ressources : quotas par locataire et limitation de débit, plafonds sur les opérations coûteuses, tâches lourdes déplacées en dehors des pics, et télémétrie étiquetée avec l'identifiant du locataire pour que la cause soit visible. Les locataires peuvent ensuite être rééquilibrés ou déplacés vers une capacité dédiée.

Quand une seule base de données ne suffit pas, le **sharding par locataire** utilise l'identifiant du locataire comme clé de partition. Dans Citus, par exemple, chaque table porte la colonne du locataire, qui entre aussi dans les clés primaires et étrangères : les lignes d'un locataire restent ainsi ensemble, et les requêtes et transactions à locataire unique s'exécutent sur un seul nœud comme du PostgreSQL ordinaire. Le hachage ne résout pas le déséquilibre : une très grande chaîne peut donc avoir besoin de son propre shard ; le rééquilibreur de Citus ne tient pas compte des locataires qui partagent un shard, et de tels déplacements exigent de l'outillage. Les rapports couvrant tous les locataires ont leur place dans un entrepôt de données. Décidez tôt pour les restaurations aussi : Microsoft note que restaurer un locataire à partir d'une base partagée peut obliger à restaurer toute la base ailleurs et à récupérer sélectivement les données de ce locataire. Les cellules poussent la même idée plus loin, pour toute la pile.

Index secondaires entre partitions

Le partitionnement fonctionne sans heurt pour les recherches par clé de partition. Les requêtes portant sur autre chose ont besoin d'un index secondaire, et il existe deux façons de le partitionner. Un **index local** (partitionné par document) vit avec chaque partition et ne couvre que les lignes de celle-ci. Les écritures ne mettent à jour qu'une partition, mais une requête sur l'attribut indexé doit être envoyée à toutes les partitions et les résultats fusionnés (*scatter-gather*, dispersion puis regroupement), si bien que sa latence est fixée par la partition la plus lente et que son coût augmente avec le nombre de partitions. Un **index global** (partitionné par terme) est lui-même partitionné selon la valeur indexée, si bien qu'une requête lit une seule partition d'index ; mais une seule écriture peut alors devoir mettre à jour des partitions d'index situées sur d'autres nœuds, ce qui se fait généralement de façon asynchrone, de sorte que l'index peut brièvement être en retard sur les données. DynamoDB illustre les deux : un index secondaire local partage la clé de partition de la table, tandis qu'un index secondaire global a la sienne et offre une cohérence à terme.

Index secondaires locaux et globaux



Un index local rend les écritures peu coûteuses et disperse les requêtes ; un index global rend les requêtes ciblées et les mises à jour d'index distantes et généralement asynchrones.

EXEMPLE COMMENTÉ

Exemple commenté : dimensionner et choisir une clé de sharding pour les commandes

La plateforme de livraison de repas s'est étendue à 30 pays et traite 10 millions de commandes par jour, avec un pic en soirée à trois fois le rythme moyen. Chaque commande génère environ 10 écritures au cours de sa vie (création, lignes, changements de statut, références de paiement) et occupe environ 4 Ko, index compris. Les clients peuvent consulter tout leur historique de commandes ; la réglementation exige que les commandes puissent être retrouvées pendant six ans, mais le support n'a besoin d'un accès en ligne qu'aux deux dernières années.

Débit d'écriture	10 000 000 / 86 400 s = environ 116 commandes par seconde en moyenne ; pic d'environ 350 par seconde ; à 10 écritures chacune, environ 3500 écritures par seconde au pic. Un seul primaire bien réglé peut absorber cela ; le débit à lui seul n'impose pas le sharding.
Croissance du stockage	10 000 000 x 4 Ko = 40 Go par jour, environ 14,6 To par an, environ 29 To pour deux ans en ligne.
Faut-il partitionner ?	Si seuls 90 jours devaient rester en ligne (3,6 To), un primaire plus des répliques, avec un partitionnement des tables par date et un archivage vers l'entrepôt de données, serait la réponse ennuyeuse et correcte. Avec deux ans en ligne, soit 29 To, les temps de restauration et de reconstruction des répliques sur un seul nœud deviennent inacceptables : l'équipe passe donc au sharding.
Dimensionner les shards	Viser environ 2 To par shard pour qu'une réplique puisse être reconstruite en quelques heures : 29 To / 2 To = environ 15, donc 16 shards, chacun composé d'un leader et de deux suiveurs. Créer 1024 partitions logiques réparties sur les 16 shards, pour que la croissance se traduise par des déplacements de partitions entières, sans rehachage. Écritures au pic par shard : 3500 / 16 = environ 220 par seconde.
Candidate : order_id (hachage)	Des écritures parfaitement réparties, mais l'historique d'un client comme la file en direct d'un restaurant deviennent du scatter-gather sur 16 shards. Rejetée comme clé principale de placement.
Candidate : placed_at (plage)	Idéale pour les rapports par période, mais chaque nouvelle commande s'écrit dans la partition la plus récente : un point chaud garanti. Rejetée.
Candidate : restaurant_id	La file d'un restaurant est locale, mais les grandes chaînes et les promotions créent des clés chaudes, et l'historique des clients se disperse. Rejetée.

Candidate : pays ou ville	Favorise la résidence des données et les requêtes locales, mais une grande ville peut représenter 15 % de toutes les commandes : un déséquilibre. Conservée uniquement comme premier niveau grossier pour la résidence (clusters UE et hors UE), pas comme clé de sharding.
Décision	Partitionner par `hash(customer_id)`, avec des lignes triées par `placed_at` dans chaque partition. L'id de commande intègre le numéro de partition du client, si bien qu'une recherche par id de commande va directement vers un seul shard.
Couvrir les autres requêtes	La file en direct du restaurant est servie par un modèle de lecture distinct, indexé par restaurant et construit à partir des événements de commande (un index global délibéré, à cohérence à terme). L'analytique s'exécute dans l'entrepôt de données. Les transactions entre clients ne sont pas nécessaires : chaque commande ne concerne qu'un client.

Les chiffres ont changé la question. Le débit n'a jamais exigé le sharding ; ce sont les données conservées en ligne qui l'ont imposé. Réduire la fenêtre en ligne aurait entièrement supprimé ce besoin, ce qui mérite d'être discuté avec le métier avant de construire un système partitionné.

La clé de sharding a été choisie à partir des requêtes les plus importantes et de leurs besoins de cohérence, pas à partir du seul modèle de données. Chaque autre requête a reçu un chemin explicite, conçu séparément.

Le choix d'une clé de sharding est l'une des décisions sur lesquelles il est le plus difficile de revenir : la changer plus tard signifie déplacer chaque ligne. Consignez-la dans un ADR avec les volumes et les requêtes qu'elle supposait.

Erreurs courantes

Traiter les répliques comme des sauvegardes. Une réplique copie fidèlement un `DELETE` erroné en quelques millisecondes ; les sauvegardes et la restauration à un instant donné restent nécessaires (chapitre 0). **Lire sur les répliques par défaut**, sans décider quelles requêtes tolèrent le retard. **Se fier à un failover automatique jamais répété**, ou capable de promouvoir un nœud alors que l'ancien leader accepte encore des écritures. **Partitionner trop tôt** : une base de données partitionnée complique les jointures, les transactions, les changements de schéma et l'exploitation, et les nœuds uniques modernes sont très puissants ; épuisez d'abord l'indexation, la mise en cache, les répliques en lecture, l'archivage et le passage à l'échelle vertical. **Partitionner selon une clé qui croît de façon monotone**, ce qui concentre chaque insertion sur une seule partition. **Oublier les chemins d'accès secondaires**, et découvrir après le découpage qu'une requête importante doit désormais interroger chaque shard.

Comme le montre la suite de ce chapitre, les bonnes alternatives passent souvent par des données dérivées : des modèles de lecture, des caches et des stockages analytiques, chacun construit pour un type de requête.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi la réplication asynchrone risque-t-elle de perdre des écritures dont les clients ont été informés qu'elles étaient validées ?

Le leader confirme avant qu'un suiveur ait la modification ; s'il tombe en panne avant de la transmettre, le suiveur promu n'a jamais reçu ces écritures.

▼ Un utilisateur enregistre une modification et voit aussitôt l'ancienne valeur. Quelle garantie n'est pas respectée, et quel est un remède possible ?

La lecture de ses propres écritures (read-your-writes). Servez depuis le leader les lectures de cet utilisateur portant sur des données récemment modifiées, ou attendez que le suiveur ait appliqué la position de l'écriture dans le journal.

▼ Pourquoi hash mod N est-il un mauvais moyen d'affecter les clés aux nœuds ?

Changer N déplace la plupart des clés (environ 80 % en passant de 4 à 5 nœuds), alors que le hachage cohérent ou des partitions logiques fixes n'en déplacent qu'environ 1/N.

▼ Quel est le compromis entre index secondaires locaux et globaux ?

Les index locaux gardent les écritures sur une seule partition mais obligent les requêtes à interroger toutes les partitions ; les index globaux permettent aux requêtes de cibler une seule partition d'index mais exigent des mises à jour d'index entre partitions, généralement asynchrones.

▼ À vous : Une plateforme reçoit 6 millions de commandes par jour, chacune causant 10 écritures, avec un pic à trois fois la moyenne. Chaque shard absorbe sans risque 500 écritures par seconde. Combien de shards ? Comparez l'identifiant client et l'horodatage de commande comme clé de sharding.

Moyenne : 60 000 000 / 86 400 = 694 écritures par seconde. Pointe : environ 2083. Shards : 2083 / 500 = 4,2, donc 5, plus de la marge pour la croissance. Un hachage de l'ID client répartit la charge uniformément et garde ensemble l'historique d'un client, la requête la plus importante. Une clé d'horodatage envoie toutes les nouvelles écritures vers un seul shard, créant un point chaud. Les rapports par plage de dates exigent alors du scatter-gather ou, mieux, un magasin d'analyse séparé. Ajoutez des répliques à chaque shard pour la disponibilité.

SOURCES

- DeCandia et al.: Dynamo: Amazon's Highly Available Key-value Store (SOSP 2007): https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- GitHub: October 21 post-incident analysis (2018): https://github.blog/2018-10-30-oct21-post-incident-analysis/
- PostgreSQL documentation: High Availability, Load Balancing, and Replication (log-shipping standby servers): https://www.postgresql.org/docs/current/warm-standby.html
- Amazon DynamoDB Developer Guide: Best practices for designing and using partition keys effectively: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-partition-key-design.html
- Microsoft Azure Architecture Center: Sharding pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/sharding
- AWS SaaS Lens: Silo, pool, and bridge models: https://docs.aws.amazon.com/wellarchitected/latest/saas-lens/silo-pool-and-bridge-models.html
- PostgreSQL documentation: Row Security Policies: https://www.postgresql.org/docs/current/ddl-rowsecurity.html
- PostgreSQL documentation: SET: https://www.postgresql.org/docs/current/sql-set.html
- PgBouncer: Features: https://www.pgbouncer.org/features.html
- Microsoft Learn: Noisy Neighbor antipattern: https://learn.microsoft.com/en-us/azure/architecture/antipatterns/noisy-neighbor/noisy-neighbor
- Citus documentation: Multi-tenant applications: https://docs.citusdata.com/en/stable/use_cases/multi_tenant.html
- Microsoft Learn: Multitenant data approaches: https://learn.microsoft.com/en-us/azure/architecture/guide/multitenant/approaches/storage-data
- GitLab: Postmortem of database outage of January 31 (2017): https://about.gitlab.com/blog/2017/02/10/postmortem-of-database-outage-of-january-31/
- PostgreSQL documentation: Continuous archiving and point-in-time recovery: https://www.postgresql.org/docs/current/continuous-archiving.html

SECTION 7.5 SUR 7.8 · AVANCÉ · 26 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

CQRS et event sourcing

AVANT DE LIRE

Cette section s'appuie sur :

- 5.7 Conception pilotée par le domaine
- 6.4 Pilotée par les événements
- 7.3 Transactions

Termes déjà vus qu'elle utilise : Agrégat, Contexte délimité (bounded context), Cohérence à terme (eventual consistency), Contrôle de concurrence optimiste

L'IDÉE EN BREF

Une banque tient un grand livre dans lequel chaque dépôt et chaque retrait est inscrit, ligne après ligne, et rien n'est jamais effacé ; les erreurs se corrigent en ajoutant une ligne de correction. Votre solde est simplement le total de toutes les lignes. Votre relevé mensuel est un résumé soigné, établi à partir de ce grand livre pour être facile à lire. L'event sourcing stocke les données comme le fait le grand livre : sous la forme d'une liste de choses qui se sont produites. CQRS sépare le grand livre (pour effectuer des changements) des relevés (pour consulter les informations). Les deux sont puissants, et les deux ajoutent une réelle complexité.

CE QU'IL FAUT SAVOIR

- La séparation commande-requête (Bertrand Meyer, 1988) veut qu'une méthode change l'état ou renvoie une réponse, mais pas les deux ; CQRS (Greg Young, vers 2010) applique la même séparation à des modèles entiers.
- Dans CQRS, un modèle d'écriture applique les règles métier aux commandes, tandis que des projections tiennent à jour un ou plusieurs modèles de lecture dénormalisés pour les requêtes ; les modèles de lecture sont des données dérivées qui peuvent être reconstruites.
- L'event sourcing stocke chaque changement sous la forme d'un événement immuable, nommé au passé, dans un flux en ajout seul (append-only) propre à chaque agrégat ; l'état courant est un pli à gauche (left fold) déterministe des événements.
- L'event sourcing offre une piste d'audit complète, des requêtes temporelles, des modèles de lecture reconstructibles et rétroactifs, et la relecture (replay) pour le débogage ; les snapshots raccourcissent les longs plis, mais ne sont qu'un cache.

- Les coûts sont réels : complexité, cohérence à terme des modèles de lecture, versionnement des événements par upcasting ou par de nouveaux types d'événements, et effacement des données personnelles d'un journal immuable, généralement par crypto-shredding.
- Martin Fowler avertit que CQRS ajoute une complexité risquée pour la plupart des systèmes ; appliquez l'un ou l'autre patron à des contextes délimités précis dont les besoins de lecture et d'écriture sont réellement différents, jamais à un système entier par défaut.

De la séparation commande-requête à CQRS

L'idée part de peu. Dans *Object-Oriented Software Construction* (première édition en 1988), Bertrand Meyer a proposé la séparation commande-requête : chaque méthode doit être soit une commande, qui change l'état et ne renvoie rien, soit une requête, qui renvoie une réponse et ne change rien. Poser une question ne devrait jamais changer la réponse. Martin Fowler fait remarquer que la règle admet des exceptions raisonnables (dépiler un élément d'une pile modifie la pile et renvoie l'élément), mais, comme règle par défaut, elle rend le code plus facile à comprendre, car les requêtes peuvent être appelées librement, dans n'importe quel ordre, sans effets de bord.

Vers 2010, Greg Young a étendu le principe des méthodes à des modèles entiers et a appelé le résultat Command Query Responsibility Segregation (CQRS), la séparation des responsabilités entre commandes et requêtes. Au lieu d'un seul modèle objet, et généralement d'un seul ensemble de tables, servant à la fois aux mises à jour et aux lectures, le système dispose d'un **modèle d'écriture** qui traite les commandes et applique les règles métier, et d'un ou plusieurs **modèles de lecture** façonnés pour les questions que posent les utilisateurs. L'idée clé est qu'écrire et lire sont deux métiers différents. L'écriture exige des règles strictes, de petites transactions et une structure normalisée, pour qu'un fait soit stocké une seule fois. La lecture exige de la vitesse, des formes dénormalisées qui correspondent aux écrans et aux rapports, et doit souvent absorber bien plus de trafic. Un modèle unique qui essaie de bien faire les deux a tendance à mal faire les deux dès que le domaine devient riche.

Ce que CQRS change concrètement

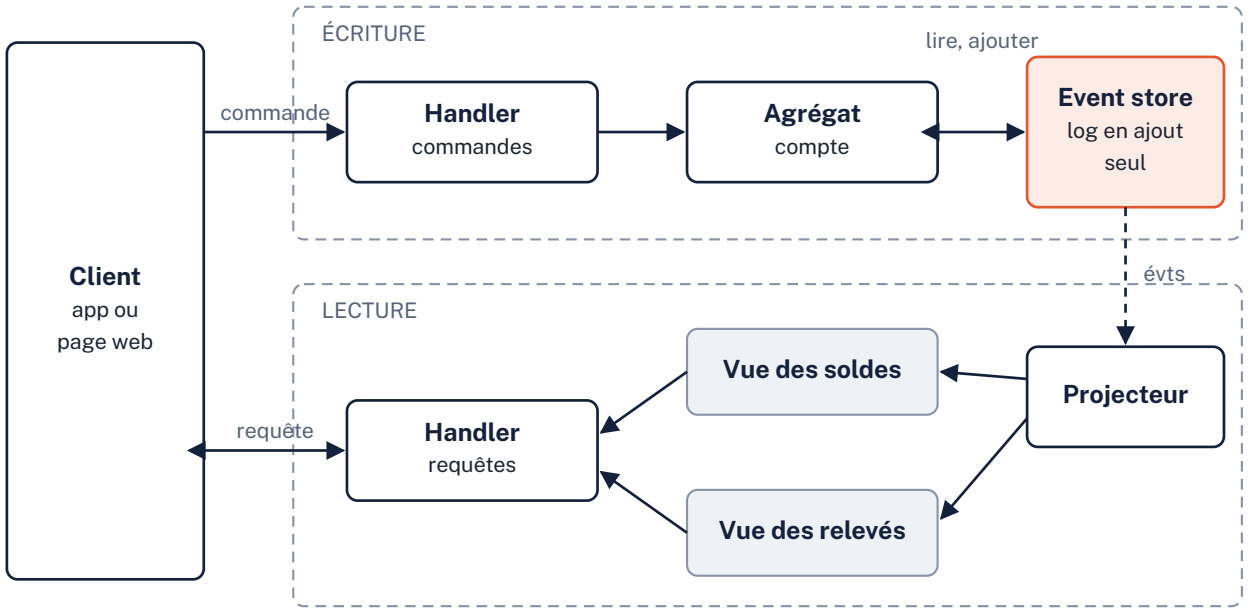
Côté écriture, une commande comme `WithdrawMoney(account 42, £50)` arrive à un gestionnaire de commandes (command handler). Le gestionnaire charge l'agrégat concerné (le groupe de données qui doit rester cohérent au sein d'une même transaction, un terme de la conception pilotée par le domaine présenté à la section 5.7), lui demande d'appliquer les règles métier et enregistre le résultat. Les commandes sont des demandes et peuvent être refusées : le compte peut être gelé ou les fonds insuffisants. Côté lecture, une requête comme «montre-

moi les 30 dernières transactions avec les soldes cumulés» reçoit sa réponse d'un modèle de lecture construit exactement pour cet écran, sans jointures, sans logique métier et sans verrous sur les tables d'écriture.

Il faut bien que quelque chose tienne les modèles de lecture à jour. Cette tâche revient à une projection : du code qui écoute les changements côté écriture et met à jour un modèle de lecture. Une projection peut maintenir une table des soldes, une autre un index de recherche des transactions, d'autres encore un document de relevé mensuel et un feature store de scoring de fraude, chacun à partir du même flux de changements. Les modèles de lecture sont donc des **données dérivées** : on peut toujours les jeter et les reconstruire à partir du côté écriture. C'est cette propriété qui permet d'ajouter sans risque un nouveau modèle de lecture l'an prochain, ou de modifier la forme d'un ancien.

CQRS se décline par degrés, et le degré le plus léger est souvent tout ce dont un système a besoin. Au plus léger, les modèles d'écriture et de lecture sont des classes différentes au-dessus de la même base de données, les lectures étant servies par des vues ou par du code de requête dédié. Au niveau intermédiaire, les modèles de lecture vivent dans des tables séparées, mises à jour dans la même transaction que l'écriture. Au plus lourd, les modèles de lecture vivent dans des stockages différents (un moteur de recherche, une base de données orientée documents, un cache) et sont mis à jour de façon asynchrone à partir d'événements, ce qui apporte scalabilité et flexibilité, mais aussi la cohérence à terme. CQRS n'exige ni event sourcing, ni courtier de messages (message broker), ni microservices ; ce sont des décisions distinctes, souvent prises ensemble, mais pas nécessairement.

CQRS avec event sourcing



Les commandes et les requêtes empruntent des chemins différents ; l'event store est la source de vérité, et chaque modèle de lecture en est dérivé par un projecteur.

SI VOUS DÉBUTEZ

Si vous débutez

Pensez au grand livre d'une banque et à un relevé bancaire. Le grand livre est l'endroit où chaque dépôt et chaque retrait est inscrit, une ligne à la fois, selon des règles strictes. Le relevé est un résumé établi à partir du grand livre dans un format facile à lire. Vous n'écrivez jamais sur le relevé pour déplacer de l'argent, et vous ne lisez jamais le grand livre ligne par ligne pour connaître votre solde. CQRS donne au logiciel la même séparation : un modèle pour changer les choses, d'autres modèles pour les consulter. L'event sourcing va un peu plus loin et fait du grand livre lui-même la seule chose stockée de façon permanente.

Event sourcing : l'état comme suite de faits

La plupart des systèmes stockent l'état courant. Quand Asha retire £120, la ligne de `accounts` passe de £500 à £380 et l'ancienne valeur disparaît, à moins que quelqu'un n'écrive aussi une ligne d'audit. L'event sourcing, décrit par Martin Fowler en 2005, renverse cette logique : le système stocke chaque changement sous la forme d'un événement immuable dans un journal en ajout seul, et l'état courant est ce que l'on obtient en rejouant les événements dans l'ordre. Les événements sont la source de vérité ; le solde est un calcul.

Les événements sont nommés au passé, car ils enregistrent des faits qui se sont déjà produits et ont été acceptés : `AccountOpened`, `MoneyDeposited`, `MoneyWithdrawn`. Une commande (`WithdrawMoney`) est une demande qui peut être refusée ; un événement (`MoneyWithdrawn`) ne peut plus être refusé, seulement compensé par un événement ultérieur. Chaque agrégat a son propre flux d'événements, numéroté à partir de la version 1, et l'event store garantit que les événements d'un flux sont conservés dans l'ordre.

Reconstruire l'état est un **pli à gauche** (left fold) : partir de rien, appliquer le premier événement, appliquer le deuxième au résultat, et ainsi de suite. La fonction d'application doit être pure et déterministe : elle ne peut appeler ni une horloge, ni un générateur de nombres aléatoires, ni un autre service, car rejouer les mêmes événements doit toujours donner le même état. Toute valeur venue de l'extérieur (un taux de change, un horodatage, un score de fraude) est enregistrée dans l'événement au moment où elle est décidée pour la première fois.

L'écriture suit un cycle fixe. Le gestionnaire de commandes charge le flux, le replie en état courant, exécute la règle métier et ajoute les nouveaux événements avec la version qu'il s'attend à écrire. Si une autre commande a ajouté des événements au même flux entre-temps, l'ajout échoue et le gestionnaire réessaie avec un état à jour. C'est la concurrence optimiste de la section 7.3, appliquée flux par flux, et c'est pourquoi deux retraits simultanés ne peuvent pas dépenser tous deux les mêmes £100.

EXEMPLE COMMENTÉ

Exemple commenté : un compte bancaire sous forme d'événements

Le compte courant d'Asha est le flux `account-42`. Les montants sont stockés en pence, comme la section 0.2 le recommande pour l'argent. Cinq événements ont été ajoutés ; elle essaie ensuite de retirer £200.

v1 AccountOpened (découvert autorisé £0)	État : solde £0,00, version 1
v2 MoneyDeposited £500,00	£0,00 + £500,00 = £500,00, version 2
v3 MoneyWithdrawn £120,00	£500,00 − £120,00 = £380,00, version 3
v4 MoneyDeposited £75,50	£380,00 + £75,50 = £455,50, version 4
v5 MoneyWithdrawn £300,00	£455,50 − £300,00 = £155,50, version 5
Commande : WithdrawMoney £200,00	Règle : le solde moins le montant ne doit pas descendre sous moins le découvert autorisé. £155,50 − £200,00 = −£44,50, sous £0. Commande refusée ; aucun événement n'est ajouté.
Commande : WithdrawMoney £20,00	£155,50 − £20,00 = £135,50 est autorisé. Le gestionnaire ajoute v6 MoneyWithdrawn £20,00, en s'attendant à ce que la version 6 soit libre.
Requête temporelle : solde après v3	Replier seulement v1 à v3 : £380,00. Aucune table d'historique n'a été nécessaire.
Question d'audit : pourquoi le solde est-il de £135,50 ?	Les six événements sont la réponse complète, dans l'ordre, chacun indiquant qui, quand et pourquoi.

La commande refusée n'a laissé aucune trace dans le flux. Si la banque doit enregistrer les refus (pour l'analyse de la fraude, par exemple), elle ajoute un événement distinct `WithdrawalRefused`, qui ne modifie pas le solde.

Un grand livre en partie double est le plus ancien système en event sourcing qui soit : les comptables n'effacent jamais une ligne, ils passent une écriture de correction. Les domaines bancaires se prêtent donc naturellement à l'event sourcing, ce qui explique qu'on l'enseigne souvent avec un compte.

```
# A bank account as a stream of events, folded into current state.
# Amounts are in pence so that no floating-point rounding creeps in.
from functools import reduce

events = [
    {"v": 1, "type": "AccountOpened",  "overdraft_limit": 0},
    {"v": 2, "type": "MoneyDeposited", "amount": 50000},
    {"v": 3, "type": "MoneyWithdrawn", "amount": 12000},
    {"v": 4, "type": "MoneyDeposited", "amount": 7550},
    {"v": 5, "type": "MoneyWithdrawn", "amount": 30000},
]

def apply(state, e):                      # pure: old state + event → new state
    if e["type"] == "AccountOpened":
        return {"balance": 0, "limit": e["overdraft_limit"], "version": e["v"]}
    sign = 1 if e["type"] == "MoneyDeposited" else -1
    return {**state, "balance": state["balance"] + sign * e["amount"], "version": e["v"]}

def decide_withdraw(state, amount):       # command side: check the rule, emit events
    if state["balance"] - amount < -state["limit"]:
        raise ValueError("insufficient funds")
    return [{"v": state["version"] + 1, "type": "MoneyWithdrawn", "amount": amount}]

now = reduce(apply, events, None)         # {'balance': 15550, 'limit': 0, 'version': 5}
as_at_v3 = reduce(apply, events[:3], None) # temporal query: balance 38000
```

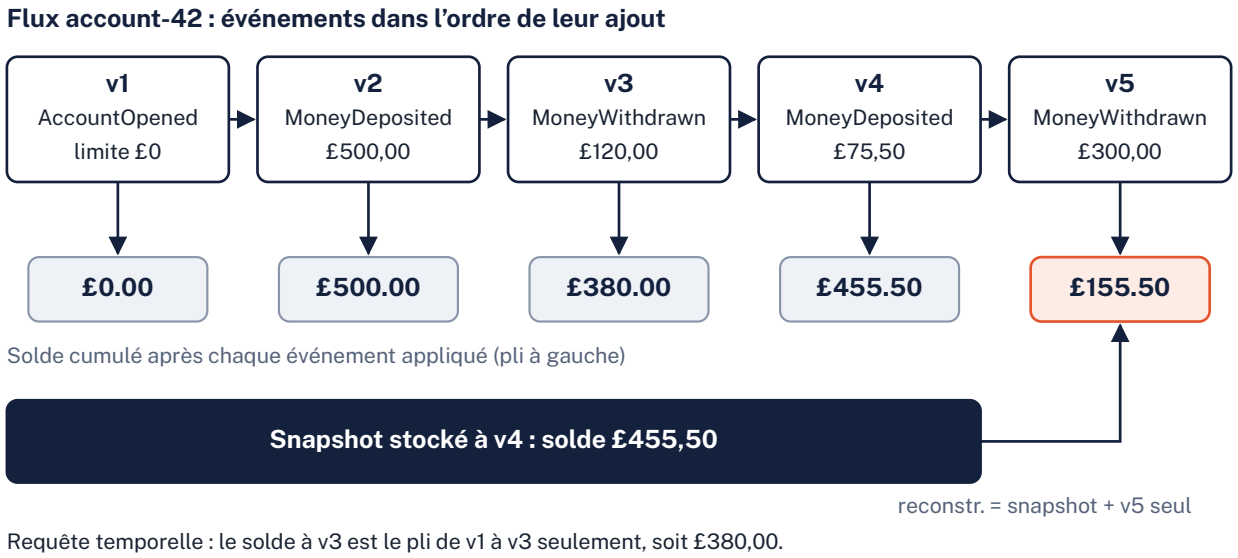
Replier le flux en état courant, décider d'une commande d'après cet état, et répondre à une requête temporelle en ne repliant qu'une partie du flux.

```
-- An event store can be an ordinary table. The primary key enforces
-- optimistic concurrency: only one writer can append version 6.
CREATE TABLE account_events (
  account_id  uuid        NOT NULL,
  version     integer     NOT NULL,
  event_type  text        NOT NULL,
  data        jsonb       NOT NULL,
  recorded_at timestamptz NOT NULL DEFAULT now(),
  PRIMARY KEY (account_id, version)
);

-- The command handler read the stream up to version 5, decided, and now appends:
INSERT INTO account_events (account_id, version, event_type, data)
VALUES ('2f9c0d6e-4a51-4c7e-9a43-6d1f2b7e8a10', 6, 'MoneyWithdrawn', '{"amount": 2000}');
-- If a concurrent command already appended version 6, this fails with a
-- unique violation; the handler reloads the stream and decides again.
```

Un event store minimal dans une base de données relationnelle : la clé primaire composite transforme un ajout concurrent en conflit détectable.

Pli, snapshots et requêtes temporelles



Requête temporelle : le solde à v3 est le pli de v1 à v3 seulement, soit £380,00.

L'état courant est le pli de tous les événements ; un snapshot raccourcit le pli, et replier un préfixe du flux répond aux questions sur le passé.

Ce que vous apporte l'event sourcing

Une piste d'audit complète par construction. Dans un système fondé sur l'état, le journal d'audit est une seconde écriture que quelqu'un peut oublier, et il peut contredire les données. Dans un système en event sourcing, le journal est la donnée : il ne peut donc pas diverger. Les régulateurs de la banque, de l'assurance et de la santé y attachent une grande valeur.

Requêtes temporelles. «Quel était le solde à 23 h 59 le 31 mars ?» ou «Quelles commandes étaient au statut PREPARING quand l'écran de la cuisine est tombé en panne ?» deviennent des plis sur un préfixe de chaque flux, plutôt qu'un travail d'enquête sur les sauvegardes.

Reconstructions et nouveaux modèles de lecture. Comme les modèles de lecture sont dérivés, une équipe peut corriger un bug dans une projection, supprimer ce qu'elle a produit et rejouer tous les événements pour obtenir une vue corrigée. Elle peut aussi construire un modèle de lecture auquel personne n'avait pensé lors de l'enregistrement des événements, comme un rapport sur la fréquence à laquelle les clients retirent de l'argent liquide dans l'heure qui suit le versement de leur salaire, sur tout l'historique.

Débogage et analyse de scénarios (what-if). Copier un flux dans un environnement de test et le rejouer reproduit exactement l'état qui a causé un bug. Rejouer les mêmes événements avec des règles modifiées montre ce qu'auraient fait les nouvelles règles.

Une source naturelle d'événements d'intégration. Les événements peuvent alimenter d'autres services, même s'il vaut mieux traiter les événements stockés dans un flux comme un détail privé du service propriétaire. Les publier directement couple chaque consommateur au modèle interne ; beaucoup d'équipes les traduisent plutôt en un ensemble plus restreint d'événements d'intégration publics, conçus comme les contrats publiés et durables décrits à la section 6.4.

Snapshots et flux longs

Replier cinq événements est instantané ; replier 400 000 événements pour un compte commerçant très actif à chaque commande ne l'est pas. Un snapshot (instantané) stocke l'état replié d'un flux à une version connue, si bien que le chargement devient «lire le dernier snapshot, puis n'appliquer que les événements qui le suivent». Les snapshots sont un cache, pas une source de vérité : ils peuvent être supprimés et recréés à tout moment, en arrière-plan, sans arrêter le système. Prenez-en un tous les N événements ou selon un calendrier, et stockez avec eux la version du code d'application, car un snapshot produit par un ancien code est faux dès que la logique du pli change.

Avant de recourir aux snapshots, vérifiez la conception des flux. Des flux très longs signifient souvent que l'agrégat est trop gros. Un compte bancaire qui vit 40 ans peut être modélisé comme une série de flux plus courts, un par période de relevé, chacun ouvert par un événement `BalanceCarriedForward` qui enregistre le solde de clôture de la période précédente. Cela reflète la façon dont les comptables clôturent leurs comptes, et cela garde chaque flux assez court pour être chargé rapidement.

Les coûts, en toute honnêteté

Complexité et manque de familiarité. L'event sourcing change la façon dont chaque développeur de l'équipe pense la persistance. Des questions simples comme «quels clients habitent à Leeds ?» ne peuvent pas du tout être posées à l'event store ; elles exigent une projection. Les outils pour inspecter, migrer et réparer les event stores sont moins matures que pour les tables relationnelles, et moins d'ingénieurs ont fait tourner un tel système en production.

Cohérence à terme des modèles de lecture. Quand les projections s'exécutent de façon asynchrone, un utilisateur peut effectuer un retrait puis voir un solde qui ne l'inclut pas encore. Les parades habituelles consistent à renvoyer la nouvelle version du flux depuis la commande et à laisser le côté lecture attendre que sa projection ait traité cette version (une garantie de lecture de ses propres écritures, comme à la section 7.4) ; à mettre à jour l'interface utilisateur de manière optimiste à partir du résultat de la commande ; ou à mettre à jour les modèles de

lecture critiques de façon synchrone, dans la même transaction que l'ajout. Les projections doivent aussi être idempotentes et enregistrer la position qu'elles ont traitée, car des événements peuvent être livrés plus d'une fois après un crash.

Versionnement des événements. Les événements sont stockés pour toujours, si bien que leur schéma ne peut jamais être simplement modifié. Quand `MoneyDeposited` gagne un champ de devise, des années d'anciens événements ne l'ont toujours pas. Les techniques standard, cataloguées par Greg Young dans *Versioning in an Event Sourced System*, comprennent les lecteurs tolérants, qui fournissent des valeurs par défaut pour les champs manquants, l'upcasting (une fonction qui convertit une ancienne version d'événement dans la forme actuelle au moment de la lecture, afin que le code du domaine ne voie qu'une seule version), l'introduction d'un nouveau type d'événement à côté de l'ancien et, en dernier recours, la copie d'un flux entier dans un nouveau flux avec des événements transformés. La section 7.7 présente les règles générales de compatibilité.

Supprimer des données personnelles. Un journal immuable se heurte au droit à l'effacement prévu par la législation sur la protection des données. La technique courante est le crypto-shredding (effacement cryptographique) : les champs personnels des événements sont chiffrés avec une clé propre à chaque client, conservée dans un stockage de clés séparé, et l'effacement consiste à détruire cette clé afin que le texte chiffré ne puisse plus être lu. L'autre solution consiste à garder entièrement les données personnelles hors des événements et à ne stocker qu'une référence à un enregistrement qui peut être supprimé. Savoir si la destruction d'une clé vaut effacement relève d'une appréciation juridique plutôt que technique ; convenez donc de l'approche avec le délégué à la protection des données (DPD). La section 8.7 présente les obligations plus larges.

Charge d'exploitation. Reconstruire une projection sur un milliard d'événements peut prendre des heures ; les équipes ont donc besoin d'un moyen d'exécuter la nouvelle projection à côté de l'ancienne et de basculer quand elle a rattrapé son retard. Les event stores croissent sans limite, et le stockage ainsi que les temps de sauvegarde et de relecture croissent avec eux.

ATTENTION

L’avertissement de Fowler

L’article de Martin Fowler de 2011 sur CQRS est d’une franchise inhabituelle : il écrit que «pour la plupart des systèmes, CQRS ajoute une complexité risquée», et il recommande de ne l’utiliser que pour des parties précises d’un système, typiquement un contexte délimité dont les besoins de lecture et d’écriture sont réellement différents, jamais comme architecture du système entier. Cela vaut à plus forte raison pour l’event sourcing. Des équipes qui ont appliqué les deux partout parce qu’ils étaient à la mode ont fait état d’années de complexité accidentelle, en particulier dans des zones simples de type CRUD (création, lecture, mise à jour, suppression), comme les données de référence ou les profils utilisateur.

GUIDE DE DÉCISION

Guide de décision : cette partie du système doit-elle utiliser CQRS ou l'event sourcing ?

Posez ces questions pour un contexte délimité, pas pour le système entier.

Q1 **Cette zone se résume-t-elle surtout à de la création, de la lecture, de la mise à jour et de la suppression simples, avec peu de règles métier ?**

- **Oui** Utilisez un modèle classique avec une seule base de données. Aucun des deux patrons ne sera rentable ici.
- **Non** Continuez : le domaine a un véritable comportement qui mérite d'être modélisé.

Q2 **Les lectures et les écritures diffèrent-elles fortement par leur forme ou leur volume (par exemple, de nombreux écrans dénormalisés, de la recherche, ou mille lectures pour une écriture) ?**

- **Oui** Envisagez CQRS. Commencez par la forme la plus légère : un code de lecture séparé ou des vues sur la même base de données.
- **Non** Gardez un modèle unique ; ajoutez une réplique en lecture ou un cache si la charge de lecture est le seul problème.

Q3 **L'historique a-t-il une valeur en soi : audit réglementaire, requêtes temporelles, relecture avec une logique corrigée, ou nombreux futurs modèles de lecture à partir des mêmes faits ?**

- **Oui** Envisagez l'event sourcing pour ce contexte, idéalement un domaine de type grand livre, comme les comptes, les paiements ou le cycle de vie des commandes.
- **Non** Stockez l'état courant ; publiez des événements avec une outbox si d'autres services doivent être informés des changements.

Q4 **L'équipe a-t-elle l'expérience, les outils et le temps nécessaires pour gérer le versionnement des événements, la reconstruction des projections et la cohérence à terme ?**

- **Oui** Allez-y, gardez la zone en event sourcing réduite, et consignez la décision et ses raisons dans un ADR.
- **Non** Stockez l'état courant plus une table d'audit en ajout seul, qui apporte l'essentiel du bénéfice d'audit pour une fraction du coût.

Quand n'utiliser ni l'un ni l'autre, et erreurs courantes

Aucun des deux patrons ne convient à un domaine qui consiste surtout en formulaires sur des données (forms over data), à une petite équipe qui construit sa première version, ou à un système dont le seul problème est la charge de lecture (une réplique en lecture ou un cache

coûte bien moins cher). L'event sourcing est aussi mal adapté lorsque des données doivent être réellement supprimées à la demande et en grand volume, ou lorsque la plupart des requêtes recoupent de nombreux agrégats, puisque chacune de ces requêtes a besoin de sa propre projection.

Les erreurs courantes sont prévisibles. Les équipes appliquent les patrons à l'ensemble du système au lieu d'un seul contexte. Elles exposent les événements stockés comme contrat public, figeant ainsi leur modèle interne. Elles mettent des valeurs dérivées ou des appels externes dans la fonction d'application, si bien que la relecture donne des réponses différentes. Elles conçoivent les événements comme des blocs génériques `AccountUpdated`, qui indiquent que quelque chose a changé mais pas ce qui s'est passé ni pourquoi, et perdent l'essentiel de leur valeur. Enfin, elles oublient qu'un modèle de lecture est un cache : elles se mettent à y écrire directement, après quoi il ne peut plus être reconstruit.

Utilisée de façon sélective, la combinaison est puissante. Une plateforme de livraison de repas peut mettre en event sourcing le cycle de vie des commandes (passée, acceptée, en préparation, récupérée, livrée, remboursée), car les litiges et les remboursements exigent l'historique exact, tout en conservant les cartes, les profils de restaurants et les paramètres utilisateur dans des tables ordinaires. Une banque peut mettre en event sourcing le grand livre et les paiements tout en stockant de façon classique les coordonnées des clients, ce qui garde aussi l'essentiel des données personnelles hors du journal immuable.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi la fonction qui applique les événements à l'état ne doit-elle jamais appeler l'horloge système ?

Parce que rejouer les mêmes événements doit toujours produire le même état. Une horloge (ou un nombre aléatoire, ou un appel externe) ferait donner à la relecture des réponses différentes ; une telle valeur doit être capturée dans l'événement au moment où elle est décidée pour la première fois.

▼ Quelle est la différence entre la commande WithdrawMoney et l'événement MoneyWithdrawn ?

La commande est une demande qui peut être refusée par les règles métier ; l'événement est un fait enregistré qui a déjà été accepté et ne peut être que compensé par un événement ultérieur, jamais supprimé.

▼ **Une projection comporte un bug qui a produit des relevés erronés pendant trois mois. En quoi l'event sourcing aide-t-il ?**

Comme le modèle de lecture est dérivé, l'équipe corrige la projection, construit un nouveau modèle de lecture en rejouant tous les événements (généralement à côté de l'ancien) et bascule dessus, sans avoir à réparer les données à la main.

▼ **Un snapshot fait-il partie de la source de vérité ?**

Non. C'est un pli mis en cache d'un flux à une version donnée, qui peut être supprimé et recréé à partir des événements à tout moment.

▼ **À vous : L'architecte veut de l'event sourcing partout. Considérez un registre de points de fidélité exigeant une piste d'audit complète, et la table des horaires d'ouverture des salles. L'appliqueriez-vous à chacun ? Nommez deux coûts que vous signaleriez pour le registre.**

Utilisez-le pour le grand livre, si tant est que vous l'utilisiez : un historique en ajout seul fournit la piste d'audit, et les soldes peuvent être reconstruits en rejouant les événements. Les horaires des salles sont un simple CRUD : une table ordinaire suffit, et l'event sourcing n'ajouterait que du risque. Pour le grand livre, signalez le versionnage des événements, puisque les anciens événements restent à jamais, et l'effacement des données à caractère personnel d'un journal immuable, généralement par crypto-shredding. Les modèles de lecture seront aussi à cohérence éventuelle. Appliquez le pattern par contexte, jamais par défaut.

SOURCES

- Martin Fowler: CQRS (2011): https://martinfowler.com/bliki/CQRS.html
- Martin Fowler: Event Sourcing (2005): https://martinfowler.com/eaaDev/EventSourcing.html
- Martin Fowler: Command Query Separation: https://martinfowler.com/bliki/CommandQuerySeparation.html
- Microsoft Azure Architecture Center: Event Sourcing pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing
- Greg Young: Versioning in an Event Sourced System (Leanpub): https://leanpub.com/esversioning

SECTION 7.6 SUR 7.8 · INTERMÉDIAIRE · 24 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Stratégies de mise en cache et leurs pièges

AVANT DE LIRE

Cette section s'appuie sur :

- 0.2 Données et algorithmes
- 0.4 Comment les ordinateurs communiquent
- 2.4 Performances et scalabilité

Termes déjà vus qu'elle utilise : Latence, Débit (throughput), HTTP, Système de référence (system of record), Lecture de ses propres écritures (read-your-writes)

L'IDÉE EN BREF

Imaginez un café très fréquenté. Au lieu de moudre des grains pour chaque commande, le barista garde sur le comptoir un petit bocal de café fraîchement moulu. La plupart des commandes sont servies à partir du bocal en quelques secondes ; ce n'est que lorsqu'il est vide que quelqu'un va à la réserve. Le bocal est un cache. Il rend le service rapide et ménage la réserve, mais il soulève des questions : combien de temps le café moulu peut-il attendre avant d'avoir un goût de rassis, que se passe-t-il quand le bocal se vide pendant le coup de feu du matin, et qui pense à le jeter quand les grains changent ? Les caches logiciels soulèvent exactement les mêmes questions.

CE QU'IL FAUT SAVOIR

- Un cache échange une seconde copie des données, potentiellement périmée, contre une latence plus faible, un débit plus élevé et la protection de la source ; décidez d'abord ce qui peut être périmé, dans quelle mesure et pour qui.
- Cache-aside est le choix sûr par défaut ; read-through déplace le chargement dans la couche de cache ; write-through garde le cache à jour au prix d'écritures plus lentes ; write-behind risque de perdre des données et ne doit jamais contenir d'argent ; refresh-ahead convient à quelques clés très demandées.
- Combinez l'invalidation lors des changements avec un TTL comme filet de sécurité ; lors des écritures, mettez à jour la base de données, puis supprimez l'entrée du cache au lieu de l'écraser.
- Prévenez les ruées sur le cache avec le regroupement des requêtes, l'expiration avec gigue aléatoire (jitter), stale-while-revalidate, le rafraîchissement anticipé probabiliste et le préchauffage du cache.

- Le taux de succès pèse bien plus sur la charge de la base de données que sur la latence moyenne : chaque « neuf » de succès supplémentaire divise les requêtes à la base de données par environ dix, et un cache froid peut submerger une base de données dimensionnée pour le trafic mis en cache.
- Les caches existent dans les navigateurs, les CDN, les reverse proxies, les applications et les bases de données ; HTTP les contrôle avec Cache-Control, ETag et les requêtes conditionnelles (RFC 9111, RFC 9110).
- LRU et LFU sont les politiques d'éviction courantes ; les clés doivent inclure tout ce qui change la réponse, et les données personnelles ou financières doivent rester hors des caches partagés.

Pourquoi mettre en cache, et ce que cela coûte

Un cache conserve une copie des données dans un endroit plus rapide ou plus proche que leur emplacement d'origine, afin que les requêtes répétées ne paient pas le prix fort à chaque fois. La section 0.2 a présenté l'idée comme un échange d'espace contre du temps. À l'échelle de l'architecture, l'échange est plus intéressant, car un cache modifie trois qualités à la fois. Il réduit la latence (une consultation de Redis dans le même centre de données prend généralement bien moins d'une milliseconde ; une requête qui joint cinq tables peut prendre des dizaines de millisecondes). Il augmente le débit et protège la base de données, car chaque succès (hit) est une requête que la base de données ne voit jamais. Et il peut améliorer la disponibilité, car un cache capable de servir des données légèrement anciennes continue de fonctionner pendant que la source est lente ou indisponible.

Le prix se paie en exactitude et en complexité. Un cache est une seconde copie des données, et deux copies peuvent diverger. Chaque cache incarne donc une décision sur le degré de péremption autorisé pour une réponse, et pour qui. Une carte de restaurant vieille de cinq minutes est acceptable ; un solde bancaire vieux de cinq minutes, montré à un client qui se demande s'il peut se permettre un paiement, ne l'est pas. Les caches ajoutent aussi un composant qui peut tomber en panne, consomment de la mémoire qui coûte de l'argent et créent de nouveaux modes de défaillance, comme les ruées et les démarrages à froid, qui n'existeraient pas sans eux. La première question n'est donc pas «quel cache ?» mais «qu'est-ce qui peut être périmé, dans quelle mesure, et que se passe-t-il si c'est le cas ?»

L'efficacité d'un cache se mesure par son taux de succès : le nombre de succès divisé par le nombre total de consultations. Le taux de succès dépend de l'asymétrie du profil d'accès. Dans la plupart des systèmes grand public, le trafic de lecture est très asymétrique (une petite fraction des restaurants, des produits ou des comptes reçoit l'essentiel des lectures), si bien qu'un cache contenant quelques pour cent des données peut souvent répondre à la plupart des lectures. Un accès aléatoire uniforme, en revanche, se met mal en cache quoi que vous fassiez.

Les cinq patrons classiques

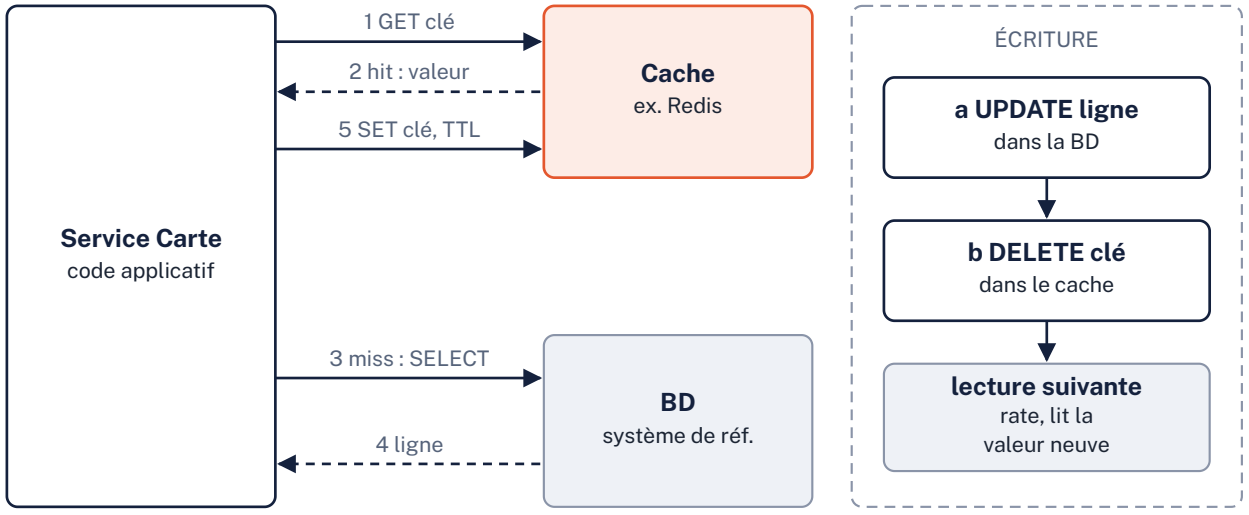
Les patrons de mise en cache diffèrent par l'acteur qui charge le cache, par le moment où les écritures atteignent la source, et donc par le degré de péremption et de durabilité des données. Le tableau les compare ; les paragraphes qui le suivent expliquent où chacun trouve sa place.

Comparaison des patrons de mise en cache

Patron	Fonctionnement des lectures	Fonctionnement des écritures	Points forts	Risques
Cache-aside (chargement paresseux)	L'application consulte le cache ; en cas de défaut (miss), elle lit la base de données et stocke le résultat	L'application écrit dans la base de données, puis supprime (ou met à jour) l'entrée du cache	Simple, ne met en cache que ce qui est utilisé, une panne du cache se traduit seulement par des lectures plus lentes	La première lecture après expiration est lente ; des situations de concurrence peuvent laisser des entrées périmées ; ruées
Read-through	L'application interroge le cache ; la bibliothèque de cache charge elle-même depuis la base de données en cas de défaut	Généralement combiné avec des écritures write-through ou cache-aside	La logique de chargement est à un seul endroit	Mêmes problèmes de péremption et de ruée ; la couche de cache doit connaître la source de données
Write-through	Comme read-through ou cache-aside	L'écriture va dans le cache et dans la base de données de façon synchrone avant l'accusé de réception	Le cache n'est jamais plus ancien que la base de données pour les données écrites à travers lui	Chaque écriture paie les deux latences ; met en cache des données qui ne seront peut-être jamais lues
Write-behind (write-back)	Servies depuis le cache	L'écriture est confirmée dès qu'elle est dans le cache, puis vidée vers la base de données plus tard, souvent par lots	Écritures très rapides ; absorbe les pics ; regroupe le travail de la base de données par lots	Perte de données si le cache tombe en panne avant le vidage ; retard de la base de données ; ordre et conflits
Refresh-ahead	Servies depuis le cache	Ne concerne pas les écritures : les entrées très demandées sont rechargées peu avant leur expiration	Les éléments populaires n'expirent jamais sous les yeux des utilisateurs	Rechargements inutiles d'éléments qui n'auraient pas été lus ; nécessite des statistiques d'accès

Cache-aside est le choix par défaut pour de bonnes raisons : il est facile à comprendre et échoue sans danger, car si le cache disparaît, l'application lit simplement la base de données (plus lentement). Read-through déplace la même logique dans une bibliothèque ou un produit de cache, ce qui est plus net, mais ne change rien au comportement. Write-through convient aux données lues peu après leur écriture, comme la session d'un utilisateur ou son panier. Write-behind est le plus risqué : il fait du cache le système de référence temporaire, si bien qu'il ne convient que lorsque perdre les dernières secondes d'écritures est acceptable (compteurs de vues, événements analytiques, scores de jeu) ou lorsque le cache lui-même est répliqué et durable. N'utilisez jamais write-behind pour de l'argent. Refresh-ahead est une optimisation pour un petit ensemble de clés très demandées, comme la liste des restaurants de la page d'accueil dans une grande ville animée.

Cache-aside en pratique



Les lectures remplissent le cache paresseusement en cas de défaut ; les écritures modifient d'abord la base de données, puis suppriment la copie en cache au lieu de l'écraser.

```
import json, random

TTL = 300  # seconds

def get_menu(restaurant_id):
    key = f"menu:{restaurant_id}:v3"            # version the key with the data shape
    cached = cache.get(key)
    if cached is not None:
        return json.loads(cached)                # hit
    with cache.lock(f"lock:{key}", timeout=5):  # coalesce: one loader per key
        cached = cache.get(key)                  # someone may have filled it meanwhile
        if cached is not None:
            return json.loads(cached)
        menu = db.load_menu(restaurant_id)       # miss: go to the system of record
        jitter = random.randint(0, TTL // 10)    # spread expiry times out
        cache.set(key, json.dumps(menu), ex=TTL + jitter)
        return menu

def update_price(restaurant_id, item_id, price):
    db.update_price(item_id, price)              # 1. change the system of record
    cache.delete(f"menu:{restaurant_id}:v3")     # 2. invalidate, do not overwrite
```

Cache-aside avec une clé versionnée, un verrou par clé pour regrouper les défauts concurrents, une expiration avec gigue aléatoire, et une invalidation par suppression après l'écriture en base de données.

Durée de vie et invalidation

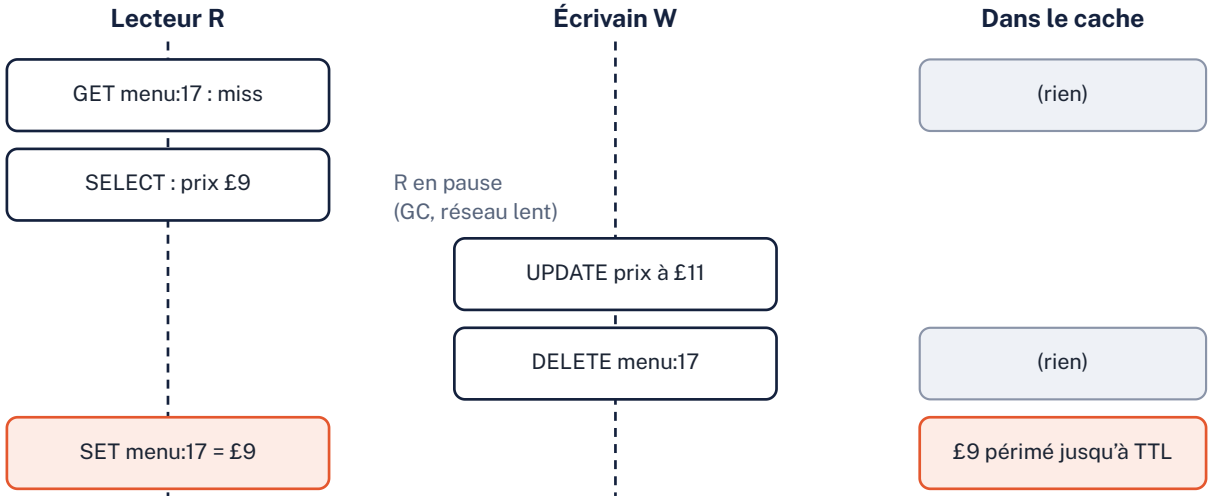
Il y a deux façons d'empêcher un cache de servir des données qui ont changé. La première est une durée de vie (TTL) : chaque entrée expire après une période fixe, ce qui borne la péremption sans aucune coordination. Un TTL de cinq minutes sur les cartes signifie qu'un changement de prix est visible partout en moins de cinq minutes. La seconde est l'invalidation explicite : quand la source change, quelque chose supprime ou met à jour la copie en cache. L'invalidation donne des données plus fraîches, mais exige que chaque chemin de code qui modifie les données connaisse chaque cache qui les contient, et c'est exactement ce qui se dérègle à mesure que les systèmes grandissent. Dans les systèmes distribués, l'invalidation est souvent pilotée par des événements : le service Carte publie `MenuChanged`, et chaque service qui en détient une copie supprime son entrée. La plupart des conceptions professionnelles combinent les deux : invalider lors des changements pour la fraîcheur, et garder un TTL comme filet de sécurité pour l'invalidation qui aura été manquée.

La difficulté est résumée par une boutade attribuée à Phil Karlton, de Netscape : «Il n'y a que deux choses difficiles en informatique : l'invalidation de cache et le nommage des choses.» C'est difficile parce que l'invalidation est un problème de systèmes distribués déguisé.

L'écriture en base de données et la suppression dans le cache sont deux opérations sur deux systèmes sans transaction commune ; l'une peut donc réussir pendant que l'autre échoue, et des opérations issues de requêtes différentes peuvent s'entrelacer.

Deux règles évitent la plupart des problèmes. D'abord, lors d'une écriture, mettez à jour la base de données, puis **supprimez** l'entrée du cache ; n'écrivez pas la nouvelle valeur dans le cache. Deux écrivains concurrents qui mettent chacun le cache à jour peuvent terminer dans l'ordre inverse de leurs commits en base de données et laisser l'ancienne valeur en cache indéfiniment, alors qu'une suppression ne peut causer qu'un défaut supplémentaire. Ensuite, acceptez que même la suppression à l'écriture laisse une étroite situation de concurrence, illustrée ci-dessous, et bornez-la avec un TTL. L'article de Facebook de 2013 sur le passage à l'échelle de memcache décrit des **baux (leases)** pour ce cas : le cache remet un jeton au client qui a subi le défaut, et refuse l'écriture ultérieure de ce client si la clé a été supprimée entre-temps.

La situation de concurrence du SET périmé



Supprimer à l'écriture réduit la fenêtre de données périmées sans la fermer ; un TTL borne la durée de survie de la valeur périmée.

SI VOUS DÉBUTEZ

Pas à pas : pourquoi supprimer plutôt que mettre à jour

L'idée qui demande le plus d'attention ici est l'invalidation du cache : modifier la base de données et modifier le cache sont des étapes distinctes sur deux systèmes, si bien que des requêtes qui se chevauchent peuvent laisser une ancienne valeur derrière elles. Deux changements d'adresse rapprochés montrent pourquoi il faut supprimer, et non écraser.

- La plateforme met en cache le profil de chaque client, adresse de livraison comprise, en cache-aside : une requête qui rate le cache charge le profil depuis la base de données et le stocke. Le cache indique Appartement 1. En moins d'une seconde, le client enregistre Appartement 2, puis corrige en Appartement 3 : ce sont les requêtes A et B.
- Supposons que chaque enregistrement fasse un commit dans la base de données, puis recopie son adresse par-dessus celle du cache. A enregistre Appartement 2, B enregistre Appartement 3 : la base de données est juste.
- Mais A se bloque entre ses deux étapes. B met Appartement 3 dans le cache ; puis A, en retard, y met Appartement 2. Le cache contredit maintenant la base de données, et chaque lecteur voit le mauvais appartement.
- Rien ne le répare : aucun autre enregistrement n'arrive, donc la mauvaise adresse reste jusqu'à l'expiration de l'entrée, ou indéfiniment si elle n'expire jamais.
- Faites maintenant en sorte que chaque enregistrement supprime le profil en cache. Quel que soit l'ordre dans lequel les suppressions arrivent, le cache finit vide ; la requête suivante rate le cache et stocke Appartement 3 depuis la base de données. Une suppression tardive coûte au plus un défaut supplémentaire.
- Supprimer réduit l'écart sans le combler : un lecteur lent peut encore stocker une ancienne valeur juste après une suppression, comme le montre la situation de concurrence ci-dessus. Une durée de vie sur chaque entrée limite la durée de survie de cette valeur.
- **La leçon :** lors d'un changement, faites d'abord le commit dans la base de données, puis supprimez la copie en cache au lieu de la remplacer, et donnez à chaque entrée une durée d'expiration pour rattraper ce que l'invalidation manque.

Ruées et effet de troupeau

Une ruée sur le cache (cache stampede) se produit quand une entrée populaire expire, ou qu'un nœud de cache redémarre vide, et que des centaines de requêtes concurrentes ratent toutes le cache au même instant. Chacune va à la base de données pour calculer la même valeur, et la base de données, dimensionnée pour le taux de défaut normal du cache, est submergée. Des réponses plus lentes font s'accumuler davantage de requêtes, et la surcharge peut se propager

en cascade. Le phénomène voisin de l'**effet de troupeau (thundering herd)** désigne de nombreux clients en attente réveillés en même temps pour se disputer une seule ressource. Facebook a indiqué que, pour un ensemble de clés particulièrement sujettes à ce problème, les baux ont fait passer le pic de requêtes à la base de données de 17 000 à 1300 par seconde.

Les parades se combinent bien. Le regroupement des requêtes (request coalescing, aussi appelé single flight) ne laisse qu'une requête par clé aller chercher la valeur à la source, pendant que les autres attendent son résultat ; au sein d'un processus, c'est une table de futures en cours ; entre processus, un verrou court dans le cache, comme dans le code ci-dessus. L'**expiration avec gigue aléatoire** (jitter) ajoute une durée aléatoire à chaque TTL afin que des entrées écrites ensemble (après un déploiement ou un vidage du cache) n'expirent pas ensemble. **Stale-while-revalidate** sert la valeur expirée à tout le monde pendant qu'une requête en arrière-plan en récupère une nouvelle, ce qui supprime entièrement le pic de latence quand une légère péremption est acceptable. L'**expiration anticipée probabiliste** (Vattani, Chierichetti et Lowenstein, VLDB 2015) fait décider à chaque lecteur, au hasard et avec une probabilité qui augmente à l'approche de l'expiration, de rafraîchir l'entrée en avance, de sorte qu'une requête la rafraîchit généralement avant qu'elle n'expire réellement. Enfin, le **préchauffage** d'un nouveau cache à partir d'une liste de clés très demandées, avant qu'il ne reçoive du trafic, évite la ruée du démarrage à froid après un redémarrage.

EXEMPLE COMMENTÉ

Exemple commenté : ce que le taux de succès fait à la latence et à la charge de la base de données

Le service Carte de la plateforme de livraison de repas reçoit 20 000 lectures de carte par seconde au pic du soir. Une consultation du cache prend 0,5 ms ; une requête à la base de données prend 8 ms. En cas de défaut, le service paie les deux (8,5 ms) et stocke le résultat de façon asynchrone. La base de données peut soutenir environ 3000 requêtes de ce type par seconde avant que la latence ne grimpe en flèche.

Formule	Latence moyenne = 0,5 + (1 − h) × 8 ms. Charge de la base de données = (1 − h) × 20 000 requêtes par seconde.
Aucun cache	8 ms en moyenne ; 20 000 requêtes par seconde, presque sept fois ce que la base de données peut soutenir.
Taux de succès 80 %	0,5 + 0,2 × 8 = 2,1 ms ; 4000 requêtes par seconde : encore au-dessus de la capacité, donc les requêtes s'accumulent dans la file d'attente de la base de données et la latence augmente.
Taux de succès 95 %	0,5 + 0,05 × 8 = 0,9 ms ; 1000 requêtes par seconde : confortable.
Taux de succès 99 %	0,5 + 0,01 × 8 = 0,58 ms ; 200 requêtes par seconde.
De 95 % à 99 %	La latence moyenne ne s'améliore que de 0,32 ms, mais la charge de la base de données est divisée par cinq, de 1000 à 200 requêtes par seconde.
Le cluster de cache redémarre vide	Le taux de succès tombe près de 0 % pendant un moment : 20 000 requêtes par seconde frappent une base de données dimensionnée pour 3000. Sans regroupement, préchauffage ni limitation de débit, la base de données s'effondre.
Percentiles	À 95 % de succès, 5 % des requêtes prennent 8,5 ms, donc le p95 est à la limite et le p99 vaut 8,5 ms, pas 0,9 ms.

Le taux de succès compte bien plus pour la base de données que pour la latence moyenne : chaque « neuf » de succès supplémentaire divise la charge de la base de données par environ dix.

Un cache sans lequel le système ne peut pas fonctionner n'est plus une optimisation ; c'est une dépendance critique, et sa défaillance doit être prévue comme la section 2.5 le prévoit pour toute autre.

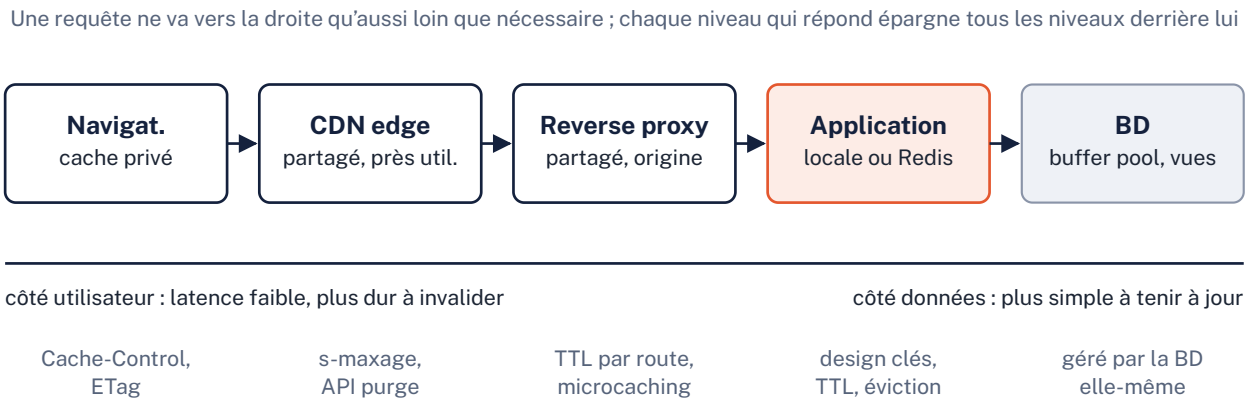
Les moyennes flattent les caches ; comme l'a montré la section 2.4, ce sont les défauts qui définissent la queue de distribution.

Les niveaux de cache

Les caches existent à chaque niveau entre l'utilisateur et le disque, et une requête reçoit sa réponse du premier niveau qui dispose d'une copie utilisable. Le **navigateur** garde un cache privé contrôlé par des en-têtes HTTP. Un réseau de diffusion de contenu (CDN) garde des copies partagées dans des emplacements en périphérie (edge), près des utilisateurs, idéal pour les images, les scripts et les réponses d'API publiques comme les cartes. Un **reverse proxy** à l'origine (Nginx, Varnish) peut mettre en cache des réponses entières pendant quelques secondes, ce qu'on appelle le microcaching, qui peut absorber un pic soudain. L'**application** met en cache des objets calculés, soit dans le processus (le plus rapide, mais chaque instance a sa propre copie et elles divergent), soit dans un cache partagé comme Redis ou Memcached. La **base de données** met en cache des pages en mémoire (les shared buffers de PostgreSQL, le buffer pool d'InnoDB) et peut précalculer des résultats coûteux dans des vues matérialisées.

Plus on est loin des données, plus l'économie par succès est grande et plus l'invalidation est difficile. Vous pouvez supprimer une clé de Redis en une milliseconde ; vous pouvez purger un CDN, plus lentement ; vous ne pouvez pas du tout atteindre le navigateur d'un client, si bien que le `max-age` que vous envoyez est une promesse avec laquelle vous devrez vivre. Les longues durées de vie dans le navigateur sont donc réservées aux fichiers dont l'URL change quand leur contenu change (`app.3f9a1c.js`), et les courtes, ou la revalidation, à tout le reste.

Où vivent les caches



Chaque niveau qui répond à une requête épargne tous les niveaux derrière lui ; les niveaux proches de l'utilisateur économisent davantage mais sont plus difficiles à invalider.

En-têtes de cache HTTP

HTTP dispose d'un modèle de cache complet, défini dans la RFC 9111 (juin 2022), et bien l'utiliser peut supprimer de la charge avant même qu'elle n'atteigne vos serveurs. `Cache-Control: max-age=60` permet à n'importe quel cache de réutiliser une réponse pendant 60 secondes ; `s-maxage` remplace cette valeur pour les caches partagés comme les CDN ;

`private` interdit aux caches partagés de la stocker (utilisez-le pour tout ce qui est personnel) ; `no-cache` signifie, de façon déroutante, «stocke-la, mais vérifie auprès du serveur avant chaque réutilisation» ; et `no-store` signifie «n'en garde jamais de copie», le bon réglage pour les soldes et les relevés bancaires. Les extensions `stale-while-revalidate` et `stale-if-error` (RFC 5861, 2010) appliquent au niveau HTTP les idées de ruée et de disponibilité vues plus haut.

Quand une copie en cache est périmée, le client n'a pas besoin de la télécharger à nouveau. Un validateur lui permet de demander si quelque chose a changé : le serveur envoie un `ETag` (un identifiant de version opaque) ou une date `Last-Modified` avec la réponse, et le client envoie plus tard `If-None-Match` ou `If-Modified-Since`. Si rien n'a changé, le serveur répond `304 Not Modified` sans corps, ce qui économise de la bande passante et souvent l'essentiel du travail du serveur. Ces requêtes conditionnelles sont définies dans la RFC 9110. L'en-tête `Vary` indique aux caches quels en-têtes de requête (comme `Accept-Language`) produisent des réponses différentes, afin qu'une carte en français ne soit jamais servie à une requête en anglais.

```
GET /restaurants/17/menu HTTP/1.1
Host: api.example-food.test

HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: public, max-age=60, s-maxage=300, stale-while-revalidate=30
ETag: "menu-17-v42"

... 18 KB of menu JSON ...

# 100 seconds later the copy is past max-age plus the stale-while-revalidate
# window, so the browser must revalidate before using it:
GET /restaurants/17/menu HTTP/1.1
If-None-Match: "menu-17-v42"

HTTP/1.1 304 Not Modified
Cache-Control: public, max-age=60, s-maxage=300, stale-while-revalidate=30
ETag: "menu-17-v42"

# A bank balance, by contrast, must never sit in any cache:
Cache-Control: no-store
```

Une réponse de carte publique que les navigateurs peuvent réutiliser pendant 60 secondes et les CDN pendant 300, revalidée à faible coût avec un ETag ; les données financières personnelles utilisent no-store.

-

Éviction, dimensionnement et risques de cohérence

Un cache est plus petit que les données qu'il couvre ; quand il est plein, il doit donc évincer quelque chose. La politique moins récemment utilisé (LRU) évince l'entrée restée inutilisée le plus longtemps et fonctionne bien quand l'usage récent prédit l'usage futur. La politique moins fréquemment utilisé (LFU) évince l'entrée la moins souvent utilisée et résiste aux parcours ponctuels (un rapport nocturne qui touche chaque restaurant une fois viderait un cache LRU de ses entrées réellement très demandées). Les caches de production approximent les deux : Redis, par exemple, échantillonne quelques clés et évince le meilleur candidat parmi elles, et propose des politiques comme `allkeys-lru`, `allkeys-lfu` et `volatile-ttl`. Sa politique par défaut, `noeviction`, rejette les écritures quand la mémoire est pleine, ce qui convient à Redis utilisé comme stockage, mais pas comme cache ; choisissez donc la politique délibérément.

Au-delà de la péremption, les caches introduisent des risques de cohérence plus subtils. Un utilisateur qui change son adresse de livraison puis voit l'ancienne, parce que la lecture provient d'un cache qu'une autre instance n'avait pas encore invalidé, subit une rupture de la garantie de lecture de ses propres écritures (section 7.4) ; une solution courante consiste à contourner le cache pour les écritures récentes de l'utilisateur lui-même. Les caches internes au processus de nombreuses instances divergent entre eux, si bien que des requêtes consécutives acheminées vers des instances différentes peuvent montrer une valeur qui revient en arrière. Mettre en cache une erreur ou un résultat vide sans TTL court transforme une brève panne en une longue. Mettre en cache des données qui mêlent plusieurs utilisateurs (une page portant le nom du client, mise en cache sous la seule URL) divulgue des données personnelles au visiteur suivant, une catégorie bien connue d'incidents de sécurité. Enfin, les clés de cache doivent inclure tout ce qui change la réponse : locataire (tenant), langue, devise, permissions et version du format des données.

ATTENTION

Erreurs courantes

- Ajouter un cache avant de mesurer : souvent, la vraie solution est un index ou une meilleure requête, qui supprime le problème sans seconde copie des données.
- Aucun TTL, en se reposant entièrement sur une invalidation qu'un seul chemin de code oublié suffira à manquer.
- Mettre à jour le cache lors des écritures au lieu de supprimer l'entrée, ce qui crée des situations de concurrence sur l'ordre entre écrivains.
- Dimensionner la base de données pour la charge mise en cache, si bien qu'un redémarrage du cache fait tomber toute la plateforme.
- Mettre en cache des données personnelles ou financières dans des caches partagés, ou oublier `private` et `Vary` dans les réponses HTTP.
- Ignorer le taux de succès, le taux d'éviction et la mémoire dans la supervision, si bien qu'un cache cesse discrètement d'être utile.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi une écriture doit-elle supprimer l'entrée du cache plutôt que d'y placer la nouvelle valeur ?

Parce que deux écrivains concurrents peuvent terminer leurs mises à jour du cache dans l'ordre inverse de leurs commits en base de données, laissant l'ancienne valeur en cache ; une suppression peut au pire causer un défaut supplémentaire, qui recharge la valeur actuelle.

▼ Une consultation du cache coûte 1 ms et une lecture en base de données 10 ms. Quelle est approximativement la latence moyenne à un taux de succès de 90 % avec cache-aside ?

1 + 0,1 × 10 = 2 ms (chaque requête paie la consultation ; une sur dix paie aussi la lecture en base de données).

▼ Quelle est la différence entre Cache-Control: no-cache et no-store ?

no-cache permet à un cache de stocker la réponse, mais l'oblige à la revalider auprès du serveur avant chaque réutilisation ; no-store interdit d'en garder la moindre copie.

▼ **Pourquoi faire passer le taux de succès de 99 % à 99,9 % peut-il compter, alors que la latence moyenne change à peine ?**

Parce que la charge de la base de données est proportionnelle au taux de défaut, qui passe de 1 % à 0,1 %, soit dix fois moins de requêtes atteignant la base de données.

▼ **À vous : Un service de carte reçoit 12 000 lectures par seconde. Une consultation du cache prend 0,5 ms, une requête à la base de données 8 ms, et la base de données supporte 1000 requêtes par seconde. Quel taux de succès vous faut-il ? Que se passe-t-il si le cache redémarre vide ?**

Les échecs de cache doivent rester à 1000 ou moins sur 12 000 : le taux de succès doit donc être d'au moins 91,7 %. À 95 %, la base de données reçoit 600 requêtes par seconde et la latence moyenne est de 0,5 + 0,05 x 8 = 0,9 ms. Un cache froid envoie les 12 000 à la base de données, douze fois sa limite, ce qui peut l'effondrer. Préchauffez le cache avant de recevoir du trafic, regroupez les requêtes portant sur la même clé et ajoutez du jitter aux délais d'expiration.

SOURCES

- RFC 9111: HTTP Caching (IETF, 2022): https://www.rfc-editor.org/rfc/rfc9111.html
- RFC 5861: HTTP Cache-Control Extensions for Stale Content: https://www.rfc-editor.org/rfc/rfc5861.html
- MDN Web Docs: HTTP caching: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Caching
- Nishtala et al.: Scaling Memcache at Facebook (NSDI 2013): https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala
- Microsoft Azure Architecture Center: Cache-Aside pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/cache-aside

SECTION 7.7 SUR 7.8 · INTERMÉDIAIRE · 25 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Évolution des schémas, migrations et compatibilité

AVANT DE LIRE

Cette section s'appuie sur :

- 0.5 Stocker les données
- 0.8 Comment les logiciels sont construits
- 6.5 Communication et API

Termes déjà vus qu'elle utilise : Changement cassant (breaking change), Protocol Buffers, Livraison continue, Retard de réplication (replication lag), Idempotent

L'IDÉE EN BREF

Imaginez un hôpital qui modifie le formulaire papier utilisé pour chaque admission alors que les patients continuent d'arriver. S'il se contente de remplacer les formulaires du jour au lendemain, le personnel infirmier de l'équipe de nuit aura en main l'ancien formulaire tandis que l'équipe de jour attendra le nouveau, et les dossiers archivés suivront toujours l'ancienne présentation. Un hôpital prudent imprime un formulaire comportant à la fois les anciennes et les nouvelles cases, l'utilise jusqu'à ce que tout le monde soit passé au nouveau et que les anciens dossiers aient été recopiés, et ne supprime qu'ensuite les anciennes cases. Les équipes logicielles modifient la structure de leurs données exactement de cette manière prudente, étape par étape.

CE QU'IL FAUT SAVOIR

- Les schémas doivent changer alors que l'ancien et le nouveau code tournent côte à côte et que d'anciennes données existent encore ; chaque état de schéma déployé doit donc fonctionner avec au moins deux versions du code.
- La rétrocompatibilité signifie que les nouveaux lecteurs peuvent lire les anciennes données ; la compatibilité future, que les anciens lecteurs peuvent lire les nouvelles données ; la compatibilité totale permet de mettre à niveau lecteurs et rédacteurs dans n'importe quel ordre.
- L'expansion-contraction (expand and contract, ou changement parallèle) découpe un changement cassant en étapes compatibles : ajouter la nouvelle structure, pratiquer la double écriture et le remplissage rétroactif (backfill), basculer les lecteurs, cesser d'écrire l'ancienne structure, puis la supprimer.

- Les migrations sont des scripts versionnés conservés avec le code et appliqués automatiquement par des outils comme Flyway, Liquibase et Alembic ; ne modifiez jamais une migration déjà appliquée, et faites une répétition sur des données de la taille de la production.
- Les gros backfills s'exécutent comme des tâches séparées, par lots, bridées et idempotentes ; des techniques en ligne comme la construction concurrente d'index et les contraintes NOT VALID évitent les verrous prolongés.
- Les registres de schémas font respecter les règles de compatibilité des schémas d'événements ; Avro s'appuie sur les valeurs par défaut et sur la résolution entre schéma du rédacteur et schéma du lecteur, Protocol Buffers sur le fait de ne jamais réutiliser un numéro de champ.

Pourquoi les schémas doivent évoluer sans tout arrêter

Toute donnée stockée ou transmise a un schéma : les tables et les types de colonnes d'une base de données, les champs d'une réponse d'API JSON, la définition d'enregistrement d'un événement dans un flux. Les schémas changent sans cesse, parce que l'activité change : la banque ajoute un second titulaire de compte, la plateforme de livraison de repas commence à enregistrer les pourboires, un régulateur exige un nouveau champ. La question architecturale est de savoir comment modifier un schéma pendant que le système continue de fonctionner, alors que l'ancienne forme est encore utilisée par du code, par des données stockées et par d'autres équipes.

Trois faits rendent cela difficile. Premièrement, dans tout système déployé sans interruption de service, l'ancienne et la nouvelle version du code tournent côte à côte : un déploiement progressif (rolling deployment) remplace les instances une par une, un déploiement canari envoie une petite part du trafic vers la nouvelle version, et un retour arrière (rollback) ramène l'ancienne version. Chaque état du schéma doit donc fonctionner avec au moins deux versions du code. Deuxièmement, les données survivent au code. Les lignes écrites il y a cinq ans, les événements d'un journal conservé indéfiniment, les messages en attente dans une file et les sauvegardes gardent tous l'ancienne forme. Troisièmement, les schémas sont partagés : une colonne est lue par des rapports, des tâches d'extraction et parfois d'autres services, et un schéma d'événement est lu par des consommateurs dont le producteur ignore parfois jusqu'à l'existence. La section 7.1 plaidait pour que les données soient détenues derrière une interface ; c'est dans l'évolution des schémas que cette discipline porte ses fruits, car un schéma privé peut être modifié avec l'accord d'une seule équipe.

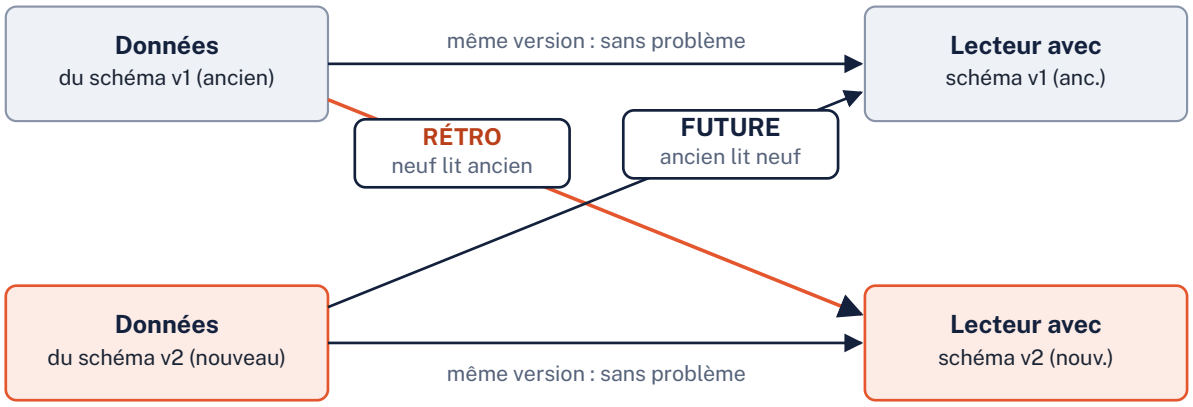
Rétrocompatibilité, compatibilité future et compatibilité totale

La compatibilité se définit en termes de lecteurs et de rédacteurs. Un changement est rétrocompatible lorsque du code utilisant le **nouveau** schéma peut lire des données écrites avec l'**ancien**. Ajouter un champ facultatif doté d'une valeur par défaut est rétrocompatible : le nouveau lecteur fournit la valeur par défaut lorsque le champ manque dans les anciennes données. Un changement offre une compatibilité future (forward compatibility) lorsque du code utilisant l'**ancien** schéma peut lire des données écrites avec le **nouveau**. Cela exige que les anciens lecteurs ignorent les champs qu'ils ne connaissent pas, ce qui est la discipline du lecteur tolérant (tolerant reader). Un changement est **totalement compatible** lorsqu'il est les deux à la fois, ce qui signifie que lecteurs et rédacteurs peuvent être mis à niveau dans n'importe quel ordre.

Le sens dont vous avez besoin découle de qui se met à niveau en premier. Si les consommateurs sont mis à niveau avant les producteurs (cas fréquent lorsqu'un flux a quelques consommateurs que vous contrôlez), il vous faut la rétrocompatibilité. Si ce sont les producteurs qui sont mis à niveau en premier, ce qui est le cas habituel pour une base de données (le nouveau code commence à écrire avant que chaque lecteur ait été redéployé) et pour les API publiques, il vous faut la compatibilité future. Lorsque vous ne pouvez pas contrôler l'ordre, ou devez pouvoir revenir en arrière d'un côté comme de l'autre, visez la compatibilité totale. Confluent Schema Registry, par exemple, applique par défaut une rétrocompatibilité vérifiée par rapport à la version précédente, et propose des variantes transitives qui vérifient par rapport à toutes les versions antérieures, ce qui compte lorsque des consommateurs peuvent rejouer d'anciennes données depuis le début d'un journal.

Un mot d'avertissement sur le vocabulaire. En conception d'API, «rétrocompatible» s'emploie souvent de façon approximative pour dire «les clients existants continuent de fonctionner», ce qui, en termes de lecteurs et de rédacteurs, correspond à la compatibilité future pour les clients qui lisent vos réponses et à la rétrocompatibilité pour les requêtes que vous lisez de leur part. Lorsqu'une discussion repose sur ce terme, demandez quel côté change et quel côté doit continuer de fonctionner.

Quel lecteur peut lire quelles données



TOTALE = les deux diagonales marchent : lecteurs et rédacteurs migrent dans tout ordre

La rétrocompatibilité protège les nouveaux lecteurs des anciennes données ; la compatibilité future protège les anciens lecteurs des nouvelles données.

Changements courants et leur compatibilité

Changement	Rétrocompatibilité (le neuf lit l'ancien)	Compatibilité future (l'ancien lit le neuf)	Approche sûre
Ajouter un champ facultatif ou une colonne nullable avec une valeur par défaut	Oui	Oui, si les lecteurs ignorent les champs inconnus	Le changement sûr par excellence
Ajouter un champ obligatoire sans valeur par défaut	Non : les anciennes données ne l'ont pas	Oui	L'ajouter en facultatif, faire le backfill, puis l'imposer
Supprimer un champ	Oui	Seulement si les anciens lecteurs ont une valeur par défaut pour ce champ	Cesser d'abord de le lire partout, puis le supprimer
Renommer un champ ou une colonne	Non	Non	Expansion-contraction : ajouter le nouveau, copier, basculer, supprimer l'ancien
Changer un type (texte en entier, int en long)	Dépend du format : élargir est souvent sûr, restreindre ne l'est pas	Cela dépend	Ajouter un nouveau champ du nouveau type et migrer
Changer le sens sans changer la forme (des pence aux livres)	Faux, sans que rien ne le signale	Faux, sans que rien ne le signale	Jamais : créer un nouveau champ sous un nouveau nom

SI VOUS DÉBUTEZ

Pas à pas : anciens lecteurs, nouveaux lecteurs

Le point qui fait le plus souvent trébucher ici est la différence entre rétrocompatibilité et compatibilité future, et la raison pour laquelle celle dont vous avez besoin dépend du côté qui se met à niveau en premier. Suivez un nouveau champ à travers la plateforme de livraison de repas.

- La plateforme commence à enregistrer les pourboires ; une nouvelle version de l'événement OrderPlaced ajoute donc un champ, tip_pence. Les événements déjà écrits et stockés n'ont pas ce champ.
- Supposons qu'un consommateur soit mis à niveau en premier. Son nouveau code lit des événements écrits avant l'existence des pourboires, y compris d'anciens événements rejoués depuis le début du journal ; il doit donc remplir le champ manquant avec une valeur par défaut, ici null, qui signifie qu'aucun pourboire n'a été enregistré. Ce cas, où la nouvelle version s'accommode des anciennes données, s'appelle la rétrocompatibilité.
- Supposons au contraire que le producteur soit mis à niveau en premier. Les consommateurs qui exécutent encore l'ancien code reçoivent maintenant des événements comportant un champ dont ils n'ont jamais entendu parler, et doivent simplement l'ignorer, comme le fait un lecteur tolérant. Ce cas, où l'ancienne version s'accommode des nouvelles données, s'appelle la compatibilité future.
- Avec des lecteurs tolérants, un champ facultatif doté d'une valeur par défaut fonctionne dans les deux sens ; le changement est donc totalement compatible : chaque côté peut être mis à niveau en premier, et chacun peut revenir en arrière.
- Comparez avec le renommage de total_pence en amount_pence, pour des lecteurs qui trouvent les champs par leur nom. Les nouveaux lecteurs cherchent amount_pence et ne le trouvent pas dans les anciens événements ; les anciens lecteurs cherchent total_pence et ne le trouvent pas dans les nouveaux. Le changement casse dans les deux sens à la fois.
- **La leçon :** avant de modifier un schéma, demandez-vous qui lira de nouvelles données avec de l'ancien code et qui lira d'anciennes données avec du nouveau code. Les changements additifs avec valeurs par défaut sont sûrs dans les deux sens ; tout le reste exige la technique pas à pas présentée ensuite.

Expansion-contraction : la technique de base

La plupart des changements cassants peuvent être découpés en une suite de changements compatibles. Ce patron s'appelle expansion-contraction (expand and contract), ou changement parallèle (la description qu'en a donnée Danilo Sato en 2014 sur martinfowler.com, sous le nom de parallel change, nomme les phases expand, migrate et contract). D'abord l'**expansion** :

ajoutez la nouvelle structure à côté de l'ancienne, de sorte que l'ancien comme le nouveau code fonctionnent. Ensuite la **migration** : faites en sorte que le code qui écrit maintienne les deux, copiez les données existantes, et faites passer les lecteurs à la nouvelle structure un par un. Enfin la **contraction** : lorsque plus rien n'utilise l'ancienne structure, supprimez-la. Chaque étape est déployée séparément, chacune est compatible avec la version qui la précède, et chacune peut faire l'objet d'un retour arrière sans perte de données.

Le prix à payer est du temps et un désordre temporaire : le changement s'étale sur plusieurs mises en production, et pendant des jours ou des semaines le schéma porte les deux formes. L'avantage est qu'aucune mise en production n'exige d'interruption de service, qu'aucun déploiement ne doit être coordonné entre équipes, et que chaque étape peut être mise en pause. *Refactoring Databases* (2006), de Scott Ambler et Pramod Sadalage, répertorie des dizaines de refactorisations de bases de données fondées sur la même idée, avec une **période de transition** pendant laquelle l'ancien et le nouveau schéma coexistent, maintenus en phase par l'application ou par des déclencheurs (triggers).

EXEMPLE COMMENTÉ

Exemple commenté : renommer une colonne sans interruption de service

La table restaurants de la plateforme de livraison de repas comporte une colonne `postcode`. La plateforme s'étend à des pays qui utilisent d'autres formats postaux, et l'équipe veut l'appeler `postal_code`. La table compte 4 millions de lignes, le service Restaurants est déployé plusieurs fois par jour par mises à jour progressives, et une tâche d'extraction nocturne lit aussi la colonne. Un simple `ALTER TABLE ... RENAME COLUMN` casserait toutes les instances en cours de l'ancien code à l'instant même de son commit.

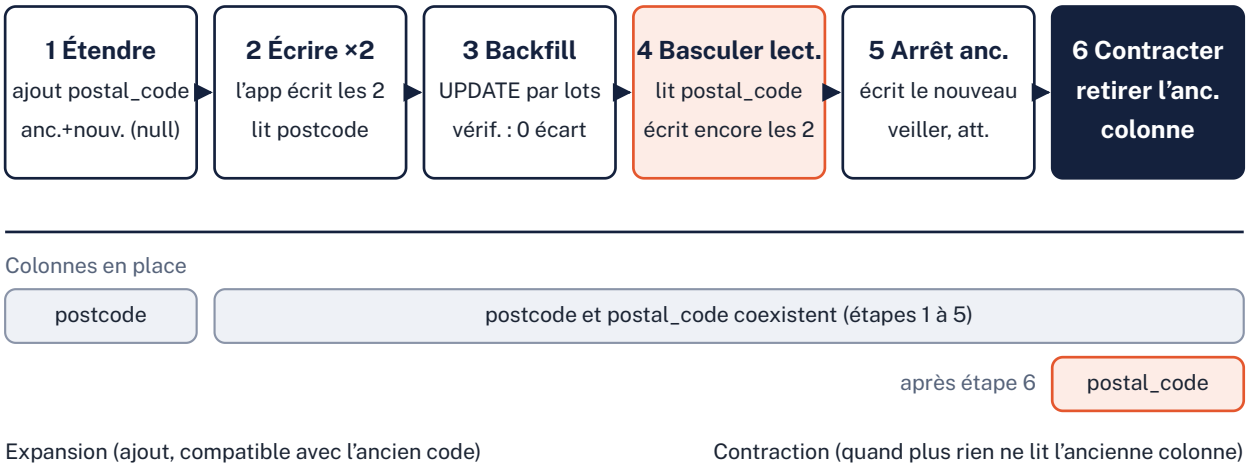
Version 1 : expansion	La migration ajoute `postal_code text NULL`. Depuis PostgreSQL 11, ajouter une colonne sans valeur par défaut ou avec une valeur par défaut constante est un changement qui ne touche que les métadonnées. L'ancien code ignore la nouvelle colonne ; rien ne casse.
Version 2 : double écriture	L'application écrit la même valeur dans les deux colonnes à chaque insertion et mise à jour, et lit toujours `postcode`. (Un déclencheur de base de données qui copie une colonne dans l'autre est une solution de rechange lorsque plusieurs applications écrivent dans la table.)
Étape 3 : backfill	Une tâche d'arrière-plan copie `postcode` dans `postal_code` pour les lignes existantes, par lots de 5 000 selon des plages de clé primaire, en marquant une pause entre les lots et en surveillant le retard de réplication. Une requête de vérification compte ensuite les lignes où les deux valeurs diffèrent : elle doit renvoyer 0.
Version 4 : bascule des lectures	L'application lit `postal_code` mais écrit toujours dans les deux colonnes, si bien qu'un retour à la version 2 ou 3 reste sûr. La tâche d'extraction est modifiée pour lire elle aussi la nouvelle colonne.
Version 5 : arrêt de l'écriture dans l'ancienne colonne	L'application n'écrit plus que dans `postal_code`. L'ancienne colonne ne doit plus être `NOT NULL` à ce stade, sinon les insertions échouent. Attendez assez longtemps pour être sûr qu'aucun retour à la version 4 ne sera nécessaire.
Version 6 : contraction	La migration impose `NOT NULL` sur `postal_code` (à l'aide d'une contrainte de vérification `NOT VALID`, validée séparément, que PostgreSQL 12 et les versions ultérieures savent utiliser pour poser `NOT NULL` sans long parcours bloquant) et supprime enfin `postcode`.

Six déploiements pour un seul renommage, cela semble excessif, mais des étapes peuvent être combinées lorsque l'équipe contrôle chaque lecteur et accepte une courte fenêtre de retour arrière ; le principe qui compte est que chaque état déployé fonctionne avec le code d'avant et d'après.

L'étape de contraction est celle que les équipes oublient. Suivez-la explicitement (un ticket, un statut d'ADR, une règle de lint qui signale l'ancienne colonne), sinon le schéma se remplit de colonnes à moitié migrées.

Le renommage en six étapes déployables

Un renommage, six étapes déployables ; chacune peut revenir sans risque à la précédente



Pendant les étapes 1 à 5, les deux colonnes existent, si bien que n'importe quelle version du code de l'étape précédente peut encore tourner.

```
-- Migration V41 (expand): metadata-only change, no table rewrite in PostgreSQL 11+
SET lock_timeout = '3s';            -- fail fast rather than queue behind long queries
ALTER TABLE restaurants ADD COLUMN postal_code text;

-- Backfill job (run repeatedly by a script, not inside one big migration)
UPDATE restaurants
SET    postal_code = postcode
WHERE  id ⩾ :from_id AND id < :from_id + 5000   -- one small batch per transaction
AND    postal_code IS DISTINCT FROM postcode;

-- Verify before switching reads
SELECT count(*) FROM restaurants WHERE postal_code IS DISTINCT FROM postcode;  -- expect
0

-- Migration V44: enforce NOT NULL without a long blocking scan
ALTER TABLE restaurants ADD CONSTRAINT postal_code_nn CHECK (postal_code IS NOT NULL) NOT
VALID;
ALTER TABLE restaurants VALIDATE CONSTRAINT postal_code_nn;  -- scans, but allows writes
ALTER TABLE restaurants ALTER COLUMN postal_code SET NOT NULL;  -- PostgreSQL 12+: uses
the check, skips scan
ALTER TABLE restaurants DROP CONSTRAINT postal_code_nn;

-- Migration V45 (contract), weeks later, once nothing reads postcode
ALTER TABLE restaurants DROP COLUMN postcode;
```

Instructions PostgreSQL pour les étapes d'expansion, de backfill, de vérification et de contraction, écrites pour éviter les verrous prolongés.

Les migrations en tant que code dans le pipeline de livraison

Les modifications de base de données ont leur place dans la gestion de versions, à côté du code de l'application, sous la forme d'une série ordonnée de migrations, chacune étant un petit script qui fait passer le schéma d'une version à la suivante. *Evolutionary Database Design* (publié pour la première fois en 2003, réécrit en 2016), de Pramod Sadalage et Martin Fowler, a posé cette pratique, et les outils sont mûrs. **Flyway** exécute dans l'ordre des fichiers SQL versionnés comme `V41__add_postal_code.sql` et consigne chacun dans une table `flyway_schema_history`. **Liquibase** décrit les changements sous forme de changesets en SQL, XML, YAML ou JSON, suivis dans une table `DATABASECHANGELOG`. **Alembic**, pour Python et SQLAlchemy, génère des scripts de révision comportant des fonctions `upgrade()` et `downgrade()` et enregistre la révision courante dans `alembic_version`. Flyway et Liquibase stockent aussi une somme de contrôle de chaque migration appliquée et s'arrêtent avec une erreur si l'une d'elles a été modifiée depuis, ce qui empêche les environnements de diverger en silence.

Dans un pipeline de livraison continue, les mêmes migrations s'exécutent sur une base de données vierge pendant les tests, puis en staging, puis en production, si bien que la production ne reçoit jamais un changement qui n'a pas été répété. Les bonnes pratiques ajoutent quelques règles. Ne modifiez jamais une migration qui a été appliquée où que ce soit ; écrivez-en une nouvelle. Placez les changements de schéma et le code qui en dépend dans des déploiements séparés, le changement de schéma en premier pour une expansion et en dernier pour une contraction. Testez les migrations sur une copie de données de la taille de la production, car une instruction qui prend 50 ms sur une base de test peut verrouiller une grande table pendant une heure. Préférez avancer avec une nouvelle migration plutôt que recourir à des migrations descendantes qui suppriment des colonnes, et les données avec elles. Enfin, laissez la base de données vous renseigner sur les verrous : dans PostgreSQL, un `ALTER TABLE` qui attend son verrou bloque toutes les requêtes arrivées après lui ; fixez donc un `lock_timeout` court et réessayez plutôt que de faire la queue derrière un rapport de longue durée.

Gros backfills et modifications de schéma en ligne

Un backfill (remplissage rétroactif) remplit une nouvelle colonne ou une nouvelle table pour les lignes existantes. Exécuté sous la forme d'un seul `UPDATE` portant sur 400 millions de transactions, il garde des verrous pendant des heures, génère un énorme volume de journal d'écriture anticipée (write-ahead log) que les répliques doivent rejouer (ce qui crée un retard de réplication), et laisse dans PostgreSQL des versions de lignes mortes que le vacuum doit nettoyer. L'approche professionnelle est une tâche séparée et redémarrable qui traite de petits lots par plages de clé primaire, valide (commit) chaque lot, enregistre sa progression, se bride elle-même lorsque le retard de réplication ou la charge de la base augmentent, et qui est idempotente, de sorte que relancer un lot ne cause aucun dégât. Exécutez-la en dehors de l'outil de migration, pour qu'un déploiement n'ait jamais à l'attendre.

Certains changements ne peuvent pas être faits sur place sans verrouillage. PostgreSQL propose `CREATE INDEX CONCURRENTLY` (qui ne peut pas s'exécuter dans un bloc de transaction, ce qu'il faut indiquer aux outils de migration) et la séquence `NOT VALID` puis `VALIDATE CONSTRAINT` présentée plus haut. Pour MySQL, des outils comme gh-ost et pt-online-schema-change construisent une copie fantôme de la table avec le nouveau schéma, la maintiennent en phase avec les écritures en cours et la mettent en place par un bref renommage. Pour changer le type d'une colonne, mieux vaut généralement une expansion-contraction vers une nouvelle colonne qu'un `ALTER` sur place, qui, dans la plupart des bases de données, réécrit toute la table.

Registres de schémas, Avro et Protocol Buffers

Pour les événements et les messages, l'équivalent d'un outil de migration est un registre de schémas. Les producteurs enregistrent chaque version de schéma ; le registre la vérifie par rapport à la règle de compatibilité configurée et rejette un changement cassant au moment du build ou du déploiement plutôt qu'en production. Chaque message porte un court identifiant de schéma, de sorte que les consommateurs peuvent récupérer le schéma exact avec lequel il a été écrit.

Apache Avro rend explicite le schéma du rédacteur : le décodage nécessite à la fois le schéma du rédacteur et celui du lecteur, et les règles de résolution d'Avro font correspondre les champs par leur nom, ignorent les champs du rédacteur que le lecteur ne connaît pas, et remplissent les champs du lecteur absents des données avec leurs valeurs par défaut déclarées. D'où la règle pratique : donnez une valeur par défaut à tout champ que vous pourriez ajouter ou supprimer. Dans l'exemple ci-dessous, `currency` et `tip_pence` ont été ajoutés dans des versions ultérieures avec des valeurs par défaut, si bien que les anciens événements se décodent toujours avec les nouveaux consommateurs.

Sur le réseau, **Protocol Buffers** identifie les champs par leur numéro, et non par leur nom. Ajouter des champs est sûr, et les anciens lecteurs conservent les champs inconnus lors d'une nouvelle sérialisation. Renommer un champ n'affecte pas l'encodage binaire (même si cela change la forme JSON). Les règles cardinales sont de ne jamais changer ni réutiliser un numéro de champ, et de marquer les numéros et les noms des champs supprimés comme `reserved` afin que personne ne les réutilise par accident. Confluent recommande la rétrocompatibilité transitive pour Protobuf, car l'ajout d'un nouveau type de message n'assure pas la compatibilité future. JSON Schema peut aussi être enregistré et vérifié, mais l'ouverture de JSON (une propriété inconnue est-elle autorisée ou non ?) rend les règles de compatibilité plus subtiles.

```
{
  "type": "record",
  "name": "OrderPlaced",
  "namespace": "food.orders.v1",
  "fields": [
    {"name": "order_id",       "type": "string"},
    {"name": "restaurant_id",  "type": "string"},
    {"name": "total_pence",    "type": "long"},
    {"name": "currency",       "type": "string", "default": "GBP"},
    {"name": "tip_pence",      "type": ["null", "long"], "default": null}
  ]
}
```

Un schéma Avro pour un événement OrderPlaced ; les champs ajoutés après la version 1 ont des valeurs par défaut, si bien que les nouveaux consommateurs peuvent lire les anciens événements.

Faire évoluer les schémas d'API et d'événements

Les API et les événements suivent les mêmes principes avec des outils différents. Pour les API, privilégiez le changement additif, exigez que les clients soient des lecteurs tolérants, et réservez les changements cassants à une nouvelle version exécutée à côté de l'ancienne, avec une supervision de l'usage pour savoir quand l'ancienne version peut être retirée (la section 6.5 présente les approches de versionnage). Les tests de contrat pilotés par le consommateur permettent à chaque consommateur de publier les champs dont il dépend réellement, si bien que le fournisseur sait quels changements sont sûrs.

Les événements sont plus difficiles que les API parce qu'ils sont stockés : un consommateur peut rejouer un topic depuis le début, et un système fondé sur l'event sourcing (section 7.5) reconstruit son état à partir d'événements écrits il y a des années. Traitez un schéma d'événement publié comme un contrat public, versionnez-le dans le registre et, lorsqu'un changement est réellement cassant, publiez pendant un certain temps un nouveau type d'événement ou un nouveau topic (`OrderPlaced.v2`) à côté de l'ancien plutôt que de modifier l'existant. La section 6.4 traite plus généralement de la conception des événements.

Liste de contrôle : avant de livrer un changement de schéma

- [ ] **Le changement est-il additif ?**
  Sinon, découpez-le en étapes d'expansion, de migration et de contraction, chacune déployable seule.
- [ ] **Chaque état déployé fonctionne-t-il avec la version précédente et la version suivante du code ?**
  Avec les déploiements progressifs et les retours arrière, deux versions tournent toujours ensemble.
- [ ] **Qui d'autre lit ces données ?**
  Recherchez les rapports, les tâches d'extraction, les autres services et les consommateurs d'événements ; vérifiez les abonnés du registre et le lignage dans le catalogue de données.
- [ ] **La migration a-t-elle été exécutée sur des données de la taille de la production ?**
  Mesurez la durée des verrous, les réécritures de tables et le volume de journal ; utilisez la construction concurrente d'index et les contraintes NOT VALID lorsqu'elles sont disponibles.
- [ ] **Un délai d'expiration de verrou est-il défini ?**
  Échouez vite et réessayez plutôt que de laisser un ALTER en attente bloquer tout le trafic.
- [ ] **Le backfill est-il séparé, par lots, bridé, idempotent et reprenable ?**
  Il ne doit jamais bloquer un déploiement ni submerger les répliques.
- [ ] **Les nouveaux champs ont-ils des valeurs par défaut, et aucun numéro ni nom de champ n'est-il réutilisé ?**
  Indispensable pour Avro et Protocol Buffers ; imposé par la vérification de compatibilité du registre.
- [ ] **Le sens d'un champ a-t-il changé ?**
  Un changement d'unité, de devise ou de sémantique exige un nouveau nom de champ, jamais un nom réutilisé.
- [ ] **L'étape de contraction est-elle suivie ?**
  Ouvrez un ticket ou un ADR pour que l'ancienne structure soit réellement supprimée une fois inutilisée.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Une équipe ajoute un champ obligatoire sans valeur par défaut à un schéma Avro. Quelle compatibilité cela casse-t-il, et pourquoi ?**

La rétrocompatibilité : un consommateur qui utilise le nouveau schéma pour lire d'anciennes données constate que le champ manque et n'a aucune valeur par défaut pour le remplir.

▼ **Pourquoi l'étape de double écriture se poursuit-elle après la bascule des lectures vers la nouvelle colonne ?**

Pour qu'un retour à la version précédente, qui lit encore l'ancienne colonne, trouve cette colonne à jour.

▼ **Pourquoi un gros backfill ne doit-il pas être écrit sous la forme d'une seule instruction UPDATE dans une migration ?**

Une énorme transaction garde des verrous longtemps, génère une rafale de journal qui crée un retard de réplication, laisse beaucoup de nettoyage, ne peut pas être reprise et bloque le déploiement ; de petits lots validés un par un évitent tout cela.

▼ **Qu'empêche le fait de marquer un numéro de champ Protocol Buffers comme reserved ?**

Il empêche quiconque de réutiliser le numéro d'un champ supprimé pour un nouveau champ, ce qui ferait décoder d'anciennes données comme le mauvais champ.

▼ **À vous : Une table bookings a une colonne, name, et il vous faut first_name et last_name. Les services utilisent des mises à jour progressives et une tâche nocturne lit name. Énumérez les étapes pour y parvenir sans interruption de service, et dites pourquoi une migration unique est dangereuse.**

Un changement unique casserait toutes les instances en cours d'exécution de l'ancien code. Utilisez l'expansion-contraction. Ajoutez les deux nouvelles colonnes acceptant NULL. Déployez du code qui écrit à la fois l'ancienne et les nouvelles. Faites le backfill des anciennes lignes par petits lots idempotents et limités en débit. Basculez les lecteurs, y compris le job nocturne, vers les nouvelles colonnes. Cessez d'écrire name, puis supprimez-la une fois que plus rien ne l'utilise. Chaque étape fonctionne avec deux versions du code, et toutes peuvent être mises en pause sans risque.

SOURCES

- Danilo Sato: ParallelChange (martinfowler.com, 2014): https://martinfowler.com/bliki/ParallelChange.html

- Pramod Sadalage and Martin Fowler: Evolutionary Database Design: https://martinfowler.com/articles/evodb.html
- Confluent documentation: Schema evolution and compatibility: https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html
- Protocol Buffers: Language Guide (proto3), updating a message type: https://protobuf.dev/programming-guides/proto3/
- PostgreSQL documentation: ALTER TABLE: https://www.postgresql.org/docs/current/sql-altertable.html

Analytique, streaming, entrepôts de données, data lakes et data mesh

AVANT DE LIRE

Cette section s'appuie sur :

- 0.5 Stocker les données
- 6.4 Pilotée par les événements
- 7.1 Propriété des données

Termes déjà vus qu'elle utilise : Contrat de données, Schéma, Clé étrangère, Jointure (join)

L'IDÉE EN BREF

Les caisses d'un supermarché sont conçues pour servir chaque client rapidement, un panier à la fois. Personne ne demanderait au personnel de caisse de s'arrêter pour calculer quels produits se sont le mieux vendus dans chaque magasin au cours des deux dernières années. À la place, chaque soir, les ventes de la journée sont envoyées au siège, où des analystes conservent des années d'historique, organisées pour ce genre de question. Certains chiffres ne peuvent pas attendre le lendemain, comme une ruée soudaine sur le pain ; ils sont donc aussi surveillés en direct. L'architecture analytique consiste à construire le siège et la surveillance en direct sans ralentir les caisses.

CE QU'IL FAUT SAVOIR

- L'OLTP sert de nombreuses petites transactions par clé ; l'OLAP parcourt et agrège de gros volumes ; les séparer protège les systèmes opérationnels et permet à l'analytique d'utiliser des conceptions en colonnes et dénormalisées.
- Inmon construit d'abord un entrepôt d'entreprise intégré et normalisé ; Kimball construit des schémas en étoile dimensionnels par processus métier, reliés par des dimensions conformes ; toute table de faits a besoin d'un grain déclaré.
- L'ELT charge les données brutes et les transforme dans l'entrepôt ; la capture des données modifiées diffuse les changements à partir des journaux de la base de données, mais des contrats de données publiés sont plus robustes que la lecture des tables d'un autre service.
- Un lakehouse ajoute transactions, évolution des schémas, suppressions et voyage dans le temps à des fichiers ouverts sur un stockage objet, grâce à des formats de table ouverts comme Delta Lake, Apache Iceberg et Apache Hudi.

- Le traitement de flux doit calculer selon le temps de l'événement, découper les flux en fenêtres bascules, glissantes ou de session, et utiliser des filigranes ainsi qu'une règle explicite pour les données tardives.
- Lambda (Marz) combine une couche batch et une couche rapide au prix d'une logique dupliquée ; Kappa (Kreps, 2014) utilise un seul processeur de flux et retraite en rejouant un journal conservé.
- Le data mesh (Dehghani, 2019) repose sur la propriété par domaine, la donnée en tant que produit, une plateforme en libre-service et une gouvernance computationnelle fédérée ; il convient aux grandes organisations dotées d'équipes de domaine matures.

Deux charges de travail différentes : OLTP et OLAP

Tout ce chapitre a jusqu'ici porté sur le traitement transactionnel en ligne (OLTP) : de nombreuses lectures et écritures petites et rapides d'enregistrements individuels, comme passer une commande, enregistrer un paiement ou mettre à jour la position d'un livreur. L'analytique pose un autre type de question. «Quel a été le chiffre d'affaires par type de cuisine et par région, semaine après semaine, sur les deux dernières années ?» touche des millions de lignes, ne lit que quelques colonnes de chacune et ne modifie rien. C'est le traitement analytique en ligne (OLAP), et il diffère tellement de l'OLTP qu'exécuter les deux sur la même base de données ne sert généralement bien ni l'un ni l'autre : une seule grosse requête analytique peut saturer les disques et le cache de tampons dont dépend l'étape de paiement, et le schéma normalisé qui garantit la justesse des transactions oblige les analystes à joindre une douzaine de tables.

La réponse architecturale classique consiste à séparer les charges de travail. Les systèmes opérationnels gardent leurs propres bases de données, détenues par leurs services comme l'a décrit la section 7.1. Les données sont copiées, en continu ou périodiquement, vers une plateforme analytique conçue pour les parcours de données : un stockage en colonnes qui ne lit que les colonnes nécessaires, une forte compression, des moteurs de requêtes massivement parallèles et des schémas pensés pour l'analyse. Le reste de cette section explique comment cette plateforme est conçue, comment les données y parviennent, ce que le streaming change et à qui appartient le résultat.

OLTP et OLAP comparés

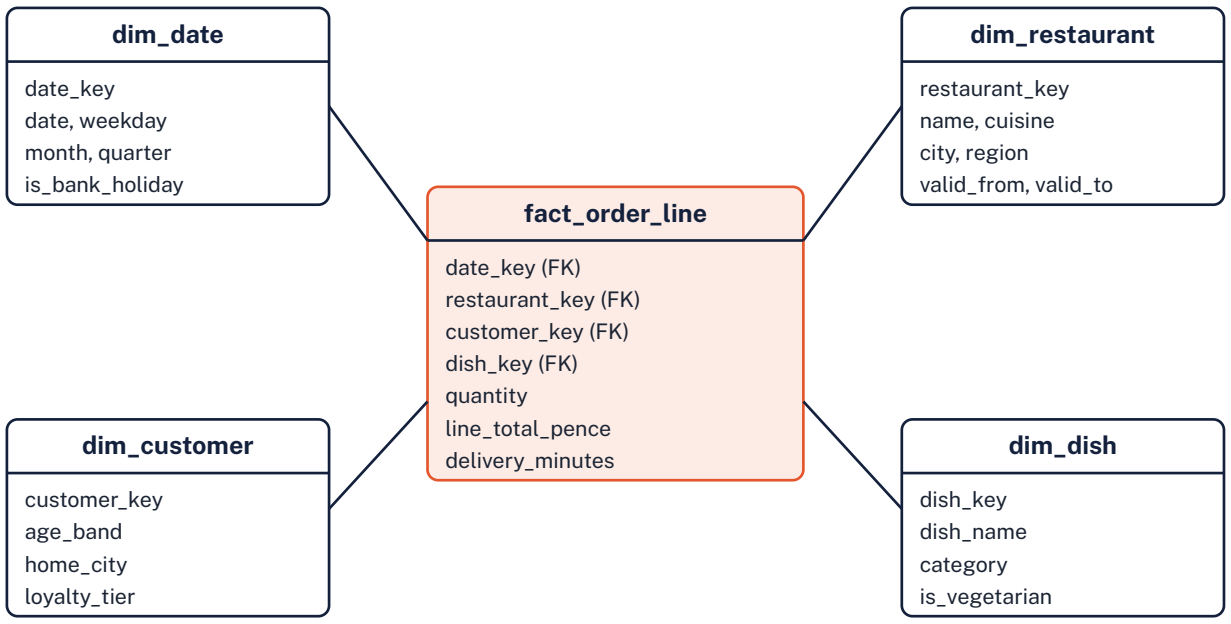
Aspect	OLTP (opérationnel)	OLAP (analytique)
Question type	Récupérer la commande 81723 ; la marquer comme livrée	Délai moyen de livraison par ville et par heure, sur les 12 derniers mois
Lignes touchées par requête	Une à quelques centaines, par clé	Des millions à des milliards, par parcours
Écritures	Petites insertions et mises à jour constantes	Chargements en masse ou ajouts ; peu de mises à jour
Schéma	Normalisé pour l'intégrité (section 0.5)	Dénormalisé, dimensionnel ou en tables larges
Organisation du stockage	Orienté lignes, indexé par clé	Orienté colonnes, compressé, partitionné par date
Usagers	Clients et applications	Analystes, data scientists, tableaux de bord, pipelines de ML
Qualités clés	Faible latence, cohérence, disponibilité	Débit, souplesse des requêtes, historique
Exemples	PostgreSQL, MySQL, DynamoDB	Snowflake, BigQuery, Redshift, ClickHouse, Databricks SQL

Entrepôts de données et modélisation dimensionnelle

Un entrepôt de données (data warehouse) est une base de données conçue pour l'analyse, qui intègre des données nettoyées provenant de nombreux systèmes opérationnels et en conserve l'historique. Deux écoles ont façonné la conception des entrepôts, et les praticiens en débattent encore. **Bill Inmon**, souvent appelé le père de l'entrepôt de données (*Building the Data Warehouse*, 1992), prône une conception descendante : un entrepôt intégré, normalisé, à l'échelle de l'entreprise, est construit d'abord comme source unique de vérité, et des data marts départementaux en sont ensuite dérivés. **Ralph Kimball** (*The Data Warehouse Toolkit*, publié pour la première fois en 1996, troisième édition avec Margy Ross en 2013) prône une conception ascendante : construire des modèles dimensionnels un processus métier à la fois, et les intégrer au moyen de **dimensions conformes** partagées (une seule définition convenue du client, de la date ou du restaurant, utilisée par tous les modèles). L'approche d'Inmon offre une intégration plus forte pour un coût initial plus élevé ; celle de Kimball produit plus tôt des résultats exploitables, mais demande de la discipline pour que les dimensions restent conformes. De nombreuses plateformes modernes mélangent les deux : une couche nettoyée et intégrée qui alimente des modèles dimensionnels destinés à la consommation.

Le schéma en étoile de Kimball reste le modèle analytique le plus enseigné. Une table de faits centrale enregistre des événements métier à un **grain** déclaré (une ligne par ligne de commande, par paiement, par signal de position d'un livreur), avec des mesures numériques comme la quantité, le montant et les minutes, et des clés étrangères vers des tables de dimensions. Chaque dimension contient des attributs descriptifs servant à filtrer et à regrouper : les dates (avec jour de la semaine, mois et indicateur de jour férié précalculés), les restaurants (nom, type de cuisine, ville), les clients et les plats. Les requêtes se réduisent à une table de faits jointe à quelques petites dimensions, ce que les moteurs en colonnes exécutent très efficacement. Les attributs qui changent au fil du temps, comme un restaurant qui change de ville, sont gérés par des techniques de **dimension à évolution lente** (slowly changing dimension) ; la plus courante, le type 2, ajoute une nouvelle ligne de dimension avec des dates de validité, si bien que les faits historiques pointent toujours vers les attributs qui étaient vrais à l'époque.

Un schéma en étoile pour les commandes de livraison de repas



Grain : une ligne par plat d'une commande livrée. Faits numériques et additifs ; les dimensions les décrivent.

Une table de faits à un grain déclaré, entourée des dimensions qui décrivent chaque fait.

```
-- Revenue and average delivery time by cuisine and weekday for last quarter.
-- One join per dimension; the fact table is scanned, never updated.
SELECT r.cuisine,
       d.weekday,
       SUM(f.line_total_pence) / 100.0 AS revenue_gbp,
       AVG(f.delivery_minutes)          AS avg_delivery_min
FROM   fact_order_line f
JOIN   dim_date       d ON d.date_key       = f.date_key
JOIN   dim_restaurant r ON r.restaurant_key = f.restaurant_key
WHERE  d.quarter = '2026-Q2'
  AND  r.region  = 'North West'
GROUP  BY r.cuisine, d.weekday
ORDER  BY revenue_gbp DESC;
-- Caveat: delivery_minutes is per order, so averaging it over order LINES
-- weights big orders more. Grain decisions matter (see the worked example).
```

Une requête type sur un schéma en étoile : filtrer et regrouper selon des attributs de dimension, agréger les mesures des faits.

EXEMPLE COMMENTÉ

Exemple commenté : déclarer le grain

L'équipe analytique construit la table de faits ci-dessus au grain d'une ligne par plat d'une commande livrée. La mesure de niveau commande `delivery_minutes` a été copiée sur chaque ligne. Deux commandes sont livrées : la commande A a 1 ligne et a pris 20 minutes ; la commande B a 4 lignes et a pris 40 minutes.

Délai moyen de livraison correct	Par commande : (20 + 40) ÷ 2 = 30 minutes.
Moyenne sur les lignes de faits	Lignes : 20, 40, 40, 40, 40. Somme 180 ÷ 5 lignes = 36 minutes. Faux : les grosses commandes comptent quatre fois.
Somme des minutes de livraison	180, ce qui n'a aucun sens : la mesure n'est pas additive à ce grain.
Correction	Gardez les mesures de niveau ligne (quantité, total de ligne) dans la table de faits des lignes de commande, et placez les mesures de niveau commande (minutes de livraison, frais de livraison, pourboire) dans une table fact_order distincte, au grain d'une ligne par commande, qui partage les mêmes dimensions conformes.
Vérifier	Dans fact_order : (20 + 40) ÷ 2 = 30 minutes, et SUM(delivery_fee) compte désormais chaque montant de frais une seule fois.

La première étape de conception de Kimball consiste à déclarer précisément le grain, avant de choisir les dimensions ou les faits, car chaque mesure doit être vraie à ce grain.

Mélanger des grains dans une même table est l'une des causes les plus courantes de tableaux de bord qui se contredisent.

Faire entrer les données : ETL, ELT et capture des données modifiées

Les données parviennent à l'entrepôt par des pipelines. Dans l'ETL classique (extract, transform, load : extraire, transformer, charger), un outil séparé extrait les données des systèmes sources, les nettoie et les remodèle sur ses propres serveurs, et ne charge que le résultat. Depuis que les entrepôts cloud ont rendu le stockage bon marché et le calcul élastique, l'**ELT** est devenu courant : extraire et charger d'abord les données brutes, puis les transformer à l'intérieur de l'entrepôt avec du SQL, souvent géré sous forme de modèles versionnés et testés avec un outil comme dbt. L'ELT conserve les données brutes, de sorte que les transformations peuvent être corrigées et relancées ; son risque est que des données à caractère personnel brutes se diffusent plus loin que nécessaire.

La manière dont les données sont extraites compte tout autant. Les copies complètes nocturnes sont simples, mais lentes et vite périmées. La **capture des données modifiées (CDC)** lit le journal de réplication propre à la base de données (le write-ahead log de PostgreSQL, le binlog de MySQL) avec un outil comme Debezium et diffuse chaque insertion, mise à jour et suppression sous forme d'événement, avec une faible latence et peu de charge sur la source. Quel que soit le mécanisme, lire directement les tables d'un autre service couple la plateforme analytique au schéma privé de ce service, exactement le problème décrit à la section 7.1. Le patron le plus robuste consiste pour chaque service à publier délibérément ses données, par des événements ou un export convenu, dans le cadre d'un contrat de données.

Data lakes, lakehouses et formats de table ouverts

Un data lake (lac de données), terme attribué à James Dixon (2010), stocke des données brutes provenant de nombreuses sources sous leur forme d'origine, généralement sous forme de fichiers sur un stockage objet bon marché comme Amazon S3, souvent dans des formats en colonnes comme Apache Parquet. Il reporte le schéma au moment de la lecture des données, ce qui convient aux data scientists qui veulent le détail brut et aux données dont l'usage n'est pas encore connu. Les lacs se sont fait la réputation de devenir des «marécages de données» (data swamps) : sans propriétaires, documentation ni contrôles de qualité, personne ne fait confiance à ce qu'ils contiennent. Les fichiers sur un stockage objet n'ont pas non plus ce qu'offre un entrepôt : transactions, mises à jour et suppressions, application du schéma et lectures cohérentes pendant l'écriture des données.

Le lakehouse (Armbrust, Ghodsi, Xin et Zaharia, CIDR 2021) ajoute ces fonctionnalités d'entrepôt par-dessus les fichiers ouverts du lac. La technologie qui le permet est le format de table ouvert (open table format) : **Delta Lake** (créé chez Databricks), **Apache Iceberg** (créé chez Netflix) et **Apache Hudi** (créé chez Uber). Chacun tient une couche de métadonnées (un journal de transactions ou une arborescence de fichiers manifestes) qui enregistre quels fichiers de données composent chaque version d'une table. Cela apporte des validations atomiques, des écritures concurrentes avec contrôle de concurrence optimiste, l'évolution des schémas, les mises à jour et suppressions au niveau des lignes (nécessaires pour les demandes d'effacement) et le **voyage dans le temps** (time travel) : interroger une table telle qu'elle était dans une version antérieure. Comme le format est ouvert, plusieurs moteurs (Spark, Trino, Flink et, de plus en plus, les entrepôts cloud eux-mêmes) peuvent lire et écrire les mêmes tables, ce qui réduit la dépendance au fournisseur. De nombreux lakehouses organisent les données en couches, popularisées par Databricks sous le nom d'architecture en médaillon : bronze pour les données brutes ingérées, argent pour les données nettoyées et conformes, or pour les modèles de niveau métier comme les schémas en étoile.

Traitement de flux : temps de l’événement, fenêtres et filigranes

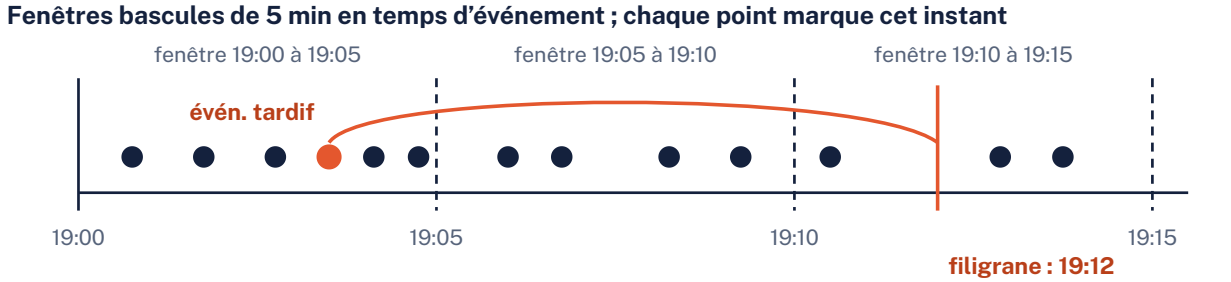
Les pipelines batch répondent à des questions sur la veille. Certaines questions ne peuvent pas attendre : cette carte est-elle utilisée frauduleusement en ce moment, les livreurs de Manchester sont-ils en retard ce soir, quels restaurants faut-il mettre en pause parce que leurs cuisines sont débordées ? Des processeurs de flux comme Apache Flink, Kafka Streams et Spark Structured Streaming calculent en continu sur des flux d’événements non bornés. Les concepts essentiels viennent du modèle Dataflow de Google (Akidau et ses collègues, VLDB 2015).

Le premier est la différence entre le temps de l’événement (event time), le moment où quelque chose s’est réellement produit, et le **temps de traitement** (processing time), le moment où le système l’observe. Ils diffèrent à cause des délais réseau, des nouvelles tentatives, du traitement par lots et des appareils qui se retrouvent hors ligne : le téléphone d’un livreur, dans un ascenseur, enregistre une remise à 19:03 mais transmet l’événement à 19:13. Tout résultat censé refléter la réalité (commandes par tranche de cinq minutes, délai moyen de livraison par heure) doit être calculé selon le temps de l’événement, sinon il se décalera dès que le pipeline ralentit et sera faux lorsqu’il sera rejoué.

Le deuxième est celui des **fenêtres**, qui découpent un flux non borné en morceaux finis à agréger : les fenêtres **bascules** (tumbling) sont fixes et ne se chevauchent pas (toutes les cinq minutes), les fenêtres **glissantes** (sliding) se chevauchent (les dix dernières minutes, mises à jour chaque minute) et les fenêtres de **session** regroupent par clé des rafales d’activité séparées par une période d’inactivité (la session de navigation d’un client).

Le troisième est le filigrane (watermark) : l’estimation évolutive, faite par le processeur, que tous les événements jusqu’à un temps d’événement donné sont probablement arrivés. Lorsque le filigrane dépasse la fin d’une fenêtre, le résultat de la fenêtre peut être émis. Les filigranes sont des heuristiques ; certains événements arrivent donc encore après la fermeture de leur fenêtre. Le pipeline a besoin d’une règle explicite pour ces **données tardives** : les ignorer, les envoyer vers une sortie secondaire pour une correction ultérieure, ou garder les fenêtres ouvertes pendant un retard autorisé et émettre des résultats mis à jour. Attendre plus longtemps donne des réponses plus complètes ; émettre plus tôt en donne de plus fraîches. Ce compromis entre latence et complétude est la version propre au traitement de flux d’un compromis entre attributs de qualité, et il doit être tranché avec le métier.

Temps de l’événement, fenêtres et événement tardif



Temps évén.
quand le téléphone du livreur a noté la remise du repas : 19:03

Temps traitement
quand le job de flux l’a vu, après le retour du réseau : 19:13

La fenêtre de 19:00 était déjà fermée quand le filigrane a passé 19:05 ; l’événement tardif exige une règle : l’ignorer, l’envoyer en sortie secondaire, ou mettre à jour la fenêtre dans le retard autorisé.

Les fenêtres sont définies par le temps de l’événement ; le filigrane décide de leur fermeture, et les événements tardifs exigent une règle explicite.

SI VOUS DÉBUTEZ

Pas à pas : compter selon le moment où cela s'est produit

L'idée la plus difficile de cette section est qu'un job de flux doit regrouper les événements selon le moment où ils se sont produits, et non selon celui où ils sont arrivés, puis décider combien de temps attendre les retardataires. Suivez un événement retardé à travers un simple décompte.

- Un job de traitement de flux (stream processing) compte les commandes acceptées par la tablette de cuisine de chaque restaurant dans des fenêtres de cinq minutes, afin de repérer les cuisines débordées. Entre 20:40 et 20:45, un restaurant très occupé accepte 25 commandes. Sa tablette perd le Wi-Fi juste après en avoir accepté une à 20:43, et n'envoie cette acceptation qu'à 20:57.
- Compté selon le temps de traitement, c'est-à-dire le moment où le job voit chaque événement, la fenêtre de 20:40 affiche 24 et la fenêtre de 20:55 gagne une commande acceptée bien plus tôt. Lors d'un rejeu le lendemain, toute la soirée est relue en quelques minutes, et les décomptes changent encore.
- Compté selon le temps de l'événement, l'heure enregistrée dans l'événement lui-même, l'acceptation retardée va dans la fenêtre de 20:40, à laquelle elle appartient, et un rejeu donne la même réponse à chaque fois.
- Cela soulève une nouvelle question : quand le job peut-il publier le décompte de 20:40, si un retardataire peut encore se présenter ? Il utilise un filigrane, son estimation évolutive que tous les événements jusqu'à un moment donné sont probablement arrivés. Lorsque le filigrane dépasse 20:45, le job publie 24.
- À 20:57, l'acceptation de 20:43 arrive, après la fermeture de sa fenêtre. Le pipeline a besoin d'une règle décidée à l'avance : l'ignorer, la mettre de côté pour une correction ultérieure, ou admettre un certain retard et publier un décompte mis à jour de 25.
- **La leçon :** calculez selon le temps de l'événement pour que les résultats reflètent ce qui s'est réellement passé, et convenez avec le métier du temps d'attente. Attendre plus longtemps donne des décomptes plus complets ; publier plus tôt en donne de plus frais.

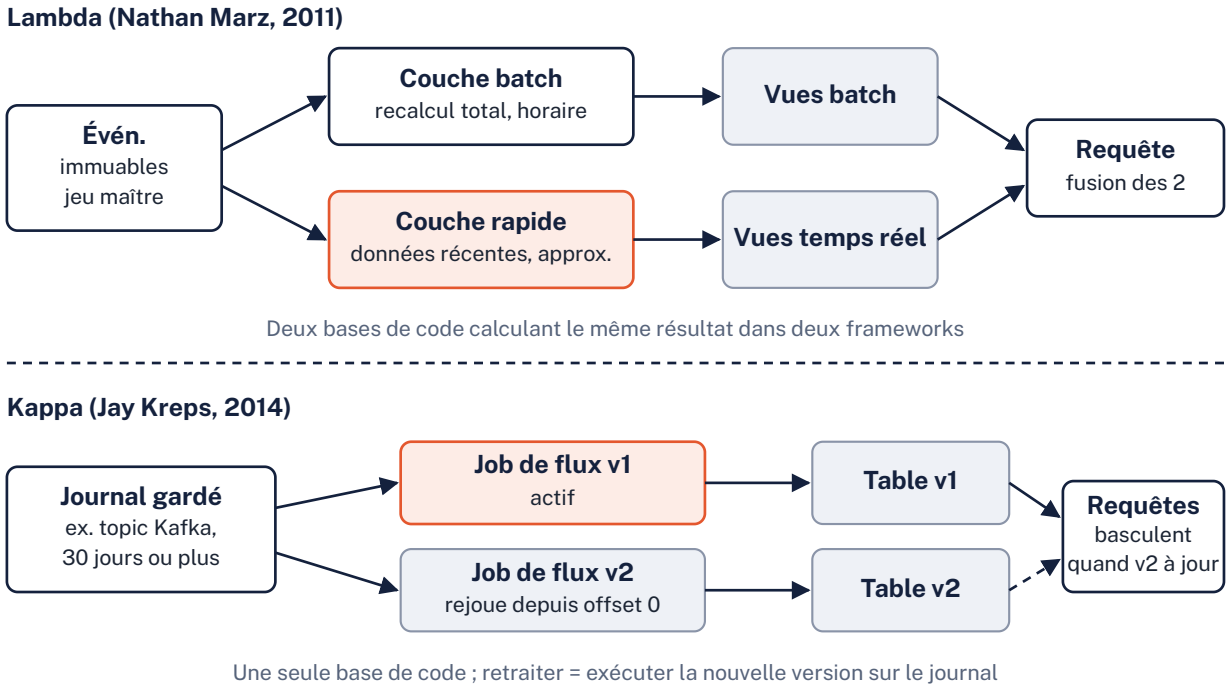
Architectures Lambda et Kappa

Comment combiner les résultats batch et streaming ? L'essai de Nathan Marz de 2011, «How to beat the CAP theorem», a décrit ce qui est devenu l'**architecture Lambda** : un jeu de données maître immuable alimente une **couche batch** qui recalcule périodiquement des vues exactes à partir de toutes les données, et une **couche rapide** (speed layer) qui calcule en temps réel des vues approximatives des données récentes ; les requêtes fusionnent les deux, et chaque exécution batch remplace les approximations de la couche rapide. Lambda apporte exactitude

et faible latence, et la couche batch sert de filet de sécurité contre les bugs du streaming. Son coût est que la même logique doit être écrite, et maintenue identique, dans deux frameworks différents.

En 2014, Jay Kreps, l'un des créateurs d'Apache Kafka, a remis en question cette conception et proposé l'**architecture Kappa**, un nom qu'il avançait à moitié en plaisantant. N'utilisez qu'un processeur de flux, et conservez l'entrée dans un journal assez longtemps pour pouvoir la rejouer. Pour changer la logique, lancez une seconde instance du nouveau job qui lit depuis le début du journal et écrit dans une nouvelle table de sortie, et faites basculer les lecteurs vers celle-ci une fois qu'elle a rattrapé son retard. Une seule base de code, et retraiter revient simplement à rejouer. Kappa dépend d'une longue conservation du journal et de processeurs de flux assez bons pour gérer correctement le temps de l'événement et l'état, ce que fournissent les moteurs modernes. Aujourd'hui, la plupart des nouvelles conceptions penchent vers un streaming de style Kappa avec un lakehouse pour l'historique, tandis que les conceptions de style Lambda persistent là où le recalcul par lots sur de très longs historiques coûte moins cher ou là où les pipelines batch existants inspirent confiance.

Lambda et Kappa côte à côte



Lambda garde des chemins batch et rapide séparés et les fusionne au moment de la requête ; Kappa a un seul chemin de streaming et retraite par rejeu.

Data mesh : la propriété des données analytiques

À mesure que les organisations grandissaient, les équipes de données centrales sont devenues des goulets d'étranglement : chaque nouvelle source et chaque nouvelle question faisaient la queue derrière une seule équipe qui ne comprenait très bien ni les systèmes sources ni les questions métier. En 2019, Zhamak Dehghani, alors chez Thoughtworks, a proposé le data mesh (développé ensuite dans son livre de 2022, *Data Mesh*). Il applique aux données analytiques la même décentralisation que les microservices ont appliquée aux systèmes opérationnels, et il repose sur quatre principes.

Propriété par domaine : les équipes les plus proches des données, comme celles des commandes, des paiements ou des livreurs, détiennent et servent leurs données analytiques, tout comme elles détiennent leurs services. **La donnée en tant que produit** : chaque équipe publie des produits de données (data products) faciles à découvrir, documentés, dignes de confiance, adressables et sécurisés, avec un propriétaire et des objectifs de qualité, en traitant les autres équipes comme des clients. **Plateforme de données en libre-service** : une équipe plateforme fournit le stockage, les pipelines, le catalogue et le contrôle d'accès qui rendent la publication d'un produit de données peu coûteuse, afin que chaque domaine n'ait pas à les réinventer. **Gouvernance computationnelle fédérée** : les règles globales (identifiants, classifications de confidentialité, normes d'interopérabilité) sont convenues conjointement et appliquées automatiquement par la plateforme plutôt que par un comité de revue central.

Le data mesh est une conception organisationnelle autant qu'une architecture, et les avis divergent sur le moment où il devient rentable. Ses partisans mettent en avant l'échelle : il supprime le goulet d'étranglement central et place la qualité des données là où se trouve la connaissance. Ses détracteurs font remarquer qu'il exige des équipes de domaine matures, une plateforme solide et un réel investissement, et que les organisations plus petites sont généralement mieux servies par une plateforme centrale bien gérée, avec une propriété claire de chaque jeu de données. La loi de Conway (section 3.7) s'applique directement : l'architecture des données reflétera la structure que l'organisation adopte réellement.

Durées de conservation, archivage et conservation légale (legal hold)

Les données ont besoin d'une date de fin autant que d'un propriétaire. Le principe de limitation de la conservation du RGPD britannique exige en général que les données à caractère personnel soient conservées sous une forme permettant l'identification pendant une durée n'excédant pas celle nécessaire ; chaque catégorie de données requiert donc une durée de conservation convenue avec des conseillers juridiques : les traces de localisation des livreurs peuvent être de courte durée, tandis que les pièces fiscales durent des années. L'**archivage** déplace vers un stockage moins cher les données qui doivent être conservées mais sont rarement lues, et il faut un moyen de les relire. La suppression doit atteindre les répliques, les

entrepôts, les journaux d'événements et les sauvegardes (section 8.7). Une **conservation légale** (legal hold) l'emporte sur le calendrier : les documents utiles à un litige peuvent devoir être préservés, et les tâches de suppression doivent donc d'abord vérifier l'existence de telles mesures. Amazon S3 Object Lock, par exemple, offre à la fois des durées de conservation et des conservations légales sans date d'expiration. Modélisez ces mesures sous forme de données, par client ou par commande, plutôt qu'en mettant une tâche en pause. Les durées dépendent du droit applicable.

Qualité, lignage, catalogues et confidentialité

Les données analytiques ne sont utiles que si l'on s'y fie. Les contrôles de **qualité des données** s'exécutent dans les pipelines et testent la fraîcheur (le chargement du jour est-il arrivé ?), le volume (le nombre de lignes a-t-il chuté de moitié ?), le schéma (une colonne a-t-elle changé de type ?), la validité (les montants sont-ils positifs et les codes postaux bien formés ?) et l'unicité. Un contrôle en échec doit arrêter le pipeline ou mettre les données en quarantaine plutôt que de les publier en silence. Le lignage des données (data lineage) enregistre d'où vient chaque jeu de données et quelles transformations l'ont produit, de sorte que lorsqu'un chiffre semble faux, ou qu'un schéma source change, vous pouvez en retracer l'impact en amont et en aval ; la spécification OpenLineage normalise la façon dont les pipelines le rapportent. Un **catalogue de données** rend les jeux de données faciles à découvrir, avec leurs descriptions, propriétaires, classifications et lignage en un seul endroit.

Les plateformes analytiques concentrent les données, ce qui les rend attrayantes pour les attaquants et risquées au regard du droit de la protection des données. Appliquez la minimisation des données (ne copiez pas les champs personnels dont aucune analyse n'a besoin), pseudonymisez les identifiants lorsque c'est possible, classifiez les colonnes pour que les politiques d'accès suivent les données, faites respecter les durées de conservation par une suppression automatisée, et veillez à ce que les demandes d'effacement atteignent le lac et l'entrepôt autant que les systèmes opérationnels ; les formats de table ouverts y aident parce qu'ils prennent en charge les suppressions au niveau des lignes. La section 8.7 présente les obligations en détail.

ATTENTION

Erreurs courantes

- Exécuter de lourdes requêtes analytiques sur la base OLTP de production, ou sur une réplique dont l'application dépend aussi pour le basculement en cas de panne.
- Construire des tables de faits sans déclarer le grain, si bien que moyennes et sommes sont faussées sans que rien ne le signale.
- Lire directement les tables privées d'autres services dans l'entrepôt, si bien que chaque changement de schéma interne casse un pipeline.
- Calculer les agrégats de streaming selon le temps de traitement, si bien que les résultats changent chaque fois que le pipeline prend du retard ou est rejoué.
- Traiter un data lake comme une décharge, sans propriétaires, contrôles de qualité ni catalogue.
- Adopter le vocabulaire du data mesh sans financer la plateforme en libre-service ni donner aux domaines une réelle propriété.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi est-ce une mauvaise idée d'exécuter un rapport de chiffre d'affaires sur deux ans sur la base de données de l'étape de paiement ?

Le rapport parcourt des millions de lignes et dispute le disque, la mémoire et le CPU aux petites transactions sensibles à la latence dont l'étape de paiement a besoin, et le schéma normalisé rend la requête complexe et lente.

▼ Quelle est la première décision que demande la méthode de Kimball lors de la conception d'une table de faits ?

Déclarer le grain : exactement ce que représente une ligne, par exemple un plat d'une commande livrée.

▼ Un job de flux compte les commandes par minute selon le temps de traitement. Que se passe-t-il lorsqu'il prend dix minutes de retard, puis rattrape ce retard ?

Les commandes sont comptées dans les minutes où elles ont été traitées plutôt que dans celles où elles se sont produites ; les décomptes sont donc faux, avec un creux suivi d'un pic ; calculer selon le temps de l'événement évite ce problème.

▼ **Comment l'architecture Kappa gère-t-elle un changement de la logique de traitement ?**

Elle exécute la nouvelle version du job de flux depuis le début du journal conservé vers une nouvelle sortie, puis fait basculer les lecteurs vers celle-ci une fois qu'elle a rattrapé son retard, et retire l'ancien job.

▼ **À vous : Une table de faits a une ligne par billet, et chaque ligne recopie les frais de la réservation. La réservation A a 1 billet et £3 de frais ; la réservation B a 3 billets et £5 de frais. Comparez les résultats corrects et naïfs. Que changeriez-vous ?**

Résultat correct : £8 de frais au total, £4 en moyenne par réservation. En additionnant les lignes de faits : 3 + 5 + 5 + 5 = £18 ; en en faisant la moyenne : 18 / 4 = £4,50. Les frais appartiennent au grain de la réservation, et non à celui du billet. Déclarez le grain, puis conservez une table de faits séparée avec une ligne par réservation, ou répartissez les frais entre les billets, de sorte que £5 deviennent environ £1,67 chacun, et indiquez-le clairement.

SOURCES

- Zhamak Dehghani: How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh (2019): https://martinfowler.com/articles/data-monolith-to-mesh.html
- Jay Kreps: Questioning the Lambda Architecture (O'Reilly Radar, 2014): https://www.oreilly.com/radar/questioning-the-lambda-architecture/
- Akidau et al.: The Dataflow Model (VLDB 2015): https://www.vldb.org/pvldb/vol8/p1792-Akidau.pdf
- Armbrust et al.: Lakehouse (CIDR 2021): https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf
- Kimball Group: Dimensional modeling techniques: https://www.kimballgroup.com/data-warehouse-business-intelligence-resources/kimball-techniques/dimensional-modeling-techniques/
- UK GDPR Article 5 (legislation.gov.uk): https://www.legislation.gov.uk/eur/2016/679/article/5
- Amazon S3 User Guide: Object Lock: https://docs.aws.amazon.com/AmazonS3/latest/userguide/object-lock.html

Lectures complémentaires

- *Designing Data-Intensive Applications (2nd edition)*, Martin Kleppmann and Chris Riccomini (2026), O'Reilly Media. La référence moderne sur la réplication, le partitionnement, les transactions, le traitement de flux et les données dérivées ; la plupart des mécanismes de ce chapitre y sont expliqués en profondeur.
  https://martin.kleppmann.com/2026/03/24/designing-data-intensive-applications-2e.html
- *Software Architecture: The Hard Parts*, Neal Ford, Mark Richards, Pramod Sadalage and Zhamak Dehghani (2021), O'Reilly Media. La propriété des données, les techniques de propriété conjointe, les patrons d'accès aux données distribuées, les désintégrateurs et intégrateurs de données, avec une étude de cas qui court tout au long du livre.
- *A Critique of ANSI SQL Isolation Levels*, Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil and Patrick O'Neil (1995), Microsoft Research technical report MSR-TR-95-51 (also ACM SIGMOD 1995). Le court article qui a défini l'isolation par instantané et les anomalies absentes de la norme SQL.
  https://www.microsoft.com/en-us/research/publication/a-critique-of-ansi-sql-isolation-levels/
- *Dynamo: Amazon's Highly Available Key-value Store*, Giuseppe DeCandia and colleagues (2007), ACM Symposium on Operating Systems Principles (SOSP). L'article à l'origine de la réplication sans leader, des quorums et du hachage cohérent avec nœuds virtuels dans Cassandra et ses proches.
  https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- *Choose Boring Technology*, Dan McKinley (2015), mcfunley.com. Un essai court et influent sur les jetons d'innovation et le coût à long terme de l'ajout de technologies, en particulier de bases de données.
  https://mcfunley.com/choose-boring-technology
- *The Data Warehouse Toolkit (3rd edition)*, Ralph Kimball and Margy Ross (2013), Wiley. Le guide de référence de la modélisation dimensionnelle, des schémas en étoile et des tables de faits et de dimensions pour l'analytique.
- *Data Mesh: Delivering Data-Driven Value at Scale*, Zhamak Dehghani (2022), O'Reilly Media. Le récit, à l'échelle d'un livre, des données analytiques possédées par les domaines, de la donnée en tant que produit, d'une plateforme en libre-service et d'une gouvernance fédérée.

Glossaire

Anomalie (phénomène)
Résultat erroné causé par l'interférence de transactions concurrentes, comme une lecture sale, une mise à jour perdue ou un write skew ; les niveaux d'isolation sont définis par les anomalies qu'ils empêchent.

Backfill (remplissage rétroactif)
Remplissage d'une nouvelle colonne ou table avec des valeurs pour les lignes existantes, généralement par petits lots afin de ne pas perturber le système en production.

Base de données d'intégration
Base de données partagée par plusieurs applications, qui s'intègrent par son intermédiaire en lisant et en écrivant dans les mêmes tables. Martin Fowler (2004) la déconseille parce qu'elle couple profondément les applications.

Base de données de séries temporelles
Base de données optimisée pour des mesures horodatées qui arrivent dans l'ordre, avec compression, sous-échantillonnage et suppression peu coûteuse des données anciennes.

Base de données vectorielle
Système de stockage pour les embeddings (des listes de nombres représentant le sens) qui trouve les éléments les plus similaires à une requête, généralement au moyen d'index de recherche approximative des plus proches voisins.

Cache stampede (ruée sur le cache)
Afflux de requêtes qui échouent toutes dans le cache au même moment, généralement quand un élément populaire expire, et frappent toutes en même temps la source plus lente ; étroitement lié au problème du troupeau tonitruant (thundering herd).

Cache write-behind
Patron de mise en cache dans lequel les écritures sont confirmées dès qu'elles atteignent le cache et sont écrites plus tard, en arrière-plan, dans le système de stockage sous-jacent ; on parle aussi de write-back.

Cache write-through
Patron de mise en cache dans lequel chaque écriture va dans le cache et dans le système de stockage sous-jacent avant d'être confirmée, de sorte que le cache n'est jamais plus ancien que le système de stockage.

Cache-aside
Patron de mise en cache dans lequel l'application consulte le cache, charge depuis la base de données en cas d'échec et place elle-même le résultat dans le cache ; aussi appelé chargement paresseux (lazy loading).

Compatibilité future (forward compatibility)
Une modification de schéma est compatible vers l'avant lorsque le code utilisant l'ancien schéma peut encore lire les données écrites avec le nouveau, généralement en ignorant les champs qu'il ne connaît pas.

Contrat de données
Accord explicite et versionné entre le propriétaire de données et ses consommateurs, couvrant le schéma, la signification, la qualité, la fraîcheur, la propriété et la manière dont les modifications sont annoncées.

Contrôle de concurrence multiversion (MVCC)
Technique dans laquelle les écritures créent de nouvelles versions des lignes au lieu de les écraser, de sorte que les lecteurs voient un instantané cohérent sans bloquer les écrivains.

Contrôle de concurrence optimiste
Détecter les conflits au lieu de les empêcher : une écriture ne réussit que si un numéro de version ou un horodatage n'a pas changé depuis la lecture des données ; sinon, l'écrivain fait une nouvelle tentative.

Crypto-shredding (effacement cryptographique)
Fait de rendre des données illisibles en détruisant la clé de chiffrement qui les protège, utilisé pour effacer des données à caractère personnel de journaux et de sauvegardes qui ne peuvent pas être modifiés.

Data lake (lac de données)
Stockage vaste et bon marché, généralement du stockage objet, qui conserve sous leur forme d'origine des données brutes provenant de nombreuses sources pour une analyse ultérieure.

Data mesh
Approche de Zhamak Dehghani pour les données analytiques, dans laquelle les équipes de domaine possèdent et publient leurs données sous forme de produits sur une plateforme en libre-service, sous une gouvernance fédérée.

Durée de vie (TTL)
La durée pendant laquelle un élément mis en cache peut être utilisé avant d'être considéré comme périmé et de devoir être rafraîchi ou revalidé.

Entrepôt de données (data warehouse)
Base de données conçue pour l'analyse, qui intègre des données nettoyées provenant de nombreux systèmes opérationnels et en conserve l'historique.

ETag
En-tête de réponse HTTP portant un identifiant d'une version d'une ressource, qu'un client peut renvoyer dans If-None-Match pour demander si elle a changé.

Expansion-contraction (expand and contract)
Réalisation d'un changement cassant par étapes sûres : ajouter la nouvelle structure à côté de l'ancienne, migrer les lecteurs, les rédacteurs et les données, puis supprimer l'ancienne structure. Aussi appelé changement parallèle (parallel change).

Filigrane (watermark)
L'estimation glissante d'un processeur de flux selon laquelle tous les événements jusqu'à un certain temps de l'événement sont probablement arrivés, utilisée pour décider quand le résultat d'une fenêtre peut être émis.

Flux d'événements
Suite ordonnée des événements d'un agrégat, numérotés par version, à partir de laquelle l'état de cet agrégat est reconstruit.

Format de table ouvert (open table format)
Spécification, comme Apache Iceberg, Delta Lake ou Apache Hudi, qui organise les fichiers de données et les métadonnées afin que de nombreux moteurs puissent les traiter comme des tables, avec transactions et voyage dans le temps (time travel).

Hachage cohérent (consistent hashing)
Manière d'attribuer des clés à des nœuds sur un anneau de hachage de sorte que l'ajout ou le retrait d'un nœud ne déplace qu'environ 1/N des clés (Karger et ses collègues, 1997).

Index secondaire
Index sur un attribut autre que la clé primaire ou la clé de partition. Dans un système partitionné, il est soit local à chaque partition, soit global et partitionné selon la valeur indexée.

Invalidation du cache
Suppression d'une copie en cache, ou son marquage comme périmée, lorsque les données d'origine changent, afin que les lectures suivantes récupèrent la nouvelle valeur.

Isolation par instantané (snapshot isolation)
Niveau d'isolation dans lequel chaque transaction lit, dès son début, un instantané cohérent des données validées, et où les écritures conflictuelles sur une même ligne sont rejetées. Il empêche les mises à jour perdues mais permet le write skew.

Journal d'écriture anticipée (write-ahead log, WAL)
Journal sur disque en ajout seul (append-only) dans lequel une base de données enregistre chaque modification avant de l'appliquer, afin que le travail validé survive à une panne ; les répliques sont souvent alimentées à partir de ce journal.

Lakehouse
Architecture qui ajoute des fonctionnalités d'entrepôt de données, telles que les transactions, les schémas et les requêtes rapides, à des données conservées dans des formats de fichiers ouverts sur un stockage objet.

Lecteur tolérant (tolerant reader)
Consommateur qui ne lit que les champs dont il a besoin et ignore tout ce qu'il ne reconnaît pas, afin que les fournisseurs puissent ajouter des champs sans le casser.

Lignage des données (data lineage)
Enregistrement de l'origine d'une donnée et de chaque transformation qu'elle a subie en chemin.

Migration
Script versionné qui fait passer un schéma de base de données ou ses données d'une version à la suivante, conservé dans la gestion de versions et appliqué automatiquement.

Mise à jour perdue (lost update)
Anomalie dans laquelle deux transactions lisent une valeur, calculent chacune une nouvelle valeur et l'écrivent, de sorte qu'une modification écrase l'autre sans avertissement.

Moins fréquemment utilisé (LFU)
Politique d'éviction qui retire, lorsque le cache est plein, l'élément lu le moins souvent.

Moins récemment utilisé (LRU)
Politique d'éviction qui retire, lorsque le cache est plein, l'élément resté le plus longtemps sans être lu.

Moteur de recherche
Système de stockage construit autour d'un index inversé des mots vers les documents, qui offre une recherche plein texte classée par pertinence et tolérante aux fautes de frappe ; généralement alimenté à partir d'un système de référence.

Niveau d'isolation
Paramètre qui indique dans quelle mesure des transactions concurrentes peuvent s'affecter mutuellement : read uncommitted, read committed, repeatable read, snapshot ou serializable. Les noms n'ont pas le même sens selon les bases de données.

OLAP
Online analytical processing (traitement analytique en ligne) : charges de travail faites de grosses requêtes qui parcourent et agrègent de nombreux enregistrements, comme le chiffre d'affaires par région et par mois.

OLTP
Online transaction processing (traitement transactionnel en ligne) : charges de travail faites de nombreuses petites lectures et écritures d'enregistrements individuels, comme passer des commandes ou enregistrer des paiements.

Partitionnement (sharding)
Découper un jeu de données de sorte que chaque machine n'en détienne qu'une partie, choisie au moyen d'une clé de partition, afin de faire passer les écritures et le stockage à l'échelle au-delà d'une seule machine.

Persistance polyglotte
Utiliser différents types de bases de données pour différentes données au sein d'un même système, choisis selon l'usage de chaque jeu de données, au prix d'un plus grand nombre de technologies à exploiter.

Produit de données (data product)
Jeu de données publié par une équipe de domaine pour que d'autres l'utilisent, avec un propriétaire, une documentation, des garanties de qualité et une interface stable.

Projection
Code qui écoute des changements ou des événements et tient à jour un modèle de lecture ; le terme désigne aussi le modèle de lecture ainsi produit.

Propriété conjointe
Situation dans laquelle plusieurs services écrivent tous les mêmes données. On peut la résoudre en scindant la table, en créant un petit domaine de données partagé, en confiant les écritures à un seul propriétaire ou en fusionnant les services.

Registre de schémas
Service qui stocke chaque version de chaque schéma de message et rejette les nouvelles versions qui enfreignent la règle de compatibilité convenue.

Regroupement des requêtes (request coalescing)
Laisser une seule requête aller chercher un élément manquant pendant que les autres requêtes portant sur le même élément attendent son résultat et le partagent ; on parle aussi de single flight.

Réplication
Conserver des copies des mêmes données sur plusieurs machines pour la disponibilité, la capacité de lecture et une latence plus faible à proximité des utilisateurs.

Réplication leader-suiveur
Schéma de réplication dans lequel un nœud (le leader, ou primaire) accepte toutes les écritures et transmet en continu ses modifications à des suiveurs (répliques), qui peuvent servir les lectures.

Réplication multi-leader
Schéma dans lequel plusieurs nœuds acceptent des écritures et se répliquent mutuellement, ce qui exige un moyen de détecter et de résoudre les écritures conflictuelles.

Réplication sans leader
Schéma inspiré de Dynamo dans lequel les clients écrivent et lisent directement sur plusieurs répliques et utilisent des quorums au lieu d'un leader unique.

Retard de réplication (replication lag)
Le délai entre une écriture sur le leader et son apparition sur un suiveur ; il conduit les utilisateurs à lire des données périmées.

Rétrocompatibilité
Une modification de schéma est rétrocompatible lorsque le code utilisant le nouveau schéma peut encore lire les données écrites avec l'ancien.

Schéma en étoile
Modèle dimensionnel composé d'une table de faits centrale contenant des mesures, entourée de tables de dimensions qui les décrivent, comme la date, le client et le produit.

Séparation commande-requête (CQS)
Principe de Bertrand Meyer selon lequel une méthode doit soit modifier l'état (une commande), soit renvoyer une réponse (une requête), mais pas les deux.

Sérialisabilité
La garantie d'isolation la plus forte : des transactions concurrentes produisent le même résultat que si elles s'étaient exécutées une par une dans un certain ordre.

Snapshot (instantané)
Copie stockée de l'état d'un agrégat à une version d'événement connue, utilisée pour éviter de rejouer un long flux depuis le début ; c'est un cache, pas une source de vérité.

Split brain (cerveau divisé)
Défaillance dans laquelle deux nœuds se croient tous deux leader et acceptent tous deux des écritures, si bien que leurs données divergent.

SQL distribué (NewSQL)
Bases de données relationnelles comme Spanner et CockroachDB qui conservent SQL et les transactions ACID tout en partitionnant et en répliquant les données sur de nombreux nœuds ou régions.

Système de référence (system of record)
La source qui fait autorité pour une donnée, dont la valeur l'emporte lorsque les copies divergent, comme le grand livre central d'une banque pour les soldes des comptes.

Table de dimensions
Table d'attributs descriptifs, comme le nom, le type de cuisine et la ville d'un restaurant, utilisée pour filtrer et regrouper les faits.

Table de faits
Table centrale d'un schéma en étoile, qui contient une ligne par événement métier à une granularité définie, avec des mesures numériques et des clés vers les dimensions.

Taux de succès du cache (cache hit ratio)
Fraction des consultations auxquelles le cache peut répondre à partir de son propre contenu ; les autres sont des échecs (misses) qui vont vers la source plus lente.

Temps de l’événement (event time)
Moment où un événement s’est réellement produit, par opposition au temps de traitement, moment où un système l’observe.

Upcasting
Convertir un événement stocké dans une ancienne version de schéma vers la version actuelle au moment de sa lecture, afin que le code de l’application n’ait à gérer qu’une seule forme.

Verrouillage pessimiste
Empêcher les conflits en verrouillant les lignes au moment de leur lecture (par exemple avec SELECT … FOR UPDATE), de sorte que les autres transactions attendent la libération du verrou.

Write skew (biais d’écriture)
Anomalie dans laquelle deux transactions lisent des données qui se recoupent, jugent chacune qu’une modification est sans danger et mettent à jour des lignes différentes, enfreignant ensemble une règle que chacune avait vérifiée.